



**MASTERING**  
**MACHINE CODE**  
on your  
**ZX SPECTRUM**

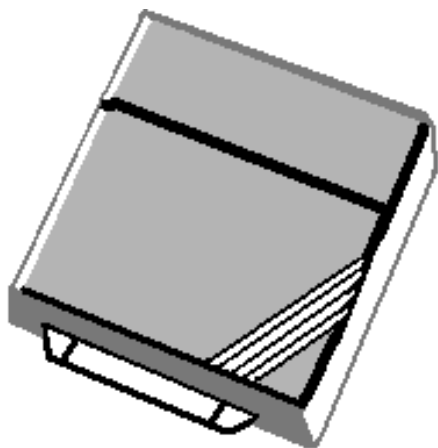
Toni Baker



# MASTERING MACHINE CODE ON YOUR ZX SPECTRUM

by Toni Baker

with illustrations by Cathy Lowe



**INTERFACE**   
**PUBLICATIONS**

First published in the UK by:  
Interface Publications,  
9-11 Kensington High Street,  
London W8 5NP.

Text © Copyright 1983, Toni Baker  
Artwork © Copyright 1983, Cathy Lowe

*All rights reserved. This book may not be reproduced in part or in whole without the explicit prior written permission of the publishers. The routines outlined in this book may not be used as part of any program offered for publication nor for programs intended to be sold as software, except as allowed by the publisher. Permission must be sought, in advance, for all applications of this material beyond private use by the purchaser of this volume.*

ISBN 0 907563 23 6

Cover Illustrator, Stuart Hughes

Typeset and Printed in England by Commercial Colour Press,  
London E7.

Further information on Spectrum Machine Code may be obtained from  
the ZX Machine Code Users Club.

Toni Baker  
37 Stratford Road  
Wolverton,  
Milton Keynes,  
MK12 5LW



# CONTENTS

## **CHAPTER ONE – INTRODUCTION**

A brief summary of the book.

## **CHAPTER TWO – INTRODUCTION TO HEXADECIMAL AND MACHINE CODE**

Computers count in sixteens not tens. The system is called hexadecimal and this chapter will help you to get to know it.

## **CHAPTER THREE – SIMPLE ARITHMETIC**

“Simple” • means *very* simple! Plusses and takes only. Shares and timeses are left till later!

## **CHAPTER FOUR – PEEKING AND POKEING AND MORE ABOUT LOADING**

An explanation of how to use memory in RAM.

## **CHAPTER FIVE – STACKING AND JUMPING**

How to use the stack to store data. Jumping and conditional jumping is explained, and the use of subroutines explained.

## **CHAPTER SIX – TAKING BYTES TO BITS**

Or how to count on your thumbs. This is called binary, and reveals the innermost secrets of how computers count!

## **CHAPTER SEVEN – PRINTING THINGS TO THE SCREEN**

In BASIC the PRINT statement is one of the most widely used of all. Here's how to use it in machine code.

## **CHAPTER EIGHT – A DICTIONARY OF MACHINE CODE**

All the instructions. A complete explanation of every single machine language instruction used by the Spectrum.

## **CHAPTER NINE – MORE PLACES TO STORE MACHINE CODE**

A very explicit guide to the use of REM statements, program area and protected areas of RAM. In addition – how to pass parameters from BASIC to machine code using FN.

## **CHAPTER TEN – A PROGRAM TO HELP YOU DEBUG**

A machine code editing program, itself written largely in machine code. The speed and power it offers will develop your machine code to fluency very rapidly.

## **CHAPTER ELEVEN – SCANNING THE KEYBOARD**

Using the keyboard in programs has obvious advantages. Here we learn how to simulate the function INKEY\$. An elegant little program called GRAFITTI is developed which demonstrates how the character set is generated.

## **CHAPTER TWELVE – DRAUGHTS PART ONE**

The first part of this program, which allows a player to input a move, and checks for cheats!

## **CHAPTER THIRTEEN – A TOUCH OF CULTURE**

Music and pictures. Music from the keyboard, and pictures from the screen. Watch out for the program LIFE.

## **CHAPTER FOURTEEN - DRAUGHTS PART TWO**

The output of the computer's move. This section doesn't do the actual "thinking", but it will output a move assuming the "thinking" to have already been done.

## **CHAPTER FIFTEEN – GRAPHICS GAMES**

Not up to the space invaders standard I'm afraid! However, listed here is a complete working BREAKOUT program, and a rather cute little program called SPIRALS.

Mastering Machine Code on your ZX Spectrum

## **CHAPTER SIXTEEN – DRAUGHTS PART THREE**

The making of the big decision... Which move?

## **CHAPTER SEVENTEEN – DISASSEMBLING THE ROM**

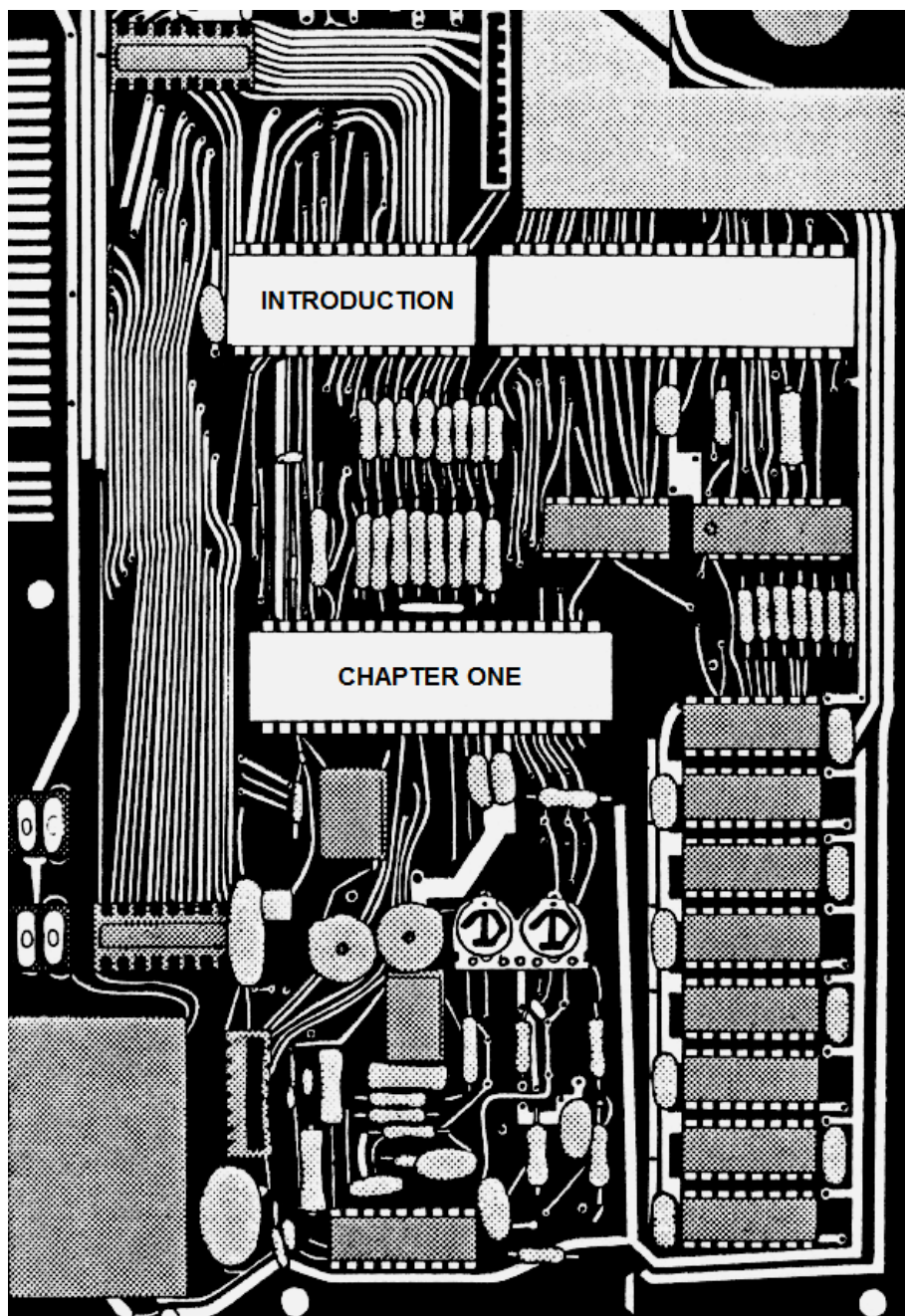
The ROM holds many secrets, but it, and any other machine code program, may be disassembled fairly simply. A full disassembling program is listed in this chapter to enable you to do this.

## **CHAPTER EIGHTEEN – FLOATING POINT ARITHMETIC**

Have you ever wondered how floating point numbers work? How can you add and subtract them in machine code? Multiply and divide them? Even take sines and cosines? This chapter will tell you how.

## **APPENDICES**

Useful information you may need when writing programs in machine code.



# CHAPTER ONE

## INTRODUCTION

This book is designed for those people who have a reasonable understanding of BASIC but whose knowledge of machine code is zero. Starting at first principles with BASIC programs, we gradually introduce the concept of a machine code subroutine and develop this theory throughout the book – eventually into full machine code programs. Before long you'll find your understanding of machine code increasing, and very soon begin writing your own machine code programs and routines.

Machine language is no more than a second computer language – very much like BASIC is, in fact. We start by learning the simplest of instructions and become familiar with them by accessing them via BASIC programs.

Printing strings is soon covered, and this involves making use of the PRINTing subroutine in the ROM. The routine is demonstrated by printing a draughts board, which later on in the book we shall make use of.

We explain the machine code equivalent of the INKEY\$ function, and use the technique of scanning the keyboard to write a typewriter-type program which uses greatly enlarged versions of the keyboard characters.

The same keyboard scanning technique is used to generate musical notes in a rather surprising manner. Two octaves are produced from your machine, enabling you to play a wide variety of tunes at a touch of the keys.

The computer is made to generate many intricate and fascinating displays in the program LIFE. It challenges the skill of an unwary human

operator in graphics games such as SPIRALS and BREAKOUT. A draughts program is also included, with several interesting features. Each program is explained in full so that you may understand the purpose of each instruction in the program.

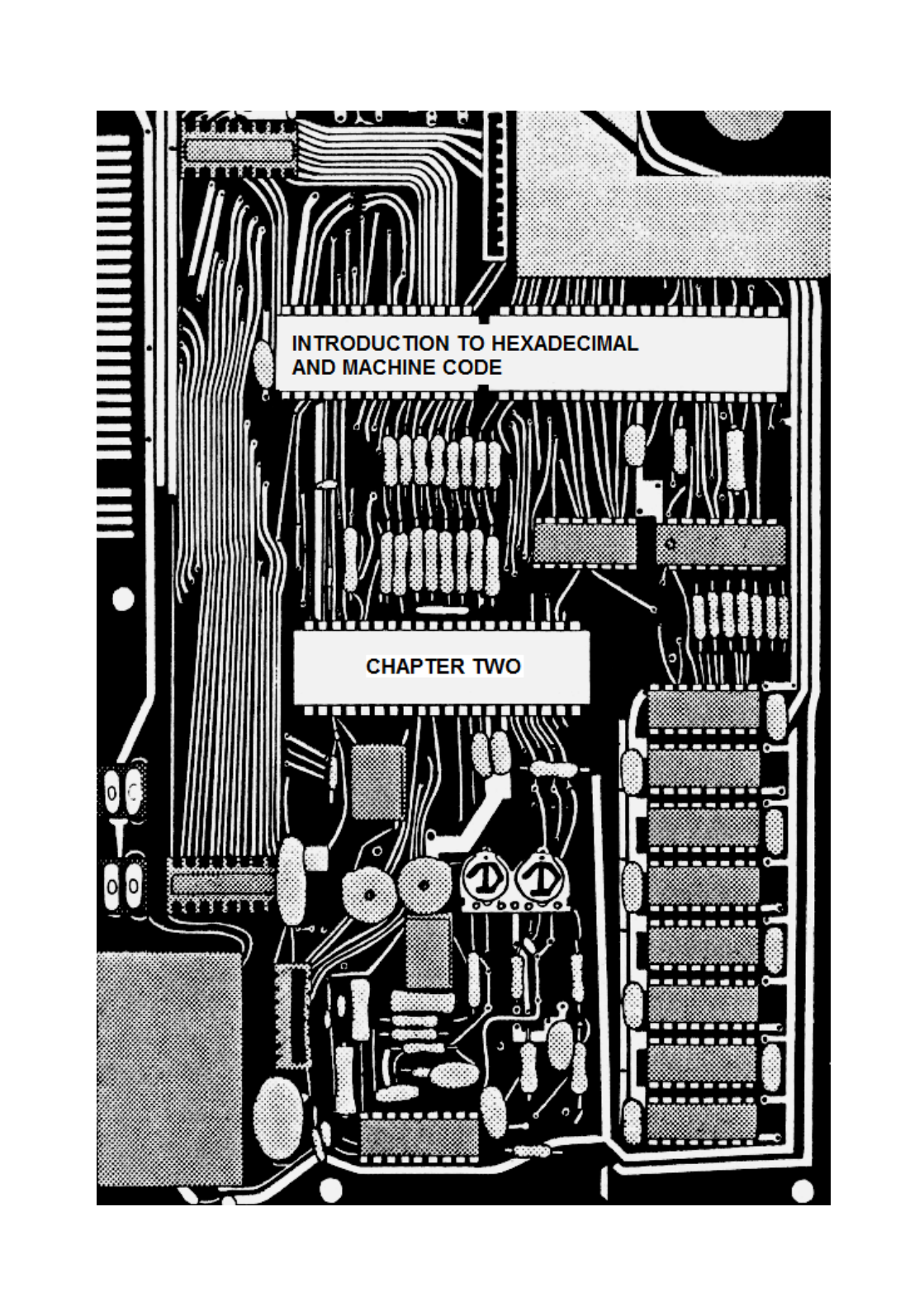
Careful study of the listings of these programs will teach you a great deal about machine code but, of course, the biggest steps in learning will come from experiment. Write your own programs – or adapt mine (by all means do – they are intended for this purpose); some, in fact, are deliberately improvable for this purpose (although, of course, all of them will work quite adequately as they stand).

To make the best use of this book you are advised to work from start to finish and where asked to alter or improve programs, you should make an attempt to do so. It's not difficult since the book progresses very slowly, but it will require some thought.

The last two chapters in the book are rather ambitious. A program by which the ROM may be disassembled is given, and the arithmetic subroutines are explained – even complicated functions like SIN and COS are explained!

The heavily packed appendices at the back are designed to be used as a source of reference throughout the book. Any piece of information you need to know can generally be found in these appendices, or in Chapter Eight which is a kind of “dictionary” of machine code.

The first real chapter starts on the next page, and begins with an introduction to the use of “hexadecimal”.



INTRODUCTION TO HEXADECIMAL  
AND MACHINE CODE

CHAPTER TWO

# CHAPTER TWO

## INTRODUCTION TO HEXADECIMAL AND MACHINE CODE

OK, so your nice new ZX Spectrum is all fired up and ready to go, and that little copyright message or report code (or even a smug little flashing “K” if you're really lucky) is sitting there glaring at you and waiting for you to type something in. What do you do?

Well, the first thing is to set up the machine so that it can accept machine code instead of BASIC. This is not difficult, but unfortunately for us, when Sinclair designed his machine he forgot to include a button saying “GO INTO MACHINE CODE”, so the procedure for doing this is going to have to be a BASIC program.

Before we can do anything, we have to make some room somewhere in the computer's memory at which to store our machine code. Type:

```
16K: CLEAR 28671  
48K: CLEAR 61439
```

The effect of this is quite straightforward. Take a look at figure 2.1 – this is a sort of “Before and After” diagram, showing what goes where at the upper end of the Spectrum's memory. (You might like to compare this with the diagram on page 165 of the Spectrum instruction manual.) The area from address 32600 (16K) or 65368 (48K) up to address 32767 (16K) or 65535 (48K) is all used by the computer, but *below* that (ie to be left of it in the diagram) is all spare down to the part of memory that is used by BASIC. We can change this address using CLEAR, which moves all the bits and pieces used by BASIC such that the very last address available to basic is the number after the word CLEAR. In the examples I gave above, CLEAR 28671 would make 28671 the last





(without quotes round it), before typing ENTER. This should stop the program with report code H STOP in INPUT. Note that although this program will *enter* machine code, it will *not* attempt to RUN it.

Now for the big question you've all been dying to ask – what exactly *is* machine code? Well, machine code, or machine language as it's otherwise known, is another computer language – much like BASIC is – only at a much lower level, which means that instructions which are even *slightly* complicated (such as FOR/NEXT loops for instance) are simply not available. However, this also makes it quite an easy language to learn. Like BASIC it consists of a set of instructions, each of which tells the computer to do a different, but quite specific, task. One such instruction is RET, which is more or less equivalent to BASIC's RETURN.

Unlike BASIC, however, the computer can't read the words if they're written in "English". You can't RUN the above BASIC program and simply input "RET" because the poor old Spectrum just wouldn't understand it. To make life a little easier for this overworked machine, every machine language instruction has its own numerical code, and these codes *can* be understood directly. For example, the code for RET is 201. Every code lies somewhere between zero and 255, and so it is usually much more convenient to write these codes in a system called *hexadecimal*.

## COUNTING IN HEXADECIMAL

Briefly, hexadecimal, or *hex* for short, is a means of counting which uses sixteen symbols instead of ten. The first ten numbers are the same as the ones we're used to. These are:

0, 1, 2, 3, 4, 5, 6, 7, 8, 9.

But there are six new symbols which represent the numbers ten to fifteen. These are:

A, B, C, D, E, F.

The full really starts when we want to represent numbers bigger than fifteen, for believe it or not, sixteen is written as 10! Worse still, seventeen is written as 11 and eighteen is written as 12. This continues up to twenty-five, which is written 19, and when we come to twenty-six we have to start using the new symbols again: twenty-six is written 1A, and so on.

If you turn to the back of the book you'll find a table called Appendix One. The first block of numbers is a complete table of all the numbers from zero to two hundred and fifty-five. This is intended to help you to understand the hexadecimal system of counting. You should try to refer to it as little as possible, but don't worry if you find yourself using it all the time at first as you'll find you get used to it much quicker than you expect.

The symbols down the left-hand side are the first hex digit, the symbols along the top are the second. The leading zeros may, of course, be omitted if they occur, but it is often more convenient to leave hex codes in two digits instead of one (ie with 7 written as 07).

If there is ever any confusion about whether a number is written in hex or not you should make it clear by writing a small letter "h" (for hex) or a small letter "d" (for decimal) after the number. Thus, 19d means nineteen, whereas 19h means twenty-five. In most cases, however, there will not be any confusion. Numbers like CD for instance cannot possibly be in decimal, so there is no need to write CDh. (CDd makes no sense at all!) Conversely, there is a convention that hex numbers are usually written with an even number of digits, so that 118 (which has an odd number of digits) means 118d. 118h could be written as 0118.

Knowing at least the fundamentals of counting in hex is virtually paramount as far as machine code is concerned, so don't be afraid to keep coming back to this section, or to keep referring to the table in Appendix One – that's what it's there for.

Incidentally, 19h is pronounced “one nine” or “one nine hex” when spoken. It is *never* pronounced “nineteen” or “nineteen hex”!

There are some fundamental differences between machine code and BASIC. One of these fundamental differences is that of *line numbers*.

As you know, every BASIC statement in a program must be preceded by a line number so that the computer knows in which order to execute them. If no line number is given, then the computer will interpret the instruction as a *command* and will execute it immediately.

In machine code, there are no line numbers. The Spectrum will not allow you to use machine code instructions as commands, they *must* form part of a program. The instructions will be executed in the order that they are stored. For example, if the computer had just finished executing the instruction which was stored at address 30000, it would then go on to execute the instruction held in location 30001. It will continue in this manner until it receives an instruction telling it to do otherwise.

Unlike BASIC, it will not STOP when it reaches the end of the program. It will plough right on through the addresses, and every time it finds a number other than zero it will treat that number as a code for some instruction and try to execute it. Usually this will result in what is called a *crash*.

## ABOUT CRASHING

Crashing is the name we give to what happens when your (up until now, at least, moderately well behaved) Sinclair machine unwittingly tries to execute something it shouldn't, or to do something it can't. Even a minor mistake in your machine coding can confuse the poor machine and give it a rather nasty headache. When the machine crashes, the keyboard will usually stop working straight away. Sometimes NEW will be executed, sometimes the screen will freeze, sometimes it will go blank, but best of all, sometimes the screen will go into its "LET'S PRODUCE SOME MODERN ART MODE". If this happens you will soon know all about it!

When this happens you will undoubtedly try to break out, by using the BREAK key, and will discover to your horror that the break key doesn't work! In fact *none* of the keys will work after a crash. This is the prime reason we dislike crashes, for the only way to get back to normal is to *disconnect the power supply!* When you reconnect it you will, of course, have lost all your program and will have to reLOAD the lot.

This is the horror of using machine code; if a BASIC program contains a mistake then it will simply *not work*, but if a machine code program contains a mistake then it will more often than not crash!

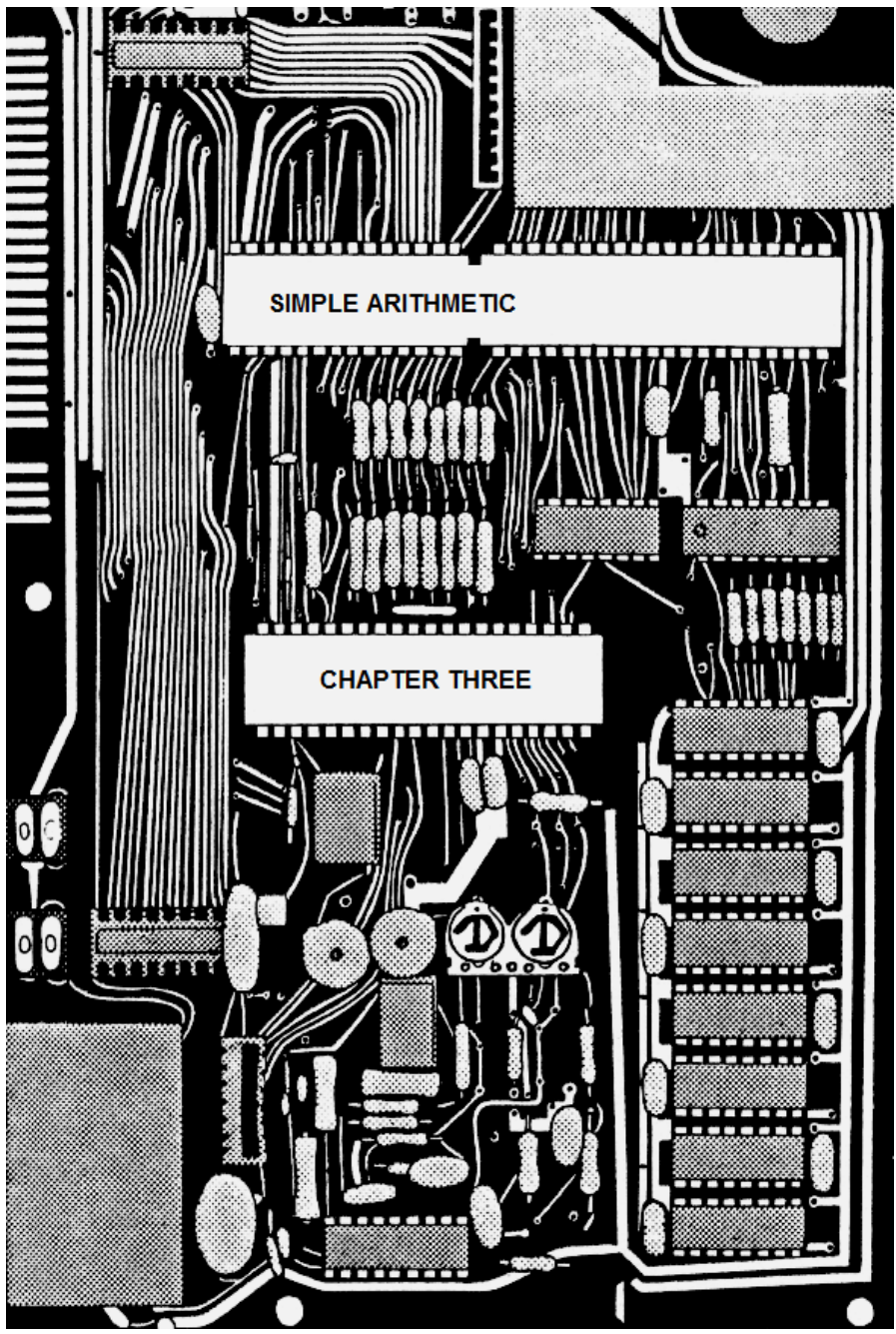
## HOW TO PREVENT CRASHES

The only way to prevent crashes is to ensure that your programs are "mistake-free". We have already stated that a program will not automatically stop at the end of a program, so it must be told to do so with a specific instruction – that instruction is RET (RETurn to BASIC).

This chapter has dealt with how to reserve space for machine code programs, and has given a program with which to load it. It has *not* told you how to make use of this program, nor how to run machine code

programs once they are loaded. The fundamentals of counting in hex have been introduced, and the notion of a crash has been mentioned.

Once you have understood this chapter you may turn to Chapter Three for your first lesson in machine code programming.



SIMPLE ARITHMETIC

CHAPTER THREE

# CHAPTER THREE

## SIMPLE ARITHMETIC

Now it's time to dig out the program I asked you to type in the last chapter. Unless you've done anything silly, like switch the machine off (in which case you'll have to retype it) it should still be sitting in front of you. SAVE it under the name "HEXLD". Type CLEAR 28671 or CLEAR 61439 if you haven't already done so, and then type RUN.

The program is now waiting for a string input – a very special kind of string input. What it actually wants is some kind of hexadecimal number. Whenever you want to input a machine language instruction you have to know its numerical code, and you have to know it in hex.

The code for RET is, as we have already stated, 201. What is this in hexadecimal? Divide it by sixteen and you get twelve remainder nine. Now the hex symbol for twelve is C, and the hex symbol for nine is 9. If you look 201 up in Appendix One you'll find that it is written in hex as C9. Is this a coincidence?

Input "C9". You have now told the computer that the first instruction of your program is RET. Now break out of the program by deleting the quote marks and inputting STOP. Your program is now complete. It consists of just one instruction – the instruction RET. We usually write this:

```
C9      RET
```

to remind us that the hex code for RET is C9. The machine language instructions are sometimes called opcodes to distinguish them from their corresponding hex codes. C9 is a hex code, RET is an opcode. Hex codes are used by the machine – it will not understand opcodes.



Conversely, opcodes are used by humans, because we would find it very difficult to think in hex codes.

If you look at the screen you'll see the messages H STOP in INPUT. The machine has gone back to command mode and is waiting for an instruction. Suppose we now wish to run the machine code program we have just typed in. We can do this either within a program, or, as we are going to do, as a direct command. Since the program was loaded to address 28672 (16K) / 61440 (48K) then the command is:

```
16K: PRINT USR 28672  
48K: PRINT USR 61440
```

You will have found that the computer will just print 28672 or 61440 onto the screen. Indeed you might just as well have typed PRINT 28672 or PRINT 61440. But think about what has happened. You gave the computer a number – the number after the word USR. The computer then carried out a machine code program consisting of the single instruction RET – ie it did nothing. Since it did nothing, it is not surprising that the number we gave it should end up unchanged, and was in fact PRINTed (by BASIC, since we actually asked it to PRINT something) unchanged. But do take some encouragement from this exercise. You have just used the instruction USR and, hopefully, you should have understood why it did exactly what it did. You have run a machine code program, though simple, but without crashing and knowing why it works. Try, if you like, experimenting with other ways of using USR, but with the same number after it, eg LET X = USR 61440 or PRINT PEEK USR 61440, and so on.

Before we can advance to learning any more instructions we are going to have to break for a while and explore the concept of *registers*. A register is like a variable, in that it has a name – usually a letter of the alphabet – and it can store numbers in the same way that BASIC variables can. The big difference is that registers can only store numbers in the range 00 to FF (or) in decimal, 0 to 255).

There are seven registers we can use easily. Their names are A, B, C, D, E, H and L. Note that because the largest number that any of them can hold is FF then a register effectively can only hold a two digit hex number. I think that this is rather easier to remember than “a number between 0 and 255”.

To give a greater degree of flexibility, some of the registers may be used in pairs. B and C, for instance, can be used as a pair. When this is done the pair is written as BC and is treated as a single unit. Since B holds two hex digits and C also holds two hex digits, BC as a pair will hold four hex digits. If B contains 4A and C contains 23h, then we say that BC holds 4A23.

We can also do the same in decimal, but it is much harder – this is why I prefer to keep everything in hex. I’ll use exactly the same numbers. If B contains 74d and C contains 35d then we say that BC holds 18979. Notice that in decimal we *can’t* do the calculation in our head – it’s not just a question of glueing the digits together as we did in hex; we can’t combine 74d and 35d and get 7435d, sorry, but it just doesn’t work as it gives the wrong answer. In decimal we need a calculator (a Spectrum will usually do) to work out 256 times the first number plus the second number, ie  $256*74+35=18979$ . I hope you get the picture. When you start using machine code, counting in hex really does make things so much simpler.

There are three register pairs possible. They are BC, DE and HL.

## THE INSTRUCTION LD

Consider the BASIC instruction LET A = 42. In machine language we assign our variables (registers) using the instruction LD. We could, for example, write LD A,2A. Note that there is no equals symbol as there is in BASIC. Instead a comma is used to separate the A from the number. The effect of this instruction is exactly what you’d expect it to be – the

previous value of A is overwritten and a new value (in this case 2A – decimal 42) is assigned in its place.

Each different LD instruction has a different code. For example, the code for LD A, is 3E. This means that the full instruction LD A,2A can be written in hex as 3E2A. Note that this instruction is *two* bytes in length (a byte is two hex digits) whereas RET occupied just *one* byte (the byte C9). In case you're interested, the BASIC instruction LET A = 42 occupies eleven bytes, plus another four for the line number (or one for a colon if it's not the first statement in a line), plus one more for the carriage return byte if it's the last statement in a line. Machine code therefore is a *significant* reduction in space.

The remaining codes are as follows:

```
LD A, 3E
LD B, 06      LD BC, 01
LD C, 0E
LD D, 16      LD DE, 11
LD E, 1E
LD H, 26      LD HL, 21
LD L, 2E
```

Using the program "HEXLD", enter the following program by inputting the symbols in the left-hand column. (The symbols in the right-hand column are only there for the benefit of humans – the computer doesn't want to know about them.) Input "0600", then "0E2A", then "C9", and then break out.

```
0600      LD B,00
0E2A      LD C,2A
C9        RET
```

The program will load B with zero and C with 2A. Hence, it will load BC with 002A. In decimal, BC will contain  $256 \times 0 + 42$ , or 42. Now type

PRINT USR 28672 (16K) or PRINT USR 61440 (48K) and what do you get? Answer – 42. Did you expect that? Now try something else. Try this program:

```
1600      LD D,00
1E2A      LD E,2A
C9        RET
```

Now when you type PRINT USR 28672 or 61440, something different happens.

This is exactly what happens when the computer comes across the phrase “USR something” in any BASIC line. Firstly, the number “something” is loaded into the BC register pair. (The number in the BASIC line will be in decimal, but you can turn it into hex if you want.) Then control will jump to address “something” and will carry out any machine code it finds there. When it hits a RET instruction it will go back to working in BASIC again, but it will “remember” the number that ended up in BC. It is quite possible then to have a line like LET A = USR 20000 + USR 30000 which will run *two* machine code programs, firstly the one at 20000, and then the one at 30000. The variable A will be assigned the sum of two different quantities: the one remembered by the first program and the one remembered by the second program. Note that the final value of DE and HL (and indeed of A) will *not* be remembered. For want of a better word, they will be “forgotten”.

HL, by the way, stands for High/Low. When HL is treated as a pair, H is called the high part and L is called the low part. BC and DE also have high and low parts: the first letter is the high part and the second letter is the low part.

We have established that forty-two in hex to four digits is 002A. What do you think the following program will do?

```
01002A      Remember 01 means LD BC,....
```

```
C9          RET
```

You may be surprised to discover that when you type PRINT USR 61440 you actually get the answer 10752 *not* 42! Now run this program:

```
012A00     Notice the 00 and the 2A have been switched around.  
C9          RET
```

Now you will get forty-two. The 00 and the 2A need to be swapped around so that they're in the "wrong order". Although this is rather strange it is, in fact usual for machine code. All of the instructions which deal with four digit numbers have the two bytes swapped around so that the low part comes *first* and the high part comes *second*. This, in fact, is the way that numbers are stored by BASIC in the system variables when they're two bytes long. In the instruction LD HL, the first byte is always 21h, the second byte is the new value of L, and the third byte is the new value of H. Note that this is always three bytes long.

## LOADING FROM ONE VARIABLE TO ANOTHER

If we were restricted in BASIC to only using LET instructions of the form, LET A = a number, we would be a bit stuck. We need to be a bit more flexible than that. For instance, something like LET A = B would be useful. Well, we can certainly manage that in machine code. The codes are listed in figure 3.1.

LD	A	B	C	D	E	H	L
A	7F	78	79	7A	7B	7C	7D
B	47	40	41	42	43	44	45
C	4F	48	49	4A	4B	4C	4D
D	57	50	51	52	53	54	55
E	5F	58	59	5A	5B	5C	5D
H	67	60	61	62	63	64	65
L	6F	68	69	6A	6B	6D	6E

Figure 3.1

In the table, you read the left-hand column registers first and the top row registers second, so that the code for LD D,A is 57 and the code for LD A,D is 7A. Notice how each of these is a mere one byte in length. Compare this with the equivalent BASIC instruction LET A = D, which takes four bytes, plus any extra for line number, colon and/or carriage return.

And now for some simple arithmetic. Those of you who have been thinking ahead may have been wondering how we can add and subtract registers in machine code. After all – if LD A,B takes just one byte then how can we cope with finding equivalents to LET A = B + C.

In fact, we use a different instruction altogether. The instruction is ADD. A useful example would be ADD HL,DE which has the effect LET HL = HL + DE. The contents of register pair HL are added to the contents of register pair DE and the result of this addition is stored in HL, overwriting its former value.

You can actually do arithmetic in hex on paper. Take a look at these sums and see how they work. Just like in decimal you add the units first, but you only carry one if you go above fifteen.

1234 + 1234 <hr style="width: 100%;"/>	1234 + 5678 <hr style="width: 100%;"/>	5678 + 9ABC <hr style="width: 100%;"/>	89AB + 9630 <hr style="width: 100%;"/>
2468	68AC	F134	11FDB

In the first example there are no surprises. In the second we have  $4 + 8 = C$  and  $3 + 7 = A$ . In the third example we need to carry:  $8 + C = 14(h)$  – that is 4 carry 1.  $7 + B + 1$  is 13h, or 3 carry 1.  $6 + A + 1$  is 11h, or 1 carry 1.  $5 + 9 + 1$  is F with no carry. If you managed to follow that then the fourth example should cause no headaches, but notice a problem: 11 FDB is too big to be stored in two registers for it has too many digits. Suppose we tried to execute ADD HL,DE when HL started off as 89AB and DE as 9630 – what do you think would happen?

## *Mastering Machine Code on your ZX Spectrum*

Before we find that out, let's demonstrate it with an easier example. Enter and run this program, and try to predict its result before you see it.

```
3EFF      LD A,FF (in decimal 255)
0601      LD B,01
80        ADD A,B (note that FF + 01 = 100 in hex)
0600      LD B,00
4F        LD C,A
C9        RET
```

Remember it is only the final value of BC that is returned to BASIC, so the last instructions simply move the answer we've calculated from A into BC – they're not part of the calculation. Look at what the program does: it tries to add one to two hundred and fifty-five, but it tries to do so in a register which can only store numbers up to 255 and no more. What will happen? Run and find out (by typing PRINT USR 28672 or PRINT USR 61440).

You should have got zero as your answer. FF + 01 is certainly 100h, but only the last two digits are stored in A.

To answer our earlier question, 89AB + 9630 may add up to 11FDB in real life, but if we tried to compute that answer in HL only the last four digits would remain: 1FDB.

There are only two registers that you can add things to: they are A and HL. You cannot, for instance, ADD B,C! Also you should note that only single registers may be added to A, and that only register pairs may be added to HL.

The instructions you are allowed are thus only those in the following list. Their hex codes are also given.

```
ADD HL,BC 09      ADD A,A 87
ADD HL,DE 19      ADD A,B 80
```

```

ADD HL,HL 29      ADD A,C 81
                  ADD A,D 82
                  ADD A,E 83
                  ADD A,H 84
                  ADD A,L 85

```

Since we've been thinking about what happens when the answer gets too big, I'd like to introduce you to a new kind of machine code "variable" – it's called a *flag*. Whereas a register may store numbers between 00 and FF, a flag is even more restricted since it can only store one of two possible values – either zero or one.

One such flag is called CARRY. It has a very special purpose. Whenever an ADD instruction takes place then CARRY is re-assigned. Its new value is always the "third digit" of a two digit sum, or the "fifth digit" of a four digit sum. If we go back to our four examples of sums in hex we can see that the following will happen:

1234 + 1234 = 2468	CARRY = 0, HL = 2468
1234 + 5678 = 68AC	CARRY = 0, HL = 68AC
5678 + 9ABC = F134	CARRY = 0, HL = F134
89AB + 9630 = 11FDB	CARRY = 1, HL = 1FDB

Do you follow? CARRY becomes 0 whenever the addition is carried out with no hiccups, but becomes 1 whenever the answer gets too big to fit into the required register.

Two special words go with flags. *Set* means "assigned one", and *reset* means "assigned zero". Thus, a flag may be *set*, or *set to one*, or *reset* or *reset to zero*. I have heard people incorrectly use the phrases "set to zero" or "reset to one", etc. Please try not to use these because they don't really make sense.

There is another instruction we can use instead of ADD. The instruction is called ADC, which stands for "ADD with Carry".



## *Mastering Machine Code on your ZX Spectrum*

It works like this. Suppose the machine comes across the instruction ADC A,B. It will take the contents of register B, and it will add this to the contents of register A, and then it will add the former value of CARRY. A will be assigned the result of this addition, and the carry will be set or reset as necessary (depending on whether the result is greater than FF).

So: ADD A,B effectively means

LET A = the last two digits of A + B

LET CARRY = 0 if A + B ≤ FF

1 if A + B > FF

Whereas: ADC A,B effectively means

LET A = the last two digits of A + B + CARRY

LET CARRY = 0 if A + B + carry ≤ FF

1 if A + B + CARRY > FF

Study the programs that follow. If the final value of the A register is unimportant, then these programs are equivalent (ie they both do the same thing) – can you understand why?

The first program is:

```
118533      LD DE,3385h
21C77B      LD HL,7BC7
19          ADD HL,DE
44          LD B,H
4D          LD C,L
C9          RET
```

The effect of both of these programs is the same. You can learn two things from this. Firstly the instruction LD does not in any way affect or alter the value of CARRY, for if it did the two LD instructions between ADD A,E and ADC A,D would really mess things up. Secondly the instruction ADD HL,DE is much shorter (and much neater) than going all

round the houses by adding each byte separately. And never forget to swap the order of the two bytes round in LD instructions on pairs.

Now run both of these programs just to verify that they are the same. What would happen if the ADC A,D in the second program were replaced by ADD A,D?

Now that you understand the difference between ADD and ADC, we shall go on to cover some other ways of adding. First, though, the codes for ADC:

ADC HL,BC ED4A	ADC A,A 8F
ADC HL,DE ED5A	ADC A,B 88
ADC HL,HL ED6A	ADC A,C 89
	ADC A,D 8A
	ADC A,E 8B
	ADC A,H 8C
	ADC A,L 8D

Notice how the codes for ADC HL, are all *two* bytes long rather than one. The first byte is always ED and the second byte depends on what you are adding. You must not think of ED as meaning ADC HL though, since ED can mean a wide variety of other things as well, depending on what comes after it.

## ADDING CONSTANTS

We can also ADD and ADC to add numerical constants to the A register (but not to HL). An example would be ADD A,3 which would, as you'd expect, add three to the current value of A. It would also assign CARRY to one or zero depending upon whether the result overflows beyond FF. (Remember FF is a *number* – it means 255 in decimal.)

The code for ADD A, is C6, and the code for ADC A, is CE. ADD A,3 then would be written in hex as C603. When adding constants two bytes are always necessary. Suppose we wished to add 003A to HL. One way would be as follows:

```
113A00      LD DE,003A
19          ADD HL,DE
```

But this method has the disadvantage of bringing the DE register into use and overwriting its former value. Another way of achieving the same thing, but this time only overwriting the A registers is this:

```
7D          LD A,L
C63A        ADD A,3A
6F          LD L,A
7C          LD A,H
CE00        ADC A,00
67          LD H,A
```

Notice how the instruction ADC A,0 was used to add any carry digit there may have been from adding 3A to L.

## **AND FINALLY (WELL, FOR ADDITION ANYWAY)....**

There is one last way that we can add constants to a register, and that is by using the instruction INC. INC A means add one to the contents of the A register. INC D means add one to the D register. INC HL means add one to the value of HL, and so on.

However, unlike any of the addition instructions we've learned so far, INC is unique in that it does not change the value of the carry. If CARRY was zero before an INC instruction then it will remain as zero afterwards. If CARRY was one before an INC instruction then it will

remain one afterwards. This rule applies even if INC “increments” (say) A from FF to 00. This then is a fundamental difference between ADD A,01 and INC A. The codes for INC are:

```

INC BC 03   INC A 3C
INC DE 13   INC B 04
INC HL 23   INC C 0C
             INC D 14
             INC E 1C
             INC H 24
             INC L 2C

```

Remember, the difference between ADD A,01 and INC A is that ADD A,01 will assign a new value to CARRY, but INC A will leave it unaltered. INC, by the way, is short for increment.

The value of CARRY can be altered *directly* without any of the other registers being affected. There is an instruction called SCF (which stands for Set Carry Flag) whose job is to assign to CARRY a value of one. It does not change the value of any of the registers. You can also set the carry to zero, with a bit of ingenuity, using the instruction ADD A,0 (think about it). Finally there is another way of changing the value of CARRY. An instruction called CCF (or Complement Carry Flag) is designed to *change* the value of the carry, either from zero to one, or from one to zero, without changing the registers. The codes for these three instructions are:

```

37           SCF           "LET CARRY = 1"
C600        ADD A,0       "LET CARRY = 0"
3F          CCF           "LET CARRY = 1 - CARRY"

```

## SUBTRACTION

In machine language there are codes for subtraction, just as there are for addition. The instruction is called SUB which is short for SUBmarine. (Only joking!) However, the only register you can subtract from is A – you can't subtract from HL. For this reason, the instruction "SUB A,B" is usually written as just SUB B. The A is just "assumed" to be there. In exactly the same way as with ADD there is also an instruction called SBC, for SuBtract with Carry. SBC, however *can* be used on HL, and so the "A" has to be written in. (Awkward isn't it.)

It works like this: SUB B will take the value of the register B and subtract it from the value of A. If A starts off greater than B then the subtraction is carried out as you'd expect and the carry flag is reset (ie to zero). If A starts off less than B, however, then it is assumed that A is representing a number which is 0100 (hex) greater than the actual value of A – in this case, the carry will be set (ie to one). Thus, if A contains 03 and B contains 05 then the actual operation carried out is not 03 – 05 (which would be negative) but is 103 – 05, or FE, and the carry set.

The allowed instructions are:

SUB A	97
SUB B	90
SUB C	91
SUB D	92
SUB E	93
SUB H	94
SUB L	95
SUB a number	D6 followed by that number

Note that there are no instructions to subtract register pairs.

SuBtract with Carry (SBC) on the other hand, *will* work for register pairs, but only with the register pair HL.

SBC A,C for instance, will subtract the value of C from the value of A, and will then subtract the initial value of CARRY from this result. CARRY will be re-assigned in the same way as for SUB. The codes for SBC are:

SBC HL,BC ED42	SBC A,A 9F
SBC HL,DE ED52	SBC A,B 98
SBC HL,HL ED62	SBC A,C 99
	SBC A,D 9A
	SBC A,E 9B
	SBC A,H 9C
	SBC A,L 9D
	SBC A, a number DE followed by that number

DEC is short for DECrement. It is, as you may have gathered from its weird sounding name, the opposite of INC. Its purpose is to decrease the value of any register by one without changing the carry flag, so DEC DE has the effect  $LET DE = DE - 1$ .

And now for a little exercise: one of the following two routines will successfully subtract DE from HL and will always get the answer right – the other will not always be so accurate. Which is which? And why?

The first routine is:

C600	ADD A,0
D602	SUB 2
ED52	SBC HL,DE

and the second routine is:

## Mastering Machine Code on your ZX Spectrum

C600	ADD A,0
3D	DEC A
3D	DEC A
ED52	SBC HL,DE

In fact, the first example is wrong. The instruction SBC HL,DE will subtract both DE and the carry flag, so we have to make sure that the carry flag is zero. This is what the instruction ADD A,0 is for. Having done that, however, the first example will alter the carry flag *again* with the instruction SUB 2. The chances are that subtracting two won't cause any overflow problems; however, if A happens to equal 01 or 00 then SUB will not only change A to FF or FE, it will also set the carry flag to *one*, so that the effect of SBC HL,DE would be to assign HL a value of  $HL - DE - 1$ , *not*  $HL - DE$ . In the second example, the instruction DEC A is used twice. DEC will not change the carry flag, so it will still be zero when the instruction SBC HL,DE is reached, and the subtraction will then go ahead correctly.

Got it? INC and DEC do not alter the value of the carry flag – the other arithmetic instructions do. The other instructions we've covered, RET and LD, do not change the value of CARRY at all.

DEC BC 0B	DEC A 3D
DEC DE 1B	DEC B 05
DEC HL 2B	DEC C 0D
	DEC D 15
	DEC E 1D
	DEC H 25
	DEC L 2D

In this chapter we have dealt with how to load machine code programs, and how to run them. The use of the instructions RET and LD were explained, and the arithmetic instructions, ADD, ADC, SUB and SBC, were introduced along with INC and DEC. The purpose of the carry flag has been covered, and the instructions SCF (Set Carry Flag) and CCF

(Complement Carry Flag) have been mentioned.

You are *not* expected to remember any of the hex codes which the computer uses – not even the experts do that! All of the codes are printed in an appendix at the back of this book. All you have to know are the words we use for them – the opcodes and what they do.

Before you proceed to Chapter Four, see if you can tackle some of the following exercises. If you find some of them difficult, don't worry about it – just take them slowly and think clearly.

Enter the following machine language program using HEXLD. You will have to look up the various hex codes yourself:

```
LD BC,0
LD HL,0
ADD HL,BC
LD BH
LD C,L
RET
```

Now run the program. What answer do you get? If you got zero, well done! But if not, then you did something fundamentally wrong. The instructions LD BC, and LD HL, are all *three* byte instructions – this means that, for instance, LD HL,0 becomes 210000, not just 2100. What instructions did you really give the computer to make it come up with your answer? Now try again until you get zero.

If you delete HEXLD by typing NEW then the machine code program will *still be there*. This is because the machine code is stored in a portion of the memory which doesn't get touched by BASIC. Now type in the following program:

```
10 INPUT A
20 INPUT B
```



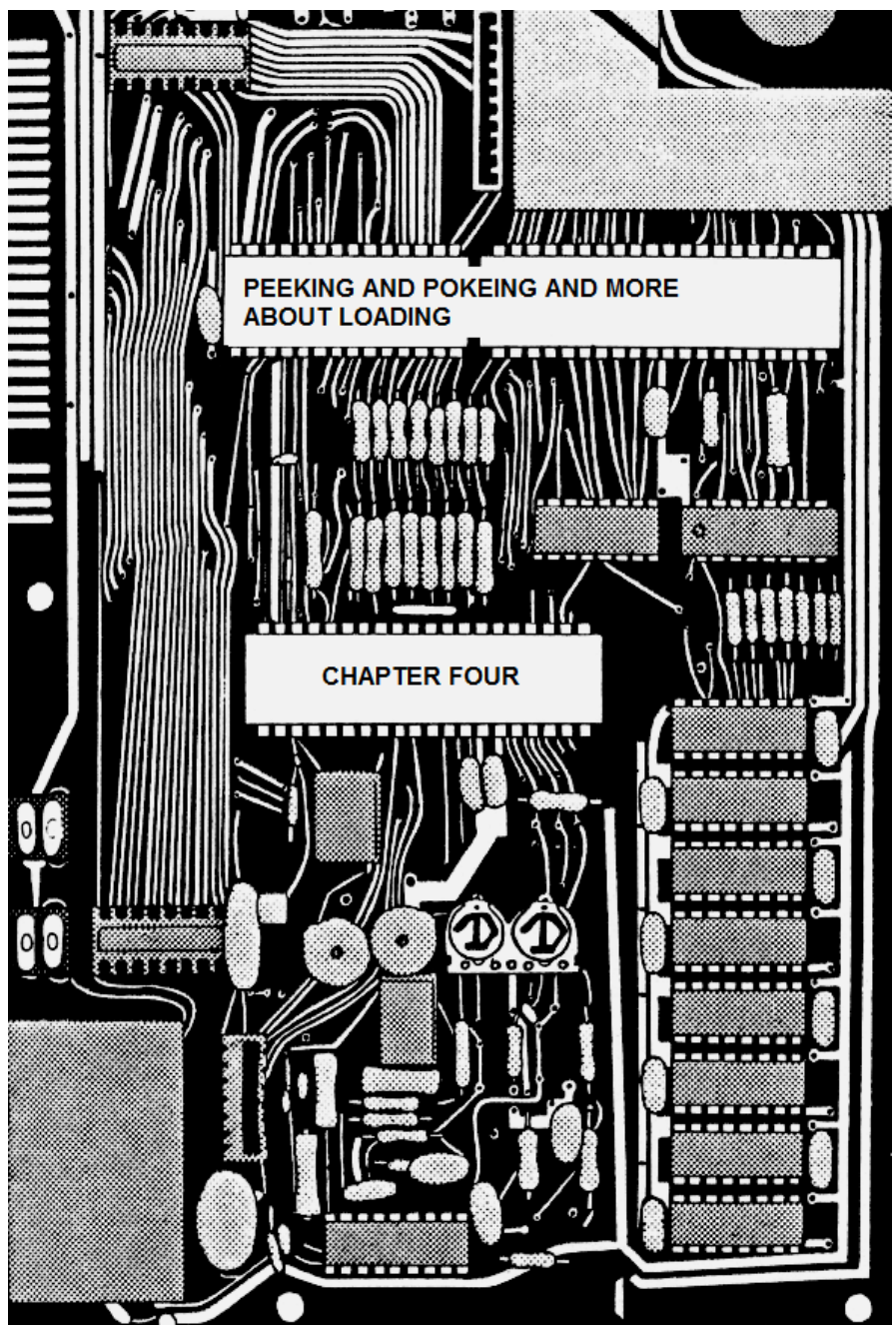
*Mastering Machine Code on your ZX Spectrum*

```
30   POKE 61441,A – 256*INT (A/256)  (use 28673 if you have 16K)
40   POKE 61442,INT (A/256)          (use 28674 if you have 16K)
50   POKE 61444,B – 256*INT(B/256)   (or 28676)
60   POKE 61445,INT(B/256)          (or 28677)
70   PRINT A,B
80   PRINT USR 61440                  (or 28672)
90   PRINT
100  GOTO 10
```

This BASIC program will overwrite the second, third, fifth and sixth bytes of the machine code program. It means that the value you input for A will be POKEd into the LD BC instruction, and the value you input for B will be POKEd into the LD HL instruction. Run the program and input some values to see what happens. Try going outside the range 0 to 65535.

Now, see if you can write a similar BASIC program which, when combined with a slightly modified version of our machine code program, will print a table of values of A and B on the screen, and the result of subtracting A from B in each case. Let A and B take on all of the values one to ten inclusive.

Write a machine code routine which will produce a one if BC is greater than or equal to DE, and a zero otherwise. How could you test this? (Hint – see previous exercise on this page.)



PEEKING AND POKEING AND MORE  
ABOUT LOADING

CHAPTER FOUR

# CHAPTER FOUR PEEKING AND POKEING AND MORE ABOUT LOADING

For those of you who thought that maybe seven registers might not be enough, it's just as well we can PEEK and POKE and thus make use of all of the addresses in RAM. (The RAM, which stands for Random Access Memory by the way, is the portion of memory which we are capable of altering – the addresses numbered from 16384 upward.) If there's any number we have to store somewhere, either permanently or temporarily, then it makes sense to just POKE that number somewhere (almost anywhere will do); then when we need it again all we have to do is PEEK at that address and *voila* – there it is!

## A LESSON IN PEEKING

If you've ever seen any machine language printed anywhere, you may have wondered why obscure brackets keep turning up all over the place. What, for example, is the difference between LD HL,5C65 and LD HL,(5C65)? It's not just for variety or to make it look pretty, they do actually mean something! Brackets around a number or register pair will refer to the contents of the *address* in the brackets, so:

	LD HL,5C65	means	LET HL = 23653
and:	LD HL,(5C65)	means	LET HL = PEEK 23653 + 256*PEEK 23654

In case you haven't worked it out yet, 23653 is the decimal equivalent of 5C65.

The second example may have confused you. The only address in brackets is 23653, so how does 23654 come into it? What has happened is a kind of side effect. H and L can each hold one byte, so HL together must hold two bytes altogether. The address 23653 only holds one byte, so another one has to come in from somewhere. In practice, this other byte comes from the next possible address – in the above case, 23654. The real effect of the instruction LD HL,(5C65) is LET L = PEEK 23653 followed by LET H = PEEK 23654.

There is also a reverse instruction which is:

```
LD (5C65),HL
```

This is effectively POKEing. The result of the instruction is:

```
POKE 23653,HL – 256*INT (HL/256)
POKE 23654,INT (HL/256)
```

Or, if you think of H and L separately:

```
POKE 23653,L
POKE 23654,H
```

In BASIC this particular pair of instructions is used quite frequently. I'll give you an example. Suppose you've just written a BASIC program and you want to know how long it is. You can find out the number of bytes your program occupies by typing in the expression PRINT (PEEK 23627 + 256\*PEEK 23628) – (PEEK 23635 + 256\*PEEK 23636). A very simple machine code program to calculate this value would be:

```
ED5B535C          LD DE,(23635d)
2A4B5C            LD HL,(23627d)
C600              ADD A,0
ED52              SBC HL,DE
44                LD B,H
```

*Mastering Machine Code on your ZX Spectrum*

4D	LD C,L
C9	RET

The ADD A,0 is used to set the carry flag to zero, so that the immediately following SBC instruction will always produce the correct answer. Remember there is no such instruction as SUB HL,DE so if we ever need to subtract DE from HL we are forced to use SBC instead. This won't subtract properly unless CARRY equals zero.

Notice how the code for LD HL,(23627d) is built up. The first byte is 2A. Now, although you're not expected to remember this, the last time we used a LD HL instruction the code was 21h. The difference is the brackets! *LD instructions which use brackets have a completely different hex code.* The next two bytes are 4B and 5C; this is the number 23672d in hexadecimal – if you divide 23627 by 256 you get ninety-two (hex 5C) remainder seventy-five (hex 4B). In the hex code, these bytes have been switched around to give 4B5C rather than 5C4B. You must always remember to do this in machine code.

If you store this machine code program above RAMTOP (using CLEAR as described earlier) then you can then type in or LOAD any BASIC program and then find its length by typing the (by now familiar) command PRINT USR 61440 (or 28672).

23627 will *always* contain the address of the first byte after the end of your BASIC program – this is its job. It is one of the *system variables* which are used by the ROM to help it keep track of what it's doing. Likewise 23635 will always store the address at which the program starts. You can verify this by reading about the system variables VARS and PROG on page 174 of the Spectrum manual, and then looking at the diagram on page 165. If you alter one of these values, either by POKEing or LDing, then the poor machine will get very confused although, as we shall see later, this is sometimes an advantage.

Make sure you understand exactly how the above program works, and why every line is needed. The most important instruction is still the first one we learned – RET. If any of the others are missing then you will get the wrong answer, but at least you'll get an answer. Without RET the program will crash!

Not all of the variables can be LDed from addresses by themselves. The instructions you are allowed to use, together with their codes and a breakdown of what they do, are listed here:

3A	LD A,(pq)	LET A = PEEK pq
ED4B	LD BC,(pq)	LETC = PEEK pq
		LET B = PEEK (pq + 1)
ED5B	LD DE,(pq)	LET E = PEEK pq
		LET D = PEEK (pq + 1)
2A	LD HL,(pq)	LET L = PEEK pq
		LET H = PEEK (pq + 1)

And POKEing:

32	LD (pq),A	POKE pq,A
ED43	LD (pq),BC	POKE pq,C
		POKE pq + 1,B
ED53	LD (pq),DE	POKE pq,E
		POKE pq + 1,D
22	LD (pq),HL	POKE pq,L
		POKE pq + 1,H

You will notice that only the variable A may be assigned a PEEK value, or POKEd anywhere, by itself – all of the other registers must be used in pairs. Usually this is quite a useful feature, but there will be times when you'll want to assign a single register (a usual choice is L) without disturbing the value of A. There isn't really any way round this I'm afraid, but what you can do is to assign both halves of a register pair as

## Mastering Machine Code on your ZX Spectrum

described above, and then reset the appropriate single register to zero afterwards.

Suppose you want to know how far down the screen the PRINT position is. If you look in the instruction manual you'll find that PEEKing 23689 will tell you exactly that (well, 24d minus that anyway). The problem is to LD this into BC, because the number we're after is *one byte* long – it isn't stored in either 23688 or 23690. One way of doing it is this:

```
ED4B895C          LD BC,(23689d)
0600              LD B,00
C9                RET
```

As you can see, the first instruction will successfully load the contents of 23689 into the C register as required, but it will also (as a side effect) load the contents of 23690 into the B register. So the B register must therefore be reset to zero in order that BC contains the same value as C alone before we return to BASIC, otherwise the figure printed at the end will be virtually meaningless.

The other way of getting PEEK 23689 into BC is to go via the A register, since the A register can be LDED directly all by itself. But as you will see this offers no advantages since we still need to reset B anyway:

```
3A895C           LD A,(23689d)
0600             LD B,00
4F              LD C,A
C9              RET
```

If you still aren't convinced that the second instruction is necessary, try omitting it to see what happens. Can you see what is wrong? If the value of B is not changed then the value of BC will be wrong (by an amount 256d\*B).

Both of the above programs, as they are written, will have the same effect – they will tell you twenty-four minus the first co-ordinate of the current PRINT position. That is, they will tell you the number of squares below the PRINT position including all the lines in the lower half (half?) of the screen.

Try feeding in one of the above programs, and then enter this BASIC program:

```
10  FOR I = 0 TO 20
30  PRINT USR 61440          (or 28672 for 16K users)
50  NEXT I
```

Run the program and see if you can work out why it does what it does. Now insert more lines:

```
20  FOR J = 0 TO 3
30  PRINT TAB (8*J);USR 61440  (this overwrites previous line 30)
40  NEXT J
```

And again, RUN it and see what happens. Interesting, no?

## POKEING IN MACHINE CODE

POKEing is just as easy. To put line 50 of any BASIC program at the top of the screen at the next automatic listing you can POKE 23660,50 then POKE 23661,0. (This will only work if the current line – that is the line with the “greater-than” cursor – is actually on the screen when line 50 at the top.) In machine code:

```
213200          LD HL,0032          (0032h = 50d)
226C5C          LD (5C6C),HL        (5C6B = 23660d)
C9             RET
```



## *Mastering Machine Code on your ZX Spectrum*

Move the cursor down to any line number greater than fifty. Note that here, for the first time, it doesn't matter what number is "remembered" by the machine code routine and returned to BASIC, for the object of the exercise is simply to perform a double POKE. Since we don't care what BC ends up as there is very little point in PRINTing it, so PRINT USR is not necessary. Try using RANDOMIZE USR instead. Notice that when you do this line 50 still moves to the top of the screen (that's what the machine code's job is) but no number is printed – obviously, if you don't type PRINT then PRINT won't happen. So what happens to the number "remembered" by the routine – that is, the number left in BC when RET is reached? The answer is found by reading all about RANDOMIZE on pages 73 and 74 of the Spectrum Manual – the number becomes the new random number seed. This needn't worry us, of course. All we really need to know is that PRINT USR and RANDOMIZE USR will both run a machine code program, but PRINT will PRINT the final value of BC, whereas RANDOMIZE will not.

Now look at the hex code of LD (5C6C),HL. The first byte is 22h. This is the code for LD (pq),HL, where pq represents some arbitrary address. The remainder of the code is 6C5C, which is the number 23660d in hexadecimal (with of course the first and last byte switched around). So even though we humans would write our opcode with the (5C6C) first and the HL second, the machine language code always requires the instruction itself first, even though the instruction actually incorporates the HL at the end of the opcode.

There are some other PEEK and POKE instructions, but which use register names throughout. These are:

0A	LD A,(BC)	LET A = PEEK BC
1A	LD A,(DE)	LET A = PEEK DE
7E	LD A,(HL)	LET A = PEEK HL
46	LD B,(HL)	LET B = PEEK HL
4E	LD C,(HL)	LET C = PEEK HL
56	LD D,(HL)	LET D = PEEK HL

5E	LD E,(HL)	LET E = PEEK HL
66	LD H,(HL)	LET H = PEEK HL
6E	LD L,(HL)	LET L = PEEK HL
02	LD (BC),A	POKE BC,A
12	LD (DE),A	POKE DE,A
77	LD (HL),A	POKE HL,A
70	LD (HL),B	POKE HL,B
71	LD (HL),C	POKE HL,C
72	LD (HL),D	POKE HL,D
73	LD (HL),E	POKE HL,E
74	LD (HL),H	POKE HL,H
75	LD (HL),L	POKE HL,L

If you study the codes of the instructions that have “(HL)” in them you’ll see that they form a regular pattern. Now, if you fit these codes into a table of all of the LD instructions of the form LD A,B you’ll see an even more stunning regular pattern. Take a look at figure 4.1.

From this table it looks very much as though there ought to be a LD of (HL),(HL) instruction with code 76. In actual fact there isn’t! Code 76 represents an entirely different instruction called HALT, but we’ll cover that later on.

Why is any variable in brackets a register pair rather than a single register? Why is any variable not in brackets a single register rather than a register pair? If HL contained 5C8F, what would be the difference between LD B,(HL) and LD BC,(5C8F)? What is the precise effect of each? See if you can write a machine code routine to assign to HL a value of PEEK 23693 *only*, using one of the LD,(HL) instructions.

LD	B	C	D	E	H	L	(HL)	A
B	40	41	42	43	44	45	46	47
C	48	49	4A	4B	4C	4D	4E	4F
D	50	51	52	53	54	55	56	57
E	58	59	5A	5B	5C	5D	5E	5F
H	60	61	62	63	64	65	66	67
L	68	68	6A	6B	6C	6D	6E	6F
(HL)	70	71	72	73	74	75	76	77

Figure 4.1

We have now covered all of the basic LD instructions which operate on the registers A, B, C, D, E, H and L. We shall now take a look at some of the other ways of loading these variables.

## HOW TO LOAD BLOCKS

Loading *blocks* means loading huge chunks of memory all in one go. For example, if you had a machine code routine at address 30000 and you wanted to move it to location 20000, then if you were really patient you could do something like this:

```

11204E          LD DE,20000d
213075          LD HL,30000d
7E             LD A,(HL)
12             LD (DE),A
23             INC HL
13             INC DE
7E             LD A,(HL)
12             LD (DE),A
23             INC HL
13             INC DE
....           ....
and so on.
```

You could shorten things a bit if you knew about the instruction LDI, which means to LoaD with Increment. This is a very special instruction which does four things all in one go. First of all, it will transfer the contents of the address stored in HL into the address stored in DE, then it will increment both HL and DE and finally, it will decrement BC. It will not alter the value of register A. To summarise:

EDAd0	LDI	POKE DE,PEEK HL
		LET HL = HL + 1
		LET DE = DE + 1
		LET BC = BC - 1

The above program could therefore have been completely re-written as:

11204E	LD DE,20000d
213075	LD HL,30000d
EDA0	LDI
EDA0	LDI
EDA0	LDI
....	....
	and so on.

There is no list of variables after the opcode LDI because the instruction will *always* load from (HL) to (DE). You must not write LDI (DE),(HL) because it is totally pointless – you can't use LDI to load any other combination. Loading from (HL) to (BC), for example, simply cannot be done in a single instruction.

There is also an instruction called LDD (LoaD with Decrement) which has the same effect as LDI except that DE and HL are decremented instead of incremented. BC is still decremented as before. Neither of these instructions will change the value of CARRY (as with all LD type instructions). The code for LDD is EDA8.

## REPEATING THINGS

Even with LDI and LDD at our disposal, it would still be a very tedious affair to move something from, say, 30000 to 20000 if that something were around fifty bytes long. If it were a hundred we'd probably give up in despair. Fortunately for us, both LDI and LDD have a *repeat* facility. If we wrote LDIR instead of just LDI (with the extra R standing for Repeat) then the instruction LDI would be carried out over and over again, and would not stop until BC was zero. So, if the routine we wanted to move was in fact one hundred bytes long, then we could move it using the routine:

```
016400          LD BC,100d
11204E          LD DE,20000d
213075          LD HL,30000d
EDB0           LDIR
```

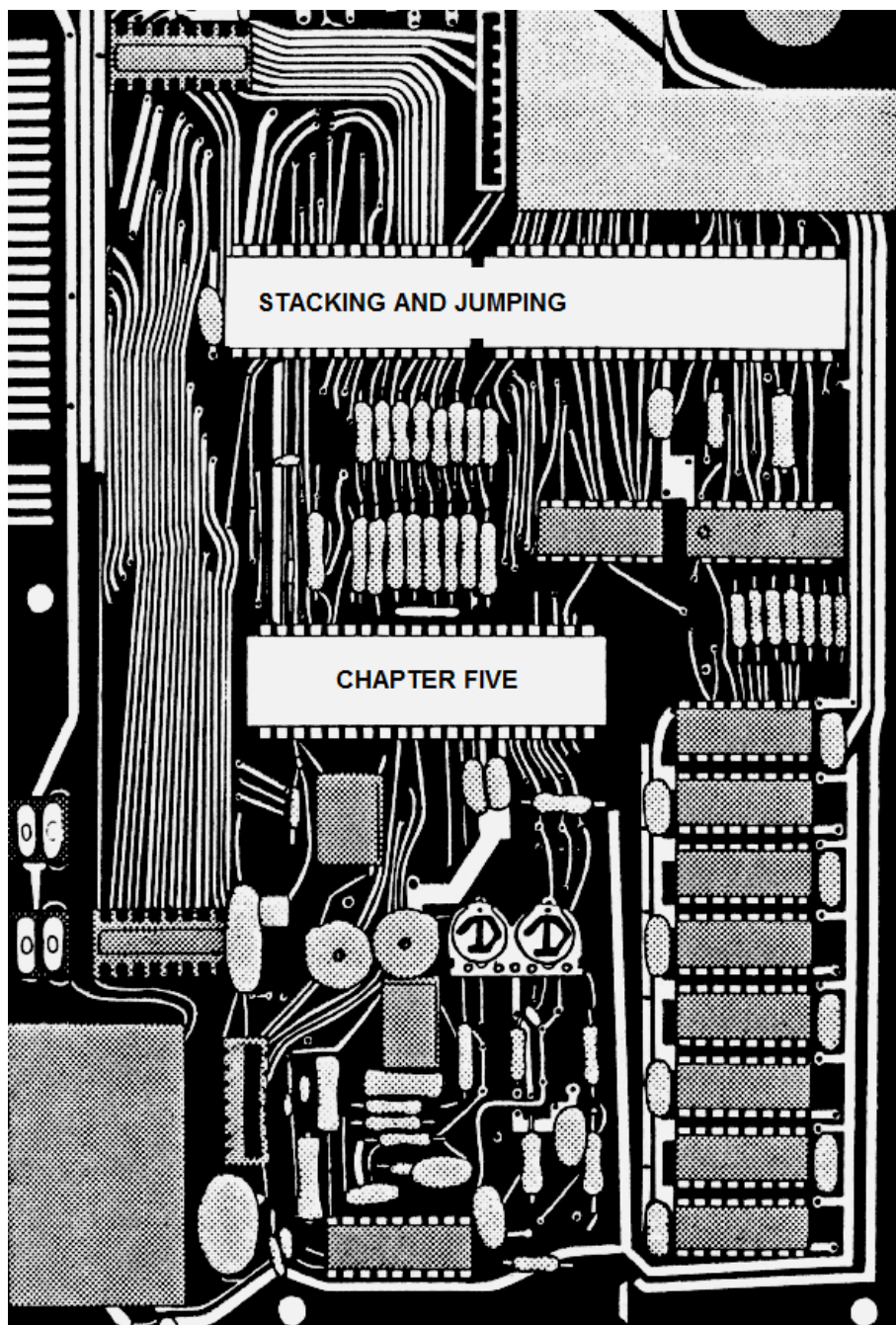
When the machine reaches the instruction LDIR, BC will contain a value of one hundred. After LDI had been carried out once the first byte would have been transferred, DE would be incremented to 20001 d, HL would be increased to 30001 d and BC would be decreased to ninety-nine. After a second attempt, the second byte would have been transferred and BC would contain ninety-eight. After LDI had been carried out one hundred times the whole routine would have successfully transferred and BC would contain a value zero, which means that the program would then continue with the next instruction. If this routine were the entire program then the next instruction should, of course, be RET.

The four instructions, LDI, LDD, LDIR and LDDR, each do slightly different things. Make sure you understand the difference between them. They also each have a different code, all beginning with ED. The codes are:

```
EDA0           LDI
```

EDA8  
EDB0  
EDB8

LDD  
LDIR  
LDDR



# CHAPTER FIVE

## STACKING AND JUMPING

### THE STACK

There is an area of RAM that is set aside for storing various pieces of information to help the machine know what it's doing. It works as follows:

The word "stack" is something that the computer people have got straight out of a dictionary. It means exactly what it sounds like! Imagine a stack of cardboard boxes. Each box is really a memory location, so each has an address – but if you want to know what's inside any particular cardboard box then the only one you can easily look at is the top one. If you tried to pull one of the boxes from somewhere in the middle then all the boxes above it would fall down. Conversely, to add a new box to the stack, the only place you can easily put it is at the top.

The memory locations in the stack are just like that. You can put things on top of it, but *only* at the top, and you can take things *from the top*. There are two special words which go with the stack – one which means "stacking a new number onto the top" and another which means "removing a number from the top"; the first word is PUSH, and the second word is POP. If you PUSH the number five onto the stack, and then you PUSH the number 9ABC, and then you PUSH, say, 8000h, then the first number you can POP is 8000h, since this number is now at the top (because it was PUSHed there last); the second number you can POP is 9ABC, and the third is five.

The stack is stored very high amongst the Spectrum's addresses, so that there is less chance of the BASIC program "colliding" with the stack as either one or the other is built up. The stack is actually very peculiar, because it's *upside down*! The bottom of the stack is near the top of available memory, and the top of the stack is below it. It turns out to be



more efficient this way. It's not really a deliberate plot to confuse the entire human race so that the world may be taken over by ZX computers, even if it does at times seem like it. So remember, the stack, or *machine stack* as it's sometimes called, is like a stack of cardboard boxes piled upon a shop floor, except that in a daring feat of defiance of Newton's laws this stack instead decides to reside on the ceiling and build up downwards! The top – the only part you can actually get at – is lower than the bottom!

The stack is so important to the computer that a special register is set aside just to store the position of the *top* of the stack (the part with the lowest address – the part we can get to). That register is called SP, which stands for Stack Pointer. It is actually a register pair because it stores two separate bytes, but unlike the other register pairs (BC, DE and HL) we *cannot* separate the two halves – they really are glued together quite solidly.

Here's how the instructions PUSH and POP work. I'll do this in hex because I think it's easier that way. Suppose HL contained a value of ABCD. This means that H contained AB and L contained CD. Now, the instruction PUSH HL would store the number ABCD at the top of the stack. It would do so by first of all stacking the *high* part (AB), and then stacking the *low* part (CD). It would alter the value of SP accordingly since two more bytes have been added to the stack, and the position of the top will therefore have moved (down) by two addresses.

It is unfortunately not possible to PUSH single registers onto the stack. You may only PUSH register *pairs*, so BC may be PUSHed, but B on its own may not. It is worth noting that PUSH BC will not in any way alter the value of BC, but will simply copy it without changing it. This, of course, goes for all PUSH instructions.

PUSH can be thought of in BASIC as three separate statements:

PUSH HL                    LET SP = SP - 2

```
POKE SP + 1,H
POKE SP,L
```

POP, of course, works the other way round. POP HL will first of all remove L from the stack, and will then remove H. SP will be changed, since the top of the stack will have moved.

```
POP HL          LET L = PEEK SP
                LET H = PEEK (SP + 1)
                LET SP = SP + 2
```

Verify, using the BASIC equivalents given, that PUSH HL followed by POP DE is the same thing as LD D,H followed by LD E,L.

## PUSH

Here are the codes for the instruction PUSH. One of them will require a small degree of explanation.

```
F5          PUSH AF
C5          PUSH BC
D5          PUSH DE
E5          PUSH HL
```

The register pair AF, which normally cannot be used as such, is made up of the smaller single registers A and F, in the same way that BC is composed of B and C. A is the register which we've been using throughout the book so far, but F is something completely different. The F stands for Flags. To understand the workings of F you have to look at it not in hex, but in *binary*. F could, for instance, have a value 41h, which in binary is 01000001. Each of the digits is either zero or one, and each of the different digits has a different meaning for they are all different flags. One of these flags we've seen already, the carry flag, is actually stored as the rightmost binary digit of F. We need not concern ourselves

with the workings of F at the moment as I'll go over it in a bit more detail later on. In the meantime, see if it makes any sense to you why this short routine will set the carry flag (without using SCF); it doesn't matter if it doesn't right now, but just to set your mind at work:

```
OE01          LD C,01
C5            PUSH BC
F1           POP AF
```

## **POP**

The codes for the POP instructions are very similar to the codes used for PUSH. They are:

```
F1           POP AF
C1           POP BC
D1           POP DE
E1           POP HL
```

One of the major uses of PUSH AF and POP AF is simply to PUSH and POP the value of A. The fact that F has been stacked alongside it can be conveniently forgotten about. PUSH AF will certainly stack the value of A until it's needed again, at which point it may be recovered by the use of POP AF. This could be useful if you have to use the A register to perform calculations of some kind that could not be performed in another register, but where the value of A will still be needed later on in the program.

For example, to add twenty-five to the value of B without altering the value of A (or any other register):

```
F5           PUSH AF
78           LD A,B
C619        ADD A,19h ( = 25d)
```

```

47          LD B,A
F1          POP AF

```

Why will only B and no other register be altered? (Not even the carry flag!) See if you can work out precisely what the above routine is doing, before you read on.

## ALTERING SP

We can actually use SP in much the same way that we use BC and DE. We can add and subtract it, and we can load it. The hex codes are:

```

F9          LD SP,HL
31????     LD SP,mn
ED7B????   LD SP,(pq)
ED73????   LD (pq),SP
39          ADD HL, SP
ED7A       ADC HL,SP
ED72       SBC HL,SP
33          INC SP
3B          DEC SP

```

This is very powerful, and very useful. Suppose you wanted to exchange the values of D and E without altering anything else. The following routine will do just that.

```

D5          PUSH DE
D5          PUSH DE
33          INC SP
D1          POP DE
33          INC SP

```

The final INC SP was necessary in order to restore the stack pointer to its original value. If this is not done you may get a pretty nasty crash.

SP is not the only specialised register in use. There is another two byte register called PC, standing for Program Counter. Its job is to remember whereabouts we are in the program. Every time the Spectrum has to execute an instruction it will take a look at what PC says. If it says F004 then it will execute the instruction at location F004. It will then increment the value of PC by the number of bytes in that instruction, so that *next* time round it will be looking at the next instruction in sequence. For example, if F004 contained the instruction LD A,B, then this would be carried out and PC would be incremented to F005. If the instruction at F005 was LD B,2 then once this was carried out, PC would be increased by *two*, since LD B,2 is a *two byte* instruction. PC would then be reading F007 where the next instruction begins.

If you alter the value of PC then the effect is like a BASIC GO TO. The only difference is that machine code does not use line numbers, so you have to GO TO the right *address* rather than the right line number. The machine language instruction to do this is called JP which, of course, is short for Jump. JP F000 means GO TO address F000 and continue executing the machine code from there. Of course, all this instruction really does is to load the number F000 into PC (but without incrementing it at the end of the instruction) so that it thinks F000 is the next address in the program. It is far more useful for us human beings to think of it as a kind of GO TO though, because that's what we're used to.

Be careful with JP though. If you create an infinite loop in machine code then *tough!* You're stuck with it – you can never break out unless you actually switch the machine off at the mains. An example of an infinite loop would be:

77	F000	LD (HL),A
23	F001	INC HL
C300F0	F002	JP F000

I've written the actual addresses in the middle column. Usually this isn't

done and important lines are marked with *labels*, or words which tell us which lines do what. These labels do not appear in the hex, and we only in fact write them out for our own convenience. If, for instance, we decided to call the first line START then our pretty bad program could be written as:

```
77          START      LD (HL),A
23          INC HL
C300F0     JP START
```

There is another instruction (similar to JP) called JR, or Jump Relative. It means jump forward a given number of bytes. In many ways it is better than JP because it is only two bytes long instead of three, and because a whole routine may be relocated without changing JP destinations all over the place. JR 0 has no effect whatsoever, and the next instruction will be executed in sequence; however, JR 1 will cause the next instruction (assuming it to be a single byte instruction) to be skipped. To skip over two byte instruction, or two single byte instructions, you will need to use JR 2.

It is also possible to jump *backwards* using JR, since there is a convention that any hex number between 80 and FF will be taken to mean a negative number (actually 256d less than the number it would normally represent). To make life easier I have included a table of hexadecimal numbers between 80 and FF and the decimal (negative) numbers they represent. This table may be found as the second table of Appendix One at the back of this book. Take a look at it now and compare it to the table above it. Note that the number minus five, for example, is represented by FB, and so therefore it is possible to use the instruction JR - 5 - but note that because of this convention we are unable to say JR 129d, for instance, because 129d in hex is 81, which would be taken to mean - 127d and would be a jump backwards. The range we are limited to is, therefore, - 128d to 127d.

JR 0, as has been said, does absolutely nothing. It will continue with the next instruction. It is important to remember that relative jumps are counted from the *next* instruction. JR 0 means execute the *next plus zero* instruction. JR 1 means execute the *next plus one* instruction. Consequently, if we were to say JRs - 2 then you must count backwards for two bytes starting at zero with the *next* instruction. You will find that two bytes leads you to precisely the instruction we have just executed - the instruction JR - 2. JR - 2 is therefore an infinite loop all by itself, and is thus not really a recommended instruction to use in a program.

The (rather silly) infinite loop program above can now be rewritten in one byte less using JR instead of JP.

```
77          START      LD (HL),A
23                          INC HL
18FC                          JR START
```

Notice how I wrote “JR START” instead of “JR - 4”. This makes the program much easier to follow since all we have to do is look for the label START in order to know where the JR is taking us.

JR and JP are more or less useless on their own without conditions attached, in the same way that BASIC GO TO would be useless if the instruction IF/THEN GO TO were not around. We need some kind of *conditional* jump, so that we can say IF some condition is true THEN jump to a new address. Without this facility JP and JR can only really be used to create infinite loops. Although machine code does not have quite the same degree of flexibility as BASIC, it does allow us to check for four conditions on JR, or eight conditions on JP. These are (for JR):

18ee	JR e	Jump Relative by e bytes
20ee	JR NZ,e	IF the last result calculated was non zero THEN Jump Relative by e bytes.

28ee	JR Z,e	IF the last result calculated was zero THEN Jump Relative by e bytes.
30ee	JR NC,e	IF CARRY = 0 THEN Jump Relative by e bytes.
38ee	JR C,e	IF CARRY = 1 THEN Jump Relative by e bytes.

And for JP:

C3qqpp	JP pq	Jump to address pq.
C2qqpp	JP NZ,pq	IF the last result calculated was non zero THEN jump to address pq.
CAqqpp	JP Z,pq	IF the last result calculated was zero THEN jump to address pq.
D2qqpp	JP NC,pq	IF CARRY = 0 THEN jump to address pq.
DAqqpp	JP C,pq	IF CARRY = 1 THEN jump to address pq.
E2qqpp	JP PO,pq	See below.
EAqqpp	JP PE,pq	See below.
F2qqpp	JP P,pq	IF the last result calculated was positive (Plus) THEN jump to address pq.
FAqqpp	JP M,pq	IF the last result calculated was negative (Minus) THEN jump to address pq.

Now although this is a far cry from IF A\$ = "HELLO" THEN PRINT "GOODBYE" as you're used to, you'll soon see that even this horrendous task may be carried out in machine code. First though, I'll explain about two of the instructions in the list above – JP PO and JP PE. Straightforwardly, JP PO means IF PV = 0 THEN jump to address



pq, and JP PE means IF PV = 1 THEN jump to address pq. But what exactly is PV?

Answer: PV is another flag – just like CARRY. It can only ever store one of two values – one or zero. The P stands for Parity, but I’ll explain that usage later on in the book. The V stands for oVerflow, because computer boffins, like all good mathematicians, can’t spell properly. This usage is quite easy to explain:

JP PO can also be thought of as JP NV (Jump if No-oVerflow).  
JP PE can also be thought of as JP V (Jump if oVerflow).

Technically you shouldn’t write JP NV or JP V because it’s not standard convention, but it is certainly an aid to memory. I tend to write JP NV or JP V in my programs – I don’t really think it matters if you’re conventional or not as long as you know what it means (NV = PO and V = PE).

Now, to *overflows*. If we regard numbers from 80 to FF as negative and numbers from 00 to 7F as positive (as described earlier in this chapter), then strange things can happen in arithmetic if we try to cross that boundary. For instance, 41h (positive) plus 41h (positive) equals 82h (negative?). This sort of blunder is called an overflow, so JP V will jump if such an overflow has occurred and JP NV will jump if such an overflow has not occurred. In simple terms, an overflow has occurred if the result of any arithmetic operation acting on numbers in positive/negative convention (also called “two’s complement<sup>2</sup> convention) gives an answer with “the wrong sign”.

To repeat once more: JP NV is *my* (non-standard) way of writing JP PO, and JP V is *my* way of writing JP PE. So NV = PO and V = PE. You should invent some way of remembering this.

All of these various tests, if combined with other instructions properly, can really check for any situation conceivable. In fact, there’s only one

other kind of instruction you need in order to make JP and JR as powerful as IF/THEN/GO TO – that instruction is called CP, or ComPare.

CP will compare the register A with any other register or with any numeric constant. For instance, we could have CP B (Compare A with B) or CP L (Compare A with L) or CP 3E (Compare A with the number 3E). What this instruction actually does can be thought of in BASIC as, for CP B, LET DUMMY = A – B.

In other words, a subtraction is performed, but the *result* of that subtraction is “forgotten”, and the value of A never changes. The flags, however, *will* change and tell us all about the result. If A contains 05 and B contains 06 then after a CP B instruction JP NZ will work (since 05 – 06 is non zero), JP C will work (since 05 is less than 06), JP NV will work (there is no overflow since the result of 05 – 06 = FF which is negative and which is *supposed* to be negative) and JP M will work (since FF is negative). Although the subtraction is actually carried out it should be emphasised that the result of the subtraction is discarded, and the value of A does not change.

You can do some useful tricks with CP:

IF A = B THEN GO TO ...	CP B / JR Z ...
IF A < B THEN GO TO ...	CP B / JR C ...
(This will only work if all numbers are assumed positive)	
IF A < B THEN GO TO ...	CP B / JP M ...

## CALLING

Even in machine code we can have *subroutines*. The machine code equivalent of GO SUB is called CALL. We write CALL pq to mean GO SUB the subroutine starting at address pq. The equivalent RETURN instruction is RET. I hope this looks familiar. RET has a dual purpose –

at the end of a subroutine it means “return from that subroutine”; if there is no subroutine to return from, it means “return to BASIC”. CALL and RET can both have conditions imposed on them as follows:

CDqqpp	CALL pq	C9	RET
C4qqpp	CALL NZ,pq	C0	RET NZ
CCqqpp	CALL Z,pq	C8	RET Z
D4qqpp	CALL NC,pq	D0	RET NC
DCqqpp	CALL C,pq	D8	RET C
E4qqpp	CALL PO,pq	E0	RET P0
	“CALL NV,pq”		“RET NV”
ECqqpp	CALL PE,pq	E8	RET PE
	“CALL V,pq”		“RET V”
F4qqpp	CALL P,pq	F0	RET P
FCqqpp	CALL M,pq	F8	RET M

As you’d probably guessed from the above, instructions like RET Z, etc., can also be used to conditionally return to BASIC, ie RET Z equals IF zero THEN RETURN to BASIC.

In BASIC there is no stack, so we never need to worry about it. In machine code, however, there is and so we *do* need to worry about it. There are two instructions other than PUSH and POP which alter the stack! These are CALL and RET!

CALL pq is equivalent to PUSH “the-address-of-the-next-instruction-to-be-executed”, followed by JP pq.

RET is equivalent to POP DUMMY followed by “jump-to-address-DUMMY”.

You should be able to see how this procedure causes CALL and RET to act as they do. Whilst this is certainly efficient, it does have some drawbacks which we must watch for.

The value of SP must not be altered during the course of a subroutine, since both CALL and RET rely on the stack. You can PUSH as many items as you like onto the stack during a subroutine, just so long as you POP an equal number before attempting to return. A clever trick you may like to know is how to actually *alter* the return address from a subroutine. Let's say you want to change the return address to E000. Watch this:

```
E1          POP HL
2100E0      LD HL,E000
E5          PUSH HL
```

The first instruction deletes the original return address from the stack. The second and third instructions replace it by a new alternative return address. When, at any later stage, a RET instruction is reached, control will “return” to address E000. Another useful trick to know is how to make absolutely sure that your subroutines will always exit with the stack “balanced”. One way of doing this is to store the value of the stack pointer somewhere and then retrieve it at the end. A good place to store this value is the SPARE system variable at address 5CB0. See how this works.

At the start of a subroutine:

```
ED73B05C          LD (5CB0),SP
```

Your subroutine may then conceivably have more PUSHes than POPs. The subroutine should end:

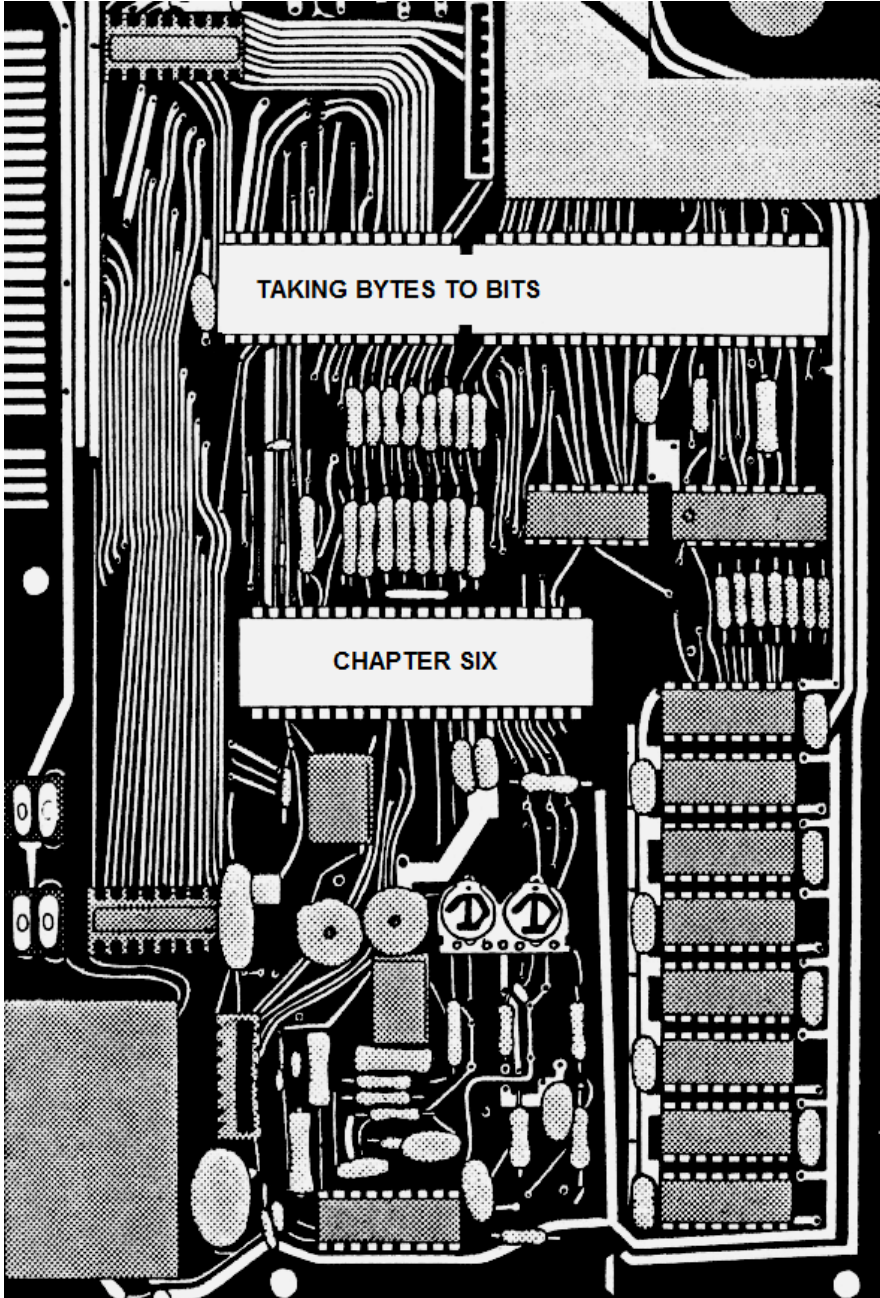
```
ED7BB05C          LD SP,(5CB0)
C9                RET
```

## **EXERCISES**

To make sure you have understood using the stack and conditional instructions, write a program which will PUSH every number between one and fifty (decimal) on to the stack (using PUSH AF) and then somehow manage a successful return to BASIC without crashing. (Hint: CP 32h (Compare with fifty decimal) is quite a useful instruction here.)

You'll need to know the various codes for CP. These are as follows:

BF	CP A
B8	CP B
B9	CP C
BA	CP D
BB	CP E
BC	CP H
BD	CP L
BE	CP (HL)
FEnn	CP n



TAKING BYTES TO BITS

CHAPTER SIX

# CHAPTER SIX

## TAKING BYTES TO BITS

### The Stack

One really good thing about the computing world is that it's got such amazingly good words in it – PEEK, POKE, PUSH, POP, etc. Another of these words is BIT. Now BIT is short for Binary DigIT. A Bit is then something that you only get in binary, so let's do a bit of binary now shall we?

I'm going to assume that you already understand hex, even if you don't have a "feel" for it yet. (It's actually quite easy to acquire a "feel" for hex numbers – two digit numbers are about  $1\frac{1}{2}$  times what they "look" like, three digit numbers are about  $2\frac{1}{2}$  times what they "look" like and four digit numbers are about four times what they "look" like.) So all the numbers I shall mention in this chapter will either be in hex or in binary. Binary and hex are very closely related. In fact, binary is the same as hex, only different!

Study this conversion table for a bit. The left column contains hex digits and the right column contains binary digits grouped in fours:

0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111

8	1000
9	1001
A	1010
B	1011
C	1100
D	1101
E	1110
F	1111

Now, every single hex digit is included in the list, but what is not quite so obvious is that every possible group of four bits (binary digits – a binary digit is either zero or one) is included in the list. It is impossible to think of a sequence of four zeros and ones which is not already listed above. To turn a hex number into binary all you have to do is convert it one digit at a time using the table above. For instance, the number 2A becomes, in binary, 00101010 since 2 corresponds to 0010 and A corresponds to 1010 – the 0010 and the 1010 are then simply stuck together side by side and in the right order (the 2 comes first, and then the A – hence 2A).

Similarly, four digit hex numbers can be turned into binary by this simple rule: E60A becomes 1110 0110 0000 1010. (I've written it with spaces inserted to make it easier to read but it could just have easily been written as 1110011000001010 or 11100110 00001010 – the spaces are for the eye, they have no real meaning.)

You can turn a number back from binary into hex by just as simple a rule. In order for this rule to work, it is essential that the binary number in question has a multiple of four digits. (Can you see why?) If this is *not* the case then your first step is to *give* it a multiple of four digits, and we do this by preceding it by zeros so, for instance, 101 becomes 0101, and 101010 becomes 00101010, and 1110001100001 becomes 0001 1100 0110 0001. (Notice how I've put spaces into the last example to make in clear.) Having grouped the number into blocks of four digits you then simply look up the blocks of digits in the table above, or in the table



in Appendix One at the back of the book, one block at a time until you get through the whole number. 0001 1100 0110 0001 therefore becomes 1C61. If you turn 1C61 back into binary you should get 0001 1100 0110 0001 again.

You don't *have* to keep looking at the table. You could learn the table given time – after all, it's quite short. But a better way is to learn what all these binary bits and pieces actually *mean*. In decimal, we're used to working in columns (at least we were, before the days of calculators) – units, tens, hundreds and thousands. In hex, we also have columns: units, sixteens, two hundred and fifty-sixes and four thousand and ninety-sixes, and so on, so when we write 12BC we technically mean the number  $4096*1 + 256*2 + 16*11 + 1*12$  (decimal).

Notice the way the column headings go – each one is sixteen times the last one. In binary we also have columns, but here the columns are units, twos, fours, eights, sixteens, thirty-twos, sixty-fours and one hundred and twenty-eights – each column is *double* the last one, so 00101010 really means (in decimal)  $128*0 + 64*0 + 32*1 + 16*0 + 8*1 + 4*0 + 2*1 + 1*0$ , or forty-two, or (in hex)  $80*0 + 40*0 + 20*1 + 10*0 + 08*1 + 04*0 + 02*1 + 01*0$ , or 2A. Notice how much easier the sum becomes to see if you do it in hex rather than decimal.

Now, bits – what's the point of them? You've probably guessed that there is some ulterior motive behind my telling you about them – surely they do have *some* sort of use. In machine code we can manipulate registers in bits using various bitwise instructions.

## BITWISE INSTRUCTIONS

The first such instruction we shall look at is cabled CPL, which as short for ComPLEMENT. It's written *just* as CPL – it has no variables written after it. What it does is this – it changes the value of A by the rule of changing every bit of A. If any given bit starts off its life as zero then CPL

changes it to one, and if any given bit starts off as one then CPL changes it to zero. Note that this will only work for the A register. To make this clear, suppose A originally contains the value 00111001, then CPL will change it to 11000110 – every zero has changed to a one and every one has changed to a zero. In hex, A has changed from 39 to C6. See if you can prove to yourself that the effect of CPL can be thought of in *hex* as each hex digit being subtracted from F ( $F - 3 = C$ ,  $F - 9 = 6$ , so CPL changes 39 to C6).

The next bitwise instruction we shall look at is called AND. AND is always written with one register name or numeric constant after it – eg AND B or AND 3F. Like CPL, its effect is to alter the value of the A register only, but it does so by comparing it bit by bit with some other value. The rule is – IF any given bit of A is one, AND the corresponding bit of the number you're comparing it to is one, THEN that bit of A remains one, otherwise that bit of A becomes zero. To make this clear, suppose A contains the value 11000101 and B contains the value 01010110 and the instruction AND B is encountered. You must convert A one bit at a time: 1 AND 0 is 0, 1 AND 1 is 1, 0 AND 0 is 0, 0 AND 1 is 0, 0 AND 0 is 0, 1 AND 1 is 1, 0 AND 1 is 0, and finally 1 AND 0 is 0. Thus, we have the final answer of 01000100, and this is the new value of A. Remember the rule – a one is generated only if a given bit of A is one AND the corresponding bit of B is one.

It is not necessary to have AND B of course – you can use AND C or AND 37h, there are many possibilities. As an exercise, see if you can work out what AND A will do, regardless of what A originally contained.

Next, we have quite a similar instruction – OR. OR, like AND, is written with one register name or numerical constant after the word OR, and its effect is to change the value of the A register by comparing it bit by bit with whatever is written after the word OR. The rule is slightly different, but just as easy to remember – IF any given bit of A is one, OR the corresponding bit of the number you're comparing it to is one, THEN

that bit of A is set to one, otherwise that bit of A remains zero. See if you can follow this rule in action by studying the example below.

11000101	11000101	11000101
CPL	AND	OR
	01010110	01010110
equals	equals	equals
00111010	01000100	11010111

So 0 OR 0 is 0, 0 OR 1 is 1, 1 OR 0 is 1, 1 OR 1 is 1. Remember the rules – a one is generated either if the given bit of A is one OR if the corresponding bit of the comparison is one.

The carry flag plays a part in all of this. The instruction CPL will change the value of A, but will leave the value of all flags *unchanged*. AND and OR on the other hand will change flags all over the place. For a start, something called the zero flag will be changed, depending on whether the final value of A is zero or not. It is this flag which is tested in instructions like JR Z or RET NZ. But one other stunning feature of AND and OR is that the carry flag will always be reset. That is, after an AND or an OR instruction CARRY will always equal zero. This means that the instruction AND A can be used instead of ADD A,0 to reset the carry flag without changing any of the registers.

Let's look now at one more bitwise instruction along the lines of AND and OR. This is an instruction called XOR, which stands for eXclusive OR. Like AND and OR, XOR is written with one register name or numerical constant after the word XOR, and its purpose in life is to alter A by comparing it bit by bit with the register or number given. Here the rule is as follows – IF any given bit of A is one OR the corresponding bit of the number you're comparing it to is one, but not both, THEN that bit of A is set to one, otherwise it is reset to zero. To use our example:

11000101	11000101	11000101	11000101
CPL	AND	OR	XOR

	01010110	01010110	01010110
equals	equals	equals	equals
00111010	01000100	11010111	10010011

Here we have 0 XOR 0 is 0, 0 XOR 1 is 1, 1 XOR 0 is 1, and 1 XOR 1 is 0. The rule is best remembered as – a one is generated if and only if the given bit of A is *different* to the corresponding bit of the comparison.

I have two exercises for you, but they're only VERIFYs and they're not hard. The first is to verify that the effect on A of XOR FF is the same as CPL, and the second is to verify that the effect on A of XOR A is the same as LD A,00.

XOR also sets or resets the zero flag as you'd expect, and will always reset CARRY to zero.

## ROTATING (OR GOING ROUND THE BEND)

The next group of bitwise instructions we shall look at are of a rather different nature – these are the shift and rotate instructions. They work on any register, not just on A, and what they do is this:

Consider the first instruction: RL (which stands for Rotate Left). You can write RL A, R L B, etc, but you *cannot* write RL 3F. You must have a register name after the word RL, not a numeric constant. RL moves every bit of the register in question one position to the left. What was once the leftmost bit moves into the carry, and what was once the carry becomes the rightmost bit; so if we assume that CARRY started off as zero and A contained 11000101 then after a RL A instruction, CARRY would contain one and A would contain 10001010. See if you can work out why RL A has exactly the same effect as ADC A,A. Note that RL B, RL C, etc, cannot be simulated in terms of ADC.

The next instruction is RR, or Rotate Right. RR moves every bit of the register in question one position to the right. What was once the rightmost bit moves into the carry, and the previous value of CARRY becomes the new leftmost bit. If CARRY = 0 and A = 11000101 then RR A would change things to CARRY = 1 and A = 01100010. Why is RR C repeated eight times the same as RL C once?

Next, we have some similar instructions RLC and RRC, which stand for Rotate Left without Carry, and Rotate Right without Carry respectively. As before, each bit of the register in question is moved one position to the left or right, but here the bit which disappears off one end re-emerges on the other end; so, for instance, if A contained 11000101 and RLC A was encountered then A would change to 10001011 – the former value of the carry would be irrelevant. CARRY itself would acquire a new value – the value of the bit which swapped ends! In this case, one.

I hope you have understood the difference between RL and RR, and RLC and RRC. RL and RR are *nine* bit cycles, whereas RLC and RRC are *eight* bit cycles. RL D repeated nine times will leave D unchanged, but RLC D repeated *eight* times leaves D unaltered. RL, RR, RLC and RRC will change all of the flags as you'd expect them to.

Speaking of flags, it's time now to re-explore that strange flag PV. PV stands for Parity/oVerflow – its use in detecting overflow we've already seen, but whenever a bitwise instruction is used PV is set or reset according to a different set of rules. Whenever the result of a bitwise instruction has an even number of ones then we say the parity is *even*, and whenever the result has an odd number of ones then we say parity is *odd*. The instructions such as JP PO (Jump if Parity Odd) or RET PE (Return if Parity Even), for instance, will reflect this count.

If we wish to use RL, RR, RLC or RRC on the A register (not any of the others) then we have a shorthand way of doing it. Four instructions – very easy to remember. RLA achieves RL A, RRA achieves RR A,

RLCA achieves RLC A, and RRCA achieves RRC A. The fact that the space is missing is very important. RLA (etc) will perform the expected operation *but without changing any of the flags except carry*. Remember, you can only do this to the A register – you are not allowed to write RLE, you must write RL E. The space *means* something.

Next we have the *shift* instructions, which are very much like the rotate instructions, except that they are largely concerned with multiplying and dividing by two. The first of these is SLA which *multiplies* by two. SLA stands for Shift Left Arithmetic and must be written with the name of a register after the word SLA. In binary, what actually happens is that each bit of the register is moved one position to the left, the former leftmost bit moves into the carry, and the new rightmost bit is always zero. Can you see why SLA A is the same thing as ADD A,A? (Note that there is no shorthand notation for this – SLAA, for instance, is not allowed.) The real effect of SLA on any register is to multiply the value of that register by two, and so this is probably the best way to remember it. An example:

```
11000101
SLA
equals
10001010
```

Next we have SRA, or Shift Right Arithmetic, whose job is to *divide* by two. For this instruction, numbers between 80 and FF are regarded as negative, and numbers between 00 and 7F are regarded as positive. (See Appendix One, table two.) In other words, suppose D contains a value FE (minus two), then after a SRA instruction D will contain FF (minus one). In binary, what this instruction does is to move each bit one position to the right, with the former rightmost bit moving into the carry, *and the leftmost bit remaining unchanged*. Thus, the leftmost and second leftmost bits are always identical after a SRA instruction. Let's do this by example again:

11000101	11000101
SLA	SRA
equals	equals
10001010	11100010

Finally, we have one last shift instruction, called SRL (Shift Right Logical). Again, its purpose is to divide by two, but note that with this instruction all numbers, even those in the range 80 to FF, are regarded as positive (see Appendix One, table one). In other words, if D contains FE then it is regarded not as minus two, but as two hundred and fifty-four. After a SRL instruction, D would then contain 7F which represents one hundred and twenty-seven, *not* FF. In binary, what SRL does is to move each bit one position to the right, with the former rightmost bit moving into the carry, *and the leftmost bit being reset to zero*. By example:

11000101	11000101	11000101
SLA	SRA	SRL
equals	equals	equals
10001010	11100010	01100010

## SETTING AND RESETTING

When written in binary, one byte is eight bits long. Each of these bits is numbered. The leftmost bit is called bit seven, and the rightmost bit is called bit zero. The in between bits are numbered in descending numerical order as you'd expect them to be.

In machine code, we have the power to play around with registers individual bits at a time. We have the instructions SET and RES (short for reset). For example, SET 7,H will set bit seven of register H. SET, remember, means change to one. RES 5,D will reset bit five of register D. RESET, of course, means change to zero. Note that we can only set or reset *specific* bits of registers. We cannot, for instance, say RES B,C

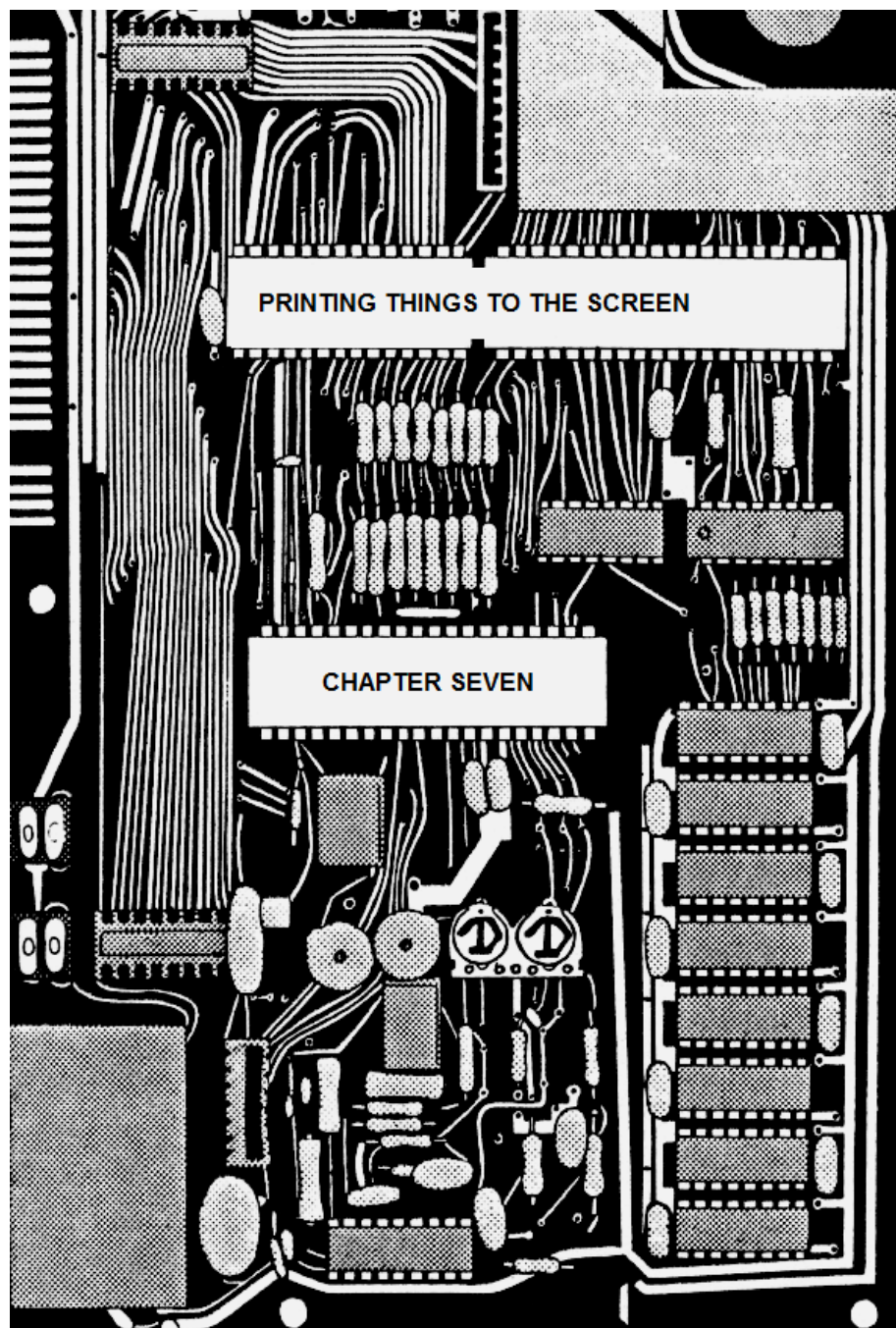
– only numerical constants (between 0 and 7) are allowed before the comma, and only register names are allowed after the comma. Incidentally, you can write “(HL)” as if it were a register in most cases, and AND (HL), OR (HL), XOR (HL), RL (HL), RR (HL), RLC (HL), RRC (HL), SLA (HL), SRA (HL), SRL (HL), SET 3, (HL) and RES 3, (HL) will all work.

Individual bits can also be *tested*. There is an instruction called BIT which works as follows.

Bit 4,B (for instance) will test bit four of register B. The actual value of B is not changed, but the zero flag will reflect the result of the test. Bit four of B can either be zero or non zero, and so instructions like JR Z or CALL NZ will all work as you’d expect them to after a BIT instruction. Of course, any register, not just B, may be tested and any bit, not just bit four, may be tested.

You may have noticed that in this chapter I have not been giving you the hex codes of any of the instructions I’ve described. This is because I’ve left it up to you to look them up in Appendix Four at the back of this book. Every instruction in machine code is listed in Appendix Four together with its hex code. My description of binary is now over. After all that slogging away, I think it’s about time we now did something useful – don’t you?





PRINTING THINGS TO THE SCREEN

CHAPTER SEVEN

# CHAPTER SEVEN

## PRINTING THINGS TO THE SCREEN

### PRINTING A DRAUGHTS BOARD

In order to write a program as extensive as draughts, we'll need a fairly powerful BASIC program in order to load it. The following is a second version of HEXLD – called HEXLD 2 – which has a couple of improvements over its predecessor. One such improvement is the ability to input strings of characters such as “To be or not to be” which will then be incorporated in the machine code one character at a time. To achieve this, when running the program, you must input “:To be or not to be:” – that is, the text must be surrounded by colons (this is very important).

HEXLD 2 lists as follows:

```
10 INPUT "WRITE TO ";A$
20 PRINT "WRITE TO ";A$
30 GO SUB 200
40 PRINT
50 LET A$ = ""
60 IF A$ = "" THEN INPUT A$: PRINT A$
70 IF CODE A$ = 58 THEN GO TO 300
80 LET Y = CODE A$ - 48: IF Y > 9 THEN LET Y = Y - 7
90 LET Z = CODE A$(2) - 48: IF Z > 9 THEN LET Z = Z - 7
100 POKE X, 16*Y + Z
110 LET X = X + 1
120 LET A$ = A$(3 TO)
130 GO TO 60
200 LET X = 0
```

```
210 FOR I = 1 TO 4
220 LET Y = CODE A$ - 48:IF Y> 9 THEN LET Y = Y - 7
230 LET X = 16*X + Y
240 LET A$ = A$(2 TO)
250 NEXT I
260 RETUR
300 LET A$ = A$(2 TO)
310 POKE X, CODE A$
320 IF CODE A$ = 226 THEN POKE X,13
330 LET A$ = A$(2 TO)
340 LET X = X + 1
350 IF CODE A$< > 58 THEN GO TO 310
360 LET A$ = A$(2 TO)
370 GO TO 60
```

This program is basically the same as HEXLD except for two features. Firstly, you are required to input the starting address (in hexadecimal) at which the machine code is to be loaded and secondly, it will allow you to input strings of data using their character codes, rather than in hex – this is what the routine starting at line 300 is for. To input text you merely have to surround the text by colons. To input an ENTER character as part of text you should use the word STOP wherever you want an ENTER symbol to be. To break out of the program you must delete the quote marks by typing EDIT and then input STOP (*not* surrounded by quotes).

## **SUBROUTINES WITH DATA**

Let's look at some uses for this. Perhaps one of the most useful subroutines we could think of would be one which prints a string of characters to the screen. There is a machine code instruction called RST 10 (which means CALL 0010) which will print a single character. Try this program. Load it to address 7000h.

```

3E2A      START          LD A, "*"
D7                RST 10
18FB                JR START

```

In order to see the program running add these lines of BASIC to HEXLD 2:

```

700 INPUT "RUN ADDRESS ";A$
710 Go SUB 200
720 RANDOMIZE USR X

```

and then type RUN 700 and input "7000". Notice the way in which USR was used here – not a number but a variable X. The address which USR calls is input in hex in line 700. On running the machine code, did you notice something odd? Try instead this slightly different machine code program:

```

AF                XOR A                (This is a short way of
                                        doing LDA,0)
323C5C            LD (TVFLAG),A
3E2A      LOOP    LD A, "*"
D7                RST 10
18FB                JR LOOP

```

Notice the difference is that the system variable TVFLAG (address 5C3C) is first of all loaded with zero before the rest of the program takes place. You should discover upon running it that the screen fills up with asterisks, and that it fills up *very, very fast*. (Much faster than PRINT "\*" / GO TO 1.) The instruction RST 10 will place the character whose code is stored in the A register at the current PRINT position on the screen.

What we want is a subroutine which can print any message, from "YES" to "Oh, what a beautiful morning" in flashing red and yellow. Suppose such a subroutine exists and it's called S\_PRINT (String PRINT). We

want to be able to use an instruction something along the lines of CALL S\_PRINT WITH DATA “*PAPER yellow INK red FLASH* on Oh, what a beautiful morning”. (The words in italics represent the control codes obtainable by pressing both shifts/6/both shifts/caps shift 2/both shifts/caps shift 9.) Here’s how it will work:

```
CD????          CALL S_PRINT
48656C6C6F      DEFM Hello
00              DEFB 00
```

Here, DEFM means DEFine Message. It’s not actually a machine language instruction but is used to specify data within a program. If you look at a table of character codes (Appendix Five at the back of this book, or page 183 of the Spectrum manual) you’ll see that 48 is hexadecimal for “H”, 65 for “e”, 6C for “I”, and 6F for “o”. DEFB is also data – it means DEFine Byte. DEFB C9 is, therefore, another way of writing RET. In the example above, we are using the byte 00 to specify the end of the string – the end of the data to be used by S\_PRINT.

We must ensure, however, that the data above is *never* executed, since the bytes given don’t make a lot of sense in machine code. Let’s take a look at how we could go about writing a subroutine such as S\_PRINT which at the same time ensures that the data is never executed (the data is the word “Hello” and the byte 00).

You may remember from earlier in the book that CALL works by PUSHing the return address onto the stack and then jumping to the required address. RET works similarly – it POPs an address from the stack and then jumps to it. Therefore, if the word “Hello” immediately follows a CALL instruction then the address at the top of the stack (throughout the course of the subroutine) will be the address of the first character of data – the “H” – we can obtain this with the single instruction POP HL. If we then increment HL by one and PUSH it back onto the stack, the effect of the next RETurn will be to jump back to the *next* address in line – the “e”. We can test for the end of data by looking

for the byte 00 (which cannot occur within the text). Follow this subroutine through:

```

E1          S_PRINT          POP HL
7E          LD A,(HL)
23          INC HL
E5          PUSH HL
A7          AND A
C8          RET Z
D7          RST 10
18F7       JR S_PRINT

```

The first four lines are designed to look at the character stored at the current return address and then *increment* the return address. The next two lines will only return from the subroutine if the byte 00 has been found. Note that AND A will compare A with 00. The RET instruction (actually a RET Z, or RETurn if Zero, but it works in precisely the same way) will, if you examine the listing closely enough, return you to the byte *after* the 00, not the 00 itself. Finally, if the 00 has not yet been found, the instruction RST 10 will be used and the single character now in the A register will be printed to the screen. The whole routine will then be repeated over and over again until the end of the message is found.

Enter the program HEXLD 2 to enable you to load machine code. RUN the program and for the starting address input "7000", type in everything you read in the left-hand column:

```

E1          S_PRINT          POP HL
7E          LD A,(HL)
23          INC HL
E5          PUSH HL
A7          AND A
C8          RET Z
D7          RST 10
18F7       JR S_PRINT

```

*Mastering Machine Code on your ZX Spectrum*

```
AF          START          XOR A
323C5C      LD (TVFLAG),A
CD0070      CALL S_PRINT
: paper yellow ink red flash on Oh, what a beautiful morning:
DEFM message
00          DEFB 00
C9          RET
```

Now do you see the purpose of the BASIC routine in HEXLD 2 which begins at line 300. Imagine how tedious it would have been to have had to type in 1106100212014F682C20... and so on instead of *:paper yellow ink red flash on Oh, what a beautiful morning:*. It has exactly the same effect. Now type RUN 700 to run the machine code and input "7009" for the starting address. This is the address of the label START in the above program. What happens?

If you take a look at line 320 of HEXLD 2 you'll see that every time you input the word "STOP" in text it will automatically be replaced by an *enter* character. This is just for convenience. We *can* input an *enter* character if we want, just by deleting the quote marks and instead typing CHR\$ 13, but it is far simpler, and far more convenient, to only have to type symbol shift A. If, of course, you ever *need* the word "STOP" then you can always type in "S", "T", "O" and "P".

Now, we have to invent a user-defined graphic. Input as a *direct command*:

```
FOR I = 0 TO 7: INPUT X:POKE USR "A" + I,X:NEXT I
```

and then input:

```
0
60
126
126
```

126  
126  
60  
0

You have now defined graphics A as a draughts piece.

The next machine code program forms the very first part of DRAUGHTS. It is the routine which enables us to print (in a sense) the playing board. In the right-hand column, the symbol @ is used to represent graphics A. You should write this to address 7000h.

E1	S_PRINT	POP HL									
7E		LD A , (HL)									
23		INC HL									
E5		PUSH HL									
A7		AND A									
C8		RET Z									
D7		RST 10									
18F7		JR S_PRINT									
AF	START	XOR A									
323C5C		LD (TVFLAG),A									
CD0070		CALL S-PRINT									
2031323334353637380D		DEFM	1	2	3	4	5	6	7	8	
312090209020902090310D		DEFM	1	@	@	@	@	@	@	1	
329020902090209020320D		DEFM	2	@	@	@	@	@	@	2	
332090209020902090330D		DEFM	3	@	@	@	@	@	@	3	
349020902090209020340D		DEFM	4	@	@	@	@	@	@	4	
352090209020902090350D		DEFM	5	@	@	@	@	@	@	5	
369020902090209020360D		DEFM	6	@	@	@	@	@	@	6	
372090209020902090370D		DEFM	7	@	@	@	@	@	@	7	
389020902090209020380D		DEFM	8	@	@	@	@	@	@	8	
203132333435363738		DEFM	1	2	3	4	5	6	7	8	
00		DEFB	00 (End of data)								
C9		RET	(Return to BASIC)								



If you now RUN 700 and input “7009” as the starting address (the address of the XOR A instruction) you should see a complete draughts board picture appear on your screen almost instantly. Try it. It doesn’t look much like a draughts board at the moment but we’ll rectify that later on – at least everything’s in the right place.

There is one thing left to rectify – that is, we cannot as yet SAVE machine code that is stored high in memory. We shall now learn how to do so. Add the following lines:

```
400 INPUT "SAVE 7000 TO ";A$
410 GO SUB 200
420 SAVE "HEXLD 2" LINE 500
430 SAVE "GRAPHIC A" CODE USR "A",8
440 SAVE "M/CODE" CODE 28672,X - 28671
450 VERIFY ""
460 VERIFY "" CODE
470 VERIFY "" CODE
480 STOP
500 LOAD "" CODE USR "A",8
510 LOAD "" CODE
520 STOP
```

Now, to SAVE your machine code type RUN 400. You will be told “Start tape then press any key” three times in a row. The program and all machine code will then VERIFY automatically (so you must rewind the tape and replay it). As you can see, the draughts piece character is also SAVED by line 430. Now, you must *never* save this program using SAVE “HEXLD 2” as you’re used to – get into the habit of typing RUN 400 whenever you want to SAVE it. The automatic VERIFYing comes in very useful because it means you’ll never forget it.

## TAKING THE SCREEN APART

To change the subject a little (just a little), let's explore the TV screen for a bit, SAVE the program, by typing RUN 400.

Now, as a direct command, just to see what happens, type the following:

```
FOR I = 1 6384 TO 23295:POKE I,41:NEXT I:PAUSE 0
```

Don't you find the result curious? It's not just the fact that we're PoKEing into the display file that's interesting, but the rather strange *order* in which things happen. This is the Spectrum memory map – it does at first seem more tied up in knots than necessary. So, I think I ought to take the screen apart for you in order to make PoKEing into it a little more sensible.

Take a look at figure 7.1. This gives us the address of the “print position” for any point of the screen, and also the “attribute byte”. For example, consider the point whose PRINT AT co-ordinates are 5,4. According to the diagram, the first two digits of the address of this print position are 40. The third digit is A (since row five is the sixth row down), and the fourth digit is 4. This gives us the address 40A4.

Try running this machine code program:

```
21A440          LD HL,40A4
36FF           LD (HL),FF
C9             RET
```

Now, confirm for yourself that the short line which has just appeared on the screen is, in fact, at co-ordinates 5,4.

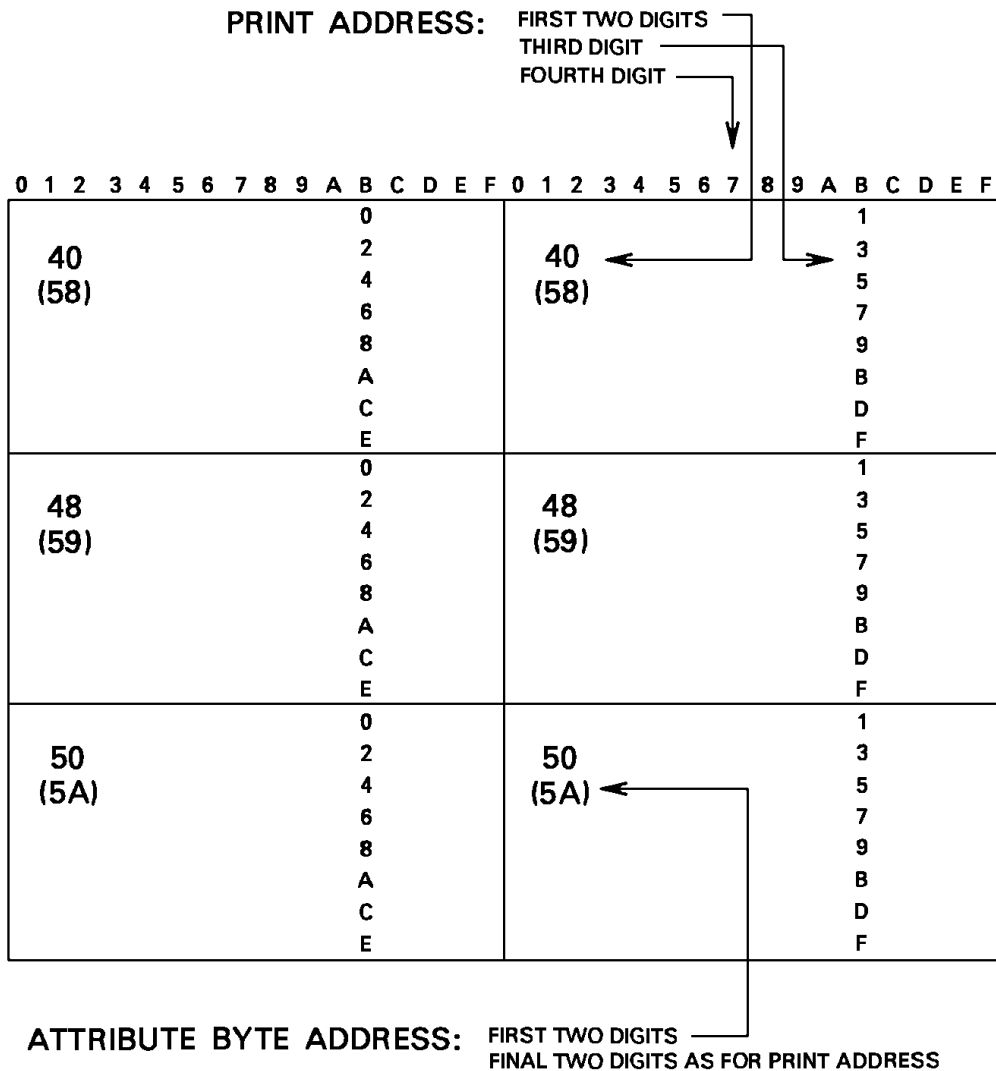


Figure 7.1

You'll also notice from the diagram that the address of the attribute byte for that position is 58A4. Try now this machine code program:

```

21A458          LD HL,58A4
36C7            LD (HL),C7
C9              RET

```

See what happens? But why did I use FF the first time, and C7 the second time? Let's take apart the printing mechanism even further now.

One very interesting part of the ROM is the last bit – the three quarters of a K that runs from 3D00 to 3FFF. It's not a subroutine, it's a table – a very long table – actually, the longest table in the ROM. It stores the dot patterns of every symbol, except graphic symbols, used by the computer – those with character codes 20 to 7F. It takes eight bytes to store a single character symbol so, for example, the characters a, b and c are represented by 00 00 38 04 3C 44 3C 00, 00 20 20 3C 22 22 3C 00, and 00 00 1C 20 20 20 1C 00. See figure 7.2.

Can you pick out the letters a, b and c from the digits in figure 7.2? The pattern is held by the positions of the “ones” amongst the “zeros”. The diagram also shows what the letters will finally look like when they appear on your TV screen. If the character code of any symbol (codes 20 to 7F only) is X then the address at which this “pixel expansion” is held in ROM is  $8 \cdot X + 3C00$ , so the address of the “a” pattern given here is 3F08 (since  $8 \cdot 61 + 3C00 = 3F08$ ).

I'd like you now to try a little experiment. Load and run this machine code program. I won't tell you what it does – but see if you can guess.

```

21A440          LD HL,40A4          (This is AT position 5,4
                                     remember)
11083F          LD DE,3F08        (The address of the “a”
                                     pattern)
0608           LD B,08
1A             LOOP LD A,(DE)
77             LD (HL),A

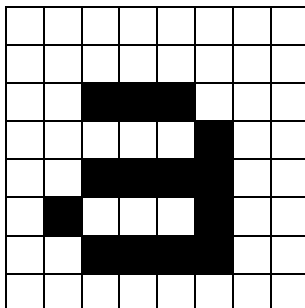
```

Mastering Machine Code on your ZX Spectrum

24	INC H	(Note: INC H, <i>not</i> INC HL)
13	INC DE	
10FA	DJNZ LOOP	
C9	RET	

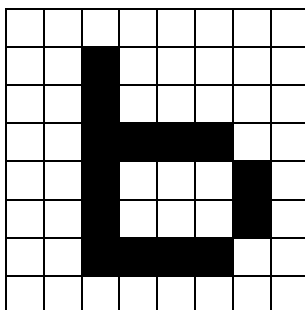
```

00  0 0 0 0 0 0 0 0
00  0 0 0 0 0 0 0 0
38  0 0 1 1 1 0 0 0
04  0 0 0 0 0 1 1 0
3C  0 0 1 1 1 1 1 0
44  0 1 0 0 0 1 0 0
3C  0 0 1 1 1 1 0 0
00  0 0 0 0 0 0 0 0
    
```



```

00  0 0 0 0 0 0 0 0
20  0 0 1 0 0 0 0 0
20  0 0 1 0 0 0 0 0
3C  0 0 1 1 1 1 0 0
22  0 0 1 0 0 0 1 0
22  0 0 1 0 0 0 1 0
3C  0 0 1 1 1 1 0 0
00  0 0 0 0 0 0 0 0
    
```



```

00  0 0 0 0 0 0 0 0
00  0 0 0 0 0 0 0 0
1C  0 0 0 1 1 1 0 0
20  0 0 1 0 0 0 0 0
20  0 0 1 0 0 0 0 0
20  0 0 1 0 0 0 0 0
1C  0 0 0 1 1 1 0 0
00  0 0 0 0 0 0 0 0
    
```

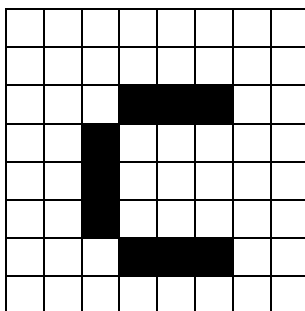


Figure 7.2

If you ran the program correctly the letter “a” should be printed at screen position 5,4, but without changing the colours at that position. Notice the INC H instruction. This is the reason that the ordering of the screen is the way it is. If the print position for a given square has address HL then the eight bytes representing the “pixel expansion” of a character must be stored at addresses HL, HL + 0100, HL + 0200, HL + 0300, HL + 0400, HL + 0500, HL + 0600 and HL + 0700. The instruction INC H is effectively the same thing as ADD HL,0100 since an overflow can never occur.

Now, imagine that instead of the screen memory map being the way it is, the consecutive addresses simply ran from the top left-hand corner of the screen to the bottom right-hand corner, with every pixel address in its supposedly logical order. If this were the case then the eight bytes for a character would have to be stored at addresses HL, HL + 0020, HL + 0040, HL + 0060, HL + 0080, HL + 00A0, HL + 00C0 and HL + 00E0. Since there is no such instruction as ADD HL,0020 the only way to point HL to the next pixel position would be PUSH DE/LD DE,0020/ADD HL,DE/POP DE. This is far, far more tedious than simply INC H. The only reason that we may use INC H for this purpose is because of the clever way the screen is laid out. My congratulations to Uncle Clive for this simplicity.

However, with all this cleverness, we still have the problem that because the screen is divided up into segments (see figure 7.1) then we will encounter problems if we try to cross from one segment to another. For instance, if the address of one particular square is HL then the address of the *next* square on the screen is *usually* HL + 0001. If HL happens to contain 40FF though, then the next print position is 4800 – *not* 4100 (which is the second row of pixels of the first square on the screen). The following routine will *always* move HL to the *next* print position:

```
CB1C          RR H
CB1C          RR H
```

*Mastering Machine Code on your ZX Spectrum*

CB1C	RR H
23	INC HL
CB14	RL H
CB14	RL H
CB14	RL H

Furthermore, the sequence of instructions RR H/RR H/RR H/LD BC,0020/ADD HL,BC/RL H/RL H/RL H will *without fail* move H L to the address of the next square down.

Let's take a look at the attribute byte now, and see what we can do with that. Each square on the board has only one attribute byte (remember it has eight bytes in the main display file). This byte stores all the information required about the colours of that square. We can think of this byte in two ways. Let's suppose that the paper colour for our square is P, and that the ink colour is I. Let's also define two more variables: FL, which will be one if the square is to be flashing or zero if the square is to be non-flashing; and BR, which will be one if the square is to be bright, or zero if the square is to be normal. The attribute byte required is then  $80*FL + 40*BR + 8*P + I$ .

However, this way of thinking of things doesn't really give us very much insight into the way the attribute byte is built up. A better way is to examine the attribute byte in binary. It is this:

FL BR P2 P1 P0 I2 I1 I0

P2, P1 and P0 represent the paper colour (in binary) and I2, I1 and I0 represent the ink colour (in binary). Notice that since the paper and ink colours can only ever be 0, 1, 2, 3, 4, 5, 6 or 7 then only three bits are needed. If you're not sure about this point, write the numbers zero to seven in binary and compare the five leftmost bits.

Using this knowledge of the attributes I'll give you a short routine which *contrasts* the ink colour to the existing paper colour. I shall assume that

the attribute byte starts off in the A register.

```

CB6F          BIT 5,A
2803          JR Z,WHITE
E6F8          AND F8          Set ink black.
C9            RET
F607          WHITE          OR 07          Set ink white.
C9            RET

```

Bit five of A is really bit two of the paper colour – it will be zero only if the paper colour is black, blue, red or magenta (ie dark). It will be one if the paper colour is green, cyan, yellow or white (ie light). I hope you can see how the routine above decides whether the ink should be black or white, and how it changes the ink colour in each case.

And now for some clever stuff. This is a subroutine. Its purpose is to point HL to a new square in the display file, and DE to a new square in the attributes file. It requires that HL is already pointing to the first row of pixels of some square, and that DE is already pointing to the attribute byte for that square. (Again, see figure 7.1.) It further requires that BC holds the *displacement* required (counting FFFF as one square left, 0001 as one square right, FFE0 as one square up and 0020 as one square down – ie the displacement simply counting squares). The subroutine has quite an interesting feature at the end:

```

7D            MOVE          LD A,L
CB1C          RR H
CB1C          RR H
CB1C          RR H
09            ADD HL,BC
CB14          RL H
CB14          RL H
CB14          RL H
EB            EX DE,HL
09            ADD HL,BC

```



*Mastering Machine Code on your ZX Spectrum*

EB	EX DE,HL
AD	XOR L
CB67	BIT 4,A
C8	RET Z
37	SCF
CB5F	BIT 3,A
C8	RET Z
7D	LD A,L
17	RLA
AD	XOR L
CB67	BIT 4,A
C0	RET NZ
37	SCF
C9	RET

The subroutine assumes that you will never want to move more than seven squares in any one direction in one go. In addition to moving HL and DE, it will return *no carry* if the displacement was acceptable or *carry* if the displacement has accidentally crossed the left or right edge of the screen. Notice the use of the instruction EX DE,HL (EXchange DE with HL) and the way in which it is used to add BC to DE. This is quite a useful trick. The logic behind the subroutine is as follows:

If bit four is unchanged then the displacement is allowed. Otherwise...  
If bit three is unchanged then the displacement is *not* allowed. Otherwise...  
If bit three is not equal to bit four then the displacement *is* allowed. Otherwise...  
The displacement is not allowed.

I shall leave it as an exercise for you to see (1) how this logic works, and (2) how this logic is implemented in the program.

Finally, now, two machine code routines you may be interested in. The first is to scroll the screen, and the second to scroll the screen

*backwards*. Neither of these routines will change the PRINT position (as used by RST 10), but they will clear the unwanted line from either the top or the bottom. Figure 7.1 should prove an immense help to understanding how the programs work.

Part one: to scroll the screen up. USR should refer to the label “UP”.

```

C5          UPLD      PUSH BC
D5          PUSH DE
E5          PUSH HL
EDB0       LDIR
E1          POP HL
24          INC H
D1          POP DE
14          INC D
C1          POP BC
C9          RET
212040     UP         LD HL,4020
110040     LD DE,4000
01E01A     LD BC,1AE0
EDB0       LDIR
21E047     LD HL,47E0
11E040     LD DE,40E0
0E20       LD C,20
3E10       LD A,10
CD????    U_LOOP    CALL UPLD
3D         DEC A
20FA      JR NZ,U_LOOP
62        LD H,D
1EE1      LD E,E1
0B        DEC BC
3E08      LD A,08
3600      U_BLANK   LD (HL),00
CD????    CALL UPLD
3D        DEC A

```

*Mastering Machine Code on your ZX Spectrum*

20F8	JR NZ,U_BLANK
265A	LD H,5A
54	LD D,H
3A485C	LD A,(BORDCR)
77	LD (HL),A
EDB0	LDIR
C9	RET

Part two: to scroll the screen down. USR should refer to the label "DOWN".

C5	DOWNLD	PUSH BC
D5		PUSH DE
E5		PUSH HL
ED88		LDDR
E1		POP HL
25		DEC H
D1		POP DE
15		DEC D
C1		POP BC
C9		RET
21DF5A	DOWN	LD HL,5ADF
11FF5A		LD DE,5AFF
01E01A		LD BC,1 AE0
EDB8		LDDR
211F50		LD HL,501F
111F57		LD DE,571F
0E20		LD C,20
3E10		LD A,10
CD????	D_LOOP	CALL DOWNLD
3D		DEC A
20FA		JR NZ,D_LOOP
62		LD H,D
1E1E		LD E,1E
0B		DEC BC

```

3E08                LD A,08
3600      D_BLANK  LD (HL),00
CD????            CALL DOWNLD
3D                DEC A
20F8              JR NZ,D_BLANK
2658              LD H,58
54                LD D,H
3A8D5C            LD A,(ATTR_P)
77                LD (HL),A
EDB8              LDDR
C9                RET

```

Notice the portion of machine code at the end of the routines which blank the unwanted lines. Notice also the difference between the system variables BORDCR and ATTR\_P. ATTR\_P stores the attribute for the top line, while BORDCR stores the attribute for the bottom line.

The following BASIC program is designed to show both of these scroll routines at work. It isn't a terrifically exciting game, or a pattern making artistic genius, but it will show you exactly what the machine code we've been working on will do. You can, of course, insert these routines into *any* program – there are one or two graphics games which would be immensely enhanced by the ability to scroll in either direction. This program sets up a striped pattern across the screen, with each stripe composed of a random character from the Spectrum character set. The pattern on the screen will then freeze until you tell it what to do. Pressing the “up” cursor key will move the pattern upwards, while pressing the “down” cursor key will move the pattern downwards. The shift key is not needed.

```

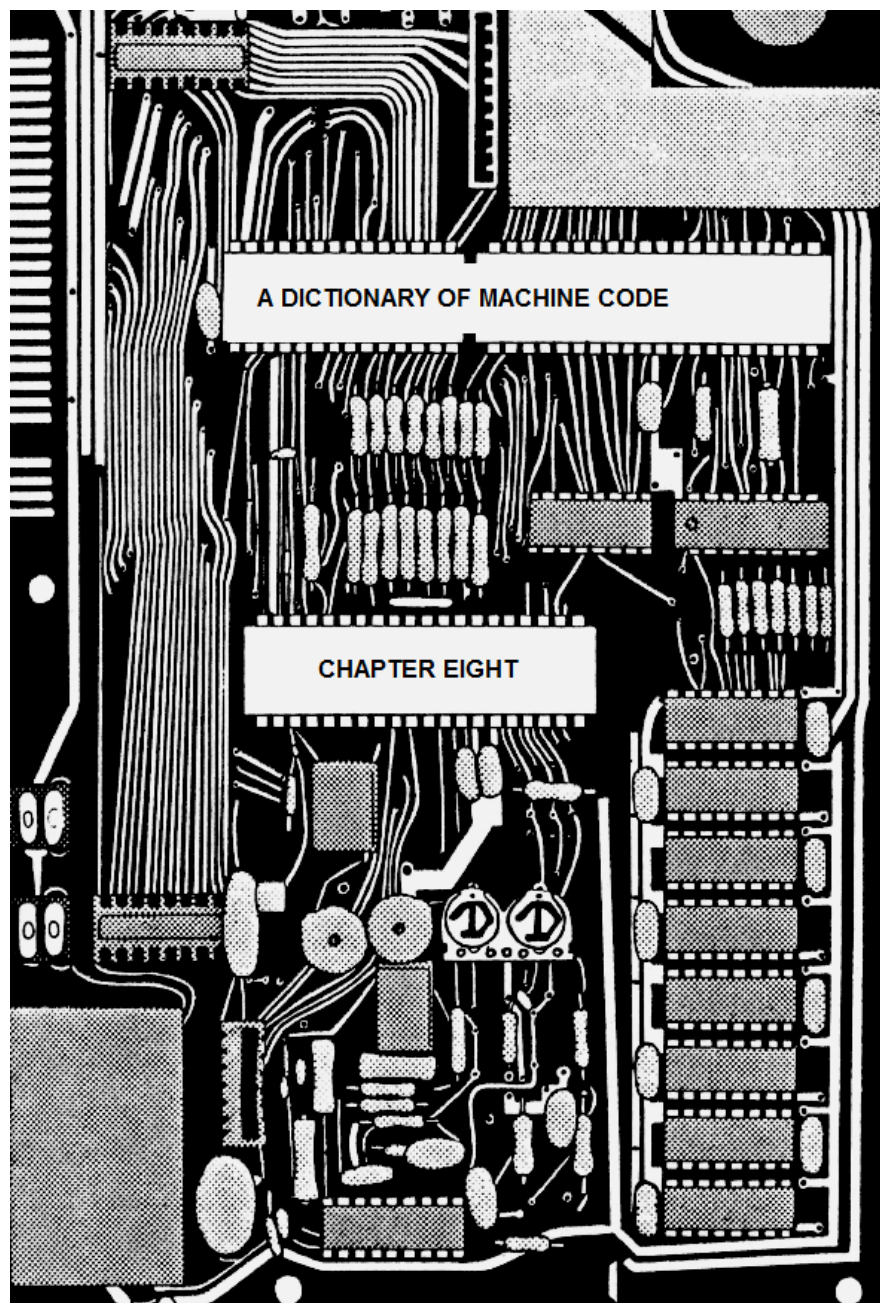
10 DIM A$(22,36)
20 FOR I = 1 TO 22
30 LET B$ = CHR$ (INT (96*RND) + 32)
40 FOR J = 0 TO 5
50 LET B$ = B$ + B$

```

*Mastering Machine Code on your ZX Spectrum*

```
60 NEXT J
70 LET A$(I) = CHR$ 16 + CHR$ INT (8*RND) + CHR$ 17 + CHR$ INT
  (8*RND) + B$
80 PRINT A$(I)
90 NEXT I
100 LET A = 1
110 PAUSE 0
120 LET B$ = INKEY$
130 LET B = A + 1 :IF B = 23 THEN LET B = 1
140 LET C = A - 1 :IF C = 0 THEN LET C = 22
150 IF B$ = "6" THEN PRINT AT 0,USR down;A$(C): LET A = C
160 IF B$ = "7" THEN PRINT AT 21,USR up; A$(A): LET A = B
170 INPUT ""
180 GO TO 110
```

Notice the INPUT "" instruction on line 170. This has the effect of clearing the lower two lines of the screen.



A DICTIONARY OF MACHINE CODE

CHAPTER EIGHT

# CHAPTER EIGHT

## A DICTIONARY OF MACHINE CODE

### MACHINE CODE REGISTERS

In addition to the registers A, B, C, D, E, H and L, and the flags sign, PV and carry, there are a couple of other registers and flags, although not all of them are useful. Let's cover the registers first.

**IX** is a register pair; however, it may not be split into its component bytes like HL can. Any instruction in machine code which involves HL (not in brackets) may be rewritten with IX instead of HL. (There are three exceptions to this rule: EX DE, HL, ADC HL, and SBC HL,.) The hex code of such an instruction is DD followed by the hex code of the corresponding instruction using HL. Any instruction in machine code which involves (HL) (with brackets) may be rewritten with (IX + dd) instead of (HL) – the dd represents any byte. For instance, since there is an instruction LD (HL),03 then there is also an instruction LD (IX + 2A),03. The displacement byte can be very useful. (There is an exception to this rule: JP (HL) may only be rewritten as JP (IX) – not JP (IX + dd).) The hex code of such an instruction is DD followed by the hex code of the corresponding instruction, but with the displacement byte inserted into the third byte of the hex code. For example, the hex code for LD (HL),03 is 3603; therefore, the hex code of LD (IX + 2A),03 is DD362A03.

**IY** is also a register pair. It is used in exactly the same way as IX except that the hex codes of IY instructions use the byte FD rather than DD. Despite the fact that both of these register names begin with I they do *not* have the high part in common, and are both independent of each other.

**F** is the flags register, sometimes called the *status* register. From time to time this register cohabitates with A in the hope that no-one will notice it.

The *bits* of F each have a specific purpose. Some of them you've seen before.

They are:

Bit 7: The *sign* flag, or S.

Bit 6: The *zero* flag, or Z.

Bit 5: Not used.

Bit 4: The *half carry* flag, or H.

Bit 3: Not used.

Bit 2: The *parity/overflow* flag, or PV, or just P.

Bit 1: The *subtract* flag, or N. (Someone can't spell!!)

Bit 0: The *carry* flag, or C. (Not to be confused with register C).

The *half-carry* flag is set by arithmetic instructions if there is a carry from bit three into bit four or, in the case of register pairs, from bit eleven into bit twelve.

The *subtract* flag is set by any instruction which involves a subtraction, or reset by any instruction which involves an addition.

There are no instructions to directly alter the value of F; however, F can be made to hold a value of, say, xx using LD C,xx/PUSH BC/POP AF. Similarly, the state of the flags H and N may be read using PUSH AF/POP BC and then examining the bits of C.

There is one more flag you should know about. It's called IFF1. It is *not* stored amongst F – it lives all on its own somewhere else in the chip. It may only be assigned by the instructions EI and DI. For more information I suggest you look these up further on in this chapter.



**SP** is the stack pointer. It's a register pair like BC or DE, however, its two component bytes may not be separated. SP *always* points to the topmost item on the machine stack. If you change the value of SP then you automatically create a new stack at the given address. Addresses immediately below the current value of SP are likely to be overwritten without prior consent by something called the Z80's interrupt handling routine (see DI).

**A** is a register you should all know about by now. It's called the *accumulator* from time to time, since it may be used in one or two ways in which the other registers may not (eg ADD A,06).

**B, C, D, E, H and L.** These are registers which you should by now be totally familiar with.

**A'** is a useful register. Note the dash – A' is not the same thing as A. A' is a register which is quite useful for temporarily storing the value of A when it would not be convenient to use the stack. There is only one instruction in machine code which uses A' and that instruction is called EX AF, AF', which will swap the value of A and A'.

**F'** is a rather odd register. It may only be accessed by the instruction EX AF,AF' – no other instruction will change its value. When the instruction EX AF,AF' is reached the F' register is exchanged with the flags register F. Therefore, this register is not really much use except for temporarily storing the values of the flags.

**B', C', D', E', H', and L'** are single byte registers. It is not possible to either assign or read their values directly; however, the instruction EXX will swap B with B', C with C', D with D', E with E', H with H', and L with L'. We sometimes write BC', DE' and HL' to mean the register pairs B'C', D'E', and H'L'. The HL' register pair *must* contain a value of 2758 before a return to BASIC is attempted, otherwise the Spectrum will crash. (The initial value of HL' is 2758 so you do not need to worry about this unless you alter it.)

**I** and **R** are two more single byte registers, however neither of these are really any use. They are primarily intended for the Spectrum's internal hardware.

## ALL THE INSTRUCTIONS

By now we've seen a fair number of Z80 instructions, so you'll be wanting to expand your vocabulary of these. Here now is a detailed list of every single instruction available to you. I shall cover them in alphabetical order so that you may use this chapter as a kind of dictionary of machine code instructions. For precisely that reason I shall also go over the ones we've already seen. You should, however, re-read them as this will provide a useful memory aid.

**ADC** Starting with ADC. It comes in two forms: ADC A,r and ADC HL,s. Here we are using r to stand for either A, B, C, D, E, H, L, a numerical constant, or the contents of an address (HL), (IX + d) or (IY + d). ADC A,r is a single byte instruction. It calculates the sum A plus r plus CARRY. The result is stored in A. ADC HL,s is a two byte instruction which evaluates HL plus s plus CARRY, and stores the result in HL. s here stands for either BC, DE, HL, IX or IY. Can you see why (ignoring flags) ADC A,A does precisely the same job as RLA?

**ADD** Very similar to ADC except that the carry flag is not used in the initial evaluation. It is, however, still altered by the final result. There are two important differences between ADC and ADD though. Firstly, the set of instructions ADD HL,s (where s means as it did in the description of ADC) are one byte instructions rather than two and secondly, it is permissible to use two further sets of instructions ADD IX,s and ADD IY,s.

**AND** Only one form here – AND r. The value of the A register is altered one bit at a time. If such a bit is zero it will remain unaltered, otherwise it will take on the value of the corresponding bit of r. Thus, AND 00 will always result in zero and AND FF will leave A unchanged. AND alters each of the flags. Specifically, the carry flag will always be reset to zero.

**BIT** The form of this instruction is BIT n,r where n is a number between zero and seven. The instruction alters the zero flag (only) according to the current value of the bit in question. If the bit is zero then the zero flag will be set, otherwise the zero flag will be reset. You can exploit this using instructions like JR Z (which will jump if the bit was zero) or RET NZ (which will return if the bit was non-zero). BIT does not alter the value of any of the registers, nor does it change the value of the carry flag. It is a two byte instruction. I tend to find it's not used all that often, but that when it is used it comes in very handy indeed.

**CALL** You've seen this one before – it's rather like GO SUB. Its exact function is as follows: PUSH the return address onto the stack, and then jump to the call address. Since the return address (now on the stack) is used by the instruction RET, it is vitally important that a subroutine should not alter the stack. You may only PUSH things onto the stack in a subroutine if you POP them off again before you attempt to return. CALL may also be used with conditions – for example, CALL Z,pq (pq is an absolute address) which means IF the zero flag is set then CALL pq, otherwise continue with the next instruction.

**CCF** Complement Carry Flag. If the carry flag was zero then set it to one. If it was one then reset it to zero.

**CP** In the form CP r, it will calculate the result of subtracting r from A, but will not store the result anywhere. The previous value of A (and, of course, r) remains unaltered. It will, on the other hand, alter all of the flags, so conditional instructions like JP Z or JP C will still work. CP r followed by JR Z will jump if A equals r (since A minus r is zero) and so on.

**CPD** Imagine this as CP (HL) followed by DEC HL followed by DEC BC. The PV flag is reset if BC decrements to zero and set otherwise. The zero flag is set if CP part of the instruction found that A was equal to (HL), otherwise it is reset.

**CPDR** Basically this is the same as CPD except that the instruction is executed over and over again – a kind of automatic loop if you like. CPDR stands for ComPare with Decrement and Repeat. The loop will end in one of two cases: (i) either the comparison found A equal to (HL) (ie if now  $A = (HL + 1)$ ) and the zero flag is set, or (ii) BC reaches zero, in which case the PV flag is reset.

**CPI** As CPD, except that HL is incremented instead of decremented.

**CPIR** As CPDR, except that HL is incremented instead of decremented.

**CPL** An abbreviation for ComPLement. The register A is altered bit by bit. If any particular bit starts off as zero then it is changed to one and *vice versa*. In other words, if A started off as 11010101 (binary) then CPL would change it to 00101010 (binary). This is equivalent to subtracting A from FF. The flags are not affected by this instruction.

**DAA** Suppose you wanted to add sixteen to twenty-six *without* converting the numbers to hex. The following seems plausible: LD A,16 then ADD A,26. Unfortunately, because the machine works in hex the final value of A will be 3C not 42. The instruction DAA (Decimal Adjust Accumulator) will then change A from 3C to 42. How it works is rather complicated – it makes a note of what's been carried where and whether you've added or subtracted and so on; but it does always work. For instance, the sequence LD A,42 then SUB 06 will again leave A with 3C, but this time round DAA will change A to 36, since forty-two minus six is thirty-six. The instruction changes every flag appropriately.

**DEC** This is another one of those instructions which comes in two forms. It can be DEC r (a single register) or DEC s (a register pair). DEC r is very simple to understand – the value of the register r is decremented (decreased by one, or changed from 00 to FF), the carry flag is unaltered and the zero flag is changed as you'd expect it to be. DEC s, however, is a sneaky little instruction, for the zero flag is *not* altered! In fact, none of the flags at all are changed! Thus DEC BC/JR NZ, -3 is either an infinite loop

or has no effect! You must be very careful to remember this – a lot of my earlier programs crashed because I didn't!

**DEFB** Technically speaking this isn't really a machine code instruction – it's what's called a directive. The word DEFB must be followed by one or more bytes of data, each separated by a comma. Usually this data is in hex, but this isn't always necessary, eg DEFB 3A,C1,45d,11011110b,"f",*yellow* is valid. The data is inserted into the machine code program at the point it occurs, and in the order it's listed. Data which forms part of a machine code program should never be executed, since the Z80 cannot distinguish between data and program.

**DEFM** Similar to DEFB except that the data which follows the word DEFM must be a string of characters (not surrounded by quotes). Commas in the text will themselves be interpreted as data, not as separators. For instance, DEFM SOLSTICE will cause the bytes 53 4F 4C 53 54 49 43 45 to be inserted into the program. Control characters may also be inserted into a DEFM directive provided they are made distinguishable from ordinary symbols (eg if they are underlined or in italics). DEFM HELLO *enter* THERE and DEFM *ink blue* WHAT? are both valid, although some convention must be established about spaces around the control word(s). I tend to write spaces in for clarity so that I would write DEFM X *enter* to mean 58 0D, and DEFM X *space enter* to mean 58 20 0D. DEFM stands for DEFine Message, as opposed to DEFB which means DEFine Byte(s).

**DEFS** Yet another of those directives. This one means DEFine Space(s). The word DEFS should be followed by one numerical constant. (Just one, mind.) The given number instructs the computer to insert *that number* of rubbish characters. So, DEFS 08 will insert eight bytes into the program at that point, but without really worrying about what those bytes actually are. DEFS is used mainly to define "variables" in RAM; eg FRED DEFS 02 (FRED is a label) and then at some later point in the program LD (FRED),HL.

**DEFW** One last directive (for now). DEFW stands for DEFine Word. It is used in a very similar way to DEFB except that the items of data are two bytes long, not one, so that (for instance) DEFW 4000 is equivalent to DEFB 00,40. Note how the bytes are switched around. Labels and expressions are permitted in DEFW, so DEFW 7000, FRED,ERIC + 3 is quite valid.

**DI** Not a Welsh name, nor is it short for Diane or Diana. It is, in fact, an abbreviation (surprise, surprise). It stands for Disable Interrupts and, although this sounds pretty confusing, its use is immensely simple. Fifty times a second a little pulse is sent down one of the pins of the Z80 chip. There is a flag called IFF1 which stands for Interrupt Flip Flop One (no, I didn't make that up!), and the effect of DI can be thought of as RES IFF1. When the Z80 receives one of these pulses it checks the value of the flag IFF1. If it is set then the computer acts as if a RST 38 instruction (or CALL 0038) has just been reached, with the return address being the next instruction in sequence. If IFF1 is *reset* however, then no action is taken and any machine code program will run as normal. The flag IFF1 must be set *before* any attempt is made to return to BASIC otherwise the Spectrum will not register any key being pressed.

**DJNZ** Yet another abbreviation. This one stands for Decrement B and Jump if Not Zero. So if B is 7, DJNZ will reduce it to 6 and then jump to a new destination. If B is zero, DJNZ will reduce it to FF and again jump to a new destination. If B is one, however, then DJNZ will change it to zero and will *not* jump to a new destination. Instead, it will simply carry on with the next instruction. The form of the instruction is DJNZ e, where e is a single byte. If B is decremented to zero then the e is ignored. If not, then the e specifies how far to jump. The displacement calculated is as in JR.

**EI** Guess what? Another abbreviation. EI stands for Enable Interrupts and is the opposite of DI. This instruction is equivalent to SET IFF1. See DI for a more complete explanation.

**EQU** Short for EQUate. This is *not* a machine code instruction, but a directive. Each EQU directive must have a label attached to it, and the word EQU must be followed by a number (in the range 0000 to FFFF) or an expression such as ERIC + 2. When this directive is reached no action is taken, and no bytes are inserted into the program – therefore, it doesn't really matter whereabouts in the program an EQU appears although it is conventional to place all EQU directives at the very start of a program. What it does is to assign a numerical value (the one given) to the label attached. In other words, if you have ANNIE EQU 9000 and then at some later stage LD HL,(ANNIE), then the instruction will be compiled as LD HL,(9000).

**EX** Short for exchange. This instruction will swap the values contained by certain register pairs. There are five EX instructions – these are EX AF,AF'; EX DE, HL; EX (SP), HL; EX (SP),IX; and EX (SP),IY. They don't alter any of the flags, all they do is swap the values over – thus, EX DE,HL replaces DE by the value HL used to contain and HL by the value DE used to contain. The last three are rather interesting – the old value of HL (or IX or IY) is swapped with the topmost item on the stack, so LD BC,0123/PUSH BC/LD HL,4567/EX (SP),HL/POP BC will leave BC with 4567 and HL with 0123. EX (SP),HL does not move the stack pointer and nor, of course, do EX (SP),IX or EX (SP),IY.

**EXX** You can imagine this as EX BC,B'C' followed by EX DE,D'E' followed by EX HL,H'L'. Basically, each of the common registers (except A) is exchanged with its corresponding alternative register. You are warned, however, that the HL' register pair must contain the value 2758 before returning to BASIC, so if you alter it in any way you must re-assign it before too long.

**HALT** When a HALT instruction is reached, control will wait at that point in the program until the next interrupt occurs. When this happens the instruction RST 38 (CALL 0038) will be executed and on return, control will then continue from the first instruction after HALT. Note that the flag IFF1 *must* be set if a HALT is to be executed, otherwise an interrupt will never

occur. In this case, HALT will literally halt forever. There will be no way of breaking out except by pulling out the plug. The effect of HALT, as far as you and I are concerned, is to wait for the next TV frame to be output – roughly equivalent to PAUSE 1 in BASIC. See DI for an explanation of the use of IFF1.

**IM** *DANGER!!!* Under no circumstances use this instruction.

**IN** This is more or less the same as the BASIC IN function. It has two forms. The first is IN A,(n) where n is a numerical constant. This is equivalent to the BASIC statement LET A = IN(256\*A + n). The second form is IN r,(C) where r is a register. This is equivalent to the BASIC statement LET r = IN(256\*B + C). The argument of IN refers to a hardware device outside the Z80 chip – a different number for each different device. In the form IN A,(n) the flags are not altered; however, the flags *are* altered by IN r,(C).

**INC** Don't panic! At long last we're back to sensible instructions we can all understand. INC r increases the value of the register r by one, but without altering the carry flag. INC s increases the value of s by one but alters no flags at all.

**IND** IN with Decrement. IND can be thought of as IN (HL),(C) followed by DEC HL followed by DEC B. The carry flag is unaltered, but the zero flag reflects the new result of B.

**INDR** As IND, except that the instruction re-executes over and over again, stopping only when B reaches zero.

**INI** As IND, except that HL is incremented instead of decremented.

**INIR** As INDR, except that HL is incremented instead of decremented.

**JP** If you can understand GO TO 10 then you can understand JP 7300. The destination is an address, not a line number, but the principle is



exactly the same. JP is the machine language GO TO. We also have conditional jumps, for example JP NZ,7300 means IF non zero THEN jump to address 7300 (In other words if the zero flag is not set.) There is another form of JP which also has an analogy in BASIC – variable destinations. If you understand GO TO N then you'll understand JP (HL). JP (HL) means GO TO HL. In this form you may not have conditions: for instance, JP NC,(HL) is not allowed. Only one of three register pairs may be used as variable destinations – these are HL, IX and IY. Even so these are very powerful instructions – HL can be the result of a calculation, possibly even generated at random.

**JR** The same as JP only slightly less powerful, but one byte shorter. Only four of the eight conditions may be used: Z, NZ, C and NC. This means it is impossible to use (for instance) JR PO. It is also impossible to say JR (HL). JR does not use an absolute address – the R stands for Relative. You write the instruction as JR e (or JR Z,e or whatever) where the e is a single byte which specifies how far to jump. JR 0 has no effect, since it jumps forward by zero bytes. JR FE is an infinite loop because control will jump back to the JR FE instruction itself. The displacement byte starts counting from the instruction immediately after the JR e instruction. If the byte is between 00 and 7F then the jump is forward, and if the byte is between 80 and FF then the jump is backwards. See table two of Appendix One.

**LD** The most used instruction in the whole of machine language. All it does is to transfer data from one location to another. It has many, many forms: the simplest being LD r1,r2 – that is to transfer data from one register to another. Other forms are LD A,(BC), LD A,(DE), and LD A,(HL) – and in reverse, LD (BC),A, LD (DE),A and LD (HL),A. Remember that the brackets mean the *contents* of an address. Registers I and R may be loaded, in conjunction with A (but only A) the registers and the register pairs may all be loaded with numerical constants, the register pairs with the contents of any address and inversely any address may be loaded from a register pair (– note that register pairs store two bytes, not one, and these are transferred to and from the address pointed to *and the address pointed to plus one*). Also allowed are LD A,(pq) and LD (pq),A (where pq

represents an address) and SP may be loaded from HL, IX, IY or (pq). ((pq) may be loaded from SP but HL, IX and IY may not.) In other words – there’s a lot you can do and a lot you can’t. You can’t say LD HL,DE, for instance (– you must use LD H,D then LD L,E or vice versa). Fortunately, since LD is used so very, very often, it is extremely easy to become familiar with its many forms.

**LDD** Load with Decrement. Effectively, LD (DE),(HL) followed by DEC HL, DEC DE and DEC BC all in one go. The carry flag and zero flag are unaltered as is the sign flag, however the PV flag is reset to zero if and only if BC decrements to zero. Thus, JP PO will jump only if BC is zero after the instruction.

**LDDR** As LDD, but the instruction is executed repeatedly until BC reaches zero.

**LDI** As LDD except that DE and HL are both incremented instead of decremented. BC is still decremented as before.

**LDIR** As LDI, but the instruction is executed repeatedly until BC reaches zero.

**NEG** NEGates the accumulator (or A register). It works by performing the subtraction 00 minus A and all the flags are changed accordingly. Thus, S reflects the sign of the result. Z will be set if and only if A is zero. P will be set if and only if A is 80. C will always be set unless A is zero. NEG is equivalent to CPL followed by INC A (ignoring flags).

**NOP** My favourite instruction. This wondrous little beastie (whose name incidentally is short for No Operation) has a very simple purpose – to waste time. NOP does absolutely nothing at all except sit around all day drinking tea, and what’s more it takes it’s time doing it! It has two major uses: (i) as a delay, or (ii) to overwrite previous machine code when debugging or editing. I suppose its nearest BASIC equivalent would be a blank REM statement. I’d say this instruction was virtually indispensable.

**OR** In the form `OR r` this instruction is almost the opposite of `AND r`. Bit by bit the value of the A register is changed. If any given bit is *one* then it remains unaltered, otherwise it takes its new value from the corresponding bit of `r`. If A contains 00 then (ignoring flags) `OR r` is the same as `LD A,r`. `OR FF` is effectively `LD A,FF`. All of the flags will change as you'd expect them to and the carry flag is reset to zero.

**ORG** Yet another of those funny directive things again. `ORG` is a directive which must *not* have a label attached. The word `ORG` must be followed by a number in the range 0000 to FFFF. It means all machine code from here on is to be written to the address given. Thus, `ORG 7000` followed by `LD A,01` means that the instruction `LD A,01` is to reside at address 7000. Unless the next thing encountered is another `ORG` directive, then the next instruction will be at address 7002 (since `LD A,01` is a two byte instruction).

**OUT** You've seen this one in BASIC. It's printed in red on the key marked "0". The machine code `OUT` instruction has two forms. The first is `OUT (n),A` – this is equivalent to the BASIC statement `OUT 256*A + n,A`. The second form is `OUT (C),r` which has the BASIC equivalent `OUT 256*B + C,r`. `OUT` sends numbers out of the Z80 chip and into the hardware outside. It has absolutely no effect on the flags.

**OUTD** `OUT` with Decrement. Equivalent to `OUT (C),(HL)` followed by `DEC HL` followed by `DEC B`. The carry flag is unchanged, but the zero flag reflects the new value of B.

**OTDR** A slightly different spelling in no way alters the fact that this is still an `OUT` with Decrement and Repeat instruction – all it does is lead us to digress from alphabetical order in order to maintain consistency. Equivalent to `OUTD` repeated over and over again until B reaches zero.

**OUTI** As `OUTD`, except that HL is incremented instead of decremented.

**OTIR** As `OTDR`, except that HL is incremented instead of decremented.

**POP** Removes two bytes of data from the top of the stack and loads them into a register pair. Register pairs BC, DE, HL, IX and IY may be used. In addition, the instruction POP AF may be used forming a pseudo “register pair” from the accumulator and the flags register. Specifically POP will remove the topmost byte from the stack into the low part of the register pair and the next byte into the high part. The stack pointer SP is updated automatically.

**PUSH** PUSH is the opposite of POP. It stores the contents of any register pair at the top of the stack. The high part is pushed first, then the low part. It “remembers” that a new item has been stacked by updating the value of SP. After a PUSH instruction, SP will always point to the low part of the topmost item on the stack.

**RES** With this instruction we can actually alter individual bits of any register. RES is short for RESet, which means “change to zero” in computing circles, so RES is the instruction which changes any required bit of a register to zero. For instance, to reset bit 3 of D you just have to say RES 3,D. RES has no effect on any of the flags.

**RET** RET is used to return from a subroutine. It works by POPping an address from the stack and then jumping to that address. It is possible to alter the address to which a subroutine will return by altering the value at the top of the stack. For example, POP HL/INC HL/PUSH HL will increase the return address by one. You could, for instance, store one byte of data immediately after the CALL instruction, then POP HL/LD A,(HL)/INC HL/PUSH HL will store that byte in A while at the same time ensuring that the subroutine will return to the address *after* that data. Another trick is to push an “artificial” return address onto the stack and then JP (or JR) to a subroutine instead of CALLing it. Now it will “return” to wherever you want it to go! Return may be used with conditions if needed. It does not alter the flags.

**RETI** Not applicable to the Spectrum.

**RETN** Not applicable to the Spectrum.

**RL** The form of this instruction is RL r. Each bit of the register specified is moved one position to the left. The leftmost bit is moved into the carry, and the rightmost bit takes on the previous value of the carry. Hence, Rotate Left. For example, if B contained 10010101 and the carry contained zero then RL B would leave B containing 00101010 and the carry containing one. RL alters all of the flags.

**RLA** Note that there is no space between the L and the A. RLA is a more efficient way of doing RL A! The instruction is one byte shorter and only the carry flag is affected by this instruction.

**RLC** Rotate Left without Carry. RLC r is almost the same as RL r, in fact, in the sense that each bit of the register in question is moved one position to the left. Here, however, the former leftmost bit becomes both the new value of CARRY and the new rightmost bit. The former value of CARRY does not enter into the process at all. All of the flags are changed.

**RLCA** In one byte instead of two, RLCA is just RLC A only quicker. Only the carry flag is changed by this operation.

**RLD** Now for a weird one. RLD is not to be confused with RL D for it is a completely different instruction which works as follows: write the value of A and the contents of address (HL) in hex. The second hex digit of (HL) is shifted left so that it becomes the new first digit. The former value of this first digit overwrites the second digit of A, which in turn becomes the second digit of (HL). Thus, if we start off with A containing 25 and (HL) containing A3 then RLD will change things to A = 2A, (HL) = 35. RLD incidentally, for some reason known only to the boffins above, is an abbreviation for Rotate Left Decimal.

**RR** As RL, except that the bits are moved right instead of left.

**RRA** As RLA, except that the bits are moved right instead of left.

**RRC** As RLC, except that the bits are moved right instead of left.

**RRCA** As RLCA, except that the bits are moved right instead of left.

**RRD** As RLD, except that the hex digits are moved right instead of left.

**RST** The same as CALL except that the instruction is but one byte long *altogether!* It is much less powerful though for two reasons: (i) you may *not* use conditions (eg RST 10 is legal but RST NZ,10 is not); and (ii) only one of eight specific addresses may be called. These are 00, 08, 10, 18, 20, 28, 30 and 38. Since the Spectrum begins executing the ROM from address 0000 onwards, RST 00 is the same as pulling out and then reconnecting the plug.

**SBC** SBC, like ADC, comes in two forms. The first is SBC A,r, which will first of all subtract r from A, and will then subtract the carry. Similarly SBC HL,s will subtract both s and the carry flag from HL. SBC A,A is quite a useful instruction – it leaves the carry unchanged but alters A to 00 if there is no carry or FF if there is a carry.

**SCF** Set the Carry Flag. All other flags are unchanged.

**SET** The opposite of RES. SET 4,H will set bit 4 of register H to one (for example). Any bit of any register may be set.

**SLA** Shift Left Arithmetic. The form of this instruction is SLA r. It is similar to RL r except that the rightmost bit is always replaced by zero. SLA r will multiply the register r by two.

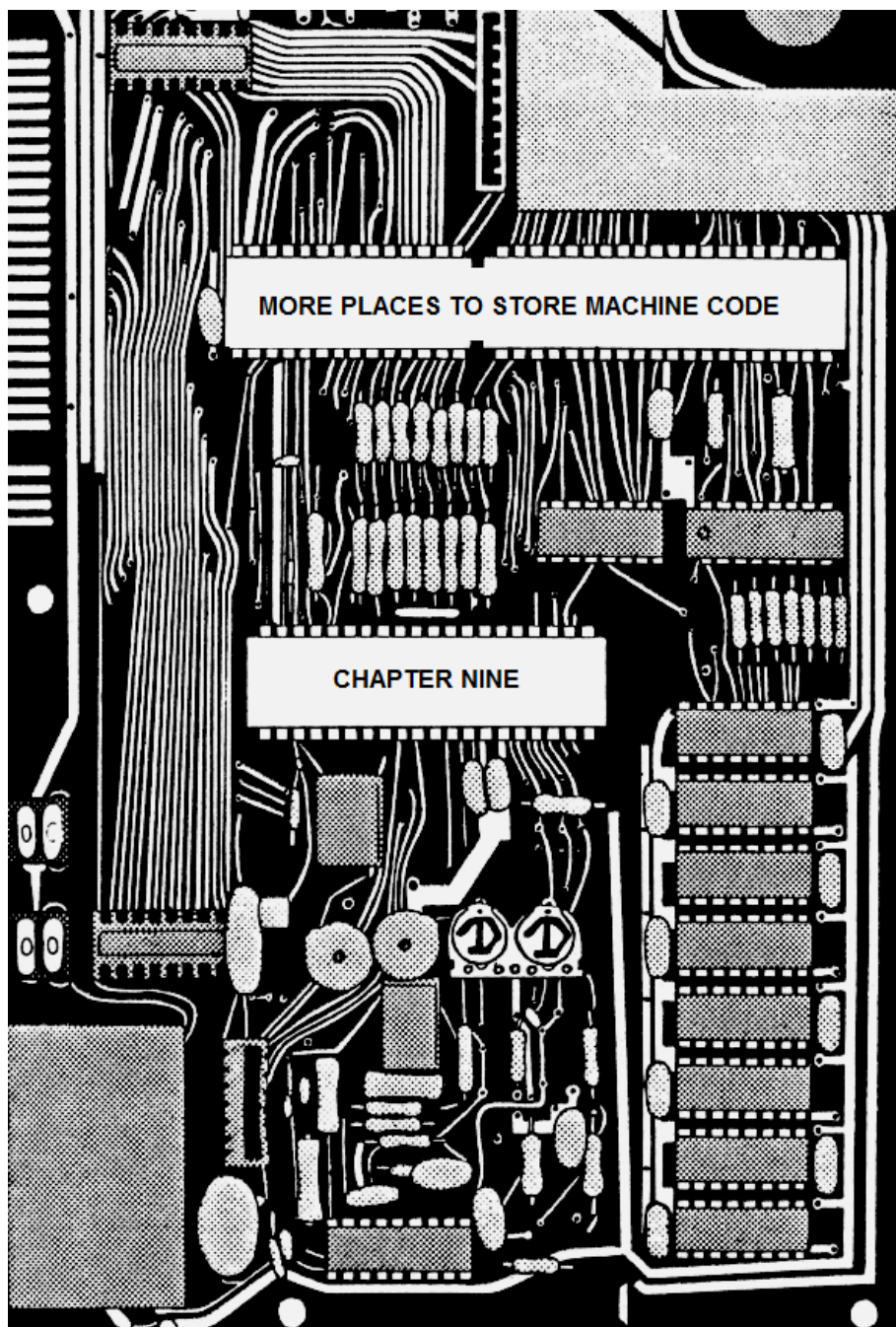
**SRA** Shift Right Arithmetic. Any register may be shifted right using the format SRA r. The instruction is similar to RR r except that the leftmost bit remains unchanged. SRA r will divide the register r by two if that register contains a number which is to be regarded as being in two's complement form (Appendix One, table two).

**SRL** Shift Right Logical. Again, similar to RR except that here the leftmost bit is replaced by zero always. `SR L r` will divide the register `r` by two if that register contains a number which is to be regarded as being in absolute value form (Appendix One, table one).

**SUB** Written as `SUB r` (but sometimes as `SUB A,r` just to be confusing). This instruction will subtract `r` from the register `A`. Note that unlike with `ADD` there is no corresponding instruction `SUB HL,s`. If you wish to subtract `s` from `HL` you must first of all reset the carry flag (usually by the use of the instruction `AND A`) and then use `SBC HL,s`.

**XOR** `XOR r` changes the value of `A` bit by bit. If any given bit of `A` is identical to the corresponding bit of `r` then that bit of `A` is reset to zero, otherwise that bit of `A` will be set. `XOR` alters all the flags and, in particular, the carry flag is always reset. Note that `XOR A` is the same as `LD A,00` (ignoring flags) and that `XOR FF` is the same as `CPL` (also ignoring flags).

**\$** The final directive. `$` is not really a directive in its own right. Technically, it's just a special symbol which may be used in an `EQU` directive. However, if the dollar symbol is used in an `EQU` directive then the word `EQU` itself may be omitted! `$` simply means the address of the next byte, so `FRED $ + 2` (which is a short way of writing `FRED EQU $ + 2`) where `FRED` is a label means "define `FRED` to mean the next address plus two". An example of its use could be `LD (HL),00/CHEAT $ - 1` (`CHEAT` is a label) and then at some later stage `LD (CHEAT),A` which forms the beginnings of a self adjusting program. Can you see that the label `CHEAT` has been defined to mean the address of the *second* byte of the `LD (HL),00` instruction. Interesting, no? I shall leave you to ponder.



MORE PLACES TO STORE MACHINE CODE

CHAPTER NINE



# CHAPTER NINE

## MORE PLACES TO STORE MACHINE CODE

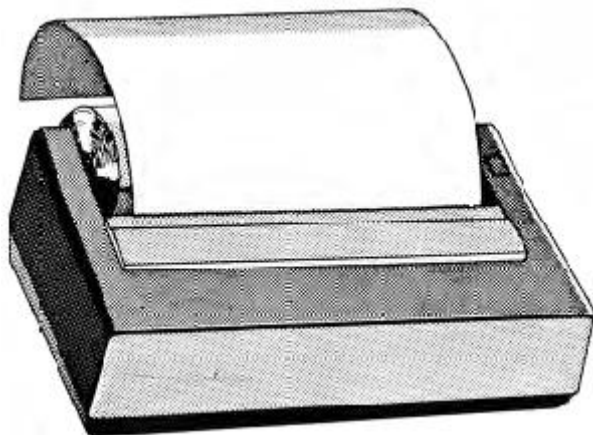
Storing machine code above RAMTOP as we have been doing so far will protect it from being erased by NEW or overwritten by a program, but it has the disadvantage that you must do two separate SAVES if you wish to store both the machine code and a surrounding BASIC program as well. There are several alternative locations in which we can store machine language programs, and we shall explore a few of the possibilities in this chapter.

### USING REM

One place to store machine code is in a REM statement. If you make line one of a BASIC program, REM followed by a number of bytes then you can store machine code in place of those bytes – ie machine code may be stored between the word REM and the end of line one. Let's see how we could do this. Suppose you wish to store a machine code routine that is fifty bytes long. Make the first line of your program:

```
1    REM 123456789012345678901234567890123456789  
    01234567890
```

This is a REM statement with fifty characters after the word REM. If your routine was sixty bytes long then you'd need sixty characters; if it was only three bytes long then you would only need three bytes after the word REM. It doesn't matter in the slightest what these characters actually are, but if you count upwards in ones, as I have done, then you have a good way of ensuring that you don't lose count halfway through.



**Figure 9.1**

You can use HEXLD 2 here if you like, but there are some alterations you must make. Firstly though, a word or two of explanation. Line *two* of BASIC should read:

```
2 DEF FN J(X) = PEEK 23635 + 256*PEEK 23636 + 5 + X
```

If you do this then the function FN J(X) will calculate the address of the (X + 1)th character after the word REM – obviously a very useful thing to know. If you then add just one more line to HEXLD 2:

```
255 LET X = FN J(X)
```

then we are ready to go. The hex number you are required to input as the starting address should now be not an absolute address but a displacement from the first character after the word REM, so that inputting “0000” will refer to the first character itself. Now you can enter a machine code program exactly as before and it will be written not into spare RAM but into the REM statement of line one. This machine code may be run either by RUN 700 (using HEXLD 2) or as a direct command: RANDOMIZE USR FN J(0).

## *Mastering Machine Code on your ZX Spectrum*

Once your machine code program is set up and working, you can then delete the program entirely except for lines one and two. You should do this not by typing NEW but by inputting the line numbers one at a time. The machine code program may be run using the BASIC statement RANDOMIZE USR FN J(0). (Or if you want to run it from the fifth byte, for instance, then RANDOMIZE USR FN J(4).)

There are advantages and disadvantages of using REM, so I'll go through them all one at a time. First the disadvantages.

The first disadvantage is that line one can never be listed – at least it shouldn't be! Certain machine code instructions will cause the listing to be corrupted. For instance, LD B,n (hex 06) will cause odd spacing, DEC C (hex 0D) will produce some weird effects, DJNZ and LD DE,mn will more often than not produce error K Invalid colour, and so on. The rest of the program may be listed quite happily (using LIST 2 for instance).

You may not, in such a REM statement, refer to any absolute addresses within the machine code program. This is because a BASIC program will tend to move up and down in memory if, for instance, a Sinclair microdrive is attached. FN J(0) may give different answers at different times.

The command NEW will erase it.

The advantage, however, is that the machine code forms an integral part of the BASIC program. It is LOAded and SAVEd simultaneously with it, and a separate LOAD instruction is not needed. VERIFY "" will verify both the program and the machine code in one go.

With REM statements you sometimes find yourself panicking. What do you do if, for instance, you find there are not enough characters after the word REM to store your machine code? It is not possible to extend the length of line one using EDIT as you'd expect if some of your machine code already exists there.

Almost any occurrence of character 10 (and several others) will cause an irritating buzz if any attempt is made to edit the line and the line will be invisible beyond that point. Any occurrences of the byte 0E in the original line will be *deleted* automatically by the ROM, as will the first five bytes after an 0E. In the listing (if you manage to get a listing) the byte 0E and the first five bytes which follow it are invisible from the listing, but are in fact still there – if you use EDIT, on the other hand, then they will tend to disappear without trace.

The byte 0E is used by the Spectrum to mean “There now follows a (floating point) number”. Whenever you use a decimal or binary (with BIN) number in a program listing, the ROM will automatically follow this number with a byte 0E, followed by five more bytes which contain the number itself in “floating point” form. Both the byte 0E and the five bytes that follow will be invisible from the listing. This is what causes all the problems in editing REM statements.

The only practical means of extending the length of a REM statement is to join two or more REM statements together. At the start of every line of program there are two invisible bytes which store the length of that line, so you have to actually POKE those invisible bytes with different values. The following is a small routine which will enable you to increase the length of a REM statement at line one.

Step one is to change the line number of the DEF FN J statement to *anything but* one or two, and then to add a new line two consisting of the word REM followed by a number of arbitrary characters. Then, at *any* point in the program, insert the following six lines (they will shortly be deleted anyway):

```
LET B = FN J(-3)
LET A = B + PEEK B + 256*PEEK (B + 1) + 4
LET A = A + PEEK A + 256*PEEK (A + 1) - B
POKE B,A - 256*INT (A/256)
```

```
POKE B + 1,INT (A/256)
STOP
```

The line `LET B = FN J( - 3)` can, of course, be replaced by `LET B = PEEK 23635 + 256*PEEK 23636 + 2`. Simply run this routine and line two will automatically become a part of line one. You can delete this routine now – its job has been done. LIST line one – you’ll see that line two still looks quite separate, but try moving the cursor down and you’ll find that it skips over line two altogether. Try deleting line two by typing in its line number – it won’t work because now the computer doesn’t even know that line two is there! Whatever the listing may *look* like, the ROM will now ignore line two altogether taking it to be a part of line one. You may now quite happily overwrite the end of line marker (character 0D) at the end of line one with no ill effects.

Conversely, the following routine will shorten a REM statement by a *minimum* of six bytes:

```
LET A = FN J( - 3)
LET B = PEEK A + 256*PEEK (A + 1)
LET C = the number of bytes you wish to preserve + 2
POKE A,C - 256*INT (C/256)
POKE A + 1,INT (C/256)
LET A = A + C + 1
LET C = B - C - 4
POKE A,13
POKE A + 1,0
POKE A + 2,2
POKE A + 3,C - 256*INT (C/256)
POKE A + 4,INT (C/256)
STOP
```

Again, you simply RUN the routine once and then delete it. Now, LIST the program and you’ll find a new line two has appeared. Delete this by typing its line number and your REM statement will be as short as you need it.

## USING PROGRAM AREA

Figure 9.2 shows the area of RAM I'm talking about here. The advantages and disadvantages of using this area are exactly the same as for REM. However, there is one further advantage that you do not need to ever worry about LISTing. The machine code will be totally invisible, and yet will still form an integral part of the program.

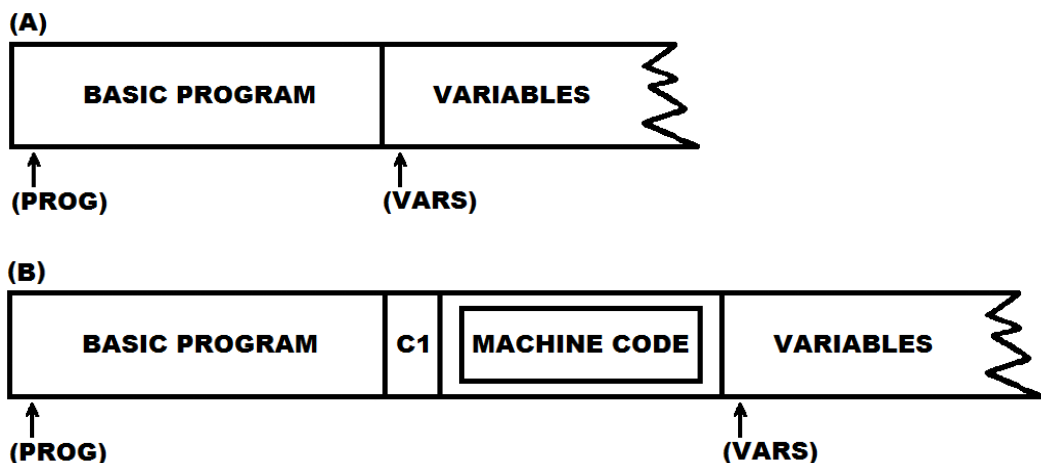


Figure 9.2

To set up this area with, say, fifty bytes you must type in the following:

```
CLEAR
DIM A$(45)           45 = five less than 50 (decimal)
LET A = PEEK 23627 + 256*PEEK 23628 + 51
                    51 = one more than 50 (decimal)
LET B = INT (A/256)
PRINT B
POKE 23627,A - 256*B
POKE 23628, the number printed
```

## *Mastering Machine Code on your ZX Spectrum*

The number in the DIM instruction should always be five less than the number of bytes you wish to preserve. The number at the end of the next instruction is one more than the number of bytes to be preserved. The process is now complete and all we need is a means to access the machine code. This is achieved by adding the following BASIC statement to your BASIC program (any line number will do):

```
DEF FN J(X) = PEEK 23627 + 256*PEEK 23628 - 50 + X
```

The number 50 refers to the fact that fifty bytes have been set aside. If you wished to set aside one hundred bytes then the DIM instruction earlier on should be DIM A\$(95), the next instruction should end + 101 and the DEF FN statement should end - 100 + X. FN J will now work exactly as it did for REM so that FN J(0) gives the address of the first byte of machine code, FN J(1) gives the address of the second byte of machine code, and so on up to FN J(49) for the last byte (or 99 if a hundred bytes were reserved, etc). Machine code placed in this area will be loaded, saved and verified along with the BASIC program so that only one SAVE is needed.

I ought to explain about the byte C1 in the diagram (figure 9.2(b)). You don't need to POKE this in – it gets left there automatically by the process we've just used. C1 is the byte used by the computer to represent the name of the string array A\$. If we'd have used DIM B\$ instead of DIM A\$ then this byte would have been C2. It doesn't really matter what byte you use to separate the BASIC from the machine code, as long as that byte is greater than or equal to hex 40.

## **PASSING PARAMETERS TO MACHINE CODE**

A change of tack completely now. We've seen one use for FN already. Now let's look at another – consider the statement:

```
DEF FN U(X)
```

*(where a machine code routine is stored in RAM at address 28672)*

Interesting, isn't it? Of course, the number involved does not have to be 28672 – it just has to be the address of a machine code program. Now, whenever FN U is used then that machine code program will be called. So far so good, but we haven't gained any real advantage over just using USR whenever we need it. But now think about the "X" in the brackets – what happens to that?

One of the system variables is called DEFADD, and it stores the address of the arguments of a FN calculation. These arguments are stored in a rather strange way (as always I suppose). Rather than spending hours trying to explain it all, what I'll do is to plunge in straight at the deep end to see if you can spot the pattern. Don't worry – I'll make it easy for you.

Consider this BASIC statement:

```
DEF FN A(A,B,C,A$,B$,C$) = USR some number
```

There are six arguments enclosed in the brackets, so the ROM keeps six records of these values. The address of the first record is pointed to by the system variable DEFADD. Take a look now at figure 9.3 which shows how these records are organised. Numerical records take up eight bytes, and string records take up nine bytes. In the case of numerical records, the first byte is the name of the record as used in the DEF statement; the second byte is always the byte 0E. The next five bytes store the number calculated within the brackets of FN (for instance, FN A(3 + 3, etc...) is allowed, but the actual result calculated is six). However, if this result is in the hex range 0000 to FFFF and is positive, then we may read its value directly as the fifth and sixth bytes of the record – the third, fourth and seventh bytes are all zero, and the eighth byte will always be 2C (comma) unless the record is the last record on file, in which case the eighth byte will be 29 (right bracket). String records are just as silly – the first two bytes store the name of the record as used in the DEF FN statement; however since the name always ends in dollar then the second byte is always 24 (dollar), the third byte will always contain the byte 0E, the fourth byte is not used and will contain rubbish, the fifth and sixth bytes will contain the address at which



*Mastering Machine Code on your ZX Spectrum*

the evaluated string begins, the seventh and eighth bytes store the length of the string and finally, the ninth byte stores comma or right bracket depending upon whether the record is the last record in the file.

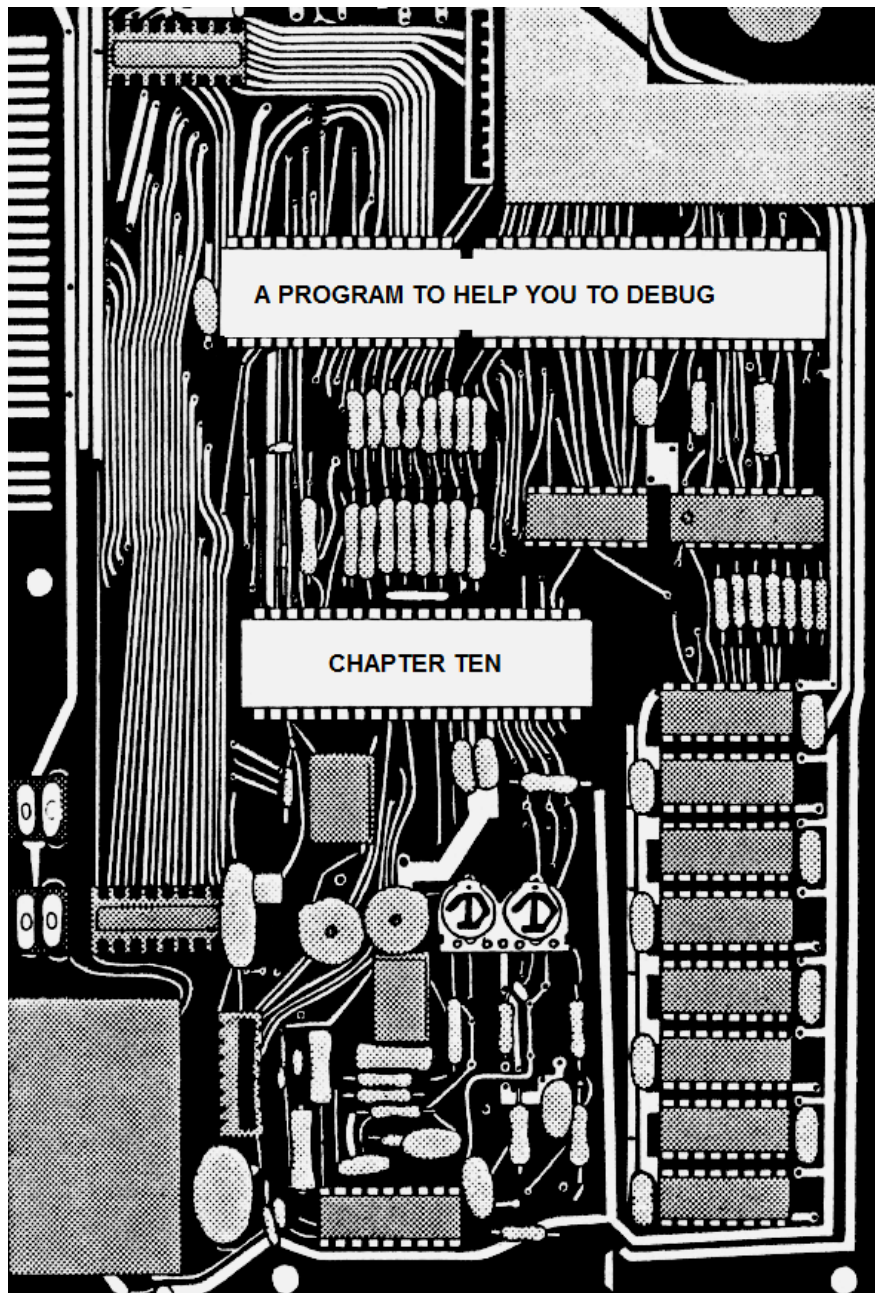
Got all that? If not it doesn't matter. The gist of it is that DEF FN U(X) = USR *something* in conjunction with, say, FN U(X + Y) will call a machine code routine. The value X + Y can be retrieved in machine code by LD HL, (DEFADD)/INC HL/INC HL/INC HL/INC HL/LD C,(HL)/INC HL/LD B,(HL).

## ARRANGEMENT OF RECORDS FOR FN A (A, B, C, A\$, B\$, C\$)

FIRST RECORD	A				FIRST PARAMETER			'	
	41	0E	00	00	aa	aa	00	2C	
		↑							
		(DEFADD)							
SECOND RECORD	B				SECOND PARAMETER			'	
	42	0E	00	00	bb	bb	00	2C	
THIRD RECORD	C				THIRD PARAMETER			'	
	43	0E	00	00	cc	cc	00	2C	
FOURTH RECORD	A				STRING ADDRESS		STRING LENGTH		'
	41	24	0E	??					2C
FIFTH RECORD	B				STRING ADDRESS		STRING LENGTH		'
	42	24	0E	??					2C
SIXTH RECORD	C				STRING ADDRESS		STRING LENGTH		)
	43	24	0E	??					29

Figure 9.3

I think I'll leave things there and move on to the next chapter. After all, you're all dying to move on aren't you?



A PROGRAM TO HELP YOU TO DEBUG

CHAPTER TEN

# CHAPTER TEN

## A PROGRAM TO HELP YOU DEBUG

Now we know more or less what machine code is, it's about time we learned a bit more about how to handle it. What we shall do now is to write to a new program – HEXLD 3 – which will allow us to do ten things: (i) input machine code; (ii) insert machine code in between previous routines, but without overwriting anything; (iii) delete machine code, closing up the gap that it occupied; (iv) save and verify machine code; (v) list machine code; (vi) replace existing segments of machine code with new code; (vii) run machine code; (viii) copy blocks of machine code from one address to another; (ix) input text as data; and (x) convert decimal to hex and vice versa. The important point about this program is that the principle parts of it will themselves be in machine code, although all of the surrounding fabric will be in BASIC. To work it, once it's written, all you will need to do is enter one of the following:

RUN	To list your stored machine code.
RUN 100	To write machine code.
RUN 200	To insert machine code.
RUN 300	To delete machine code.
RUN 400	To save and verify machine code.
RUN 500	To cancel all previous machine code and start afresh.
RUN 600	To replace machine code by new code.
RUN 700	To run machine code.
RUN 800	To copy machine code from one address to another.
RUN 900	To input text as data.
PRINT FN H(A\$)	To convert hex to decimal.
PRINT FN H\$(A)	To convert decimal to hex.
PRINT FN P(A)	To perform a "double PEEK" in decimal.

## Mastering Machine Code on your ZX Spectrum

PRINT FN P\$(A\$)	To perform a “double PEEK” in hex.
RANDOMIZE FN Q(X,Y)	To perform a “double POKE” in decimal.
RANDOMIZE FN R(A\$,B\$)	To perform a “double POKE” in hex.

More to the point – you’ll need HEXLD 2 in order to load it. The addresses used in this chapter assume that you own a 48K machine. If this is not the case then for every address I use which begins with “F” you should use the corresponding address beginning with “7” instead of “F” – for instance, use 7FFE instead of FFFE.

Let’s create it one part at a time. Firstly, let’s reserve some space, so type:

```
16K: CLEAR 28671
48K: CLEAR 61439
```

For now, we are going to overwrite part of the user defined graphics area – but don’t worry – it’s not permanent. Addresses FFF6 to FFFF we shall use as our own “system variables”, so we’ll write our first subroutine immediately below this. This subroutine is called C\_PRINT (Character PRINT) and its job is to print any character but preceded by a space. The directive ORG FFED simply means “write this routine to address FFED”.

		ORG FFED
F5	C_PRINT	PUSH AF
F5		PUSH AF
3E20		LD A, “space”
D7		RST 10
F1		POP AF
D7		RST 10
F1		POP AF
C9		RET

The subroutine requires that the character to print is stored in the A register. The next subroutine we’ll need is a mechanism for printing the *value* of the A register in hexadecimal. This subroutine may be called from

one of two addresses: H\_PRINT (Hex PRINT) for simply printing the hex value of the A register, or SH\_PRINT (Space Hex PRINT) for preceding this printout by a space. Write in these routines:

		ORG FFCB	
F5	SH_PRINT	PUSH AF	Stack the byte to be printed.
3E20		LD A,"space"	
D7		RST 10	Print a space.
F1		POP AF	Retrieve the byte to be printed.
F5	H_PRINT	PUSH AF	Store A for later use.
E6F0		AND F0	This isolates the first digit.
1F		RRA	Move this first digit
IF		RRA	to its proper
IF		RRA	position within the A register.
1F		RRA	
C630		ADD A,"0"	Change to an ASCII character.
FE3A		CP3A	Is the digit between A and F?
3802		JR C,HP_H	
C607		ADD A,07	Change to correct symbol if so.
D7	HP_H	RST 10	Print this hex digit.
F1		POP AF	Retrieve the original value of A.
E60F		AND 0F	Isolate the second digit.
C630		ADD A,"0"	Change to an ASCII character.
FE3A		CP 3A	

*Mastering Machine Code on your ZX Spectrum*

3802		JR C,HP_L	
C607		ADD A,07	Change to correct hex symbol.
D7	HP_L	RST 10	Print this hex digit.
C9		RET	

Now that we have a routine for printing the A register in hex, how about a subroutine which prints *all* of the registers in hex? The next subroutine does just that. This is called REGS:

F5	REGS	ORG FF99	
		PUSH AF	Stack the flags register.
CDD0FF		CALL H_PRINT	Print the A register in hex.
78		LD A,B	
CDCBFF		CALL SH_PRINT	Print the B register in hex.
79		LD A,C	
CDCBFF		CALL SH_PRINT	Print the C register in hex.
7A		LD A,D	
CDCBFF		CALL SH_PRINT	Print the D register in hex.
7B		LD A,E	
CDCBFF		CALL SH_PRINT	Print the E register in hex.
7C		LD A,H	
CDCBFF		CALL SH_PRINT	Print the H register in hex.
7D		LD A,L	
CDCBFF		CALL SH_PRINT	Print the L register in hex.
F1		POP AF	Retrieve the original flags

3E53	LD A,“S”	register.
FCEDFF	CALL M,C_PRINT	Print “S” if the S flag is set.
3E5A	LD A,“Z”	
CCEDFF	CALL Z,C_PRINT	Print “Z” if the Z flag is set.
3E50	LD A,“P”	
ECEDFF	CALL PE,C_PRINT	Print “P” if the P/V flag is set.
3E43	LD A,“C”	
DC		
EDFF	CALL C,C_PRINT	Print “C” if the C flag is set.
C9	RET	

The next subroutine is a cunning one – it is designed to check whether or not the *interrupts* are *enabled*. What this means is quite simple – the Spectrum has two different states of operation, called interrupts *enabled* and interrupts *disabled*. You can change between one mode and another with the machine code instructions EI (Enable Interrupts) and DI (Disable Interrupts). With interrupts disabled the Spectrum will run slightly faster, but with the disadvantage that the keyboard will not be automatically scanned fifty times a second (don’t worry – I’ll explain about the keyboard in the next chapter). The interrupts must always be enabled before returning to BASIC. This checking routine merely relies on the fact that if interrupts are enabled then the system variable FRAMES will be incremented fifty times a second, but with interrupts disabled this will not be the case:

		ORG FF89	
2A785C	ICHECK	LD HL,(FRAMES)	HL: = lowest two bytes of (FRAMES).
01AA0A		LD BC,0AAA	This represents a



Mastering Machine Code on your ZX Spectrum

			1/50th second delay.
0B	I_LOOP	DEC BC	
78		LD A,B	
B1		OR C	
20FB		JR NZ,I_LOOP	Wait for 1/50th of a second.
3A785C		LD A,(FRAMES)	A: = lowest byte of (FRAMES).
BD		CP L	Compare with previous lowest byte.
C9		RET	The subroutine will return with the zero flag indicating the current state of the interrupts.

Building up our selection of subroutines slowly, the next one can be called from one of three points and its function is to print a four digit hexadecimal value. Let's go:

		ORG FF77
010000	U_ADDR	LD BC,0000
2AF8FF	PR_ADDR	LD HL,(ADDRESS)
7C	PR_HL	LD A,H
CDD0FF		CALL H_PRINT
7D		LD A,L
CDD0FF		CALL H_PRINT
3E20		LD A, "space"
D7		RST 10
C9		RET

You see this particular routine can be called from BASIC, using PRINT USR 65399 (or 32631 if you have 16K – remember you have to subtract

32768 from all the addresses I give). The zero printed, by the way, is the final value of BC (since USR was used). You can get rid of this by instead saying PRINT USR 65399; CHR\$ 8;"" (CHR\$ 8 is backspace).

Another useful trick, which we shall use in a moment, is the instruction RST 08 (which means CALL 0008) followed by the data byte FF. This will cause an immediate return to BASIC command mode irrespective of the stack and irrespective of the value of HL'. The next piece of machine code will use this trick. Unfortunately, there is one small problem which crops up if you use RST 08 – that is that the IY register has to contain the value 5C3A before you use it. If this is not the case then an almighty crash will befall you. (Try LDIY,1234/RST 08/ DEFB FF if you want – it's quite shattering!)

There are also some warnings on the use of RST 10. As with RST 08 the IY register must contain a value of 5C3A or else the system will crash. Furthermore, RST 10 will corrupt the values of the registers A, A', B', C', D', and E', and also the flags registers F and F', so these are things we have to watch out for. The next routine is what all of this so far has been leading up to...

This is what's called a *break point* routine, and is intended to be used as a debugging aid. It will print out all of the registers and flags, including the alternative ones, and also the registers IX, IY, and the current address. Once it's done all that it will then make a successful return to BASIC command mode. To use this routine simply write CALL BREAKPT into any machine code program at any point.

		ORG FF2A	
FDE5	BREAKPT	PUSH IY	
FD213A5C		LD IY,5C3A	This is so that RST
			10 will work.
DDE5		PUSH IX	
08		EX AF,AF'	

*Mastering Machine Code on your ZX Spectrum*

F5	PUSH AF	Because RST 10 corrupts A' and F.
08	EX AF,AF'	
D9	EXX	
C5	PUSH BC	Preserve BC'.
D5	PUSH DE	Preserve DE'.
E5	PUSH HL	Preserve HL'.
D9	EXX	
F5	PUSH AF	Because RST 10 corrupts F.
3E78	LD A,78	
328F5C	LD (ATTR_T),A	Ensure that the printing will be in black on bright white.
AF	XOR A	
323C5C	LD (TVFLAG),A	Ensure that printing will be to upper part of screen.
3E0D	LD A,"enter"	
D7	RST 10	Print a carriage return.
F1	POP AF	Restore flags.
CD99FF	CALL REGS	Print all registers.
3E0D	LD A,"enter"	
D7	RST 10	Print a carriage return.
E1	POP HL	
D1	POP DE	
C1	POP BC	
F1	POP AF	
CD99FF	CALL REGS	Print all alternative registers.
3E0D	LD A,"enter"	

D7	RST 10	Print a carriage return.
E1 CD7DFF	POP HL CALL PR_HL	HL: = original IX. Print the IX register.
E1 CD7DFF	POP HL CALL PR_HL	HL: = original IY. Print the IY register.
E1	POP HL	HL: = return address from this subroutine.
2B 2B 2B CD7DFF	DEC HL DEC HL DEC HL CALL PR_HL	Print address from which subroutine was called.
CD89FF	CALL I_CHECK	Set zero flag if interrupts disabled.
3E44	LD A, "D"	"D" stands for Disabled.
2801 3C	JR Z,BC_I INC A	Change A to "E" for Enabled.
D7 BC_I	RST 10	Print "D" or "E" as required.
DD213A5CEXIT	LD IY,5C3A	This is to ensure that RST 08 will work if the routine is called from the label EXIT.
FB	EI	Enable interrupts for return to BASIC.

CF	RST 08	
FF	DEFB FF	Return to BASIC command mode.

You can now (believe it or not) actually test this routine by typing RUN 700 and inputting FF2A as the starting address. FF2A is the address of the instruction labelled BREAKPT. If your routine works you should find the following printed on your screen (the DE value may be variable):

```
2A FF 2A ?? ?? 2D 2B Z P
00 17 0F 36 9B 27 58 Z P
03D4 5C3A 2D28 E
```

The first two rows of numbers represent the current values of all the registers in the order A, B, C, D, E, H and L (then the flags) and A', B', C', D', E', H' and L' (then the alternative flags). The symbols Z and P indicate that only the Z and P flags are currently set. The bottom row contains the values of IX, IY and the address (this time in ROM) from which the subroutine was called. Note that the initial value of DE may be variable and that the initial value of BC is the USR call address, so that if you are using 16K rather than 48 then this value will in fact be 7F2A not FF2A.

Now, let's do the first real part of HEXLD 3 – the LIST part. This is a subroutine which will list machine code in hex. It makes use of three of our very own variables, so I shall explain these now:

- BEGIN (stored at FFF6) – The first address which we are allowed to list.
- ADDRESS (stored at FFF8) – The next address to list.
- LIMIT (stored at FFFE) – The last address to list, plus one.

This is the first part of HEXLD 3 that we can actually make *sensible* use of straight away. We'll do it in three parts: (i) the machine code, (ii) the BASIC, and (iii) the variables above. Type in the following machine code:

AF	H_LIST	ORG FEFO	
323C5C		XOR A	
		LD (TVFLAG),A	Print is to upper part of screen.
2AF8FF		LD HL,(ADDRESS)	HL: = address we want to list from.
ED5BF6FF		LD DE,(BEGIN)	DE: = first allowable address.
A7		AND A	
ED52		SBC HL,DE	
19		ADD HL,DE	
3003		JR NC,HL_A	
2AF6FF		LD HL,(BEGIN)	HL: = first address to list from.
ED5BFEFF	HL_A	LD DE,(LIMIT)	DE: = first illegal address.
22F8FF	HL_LOOP	LD (ADDRESS),HL	Store current address.
A7		AND A	
ED52		SBC HL,DE	
19		ADD HL,DE	Test whether address allowed.
305F		JR NC,EXIT	Exit if finished.
3E0D		LD A, "enter"	
D7		RST 10	Print a carriage return.
CD7DFF		CALL PR_HL	Print the current address.
7E		LD A,(HL)	A: = byte at this address.
CDD0FF		CALL H_PRINT	Print this byte.
7E		LD A,(HL)	Restore A: = this byte.

*Mastering Machine Code on your ZX Spectrum*

FE20		CP20	If this character is not a control code...
3807		JR C,HL_NEXT	
FEA5		CP A5	and if the character is not a keyword...
3003		JR NC,HL_NEXT	
CDEDF		CALL C_PRINT	then print this character.
23	HL_NEXT	INC HL	Point to next address.
18DE		JR HL_LOOP	

And the BASIC:

```
1000 PRINT " LIST ";
1010 GO SUB 8000
1020 RANDOMIZE USR 65264
8000 LET X = 65528

8030 INPUT "ADDRESS ";A$
8040 PRINT "ADDRESS ";A$
8050 LET Y = 0
8051 FOR I = 1 TO
8052 LET Z = CODE A$(I) - 48
8053 IF Z > 9 THEN LET Z = Z - 7
8054 LET Y = 16*Y + Z
8055 NEXT I
8060 POKE X,Y - 256*INT (Y/256)
8070 POKE X + 1,INT (Y/256)
8080 RETURN
```

*This is FEF0 in decimal (H LIST).  
This is FFF8 in decimal  
(ADDRESS).  
The line numbering is deliberate.*

*Watch the line numbering.*

We still need to assign the variables BEGIN and LIMIT, however. To do this type RUN and input (for the "WRITE TO " prompt) "FFF6" and then input "F0FE" (this is FEF0 - the address of H\_LIST - with the bytes

switched round), “0000”, “0000”, “0000”, “F6FF” (which is the address FFF6 with the bytes switched around). Break out of the program and we are ready.

To see H\_LIST actually working, type RUN 1000. You can input any address you like, but you’ll find you can’t list before FEF0 or after FFF5 – you can now use this to check the entire program if you like.

A couple of subroutines now, which are always a good thing. The first is called SET\_UP and performs two tasks. Firstly, it assigns HL with the minimum of (ADDRESS) and (BEGIN), and re-assigns (BEGIN) with this minimum if required. This is all quite straightforward. Its second job, however, requires a bit of inside information about the workings of the ROM. Its task is to load BC with the number of bytes in the string A\$ and to point HL to the byte immediately before the text of this string. To store BASIC variables, the ROM sets aside a little bit of RAM space which it keeps track of with the system variable VARS. If A\$ is the only assigned variable around, then VARS will point to a byte which will in fact be the byte “A” (hex 41). (It would have been “B” for B\$ or “T” for T\$, and so on.) Straight after this comes the length of the string and then the text of the string itself. Bearing that in mind, here’s the subroutine – note that it can also be called from the label LEN\_A\$.

		ORG FED1	
AF	SET_UP	XOR A	
323C5C		LD(TVFLAG),A	Print to upper part of screen.
3E0D		LD A, “enter”	
D7		RST 10	Print a new line.
2AF8FF		LD HL,(ADDRESS)	
ED5BF6FF		LD DE,(BEGIN)	
ED52		SBC HL,DE	(Note that RST 10 resets the carry flag)
19		ADD HL,DE	



*Mastering Machine Code on your ZX Spectrum*

3003		JR NC,SU_BEGIN	
22F6FF		LD(BEGIN),HL	(BEGIN): = minimum of BEGIN), (ADDRESS). Point HL to the BASIC variable A\$.
2A4B5C	LEN_A\$	LD HL,(VARS)	
23		INC HL	
4E		LD C,(HL)	
23		INC HL	
46		LD B,(HL)	BC: = length of string A\$.
C9		RET	

The next subroutine re-assigns (LIMIT) with the maximum of (ADDRESS) and (LIMIT). In other words, if ADDRESS points to an address higher in memory than is pointed to by LIMIT then LIMIT will be re-assigned. All aboard:

		ORG FEC1	
2AF8FF	NEW_LIM	LD HL,(ADDRESS)	
ED5BFEFF	NL_2	LD DE,(LIMIT)	
A7		AND A	
ED52		SBC HL,DE	
19		ADD HL,DE	
D8		RET C	
22FEFF		LD (LIMIT),HL	
C9		RET	

The next part of HEXLD 3 now, which is the WRITE part. Just in case you think I've been leaping around all over the place a little too quickly, I'll stop speeding for this one. I'll list the machine code all in one go, but then I'll stop to explain it all so that you can refer back to it and work out exactly what each little bit is doing (no pun intended). OK – ready for it?

CDD1 FE	WRITE	ORG FE87 CALL SET_UP	
EB		EX DE,HL	DE: = points to first byte before the text of the string A\$,HL: = address to write to.
CB38		SRLB	Divide BC by two to give the number of bytes needed to write.
CB19 CA70FF CD7DFF		RR C JP Z,EXIT CALL PR_HL	Exit if empty string. Print address to write to.
13	W_LOOP	INC DE	Point DE to first hex digit of next byte.
1A		LD A,(DE)	A: = this character ("0" to "9" or "A" to "F").
D7 1A		RST 10 LD A,(DE)	Print first hex digit. Restore A (RST 10 will erase it).
FE40 3804		CP 40 JR C,WR1	Jump if digit "0" to "9".
E6DF		AND DF	Convert to capital letter.
D607		SUB 07	Standardise character code.
87	WR1	ADD A,A	

*Mastering Machine Code on your ZX Spectrum*

87		ADD A,A	Shift A one hex digit to the left (so that 30 ("0") becomes 00, 35 ("5") becomes 50, and so on).
87		ADD A,A	
87		ADD A,A	
E5		PUSH HL	Store HL temporarily.
67		LD H,A	H: = first hex digit times sixteen.
13		INC DE	Point DE to second hex digit.
1A		LD A,(DE)	A: = second hex digit.
D7		RST 10	Print second hex digit.
1A		LD A,(DE)	
FE40		CP 40	
3804		JR C,WR2	Jump if digit "0" to "9".
E6DF		AND DF	Convert to capital letter.
D607		SUB 07	Standardise character code.
E60F	WR2	AND 0F	Consider second hex digit only.
B4		OR H	Combine with first hex digit.
E1		POP HL	Restore HL.
77		LD (HL),A	Write the byte to the required address.
23		INC HL	

22F8FF	LD (ADDRESS),HL	Store next address at which to write.
0B	DEC BC	
78	LD A,B	
81	OR C	
20D4	JR NZ,W_LOOP	

(the machine code immediately after this is the NEW\_LIM subroutine)

I'd like to explain some terminology here if I may. HEXLD 3 is a machine code program whose job it is to create and run *other* machine code programs, thus, much confusion may arise in the use of the word "program" – in order to get round this problem I refer to HEXLD 3 itself as the "object program", and the program which it is editing the "subject program". It is in a limited number of cases possible for HEXLD 3 to examine itself, in which case it becomes both the object and the subject program; however, I would still like to keep these distinctions separate, and in the most abstract sense HEXLD 3 should always be regarded as working on a separate set of data.

If the string is empty then we exit – we not only return, but we return to BASIC's *command mode*. Otherwise, the address at which to write is printed and we enter the main loop.

Each two bytes of the string correspond to one byte of hex. For instance, if the string contained "2A" then this would imply the byte 2A – however, the string itself contains the character codes 32 followed by 41. Furthermore, it is also possible for the string to contain "2a" (ie with a lower case a) in which case the character codes would be 32 followed by 61 – both of these cases should generate the byte 2A. To do this, a quick conversion takes place by which any symbol whose character code is 40 or more (ie "A" to "F" or "a" to "f") are converted to upper case (the instruction AND DF does this can you work out how? – the instruction OR 20 would have converted all letters to lower case if we'd have wanted it that way) and then seven is subtracted. This effectively "closes up the gap" between "9" and "A", so

### *Mastering Machine Code on your ZX Spectrum*

that “9” still has code 39, but “A” now has code 3A (as does “a”). It should be fairly easy for you now to follow the program through to the LD (HL),A instruction just after the label WR2 where the byte required is now calculated and loaded.

After the next address is stored a new check is made. The variable LIMIT stores the address of the first byte beyond the end of the subject program, so if we write to an address beyond or equal to (LIMIT) then this limit must be altered.

One last point to note – the instruction DEC BC does not alter the zero flag, so in order to check whether or not we are finished we have to use LD A,B/OR C. Let’s move on now to the BASIC.

Use RUN 1000 to check that your listing of the WRITE routine is correct. Type RUN to assign the variable BEGIN – you should input “FFF6” then “00F0” to do this (then break out).

Now – when everything’s all hunky dory amend the BASIC as follows:

```
10 PRINT "List ";
20 GO SUB 8000
30 RANDOMIZE USR 65264
100 PRINT "Write ";
110 GO SUB 8000
120 CLEAR
130 INPUT LINE A$
140 RANDOMIZE USR 65159
150 GO TO 130
```

You should also delete all the other lines between 1 and 199, and also between 1000 and 7999. Some of the lines in subroutine 8000 should be changed for aesthetic reasons.

```
8030 INPUT "address "; LINE A$
```

```
8040 PRINT " ";A$
```

Now SAVE this program before you attempt to RUN it. To do this delete lines 400 and 410, and change 440 to SAVE "M/CODE" CODE 61440, 4196 and then just RUN 400. From now on, RUN should *list* machine code, and RUN 100 should *write* it. Did you notice, by the way, the CLEAR instruction in the BASIC? This is to ensure that A\$ is the first thing in the variables' area. Without CLEAR the machine code won't work.

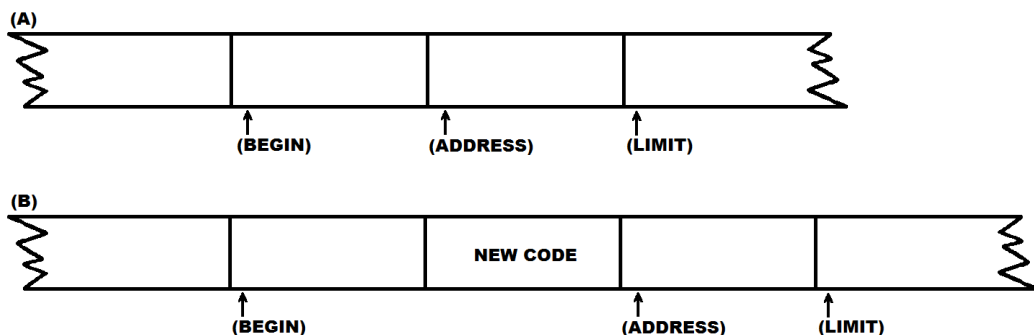


Figure 10.1

The next piece of machine code forms a rather clever task. Take a look at figure 10.1. It shows the subject program before and after the routine is called. The routine is named INSERT, and I hope you can see why. The ADDRESS variable acts as a kind of "cursor", so that what you type in gets *inserted* at the cursor position and the cursor itself moved to the end of the newly inserted bytes. No machine code is actually overwritten by this procedure as all of the code to the right of the cursor simply gets shifted up in memory. Here's the machine code to perform that task. You should use RUN 100 to enter this. Note that pressing "enter" alone will breakout.

```
CDE8FE          ORG FE63
CB38           CALL LEN_A$
                SRL B
```

*Mastering Machine Code on your ZX Spectrum*

CB19	RR C	BC: = number of bytes to insert.
CA70FF	JP Z, EXIT	Exit if the empty string given.
C5	PUSH BC	Stack number of bytes to insert.
2AFEFF	LD HL,(LIMIT)	HL: = points to first byte beyond subject program.
ED5BF8FF	LD DE,(ADDRESS)	DE: = address at which to insert.
A7	AND A	
ED52	SBC HL,DE	HL:= number of bytes of subject program which need to be moved.
23	INC HL	Add one to prevent crashing if number of bytes to move is zero.
44	LD B,H	
4D	LD C,L	BC: = the number of bytes to move.
E1	POP HL	HL: = number of bytes to insert.
ED5BFEFF	LD DE,(LIMIT)	DE: = old value of LIMIT.
19	ADD HL,DE	HL: = new value of LIMIT.
22FEFF	LD (LIMIT),HL	Store this value.
EB	EX DE,HL	DE: = new LIMIT.
EDB8	LDDR	HL: = old LIMIT. Move all required bytes.

Notice there is no return instruction at the end of this routine, and also that the routine above hasn't changed the value of ADDRESS as promised. This is because the very next machine code instruction is the start of the WRITE routine which will overwrite the now-rubbish bytes in the newly inserted space, and will then re-assign ADDRESS and return. We need some BASIC now – just type this in:

```
500 PRINT "BEGIN ADDRESS ";
510 INPUT A$: PRINT A$
520 LET X = 0
521 FOR I = 1 TO 4
522 LET Y = CODE A$(I) - 48:IF Y>9 THEN LET Y = Y - 7
523 LET X= 16*X + Y
524 NEXT I
530 LET Z = INT (X/256)
540 LET Y = X - 256*Z
550 POKE 65526,Y
560 POKE 65527,Z
570 POKE 65534,Y
580 POKE 65535,Z
590 STOP
```

*This is the address of BEGIN in decimal.*

*This is the address of LIMIT in decimal.*

You see, now that we hopefully have the ability to insert text, it is very important to ensure that HEXLD 3 is never pointing to itself when we test this (otherwise things will get really mucked up!), so the routine we've just typed in at line 500 has the job of moving the BEGIN and LIMIT pointers away from HEXLD 3 and onto somewhere safe. Now let's go back to the INSERT routine. Delete all BASIC lines between 200 and 299 and enter the following:

```
200 PRINT "Insert ";
210 GO SUB 8000
220 CLEAR
230 INPUT LINE A$
```



## Mastering Machine Code on your ZX Spectrum

```
240 RANDOMIZE USR 65123
250 GO TO 230
```

The only problem now is that the SAVE routine won't work properly (because you've just overwritten the subroutine at 200), so I'm afraid you're just going to have to alter it. To do so delete all lines between 400 and 499 and start again:

```
400 SAVE "Hexld 3" LINE 450
410 SAVE "Hexld 3 mc" CODE 63488,2048 (This will save all code from
F800 to FFFF)
420 SAVE " " CODE FN P(65526),FN P(65534) – FN P(65526) + 1
430 VERIFY "" : VERIFY "" CODE: VERIFY "" CODE
440 STOP
450 CLEAR 61439 (This frees memory from F000 upwards)
460 PAPER 7: INK 0
470 LOAD "" CODE: LOAD "" CODE
480 STOP
```

And just for completeness...

```
9000 DEF FN P(X) = PEEK X + 256*PEEK (X + 1)
```

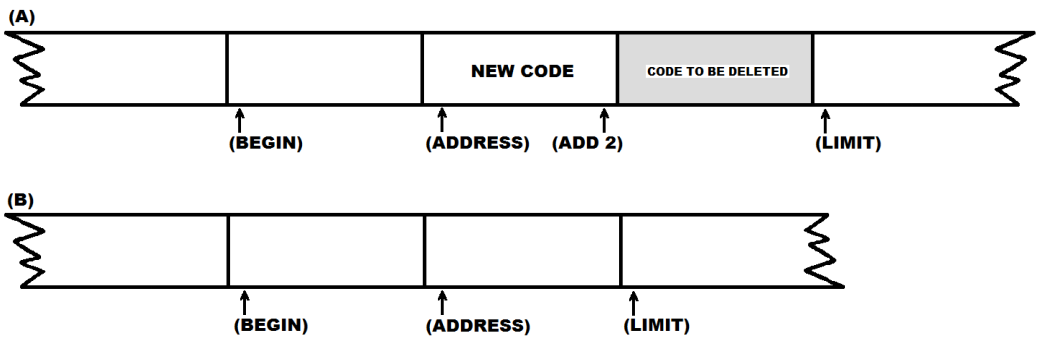
Now type RUN 400 to SAVE the program. Right, now we're all set for a little demonstration. Type RUN 500 to move the pointers away from Hexld 3 and input "F000". You should then see the message 9 STOP statement, 590:1 appear at the bottom of the screen. Now type RUN 100 to enter new code and again input the address F000. "You'll notice that the screen clears entirely – this is because of the CLEAR instruction in line 120. Now input the sequence 00/01/02/03/04/05/06/07/08/09/0A/0B/0C/0D/0E/0F/ (where "/" counts as *enter*). Press *enter* alone in order to break out. Now, if you type RUN and input F000 you should see this machine code listed out. Now for the exciting bit. Type RUN 200 and this time input the address F004. Next type in the following bytes: 20/3432/20 (again where "/" counts as *enter*). Break out by typing *enter* alone. To see what's actually

happened you must now type RUN and input F000 and briefly imagine a fanfare of trumpets in your head.

As you can see, the four bytes input in the INSERT routine have been inserted without overwriting the previous code.

And now for something completely different...

Take a look now at figure 10.2, which shows the requirements for a DELETE procedure. As you can see a new variable is brought into play here – ADD2, which is in this case the last address to delete. All machine code from (ADDRESS) to (ADD2) should be erased and overwritten by the rest of the code once it's shifted down in memory.



**Figure 10.2**

The machine code is this:

2AFEFF	DELETE	ORG FE49 LDHL,(LIMIT)	HL: = first byte beyond subject program. DE: = last byte to delete.
ED5BFAFF		LD DE,(ADD2)	

*Mastering Machine Code on your ZX Spectrum*

13	INC DE	DE: = first byte to move.
A7 ED52	AND A SBC HL,DE	HL: = number of bytes to move.
44 4D	LD B,H LD C,L	BC: = number of bytes to move.
03	INC BC	Add one to prevent crashing if an attempt is made to delete the last byte.
2AF8FF	LD HL,(ADDRESS)	HL: = address to move data to.
EB	EX DE,HL	DE: = address to move data to, HL: = address of first byte to move.
EDB0	LDIR	Move all required bytes.
1B	DEC DE	DE: = points to first byte beyond amended subject program.
ED53FEFF C9 RET	LD (LIMIT),DE	Store this new limit.

And the BASIC is:

```
300 PRINT " Delete ";  
310 GO SUB 330  
320 STOP  
330 GO SUB 8000  
340 PRINT "five spaces To ";  
350 GO SUB 8020
```

*Yes, there is a space before the D.*

```

360 RANDOMIZE USR 65097
370 RETURN
8010 GO TO 8030
8020 LET X = 65530

```

You *can* use RUN 100 to write the new machine code in, but you must type RUN 500 and input “F000” in order to move the pointers well away once it’s done. A bit of a demonstration now, just to prove it works. Type RUN 500 then input “F000”. Type RUN 100 and then F000/00/01 /02/03/04/05/06/07/08/09/0A/ 0B/0C/0D/0E/0F/ (where “/” represents *enter*). Break out by typing enter alone. RUN 300 for the magic and input F005 followed by F008. Dare I say “abracadabra”? RUN followed by F000 should prove the point.

Did you notice, by the way, how LDDR was used in the insert routine, but LDIR in the DELETE? When moving blocks of data up or down in memory, it is important to start at the right end. For instance, suppose we were moving a block of nine bytes from F001 to F000. If we let HL = F001 and DE = F000, we can use LDIR with no problems. However, had we started from the other end, with HL holding F009 and DE F008 then F008 (which has just been overwritten) to F007, then from F007 (which again has just been overwritten) to F006, and so on, the whole routine would end up slightly amiss (!). Let’s now extend HEXLD 3 without adding any more machine code at all. This is the REPLACE routine:

```

600 PRINT “Replace ”;
610 GO SUB 330
620 GO TO 220

```

Cunning isn’t it? REPLACE works by doing DELETE followed by INSERT. On running you are asked for two addresses: the first and the last address you want to replace. The stuff you input will then be inserted to this point automatically. Try it for yourself. I’ll leave you to invent your own demonstration this time though – it’s not hard.

*Mastering Machine Code on your ZX Spectrum*

Now to get rid of all the junk at line 500. Write the following machine code in:

```
                ORG FE3E
2AF8FF        BEGINMC  LD HL,(ADDRESS)
22F6FF                LD (BEGIN),HL
2B                DEC HL
22FEFF                LD (LIMIT),HL
C9                RET
```

Now delete all of the BASIC lines between 500 and 599 and replace them by the following:

```
500 PRINT "Begin ";
510 GO SUB 8000
520 RANDOMIZE USR 65086
530 GO TO 120
```

The routine at 500 now has the following effect: *to cancel any former subject program and start afresh*. Note that the routine will automatically expect you to input new code now.

Now, as you can see, we've completed most of the things that we originally wanted HEXLD 3 to be able to do. Let's now sort out the routine at line 700. First the machine code, but don't try to comprehend it all in one go because this one is quite complicated.

```
E5            H_RUN        ORG FE10
                PUSH HL                Stack the initial
                                        value of HL in case
                                        the subject
                                        program relies on
                                        it.

211CFE                LD HL,R_CH
```

E3		EX (SP),HL	Restore HL and stack the “artificial return address”
ED4BF8FF		LD BC,(ADDRESS)	R_CH. BC: = address of USR call.
79		LD A,C	A: = address of USR call (low part) just in case the subject program relies on it.
C5		PUSH BC	Stack address of USR call.
C9		RET	Call the USR subroutine.
F5	R_CH	PUSH AF	Stack the flags register.
D9		EXX	Fetch the alternative register set.
D5		PUSH DE	Stack the final value of DE’.
115827		LD DE,2758	
A7		AND A	
ED52		SBC HL,DE	Set the zero flag if HL’ equals 2758.
19		ADD HL,DE	Restore HL’.
D1		POP DE	Restore DE’.
D9		EXX	Restore all registers.
200B		JR NZ,R_EXIT	Jump if HL’ < >2758.
C5		PUSH BC	
E5		PUSH HL	

*Mastering Machine Code on your ZX Spectrum*

CD89FF		CALL I_CHECK	Check whether or not interrupts are enabled.
E1		POP HL	
C1		POP BC	
2802		JR Z,R_EXIT	
F1		POP AF	Balance the stack.
C9		RET	Return to BASIC.
F1	R_EXIT	POP AF	Restore A and the flags.
E3		EX (SP),HL	Set the topmost item on the stack to 0003 so that third item on row three of the BREAKPT output will read 0000.
210300		LD HL,0003	
E3		EX (SP),HL	
C32AFF		JP BREAKPT	

And the BASIC to accompany this routine is:

```
700 PRINT "Run ";
710 GO SUB 8000
720 RANDOMIZE USR 65040
730 STOP
```

You should delete all other BASIC lines between 700 and 799. Just in case (as is most probable) you didn't follow that machine code, I'll make a brief attempt to explain it.

The first part of the program (from H\_RUN to R\_CH minus one) ensures that the return address from the subject program is the label R\_CH (RUN CHECK), but it does so in such a way that the registers contain exactly the

same initial values as had the subject program been called directly using USR. Notice the way in which the sequence PUSH BC/RET is used to simulate “JP (BC)”. On return from the subject program, control will now be at address R\_CH where two separate checks are made on the state of affairs: Firstly, that HL’ contains a value of 2758 and secondly, that the interrupts are enabled. If either of these checks fail then the BREAK OUT routine is used which will print out the current state of all the registers and flags, including the alternative ones, the IX and IY registers, and the reference number 0000, which shows that the *run check* has failed – a successful return to BASIC command mode will then be made.

The reason that these checks are necessary is quite simple – if either of the two conditions fails and an attempt is made to return to BASIC then *crash!!* You can prove this for yourself if you don’t believe me by running the programs EXX/RET and DI/RET.

Now for an interesting part – change (using EDIT) two of the lines of BASIC as follows:

```
130 INPUT USR 65399;CHR$ 8;LINE A$
230 INPUT USR 65399;CHR$ 8;LINE A$
```

This produces an interesting effect when run. Type RUN 500 and then F000 and input some code. Notice the extra information at the bottom of the screen. This comes about because USR 65399 is the address of the routine, U\_ADDR, which loads BC with zero and then prints (ADDRESS) in hex followed by a space. Because we haven’t used XOR AF/LD (TVFLAG),A, this text is printed at the *bottom* of the screen, not the top. As a side effect of being in an INPUT statement, the final value of BC is also printed (zero) and the CHR\$ 8 (backspace) is there to “hide” it from view. The COPY routine comes next. (Don’t worry – we’re nearly finished.) This is it:



Mastering Machine Code on your ZX Spectrum

ED5BF8FF	H_COPY	ORG FDD5 LD DE,(ADDRESS)	DE: = first byte to move.
2AFAFF		LD HL,(ADD2)	HL: = last byte to move.
23		INC HL	HL: = first byte beyond data to move.
A7		AND A	
ED52		SBC HL,DE	HL: = number of bytes to move.
44		LD B,H	
4D		LD C,L	BC: = number of bytes to move.
2AFCFF		LD HL,(ADD3)	HL: = address to which data is to be moved.
A7		AND A	
ED52		SBC HL,DE	
19		ADD HL,DE	
3006		JR NC,HC_1	Jump if data is to be moved <i>upwards</i> in memory.
EB		EX DE,HL	DE: = address to which data is to be moved. HL: = address of first byte to move.
EDB0		LDIR	Move block of data as required.
D5		PUSH DE	
1808		JR HC_2	
09	HC_1	ADD HL,BC	
E5		PUSH HL	

2B		DEC HL	HL: = address of last byte of proposed new position of data.
EB		EX DE,HL	DE: = address of intended last byte of data. HL: = address of first byte to move.
09		ADD HL,BC	
2B		DEC HL	HL: = address of last byte of data.
EDB8		LDDR	Move block of data as required.
2AFCFF	HC_2	LD HL,(ADD3)	HL: = address of start of data.
ED5BF6FF		LD DE,(BEGIN)	DE: = first byte of subject program.
A7		AND A	
ED52		SBC HL,DE	
19		ADD HL,DE	
3003		JR NC,HC_3	
22F6FF		LD (BEGIN),HL	Adjust (BEGIN) if required.
E1	HC_3	POP HL	
CDC4FE		CALL NL_2	Adjust (LIMIT) if required.
C370FF		JP EXIT	

```

800 PRINT " Copy ";
810 GO SUB 8000
820 PRINT "Until ";
830 GO SUB 8020
840 PRINT "three spaces To ";

```

*Mastering Machine Code on your ZX Spectrum*

```
850 LET X = 65532
860 GO SUB 8030
870 RANDOMIZE USR 64981
```

And finally the TEXT section.

CDD1FE	TEXT	ORG FDB9 CALL SET_UP	BC: = length of string, A\$, DE: = address at which to write string, HL: = address immediately before text.
78		LD A,B	
B1		OR C	
CA70FF		JP Z, EXIT	Return if string empty.
E5		PUSH HL	
CD7AFF		CALL PR_ADDR	Print current address.
E1		POP HL	
23		INC HL	Point HL to text of string.
7E	T_LOOP	LD A,(HL)	A: = next character of text.
D7		RST 10	Print this character.
EDA0		LDI	Load byte into subject program.
EAC7FD		JP PE,T_LOOP	Repeat till finished.
ED53F8FF		LD (ADDRESS),DE	Store final address.
C3C1FE		JP NEW_LIM	Adjust (LIMIT) if required.

Notice the rather odd loop in the process – LDI followed by JP PE. LDI can be imagined as just one step of LDIR, or alternatively as LD (DE),(HL) – it also decrements BC. The P flag acts rather oddly in this case. Glancing briefly at Appendix Four under LDI we see that the P flag is reset if and only if BC finally reaches zero, so we want an instruction meaning “jump if P is set” – JP PE does just that. Note that there is no such instruction as JR PE so we are forced to use an absolute address. The LDI was used instead of LDIR because of the extra step (RST 10) in the loop.

The BASIC to accompany this machine code is this:

```

900 PRINT "Text";
910 GO SUB 8000
920 CLEAR
930 INPUT USR 65399;CHR$ 8;LINE A$
940 RANDOMIZE USR 64953
950 GO TO 930

```

A short subroutine now, just for a change of scenery, before we go on to the user-defined functions. This is a subroutine which fetches the first argument of a user-defined function. If you've forgotten how the records in the user-defined functions area are laid out then you can always recap by re-reading the end of the last chapter.

2A0B5C	FN_ARG	ORG FDAE LD HL,(DEFADD)	Point HL to user-defined function.
23		INC HL	
23		INC HL	
23		INC HL	
23		INC HL	
5E		LD E,(HL)	
23		INC HL	

*Mastering Machine Code on your ZX Spectrum*

56		LD D,(HL)	DE: = first argument.
C9		RET	
<p>The first function we shall define using this subroutine shall be called FN_H – its purpose is to convert hex into decimal:</p>			
CDAEFD	FN_H	ORG FD8F CALL FN_ARG	DE: = points to string.
23		INC HL	
46		LD B,(HL)	B: = length of string.
210000		LD HL,0000	HL: = “sum so far” (= zero).
1A	NH_LOOP	LD A,(DE)	A: = next ASCII character.
13		INC DE	DE: = points to next character.
FE40		CP 40	
3804		JR C,FNH-CHR	Jump if character “0” to “9”.
E6DF		AND DF	Convert to upper case letter if needed.
D607		SUB 07	Standardise code.
D630	FNH_CHR	SUB 30	A now contains the required hex digit.
29		ADD HL,HL	
29		ADD HL,HL	
29		ADD HL,HL	
29		ADD HL,HL	Multiply HL by sixteen.
B5		OR L	
6F		LD L,A	Add in new hex

10EC	DJ NZ FNH_LOOP	digit.
44	LD B,H	Repeat till finished.
4D	LD C,L	BC: = calculated value.
C9	RET	

And the BASIC simply is:

```
9040 DEF FN H(A$) = USR 64911
```

If you couldn't quite work out what was going on there – all that playing around in the user-defined functions area – I suggest you go back to the end of the last chapter to recap. The reverse process is a little more tricky since it is not possible to return a string value with USR. We have to do this in two steps and we'll call the first step FN K\$.

CDAEFD	FN_K\$	ORG FD69 CALL FN_ARG	DE: = first argument of function.
010700		LD BC,0007	
09		ADD HL,BC	
46		LD B,(HL)	B: = second argument of function.
04		INC B	
7B		LD A,E	
05		DEC B	
280B		JR Z,FNK_2	
05		DEC B	
2804		JR Z,FNK_1	
7A		LD A,D	
05		DEC B	
2804		JR Z,FNK_2	

*Mastering Machine Code on your ZX Spectrum*

1F	FNK_1	RRA	
1F		RRA	
1F		RRA	
1F		RRA	
E60F	FNK_2	AND 0F	A: = digit to return.
FE0A		CP 0A	
3802		JR C, FNK_3	Jump if 0 to 9.
C607		ADD A, 07	
C630	FNK_3	ADD A, 30	Convert to ASCII character code.
4F		LD C, A	
0600		LD B, 00	BC: = character code.
C9		RET	

And for some useful functions:

```
9010 DEF FN P$(A$) = FN H$(FN P(FN H(A$)))
```

```
9050 DEF FN H$(X) = FN K$(X,3) + FN K$(X,2) + FN K$(X,1) + FN  
K$(X,0)
```

```
9060 DEF FN K$(X,Y) = CHR$ USR 64873
```

And finally, those double POKEs:

CDAEFD	FN_Q	ORG FD5A CALL FN_ARG	DE: = POKE address.
010700		LD BC, 0007	
09		ADD HL, BC	
4E		LD C, (HL)	
23		INC HL	
46		LD B, (HL)	BC: = POKE argument.
EB		EX DE, HL	HL: = POKE address.

71	LD (HL),C	
23	INC HL	
70	LD (HL),B	POKE two byte number.
C9	RET	

```
9020 DEF FN Q(X,Y) =USR 64858
9030 DEF FN R(A$,B$) = FN Q(FN H(A$),FN H(B$))
```

And now for some tidying up!

If you try typing `REM graphics A graphics B graphics C... graphics U` then you'll find you get a nasty surprise – all of the graphics characters have been corrupted. (In fact they've been overwritten by the machine code of HEXLD 3.) We can cure this, and at the same time give a nice demonstration of some of the functions of our program in the following way:

```
RUN 800
Copy address 3E08
Until address 3EAF

To address FCB2
RANDOMIZE FN R("5C7B",
"FCB2")
```

These bytes are the pixel patterns of characters capital A to capital U.  
This is spare RAM.  
This is a double POKE, which loads the hex number FCB2 (the address above) into addresses 5C7B and 5C7C (the system variable UDG – User-Defined Graphics address).

Now, if you type `REM graphic A graphic B graphic C... graphic U` you should, in fact, get the letters capital A to capital U. You can now use the user-defined graphics in exactly the same way that you always have, remembering, of course, that if you do any direct POKEing (ie instead of using `USR "A"`, etc) then the graphics start at FCB2.



I'd like to do a little cosmetic surgery on the program now. I don't want to add any more bits and pieces to it – just to make its general appearance a little prettier. We'll start by improving the subroutine at 8000. Delete all lines from 8000 to 8999 and in its place, type the following:

```
8000 GO SUB 8060
8010 RANDOMIZE FN R("FFF8",A$)
8020 RETURN
8030 GO SUB 8060
8040 RANDOMIZE FN R("FFFA",A$)
8050 RETURN
8060 INPUT "address "; LINE A$
8070 IF LEN A$ < > 4 THEN CLEAR: GO TO 1 E4
8080 PRINT "address ";A$
8090 RETURN
```

Improvements are (i) the use of FN R to double-POKE either ADDRESS or ADD2; (ii) the check on the length of A\$ - note the procedure if the check fails – CLEAR will empty the GO SUB stack (obviously quite important since at this point we are nested at least two GO SUB loops down) and also clears the screen, and GO TO 1 E4 (which means GO TO 10000) is just a way of going back to command mode (STOP would have done exactly the same job, but I find a report code zero a little more aesthetically pleasing than a report code 9); and (iii) no variables other than A\$ are assigned in the subroutine.

Because subroutine 8000 now no longer assigns anything but A\$, and also in order to make the printouts nicer *delete* lines 120, 220 and 920 (all of which formerly said CLEAR).

Because the routine for assigning ADD2 has moved from 8020 to 8030 we need to change:

```
350 GO SUB 8030
830 GO SUB 8030
```

```
850 GO SUB 8060
860 RANDOMIZE FN R("FFFC",A$)
```

And for saving grace:

```
410 SAVE "Hexld 3 mc"CODE64690,846
```

*Since there's no need to oversave now that we know how long the program is.*

```
440 GO TO 1 E4
```

*Because it's nicer than STOP.*

```
450 CLEAR 64689
```

*Because there's no need to set aside more memory than is needed.*

```
460 LET P = 7:LET I= 0
```

*Set paper and ink colours.*

```
461 INK I
```

```
462 PAPER P
```

```
463 BORDER P
```

```
464 POKE 23624,I+8*P
```

```
465 FLASH 0
```

```
466 BRIGHT 0
```

```
467 OVER 0
```

```
468 INVERSE 0
```

```
469 CLEAR
```

```
470 LOAD "" CODE: LOAD "" CODE
```

```
471 RANDOMIZE FN Q(23675,64690)
```

*Double POKE the system Variable UDG.*

```
480 GO TO 1 E4
```

*Instead of STOP.*

Whenever you use HEXLD 3 to create a new program you should change lines 420 (to reflect the *name* of the new program), 450 if you need to reserve memory, and 460 if you want a different colour scheme.

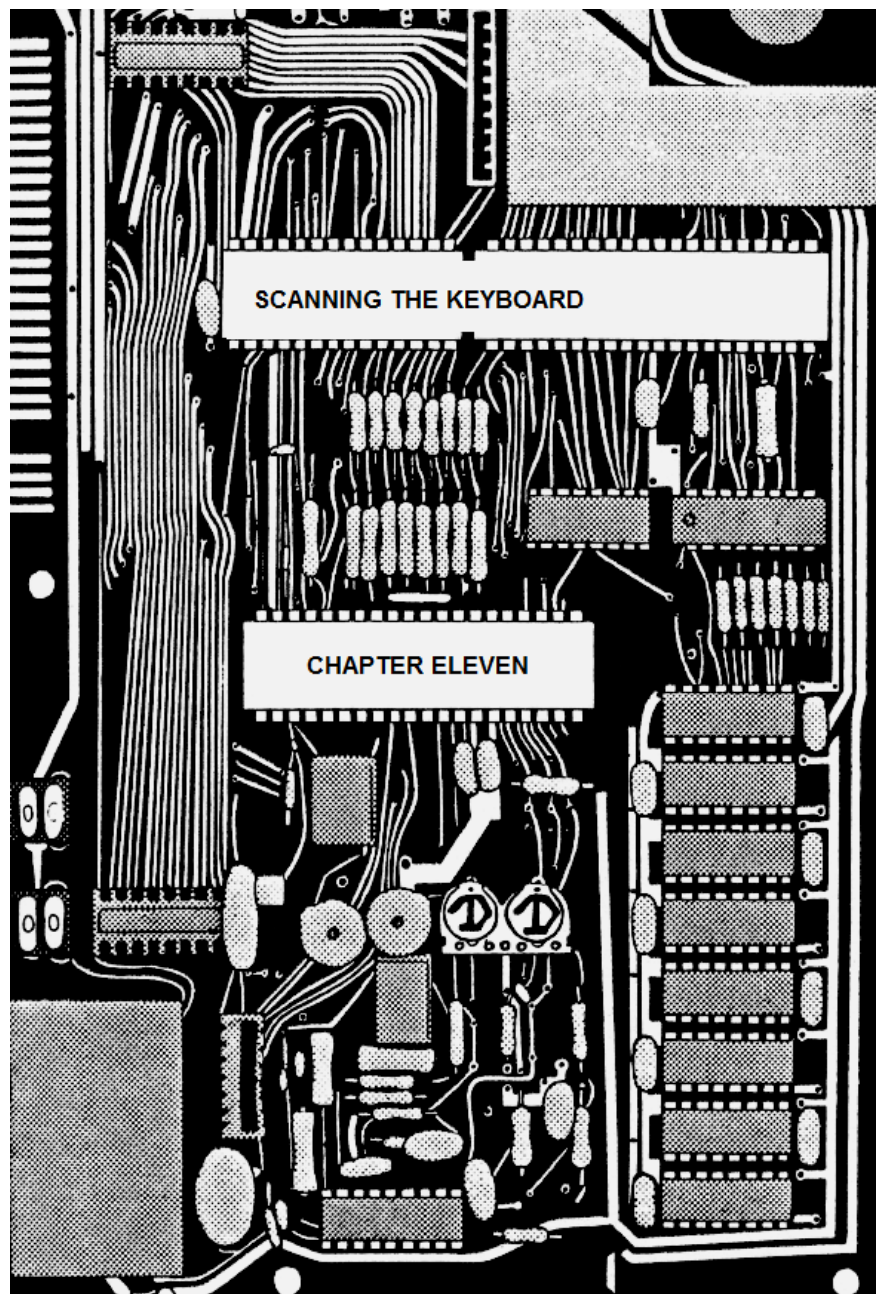
You now have at your disposal a complete machine code editing program.

Mastering Machine Code on your ZX Spectrum

RUN 400 will SAVE and VERIFY everything that is required, *including* the user-defined graphics. I'm sorry if I appear to have covered too much ground too quickly – it doesn't really matter if there are parts of this chapter that you haven't understood yet, what *does* matter is that you now have a tool with which you can create and manipulate new machine code of your own. For reference, I would like to include here a list of all the addresses used in the program.

16K		48K		
7CB2	31922	FCB2	64690	User-defined graphics.
7D5A	32090	FD5A	64858	FN_Q
7D69	32105	FD69	64873	FN_K\$
7D8F	32143	FD8F	64911	FN_H
7DAE	32174	FDAE	64942	FN_ARG
7DB9	32185	FDB9	64953	TEXT
7DD5	32213	FDD5	64981	H_COPY
7E10	32272	FE10	65040	H_RUN
7E1C	32284	FE1C	65052	R-CH
7E3E	32318	FE3E	65086	BEGIN_MC
7E49	32329	FE49	65097	DELETE
7E63	32355	FE63	65123	INSERT
7E87	32391	FE87	65159	WRITE
7EC1	32449	FEC1	65217	NEW_LIM
7EC4	32452	FEC4	65220	NL_2
7EDI	32465	FED1	65233	SET_UP
7EE8	32488	FEE8	65256	LEN_A\$
7EF0	32496	FEF0	65264	H_LIST
7F2A	32554	FF2A	65322	BREAKPT
7F70	32624	FF70	65392	EXIT
7F77	32631	FF77	65399	U_ADDR
7F7A	32634	FF7A	65402	PR_ADDR
7F7D	32637	FF7D	65405	PR_HL
7F89	32649	FF89	65417	I_CHECK
7F99	32665	FF99	65433	REGS

7FCB	32715	FFCB	65483	SH_PRINT
7FD0	32720	FFD0	65488	H_PRINT
7FED	32749	FFED	65517	C_PRINT
7FF6	32758	FFF6	65526	The variable BEGIN
7FF8	32760	FFF8	65528	The variable ADDRESS
7FFA	32762	FFFA	65530	The variable ADD2
7FFC	32764	FFFC	65532	The variable ADD3
7FFE	32766	FFFE	65534	The variable LIMIT



# CHAPTER ELEVEN

## SCANNING THE KEYBOARD

Now it's time to explore how we can make use of some of the other subroutines which are remarkably well-hidden within the ROM. Specifically, we'll cover three of these subroutines, which between them will enable us to scan the keyboard and locate which, if any, of the keys on the keyboard are being depressed. I'll also go through the processes by which these subroutines work, since a little bit of insight always comes in handy.

The first such subroutine is an amazing little keyboard scan, which begins at address 028E. It may be accessed simply by calling that address, ie CALL KEY SCAN or CD8E02 in hex. It produces a *fairly* usable answer, but still not quite what you'd hoped for. Let's see exactly what it does do.

It returns a value to the DE register pair. Actually, it returns separate and independent values – one to D and one to E. Here's how these values are interpreted.

Figure 11.1 shows the keyboard of the Spectrum. As you can see, each key has a number associated with it. Each number is between 00 and 27, and since there are exactly 28 keys and since no number is used twice, you can see that the value associated with any key is unique to that key. This number is called the "key code" – for instance, the key code of "A" is 26 and the key code of U is 0A. If you play at join the dots for a moment you'll see that the numbers are arranged in a kind of outward anticlockwise spiral starting at "B". This provides a useful method of "remembering" the key code for any key without actually having to turn back to the table the whole time.

24	1C	14	0C	04	03	0B	13	1B	23
25	1D	15	0D	05	02	0A	12	1A	22
26	1E	16	0E	06	01	09	11	19	21
27	1F	17	0F	07	00	08	10	18	20

**Figure 11.1**

The value built up in DE by the KEY SCAN subroutine is based on these key codes. If absolutely no key at all is pressed then the value in DE will be FFFF. Suppose though that one, and only one, key is depressed at the time CALL KEY SCAN was used. In this case, the value in D will still be FF, but the value in E will be the key code of that key. One important point to note is that the keys “caps shift” and “symbol shift” are *not* treated differently to any of the other keys – “symbol shift” alone produces FF18 in the same way that “S” alone produces FF1E.

Now suppose that “caps shift” is held down whilst one other key is depressed. In this case, the value of D will not be FF but will in fact be 27 – note that this is the key code of “caps shift”. The E register will contain the key code of the other key being pressed. You should note that even though in this case “caps shift” is being treated differently to any other key, that “symbol shift” is not. In other words, “caps shift” and “symbol shift” simultaneously will give 2718 in the same way that “caps shift” and “3” produces 2714.

Lastly, if “symbol shift” is held down whilst any key other than “caps shift” is depressed then the D register will contain 18 while the E register contains the key code of the key in question. Notice that 18 is the key code of “symbol shift”. It is important to remember that “caps shift”, is considered “more important” than “symbol shift”, so that both of these pressed

together will give 2718 rather than 1827.

The subroutine returns one other important piece of information – the zero flag. All of the cases above will result in the zero flag being set (ie so that JR Z will jump) but if too many keys are pressed at once then the zero flag will be *reset*. If this is the case then the value returned by the DE register pair will in general be meaningless. It is always wise then to check the zero flag before acting upon the information in DE.

It should by now be reasonably clear how each individual key produces its own unique code in the DE register pair, with or without either shift; however, we unfortunately have no real way of determining if, say, “Q”, “8”, and “T” are all pressed together. In machine code even this is possible, so we have a great advantage over BASIC where all we can use is INKEY\$. We’ll see how to go a bit deeper later on in the chapter.

The subroutine KEY\_SCAN does have one big disadvantage though – it will completely wipe out the previous values of all the registers! If you want to preserve them you’ll have to make use of the stack as follows:

```
F5      PUSH AF
C5      PUSH BC
E5      PUSH HL
CD8E02  CALL KEY_SCAN
E1      POP HL
C1      POP BC
F1      POP AF
```

Now, we want to turn those rather obscure numbers into real character codes. These character codes are stored in a table in the ROM starting at address 0205. Actually there are several tables here, but the first is the one we’re interested in. It’s called KEY TABLE and lists the keys in a familiar order.

```
0205 DEFM B H Y 6 5 T G V
```



*Mastering Machine Code on your ZX Spectrum*

020D DEFM N J U 7 4 R F C  
0215 DEFM M K I 8 3 E D X  
021D *E-control* L O 9 2 W S Z  
0225 *space enter* P 0 1 Q A

You should notice that this is the same order as the numbering of the key codes, so a fairly meaningful result can be obtained just by the sequence of instructions LD HL,0205/LD D,00/ADD HL,DE/LD A,(HL) which will generate a character. You should notice a couple of things about this table though. Firstly, that the letters in the table are *capital* letters, not lower case, although the numbers are still numbers (not edit functions) – it’s a bit like having *caps lock* on I suppose. The other thing you should notice is that the character listed as being on the *symbol shift* key is *E-control*, that is, character 0E (see Appendix Five). This is for two reasons: (i) “*symbol shift*” does not have a character code, (ii) the shift keys are sometimes used to change the mode to E-mode (Extended mode).

Having got a valid keyboard scan and having decided that too many keys were *not* pressed, the next problem is to decide if any “real” key at all was pressed. Remember that no key at all produces FFFF, “*caps shift*” only gives FF27, while “*symbol shift*” on its own comes up with FF18. All of these should be rejected. The subroutine KEY TEST at 031E solves this problem. Providing that DE starts off with valid keyboard scan (as described only a few seconds ago) then CALL KEY TEST will do a number of things:

- (i) B will be made to contain the value formerly held by D.
- (ii) D will be made to contain zero.

Following this, one of two courses of action will be taken:

- (i) If DE started off as FFFF, FF27, or FF18, then the A register will be made to contain FF, 27 or 18 accordingly, and CARRY will be reset.
- (ii) If DE did *not* start off as FFFF, FF27 or FF18, then the A register will

be made to contain the *base character* for the required key (that is, the appropriate character taken from the table KEY TABLE above), and the carry flag will be set.

So, all in all quite a useful subroutine, but we can do even better than that for there is a *third* subroutine in the ROM which is quite useful here. This subroutine is called KEY CODE, and it converts the *base character* of any key into a real Sinclair character code. It needs the registers containing all the right values though, so here are the rules.

The E register must contain the appropriate base character.

For no shifts:	B must contain FF.
For caps shift:	B must contain 27.
For symbol shift:	B must contain 18.
For G mode	C must contain 02.
For E mode	C must contain 01.
For K mode	C must contain 00 and D must contain 00.
For L mode	C must contain 00, D must contain 08, and bit three of the system variable FLAGS 2 must be reset.
For C mode	C must contain 00, D must contain 08, and bit three of the system variable FLAGS 2 must be set.

You can obtain a *current* keyboard scan by loading C with the system variable MODE, and D with the system variable FLAGS. (Incidentally, if D does not contain either 00 or 08 then only bit three is taken into consideration.)

Given all of this information, we can write a very short machine code subroutine which does everything we want it to (well, in terms of scanning the keyboard anyway, just by using these ROM subroutines. Thus:

```
CD8E02    SCAN        CALL KEY SCAN
```

## Mastering Machine Code on your ZX Spectrum

200F		JR NZ,VOID
CD1E03		CALL KEY TEST
300A		JR NC,VOID
5F		LD E,A
0E00		LD C,00
1608		LD D,08
CD3303		CALL KEY CODE
A7		AND A
C9		RET
37	VOID	SCF
C9		RET

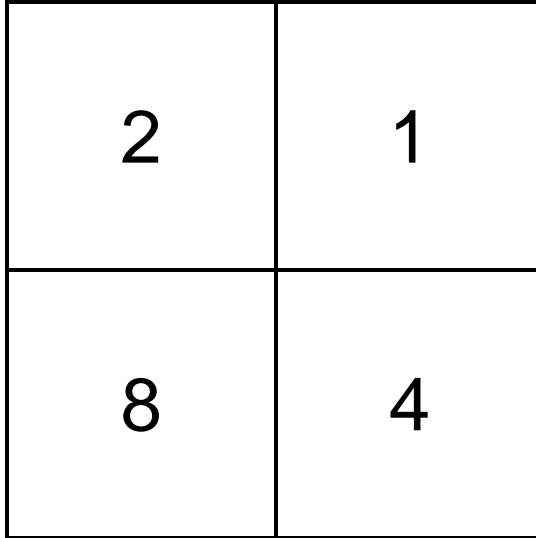
If you write this subroutine anywhere in RAM you can then instantly scan the keyboard at any time just using CALL SCAN which will return the required character in the A register. The subroutine will return a carry if the result of the keyboard scan is void – that is, if either too many keys were pressed, no keys were pressed, or one of the shift keys was pressed on its own.

The address of KEY CODE, by the way is 0333.

Let's show this by example (which is always the best way – or if not, the most fun). We'll write a program, which prints large sized characters on the screen. If you've forgotten how character symbols are stored in the ROM then go back to Chapter Seven for a quick recap while I go and have some tea, and then we can both get back to some serious programming.

The way to do this is to note that the character codes of the graphics symbols are arranged in a rather clever order. As you know, each graphics symbol is composed of four quarter-squares. If we give each quarter-square a number, 1, 2, 4 or 8 (see Figure 11.2) then you can work out the character code of any graphics symbol just by adding up the numbers in those quarter-squares which are *set* (that is printed in INK colour), and adding 80 to the result. For instance, the character *graphic-6* has the top left-hand corner and the bottom right-hand corner both set –

the number corresponding to these squares are 2 and 4 – therefore its character code is  $2 + 4 + 80$ , or 86. If you take a look either in the Sinclair manual or in Appendix Five of this book you'll see that this is the case.



**Figure 11.2**

Write the subroutine SCAN listed above to any suitable address, and then enter the following program (immediately after SCAN is usual):

AF	LARGE	XOR A	
323C5C		LD (TVFLAG),A	Send PRINT to upper part of screen.
210040	AT_0_0	LD HL,4000	
22845C		LD (DF_CC),HL	
212118		LD HL,1821	
22885C		LD (S_POSN),HL	PRINT AT 0,0.
CD????	WAIT	CALL SCAN	Scan the keyboard
38FB		JR C,WAIT	Wait for new key to

*Mastering Machine Code on your ZX Spectrum*

FE20		CP 20	be pressed
D8		RETC	Return to BASIC if
FE80		CP80	a non-printable
			character is
			pressed
D0		RET NC	
6F		LD L,A	
2600		LD H,00	HL: = character
			code of
			character pressed
29		ADD HL,HL	
29		ADD HL,HL	
29		ADD HL,HL	Multiply character
			code by eight
11003C		LD DE,3C00	
19		ADD HL,DE	Add 3C00 to this
			number (See
			Chapter Seven
			about this rule)
0E04		LD C,04	
0604	OUTER-	LD B,04	
	LOOP		
56		LD D,(HL)	Transfer two rows
23		INC HL	of pixels into DE.
5E		LD E,(HL)	
23		INC HL	
3E08	INNER-	A,08	(08 will become 80
	LOOP		once shifted left
			four times)
CB13		RL E	Compute which
			graphics character
17		RL A	is to be printed
CB13		RL E	
17		RL A	

CB12	RL D	
17	RL A	
CB12	RL D	
17	RLA	
D7	RST 10	Print this symbol
10EF	DJNZ INNERLOOP	Next print position
3E0D	LD A, "enter"	
D7	RST 10	End of current line
OD	DEC C	
20E3	JR NZ, OUTERLOOP	
18BE	JR AT_0_0	

Did you notice those instructions at the start POKEing things into the system variables? Loading DF\_CC with 4000 and S\_POSN with 1821 is equivalent to (in BASIC) PRINT AT 0,0; – in fact all that happens when PRINT AT 0,0; is encountered is that DF\_CC and S\_POSN are changed! It is the RST 10 routine which uses these values to work out where to print.

If you now run this program by typing RANDOMIZE USR ?????? (where ?????? is the address in decimal of the label LARGE) the screen will be unchanged. Press the G key and watch what happens. Now press the A key. Interesting isn't it? Incidentally, if you accidentally ran this program with *caps lock* on you won't have half as much fun as with *caps lock* off. You can break out of the program by pressing both shifts together or indeed anything that can't be printed in one character space – STOP for example.

A brief intermission from programming now while we go back to a bit of learning. As I said at the start of the chapter, I'd like to take the keyboard scanning procedure apart a bit more, so that you'll be able to write keyboard scan routines of your own without having to resort to those in the ROM. The advantage of doing this, of course, is that you can then re-write them to allow for several keys at once being pressed. It goes like this.

The keyboard is divided physically, although you can't see the divisions,

into eight different segments. You can see how these segments are arranged by looking at figure 11.3. Each segment corresponds to a different input port, which means that we have to use an IN instruction to find out what's going on.

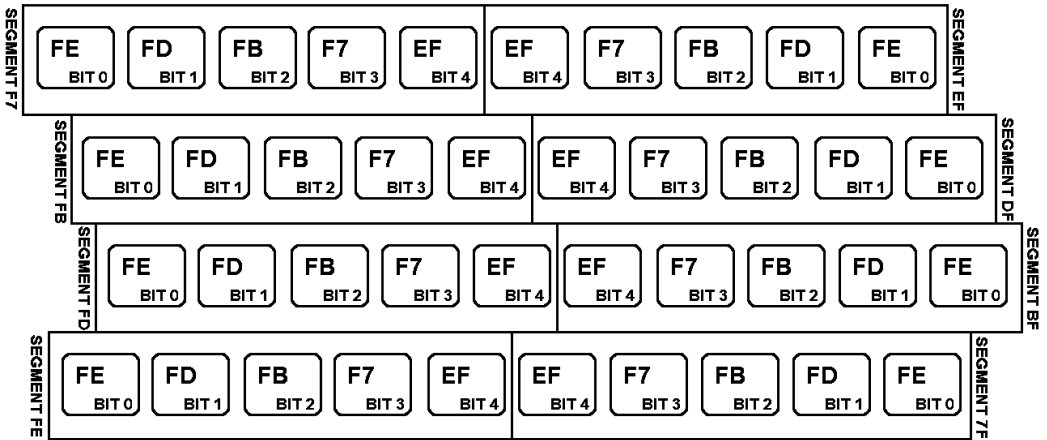


Figure 11.3

I would like to emphasise most strongly here that in the following explanations, the shift keys are regarded in exactly the same way as every other key. Machine code makes no distinction at all – it is the program in the ROM which tests for these keys and treats them separately – without using the ROM the shifts are not isolated. The key “*symbol shift*” shall now be regarded just as much a physical entity as, for instance, the key “X” – indeed, if you really wanted to you could program a routine which scanned the keyboard and regarded the “X” key as a shift!

The keyboard is divided into eight sections, and you can only scan one of them at a time. You can do so in two ways:

- (i) Load A with the section number and then use the instruction IN A,(FE).
- (ii) Load B with the section number, C with the number FE, and then use the instruction IN r,(C) (where r is one of the following registers: A, B, C, D,

E, H or L).

Suppose you wanted to scan section BF. The simplest method is then LD A,BF then IN A,(FE). This would leave a value in the A register. Bits seven, six and five of this result would be meaningless so to get rid of them you should use the instruction OR E0. If no key at all in section BF had been pressed then A would now contain FF. Suppose one, and only one, key had been pressed in section BF; other keys may have been pressed in any other section, but these are not under consideration. The value now contained by the A register would be the hex number printed in figure 11.3 on the key pressed. If more than one key in section FB had been pressed, the A register would contain the result of ANDing together the numbers for each key. Suppose the key “J” had been pressed. Figure 11.3 gives this key the number F7, so if we had been scanning segment BF as described then A would now have a value of F7. If instead the key “L” had been pressed then A would end up with a value of FD. Now, if “J” and “L” had both been pressed simultaneously then A would contain a value of F5 (= F7 AND FD).

You see, each key has a *bit* associated with it. For key J this is *bit* 3, and for key L this is *bit* 1. When you scan any section using IN the result of the scan will be, in binary, ? ? ? b4 b3 b2 b1 b0. The “b”s will be one if the corresponding key is not pressed, or zero if the corresponding key is pressed. Once you’ve straightened out the first three bits (with OR E0) then quite a sensible way of looking at things is to imagine that A starts off as FF and then has one bit reset for each key within that section which is pressed.

## GRAFFITI

It only requires a slight modification to the original version in order to make a really impressive program, demonstrating the immense speed which machine code offers over BASIC. In this program, GRAFFITI, you touch a key and an enlarged version of the required symbol appears on the screen. One interesting point about this program is that it uses PRINT AT in a



*Mastering Machine Code on your ZX Spectrum*

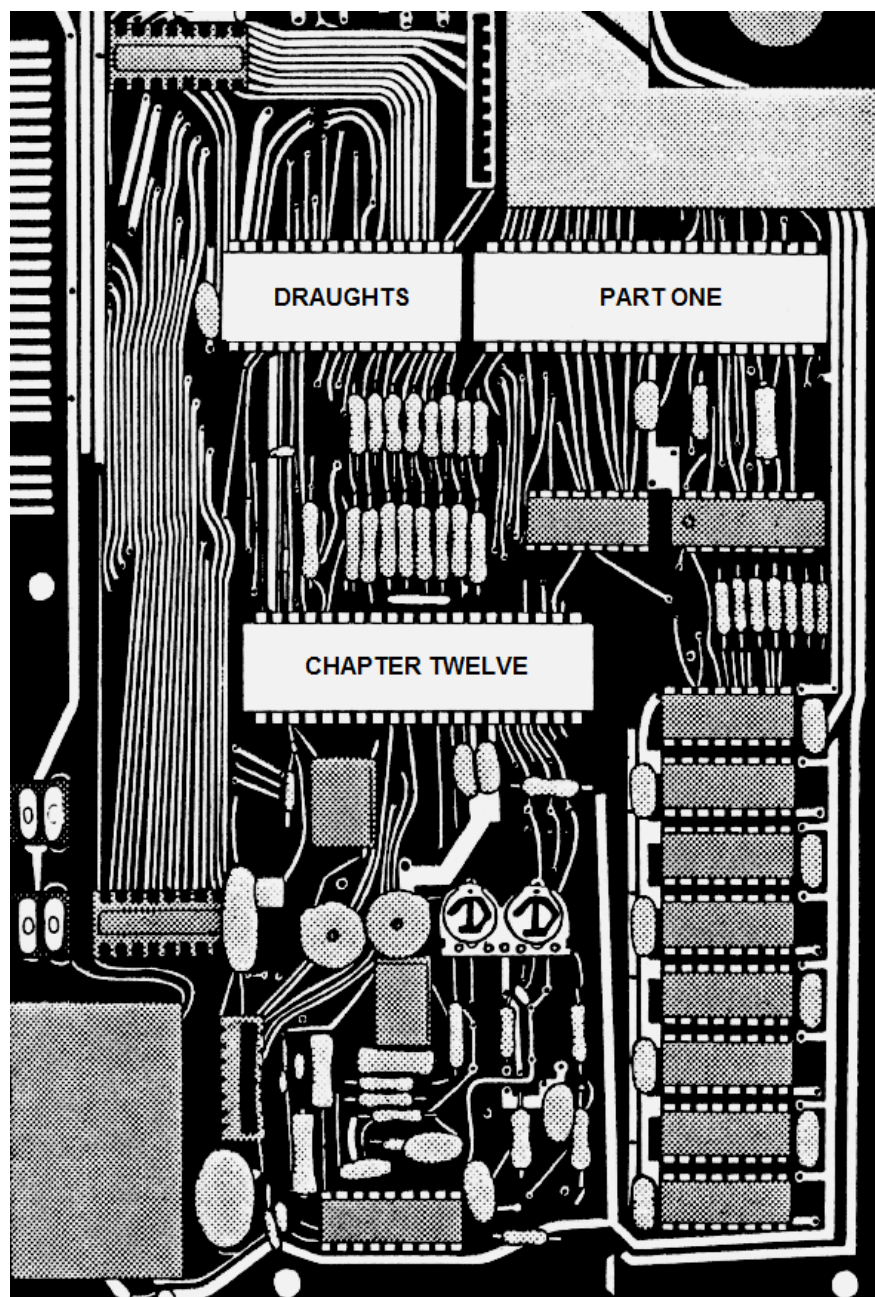
general way. To obtain PRINT AT D,E; only the following steps are required: LD A,“AT-control”/RST 10/LD A,D/RST 10/LD A,E/RST 10. Note that this procedure is very similar to the BASIC version PRINT CHR\$ 22;CHR\$ D;CHR\$ E; – the program lists as follows:

AF 323C5C	GRAFFITI	XOR A LD (TVFLAG),A	Send PRINT output to upper part of screen
ED62		SBC HL,HL	HL: = 0000 (the PRINT AT co-ordinates)
E5	MAIN	PUSH HL	Stack AT co-ordinates
CD????	PAUSE	CALL SCAN	(Listed earlier in chapter)
30FB		JR NC,PAUSE	Wait for finger to be released from key
CD???? 38FB	WAIT	CALL SCAN JR C,WAIT	Wait for new key to be pressed
FE20 3848		CP 20 JR C,EXIT	Exit if control character pressed
FE80 3044		CP 80 JR NC,EXIT	Exit if keyword token pressed
6F 2600		LD L,A LD H,00	HL: = character code
29 29 29 11003C		ADD HL,HL ADD HL,HL ADD HL,HL LD DE,3C00	

19		ADD HL,DE	HL: = point to pixel expansion of required character
0E04 D1	OUTER- LOOP	LD C,04 POP DE	DE: = PRINT AT co-ordinates
3E16 D7 7A D7 7B D7 14		LD A, "AT-control" RST 10 LD A,D RST 10 LD A,E RST 10 INC D	PRINT AT... D,... E. Point to next row down
D5		PUSH DE	Stack AT co-ordinates once more
0604 56		LD B,04 LD D,(HL)	Transfer two rows of pixels into DE
23 5E 23 3E08	INNER- LOOP	INC HL LD E,(HL) INC HL LD A,08	This will become 80 when shifted
CB13 17 CB13 17 CB12 17		RL E RLA RL E RLA RL D RLA	Compute which graphics character is to be printed
CB12 17 D7		RL D RLA RST 10	Print this symbol

*Mastering Machine Code on your ZX Spectrum*

10EF		DJNZ INNERLOOP	
0D		DEC C	
20DC		JR NZ, OUTERLOOP	
E1		POP HL	HL: = AT
7D		LD A,L	co-ordinates
			A: = column
			number of leftmost
			square of large
			character just
			printed
FE1C		CP 1C	
2008		JR NZ, NEXT_CHR	Jump unless this is
			the last character
			allowed on this row
2E00		LD L,00	Reset column
			number to zero.
			Row number is
			already correct
7C		LD A,H	A: = row number
FE14		CP 14	
C8		RET Z	Return if five rows
			of large characters
			have been printed
18AF		JR MAIN	And again for next
			character
1104FC	NEXT_	LD DE,FC04	
	CHR		
19		ADD HL,DE	Move AT
			co-ordinates for
			next character
18A9		JR MAIN	And again for next
			character
F1	EXIT	POP AF	Balance the stack
C9		RET	



DRAUGHTS

PART ONE

CHAPTER TWELVE

# CHAPTER TWELVE

## DRAUGHTS PART ONE

Now that we can enter and edit machine code, it's about time we started using it for something sensible, and (hopefully) interesting. Draughts then is our aim. This is a program we have to be very careful with. Here's what it will look like in BASIC:

```
10 RANDOMIZE USR something
20 INPUT LINE A$
30 RANDOMIZE USR something
```

As you can see, the vast, vast majority of it will be written entirely in machine code. We want to store our machine code at the lowest possible address that we possibly can. If you type RUN 700 and input the address of the BREAKPT routine of HEXLD 3 (7F2A or FF2A) you will immediately see the initial values of each of the registers. Now, the initial value of DE is the address of the first byte of spare RAM (you can verify this by adding one byte to the BASIC program – a space for instance – and doing the experiment again: DE will then start off one byte higher) – this is obviously quite useful. Now, since the BASIC area tends to leap around a bit, we obviously can't use this address itself for the start of our program since if we do this it will surely be overwritten as soon as we press any button. Therefore, I suggest that we store the machine code of DRAUGHTS at address 6800. To initialise this PRINT FN H("6800") tells us that the starting address in decimal is, in fact, 26624, so let's adjust HEXLD 3 to prepare for this program. Type or EDIT the following:

```
420 SAVE "Draughts" CODE FN P(32758),FN P(32766) – FN
      P(32758) + 1                               (16K folk)
420 SAVE "Draughts" CODE FN P(65526),FN P(65534) – FN
      (65526) + 1                               (48K folk)
450 CLEAR 26623
CLEAR 26623                                   (as a direct command)
```

```
460 LET P = 5: LET 1 = 0
RUN 460
```

(to give paper cyan and ink black)

It will be necessary to press the space key (to break out at line 470) before you can do anything else.

Finally, you should enter RUN 500 and input 6800, and type in any old rubbish – 00, for instance. Break out by pressing newline alone. At this point I would SAVE the program using RUN 400, even though we haven't properly started it yet, because it's good to have on tape what is effectively a draughts editing program, and also so that you don't have to type in the lines above again.

OK – here we go. The first part of the program you've seen before already. It's the piece of program from Chapter Seven which prints a draughts board. Firstly, you will need to redefine graphics A. To do this input as a direct command FOR I = 0 TO 7: INPUT X: POKE USR "A" + I,X: NEXT I and then input the following numbers: 0/60/126/126/126/126/60/0. Now, if you type REM *graphics A* you'll find a small blob appear before you. Type RUN 100 now and input the address 6837 (note: 6837, not 6800 – for reasons which will come clear later on). Now input the draughts board printing routine from Chapter Seven – you'll have to type it all in again I'm afraid, and you *must* remember that the address S\_PRINT is now 6837, not 7000, so that the instructions CALL S\_PRINT translates to CD3768, not CD0070.

You can make things easier for yourself just by stopping where the data starts, and then changing line 930 to read INPUT USR 32631;CHR\$ 8;A\$ 116K) or INPUT USR 65399;CHR\$8:A\$ (48K) (ie deleting the word LINE). You can then input all the text using RUN 900. To get an *enter* character into the text you should move the cursor to the right of the input quotes and type + CHR\$ 13. The *text* starting address is 6867.

```
RUN 900
Text address 6867
```

*Mastering Machine Code on your ZX Spectrum*

*"space 12345678" + CHR\$13*

*"1 space graphics A space graphics A space graphics A space graphics A  
1" + CHR\$ 13 ... and so on up to ...*

*"space 12345678" + CHR\$ 0 + CHR\$ FN H("C9")*

Now if you RUN 700 and input 6840 you should see the draughts board in its infancy appear. Now, we need to make it come to life. Add one extra line of BASIC: 715 CLS to get rid of the "RUN ADDRESS" message when using RUN 700. Now if you RUN 700 and input 6840 you'll get the same picture but without the "RUN ADDRESS" message.

Insert the following subroutine to address 6840:

3E04	ROW	LD A,04	
71	ROW_2	LD (HL),C	Colour next draughts piece
23		INC HL	
23		INC HL	Point HL to next draught piece
3D		DEC A	
20FA		JR NZ,ROW_2	
19		ADD HL,DE	Point to next row
CB45		BIT 0,L	
2002		JR NZ,ROW_3	
23		INC HL	For even rows the pointer must be adjusted
23		INC HL	
10EF	ROW_3	DJNZ ROW	Repeat for required number of rows
C9		RET	

And now insert this machine code to address 68C5 (just before the last RET instruction):

210058		LD HL,ATTRS	Point HL to attributes file
111600		LD DE,0016	
0E0A		LD C,0A	
060A	YELLOW	LD B,0A	
	_1		
3630	YELLOW	LD (HL),30	Colour next square black on yellow
	_2		
23		INC HL	Point to next square
10FB		DJNZ YELLOW_2	
19		ADD HL,DE	Point HL to start of next row
0D		DEC C	
20F5		JR NZ,YELLOW_1	
212158		LD HL,ATTRS + 21	Point HL to first square on board
1E18		LD E,18	
0E08		LD C,08	
0608	WHITE_1	LD B,08	
3638	WHITE_2	LD (HL),38	Colour square black on white
23		INC HL	Point to next square
10FB		DJNZ WHITE_2	
19		ADD HL,DE	Point HL to first square on next row
0D		DEC C	
20F5		JR NZ,WHITE_1	
212258		LD HL,ATTRS + 22	Point HL to first black square
1D		DEC E	



*Mastering Machine Code on your ZX Spectrum*

010703	LD BC,0307	B: = number of rows to colour C: = colour of draughts piece
CD4068	CALL ROW	Colour three rows white
010002	LD BC,0200	
CD4068	CALL ROW	Colour two rows black
010203	LD BC,0302	
CD4068	CALL ROW	Colour three rows red

Now if you type RUN 700 and input 6852, your draughts board should be printed complete with colour.

So now we come on to the program proper – that is, the *second* RANDOMIZE USR statement. What we want to happen is for line 710 to be repeated as soon as control returns to BASIC, but we *also* want some way of breaking out. It is quite possible to form an infinite loop if we don't have some machine code means of leaving the program. Let's carry on with draughts. Here then is the routine we need. This will break out and return to BASIC command mode if the first character of the input is STOP (*symbol shift A*), and if this is *not* the case, will ensure that on return to BASIC, the next statement executed will be line 710. The routine should be written to address 6901. It contains a couple of new tricks which I'll fully explain in a moment.

2A4B5C	MOVE	LD HL,(VARS)	Point HL to the BASIC variable A\$
23		INC HL	
5E		LD E,(HL)	E: = length of A\$
23		INC HL	
23		INC HL	Point HL to text of A\$

7E		LD A,(HL)	
FEE2		CP "STOP"	
2002		JR NZ,A\$_OK	
CF		RST 08	
FF		DEFB FF	Return to BASIC if A\$ starts with "STOP"
21C602	A\$_OK	LD HL,710d	HL: = the line number 710
22425C		LD (NEWPPC),HL	Specify a jump to line 710
3E01		LD A,01	
32445C		LD (NSPPC),A	Specify first statement in line
C9		RET	

The BASIC can now be brought up to date, otherwise jumping to line 710 won't make sense. Amend all BASIC lines between 700 and 799 as follows:

```
700 RANDOMIZE USR 26706
710 INPUT LINE A$
720 RANDOMIZE USR 26881
```

If you now type RUN 700 to start the ball rolling, the first thing you'll notice happening is the by now familiar draughts board appearing and something new – an INPUT prompt. You'll find that whatever you input gives you the same result – another input prompt. What is happening is that line 710 is continually being jumped to on return from USR – you can imagine GO TO 710 being hidden in the machine code. To break out of the program you must input STOP (*symbol shift A*). Now, I must explain how this automatic "GO TO 710" came into being. Two system variables here are assigned. The first of these is called NEWPPC which stores the line number to be jumped to. Had we loaded this variable with say, 1000d the line 1000 would be jumped to on return to BASIC. In fact, we have even finer control

than that, for the system variable NSPPC stores the statement number within that line to be jumped to, so we could even have simulated “GO TO 1000, statement 4” if we’d have wanted. It is important to note that this “GO TO” is *not immediate*. The Spectrum will carry on executing the machine code until it reaches a final RET. Then and only then will the values of NEWPPC and NSPPC be looked at by the ROM and the GO TO carried out.

Now the program seriously starts. We assume that a move has been input as A\$, which is the first item in the variable store. Here’s how to input a move. Take a look at figure 12.1. There are sixty-four squares, but only thirty-two of them are playable (the black ones). Each square has a co-ordinate from 11 to 88. Notice that these are printed without a separating comma. The first digit refers to the number down the left- (and right-) hand side of the board, and the second digit refers to the number along the top (or bottom).

There are four different directions you may move in. These are called A (up-left), B (up-right), C (down-right) and D (down-left). To input a move simply type in the co-ordinates and a letter (A, B, C, or D). There are no spaces in this input. For instance, to move from square 61 to square 52 you should input 61B. To input a double jump from square 65 firstly in direction A and then in direction B, you should input 65AB.

For the computer to do its working out, it needs somewhere to do its working *in* – a kind of notepad if you like. The area I’ve chosen for this is the “blank space” we’ve left between addresses 6800 and 6836. To fill this workspace (which will be referred to as BOARD\_2 we need to insert some machine code. Insert this to address 6901 and then I’ll explain it in detail.

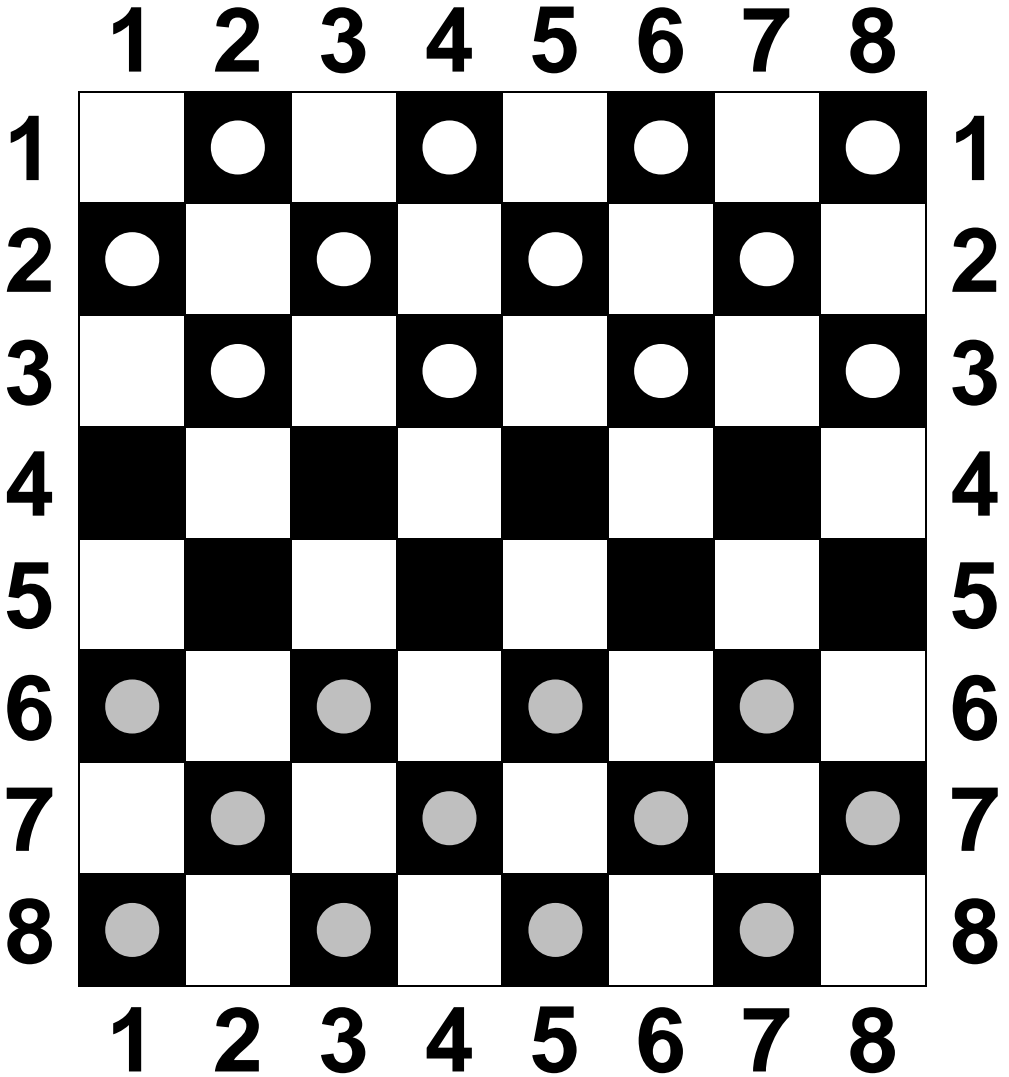
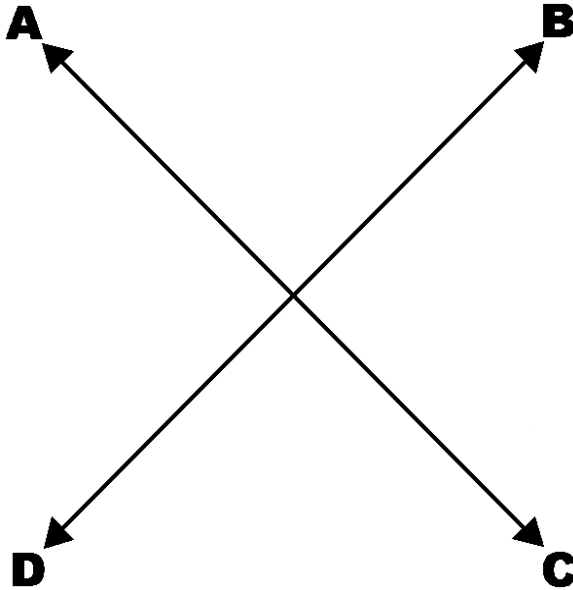


Figure 12.1a



**Figure 12.1b**

212258	COPY_B	LD HL,ATTRS + 22	Point HL to first playable square
110668		LD DE, BOARD_2 + 06	Point DE to first square in workspace
0E04		LD C,04	
0604	C_LOOP_A	LD B,04	
EDA0	C_LOOP_B	LDI	Transfer one square
03		INC BC	Restore BC
23		INC HL	Point to next square
10FA		DJNZ C_LOOP B	
13		INC DE	Skip one space in working space
D5		PUSH DE	

111700		LD DE,0017	
D1		ADD HL,DE	Point to next
19			square on screen
D1		POP DE	
0604		LD B,04	
EDA0	C_LOOP_C	LDI	Transfer one
			square
03		INC BC	Restore BC
23		INC HL	Point to next
			square
10FA		DJNZ C_LOOP_C	
13		INC DE	
13		INC DE	Skip two squares
			in workspace
D5		PUSH DE	
111900		LD DE,0019	
19		ADD HL,DE	Point to next
			square on screen
D1		POP DE	
0D		DEC C	
20DE		JR NZ,C_LOOP_A	

In order to understand this subroutine, you have to know all about the area of memory I've called the workspace and what the program does with it. Take a look now at figure 12.2 which shows exactly what's going on. Every square on the screen, at least the black squares, corresponds to one square in the workspace, so that the computer actually has a copy of the board elsewhere in memory. As you can see, in addition to there being one byte for each square, there are also additional bytes representing squares "round the edges" as it were – illegal squares. These are there to prevent pieces from moving off any one edge of the board. As things stand the routine above isn't much help, for although it doesn't fill in the unused squares it doesn't blank them out either, so we have to help it a bit. Using RUN 100 I'd like you now to fill every byte between 6800 and 6836 with FF. Having done that, run the program (RUN 700) and break out by inputting

STOP. Now if you list the area of memory between 6800 and 6836 you should find that it corresponds exactly to the diagram of figure 12.2 with FF representing illegal squares, 07 for white pieces, 02 for black (red) pieces and 00 for empty squares. This demonstration should prove that the routine above is actually doing its job properly. I hope you can follow it through and see what's going on. An important point to note is that the position of the pieces is given by looking through the *attributes* file, and not the display file. Every square in the display file which is playable contains a *graphics A*, so it would be pointless looking at that – instead, we look at the attributes, and thus distinguish the different pieces and the empty squares simply by the colours on that square.

	00		01		02		03		04	
05		06		07		08		09		0A
	0B		0C		0D		0E		0F	
10		11		12		13		14		15
	16		17		18		19		1A	
1B		1C		1D		1E		1F		20
	21		22		23		24		25	
26		27		28		29		2A		2B
	2C		2D		2E		2F		30	
31		32		33		34		35		36

**Figure 12.2**

There is one odd point about the subroutine above – the INC BC instructions. These are necessary because LDI not only loads from (HL) to (DE), incrementing both register pairs, but it also decrements BC, and so the INC BC instructions immediately after are there to restore things.

The next thing we do you might think rather odd. What it amounts to is this: PRINT AT 1,11;TAB 0; – the purpose of doing this is to erase the second line on the screen from the twelfth character onwards. Why (I hear you ask)? All will be revealed in a moment. We do this at the very start of our main piece of machine code, so insert to address 6901 the following:

AF	BLANK	XOR A	
323C5C		LD (TVFLAG),A	Print in upper part of the screen
3E16		LD A, "AT control"	
D7		RST 10	
3E01		LD A,01	
D7		RST 10	
3E0B		LD A,0B	
D7		RST 10	PRINT AT 1,11;
3E17		LD A, "TAB control"	
D7		RST 10	
AF		XOR A	
D7		RST 10	
AF		XOR A	
D7		RST 10	PRINT TAB 0;

One more piece of tidying up to do now. You remember the piece of machine code right at the very end of the program so far, which initiates GO TO 170 if A\$ doesn't start with STOP? The part of program we are going to write next relies on HL pointing to the text of A\$, so if we want this to work we are going to have to do something about preserving HL, so just type in these short steps. You can verify that they're in the right place if you like.

RUN 200/694D/E5//	("/" means <i>enter</i> . E5 is hex for PUSH HL)
RUN 200/6954/E1//	(E1 is hex for POP HL)

And now we're ready for the next section. What it will do is to make some



*Mastering Machine Code on your ZX Spectrum*

checks on the *validity* of the input: (i) that the string is at least three characters long; and (ii) that the first and second characters are both digits between one and eight. It will also, as a sideline, calculate the “square number” as shown in figure 12.2. This routine follows on from the check on A\$ = “STOP” and thus, becomes the last machine code in the program. You should insert this to address 695A:

7B	VALID_CH	LD A,E	A: = length of string A\$ A\$
FE03 381A		CP 03 JR C,INVALID	Error if LEN A\$ less than three
7E		LD A,(HL)	A: = first character of string
FE31 3815		CP “1” JR C,INVALID	
FE39 3011		CP “9” JR NC,INVALID	Error if character not a digit “1” to “8”
47 87		LD B,A ADD A,A	Multiply character code by two
87		ADD A,A	... by four
80		ADD A,B	... by five
87		ADD A,A	... by ten
80		ADD A,B	... by eleven
47 23		LD B,A INC HL	Point to second character
7E		LD A,(HL)	A: = second character
FE31		CP “1”	

3804		JR C,INVALID	
FE39		CP "9"	
3817		JR C,VALID	Error if not digit "1" to "8"
CD3768	INVALID	CALL S_PRINT	Print message
16010B		DEFM AT 1,11d	
1106		DEFM paper yellow	
494E56414C494420		DEFM INVALID	
454E545259		DEFM ENTRY	
00		DEFB 00	End of message
C9		RET	
80	VALID	ADD A,B	
CB3F		SRL A	
C6E0		ADD A,E0	
C9		RET	

What you should notice upon running the program now is that any input whose first two digits are both numbers "1" to "8" is accepted, and any other entry is rejected – the message INVALID ENTRY appearing on the screen. Now do you see the reason for the PRINT AT 1,11;TAB 0; at the start of the routine? The previous message must always be erased. The code built up in the A register is the square number of the input square. See how it is built up. It consists of the character code of the first digit multiplied by eleven (because the width of the board in the workspace is eleven squares) plus the character code of the second digit. This number is then divided by two (because only the black squares have numbers) and finally the constant E0 is added, which just happens to be the right number to map "11" to 06.

The next routine is another check – this time that the square given is, in fact, a black piece. It starts just after the SRL A instruction at the end of the routine, so use WRITE (RUN 100) to enter this with or without RUN 900. The address to load it to is 6993:

*Mastering Machine Code on your ZX Spectrum*

381C		JR C,SQ_BLACK
CD3768		CALL S_PRINT
16010B		DEFM <i>AT 1,11d</i>
1106		DEFM <i>paper yellow</i>
574849544520		DEFM WHITE
53515541524520		DEFM SQUARE
474956454E		DEFM GIVEN
00		DEFB 00
C9		RET
C6E0	SQ_BLACK	ADD A,E0
C9		RET

If you run this program now you should see both checks in action. Try breaking either rule and see what happens. Feel ready for another one yet? This next check makes sure that the square you've indicated actually has one of your own pieces in it, so that you can't cheat by moving the computer's piece instead of your own. Write this to address 69B3:

E5	PUSH HL	Stack pointer to second character of A\$
6F 2668	LD L,A LD H,68	HL points to indicated square within the workspace board copy
7E	LD A,(HL)	A: = contents of this square
E67F 201F	AND 7F JR NZ,PIECE	Jump if square occupied
CD3768 16010B 1106	CALL S_PRINT DEFM <i>AT 1,11d</i> DEFM <i>paper yellow</i>	

54484120		DEFM THAT	
53515541524520		DEFM SQUARE	
495320454D505459		DEFM IS EMPTY	
00		DEFB 00	
E1		POP HL	
C9		RET	
FE02	PIECE	CP 02	
281A		JR Z,PIECE_OK	Jump if human's piece selected
CD3768		CALL S_PRINT	
16010B		DEFM <i>AT 1,11d</i>	
1107		DEFM <i>paper white</i>	
54484154275320		DEFM THAT'S	
4D59205049454345		DEFM MY PIECE	
00		DEFB 00	
E1		POP HL	
C9		RET	
E1	PIECE_OK	POP HL	
C9		RET	

Having done just about all we can with regard to the starting square, it's about time now to start thinking about the direction(s) of movement. The next bit goes at 69F9 (overwriting the last POP HL and RET instructions).

1D		DEC E	
1D		DEC E	E: = number of moves being made
1D		DEC E	
2001		JR NZ,NOT_ZERO	
1D	LOOP_P1	DEC E	
E3	NOT_ZERO	EX (SP),HL	HL: = points to byte before next direction of movement

*Mastering Machine Code on your ZX Spectrum*

23		INC HL	HL: = points to next direction
7E		LD A,(HL)	A: = direction to move
E6DF		AND DF	Convert to upper case
FE41		CP "A"	
3804		JR C,INV_DIR	
FE45		CP "E"	
381C		JR C,VAL_DIR	Error if direction not "A", "B", "C" or "D"
CD3768	INV DIR	CALL S_PRINT	
16010B		DEFM AT 1,11d	
1106		DEFM <i>paper yellow</i>	
494E56414C494420		DEFM INVALID	
4449524543			
54494F4E		DEFM DIRECTION	
00		DEFB 00	
E1		POP HL	
C9		RET	
E1	VAL_DIR	POP HL	
C9		RET	

Study the curious treatment of the E register at the start of that routine. E starts off, if you remember, with the length of the string A\$. Now, the number of moves being input is simply this length minus two, so first of all two is subtracted. What happens next, however, is that E is decremented *again*, and furthermore, if E contained 00 then its value is changed to FF. What this means in real life is that for ordinary moves and ordinary jumps, E will contain FF, but for multiple jumps, E will contain one less than the number of jumps to take. This is so that we can check later on that the player is not cheating. If, for instance, she decided to make a multiple move in which not every move was a jump then we could fathom this out quite easily.

The direction then is now “A”, “B”, “C” or “D”. This is obviously not the final form we would like the direction in. What we would like to know is the displacement from the starting square to the finishing square. For instance, suppose the direction of movement was “C” – in this case, the displacement would be 06 since if you add 06 to any square number (figure 12.2) you get the square number of the result of moving in direction C. The displacements we require are thus “A” = FA, “B” = FB, “C” = 06 and “D” = 05.

There is also another check we have to make on the direction – whether a piece is moving forwards or backwards. In draughts, only kings may move backwards. In this version of the game, kings are represented by flashing characters – that is, characters with bit seven of the attribute byte set. The following routine then does everything that we want it to do. You should write it to address 6A28:

1606		LD D,06	D will hold displacement
3D		DEC A	A: = 40, 41, 42 or 43
CB3F		SRL A	A: = 20 or 21
3001		JR NC,DIS_1	Jump if direction “A” or “C”
15		DEC D	D: = 06 for “A” or “C”, “05 for “B” or “D”
FE20	DIS_1	CP 20	
7A		LD A,D	
2002		JR NZ,DIS_2	Jump if direction “C” or “D”
ED44		NEG	A: = correct displacement
57	DIS_2	LD D,A	D: = correct displacement

*Mastering Machine Code on your ZX Spectrum*

E3		EX (SP),HL	HL: = points to square in workspace
E680		AND 80	A: = 80 for forward move, or 00 for backward move
B6		OR (HL)	
FE82		CP 82	Check for illegal backward move
2819		JR Z,DIS_OK	
CD3768	B_MOVE	CALL S_PRINT	
16010B		DEFM AT 1,11d	
1107		DEFM paper white	
4241434B57			
4152445320		DEFM BACKWARDS	
4D4F5645		DEFM MOVE	
00		DEFB 00	
E1		POP HL	
C9		RET	
E1	DIS_OK	POP HL	
C9		RET	

Slowly, now, we're running out of things to eliminate. This next routine should, I hope be obvious. Write it to address 6A59:

4E		LD C,(HL)	C: = human's piece
3600		LD (HL),00	Delete piece from former position
7D		LD A,L	
82		ADD A,D	
6F		LD L,A	HL: = points to destination square
7E		LD A,(HL)	A: = contents of destination square
FEFF		CP FF	

2019		JR NZ,ON_BOARD	
CD3768	OFF_BOARD	CALL S_PRINT	
16010B		DEFM <i>at 1,11d</i>	
1106		DEFM <i>paper yellow</i>	
4D4F564520		DEFM MOVE	
4F464620		DEFM OFF	
424F415244		DEFM BOARD	
00		DEFB 00	
E1		POP HL	
C9		RET	
E67F	ON_BOARD	AND 7F	Disregard king status
FE02		CP 02	Check for human piece
2019		JR NZ,M_1	
CD3768	SQ__BL	CALL S_PRINT	
16010B		DEFM <i>at 1,11d</i>	
1107		DEFM <i>paper white</i>	
53515541524520		DEFM SQUARE	
424C4F434B4544		DEFM BLOCKED	
00		DEFB 00	
E1		POP HL	
C9		RET	
E1	M_1	POP HL	
C9		RET	

The next routine completes the moving section and will successfully transfer the piece to where you want it to go, although at present this new board position is stored in the board copy only. The routine also makes two final checks; that each of the steps in a multiple jump sequence is actually a jump and not a move; and that a newly-kinged piece may not be jumped backwards in the same move that it becomes a king. This segment then brings DRAUGHTS Part One very near completion. It should reside at address 6A9C:



*Mastering Machine Code on your ZX Spectrum*

A7		AND A	
2024		JR NZ,JUMP	Is this a move or a jump?
7B		LD A,E	
3C		INC A	
282C		JR C,CONTINUE	
CD3768		CALL S_PRINT	
16010B		DEFM <i>at 1,11d</i>	
1401		DEFM <i>inverse on</i>	
4D4F564520494E20		DEFM MOVE IN	
4A554D5020		DEFM JUMP	
53455155454E4345		DEFM SEQUENCE	
00		DEFB 00	
E1		POP HL	
C9		RET	
3600	JUMP	LD (HL),00	Erase computer's piece
7D		LD A,L	
82		ADD A,D	
6F		LD L,A	HL: = destination square
7E		LD A,(HL)	A: = contents of destination square
3C		INC A	
2898		JR Z,OFF_BOARD	Error if square off board
3D		DEC A	
20B4		JR NZ,SQ_BL	Error if square blocked
71	CONTINUE	LD(HL),C	Place human piece at new square
1C		INC E	
2804		JR Z,EP_1	
1D		DEC E	

```

C2FE69          JP NZ,LOOP_P1
E1              EP_1      POP HL
C9              RET

```

And now we come to the final section of Part One. This section makes kings and then (what you've all been waiting for) prints the new board out at the end of it. When you're ready, write this to address 6AD7:

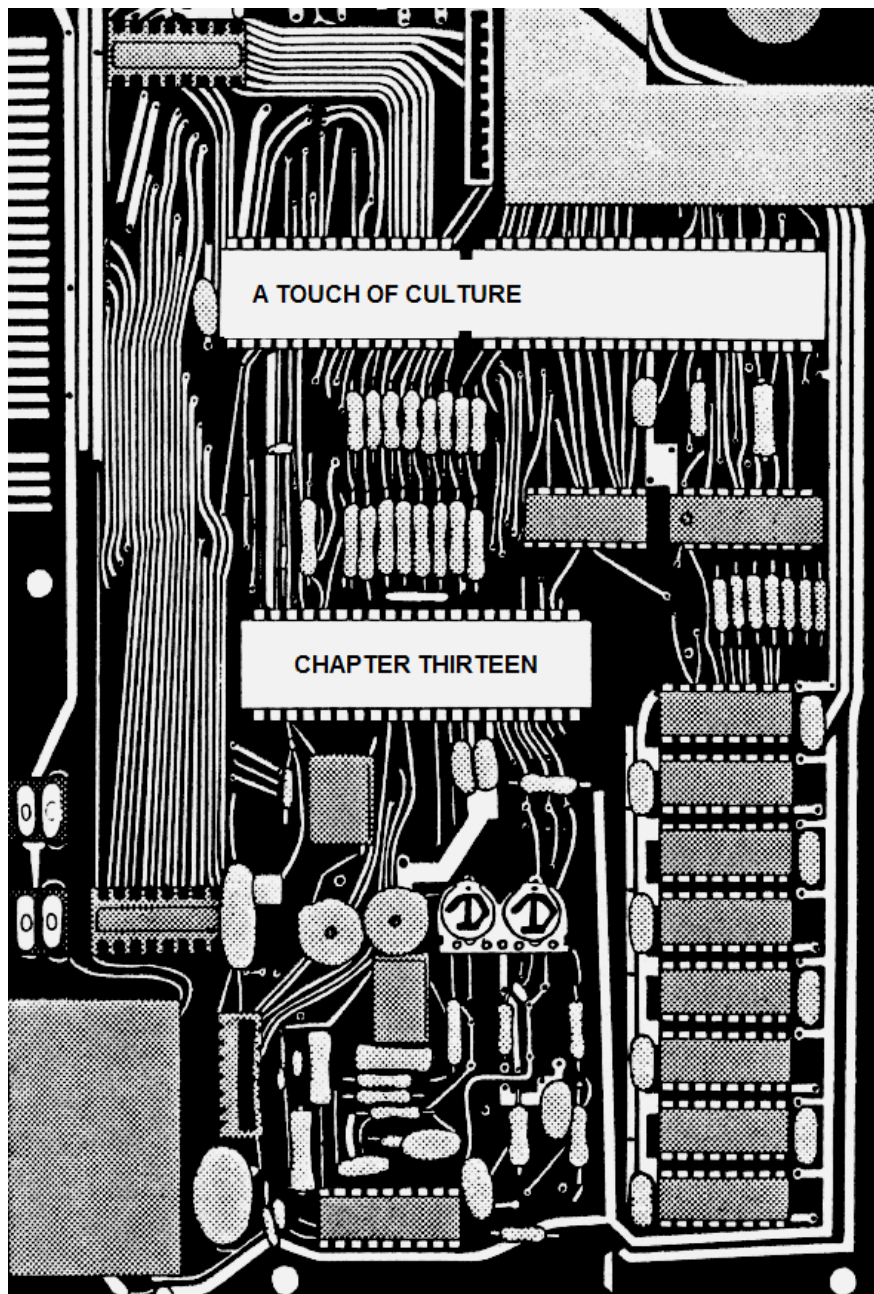
E1		POP HL	Balance the stack
210668		LD HL,BOARD_2 +6	Point HL to first square in board
0604		LD B,04	
7E	KING	LD A,(HL)	A: = contents of square
FE02		CP 02	
2002		JR NZ,M_2	If there is an
3682		LD (HL),82	un-kinged black piece on the back row then make it a king
23		INC HL	Point to next square
10F6		DJNZ KING	Repeat for all squares on back row
212258		LD HL,ATTRS + 22	Point HL to first black square on screen (in attributes file)
110668		LD DE,BOARD_2 +6	Point DE to first square in board copy
0E04		LD C,04	

*Mastering Machine Code on your ZX Spectrum*

0604 1A	PRL_1 PR L_2	LD B,04 LD A,(DE)	A: = contents of next square
13		INC DE	Point DE to next square
77		LD (HL),A	Re-print next square on board
23 23		INC HL INC HL	Point HL to next square
10F9 13		DJNZ PRL_2 INC DE	Skip over "edge marker"
D5 111700 19		PUSH DE LD DE,0017 ADD HL,DE	Point HL to first square in next row
D1 0604 1A	PRL_3	POP DE LD B,04 LD A,(DE)	A: = contents of next square
13		INC DE	Point to next square
77		LD (HL),A	Reprint next square on board
23 23		INC HL INC HL	Point to next square
10F9 13 13		DJNZ PRL_3 INC DE INC DE	Skip over two edge markers
D5 111900		PUSH DE LD DE,0019	

19	ADD HL,DE	Point HL to first square on next row
D1	POP DE	
0D	DEC C	
20DC	JR NZ,PRL_1	
C9	RET	

In part two of DRAUGHTS we shall consider physically making the move of the computer and also, of course, checking whether or not the game has actually finished. This is quite an easy check really – all you have to do is determine whether or not the remaining pieces on the board are all the same colour. Anyway – a breather for now. Let's take a rest from all this high-powered thinking and concentrate on something a little more creative.



A TOUCH OF CULTURE

CHAPTER THIRTEEN

# CHAPTER THIRTEEN A TOUCH OF CULTURE

Music from your Spectrum? Is it possible? Most of us, of course, know about the BEEP statement in BASIC, but can we do better than that? The answer is yes.

As you know, the BEEP statement enables you to play a given note for a fixed length of time. You can alter both of these “in advance”, as it were, just by altering the program, but you can’t do anything about it in “real time” – as it’s happening. Machine code, on the other hand, is an entirely different kettle of fish. In BASIC the playing of music is concrete – once the note routine is entered control will stay there until the note is finished. Then, and only then, will the keyboard be scanned once more.

It is conceivably possible to write a BASIC program which BEEPs for a bit and then checks INKEY\$ to decide on a new note, but this still isn’t true real-time control. In machine code you can write a routine of your own – a replacement for BEEP, which scans the keyboard *whilst the note is actually playing* and acts accordingly by your instructions.

This, then, is called CATHY’S PROGRAM, dedicated to someone who believes that computers should be artful, and not just attack you with space invaders. The addresses given in this listing assume that the machine code will be written to address 6800, but if you wish you may store the program absolutely anywhere in RAM, providing you remember to change the references to the absolute addresses NOTES and SOUND.

### *Cathy’s Program*

9E93464B	NOTES	G	G #	G +	F # +	Keys	B	H	Y	6
0050A9B4		–	F +	F #	F	Keys	5	T	G	V
8A813D42		A	A #	A +	G # +	Keys	N	J	U	7



C5 CD8E02	LOOP	PUSH BC CALL KEY_SCAN	DE: = keyboard scan.
C1 212027		POP BC LD HL,2720	HL: = "caps shift space" scan code.
A7 ED52 281B		AND A SBC HL,DE JR Z,EXIT	Exit if caps shift space pressed.
7B		LD A,E	A: = keyboard scan (ignoring shifts).
3C 28EF		INC A JR Z,LOOP	Loop if no key pressed.
AF 57		XOR A LD D,A	DE: = keyboard scan (ignoring shifts).
210068 19		LD HL,NOTES ADD HL,DE	Point HL to required note data byte.
46		LD B,(HL)	B: = data for required note.
B8		CP B	Compare B with zero.
28E5		JR Z,LOOP	Loop if no note on that key.
79		LD A,C	A: = border colour with bit four set.
C5 CD2868		PUSH BC CALL SOUND	First half cycle of note.
C1		POP BC	



Mastering Machine Code on your ZX Spectrum

E607		AND 07	Reset "sound bit" (bit four).
CD2868		CALL SOUND	Second half cycle of note.
18D8		JR LOOP	
FB	EXIT	EI	
C9		RET	

You now need only one more statement to make the program start running. The statement RANDOMIZE USR 26672 – this is the address of the label *start* (in hex 6830). You can (and should) now delete the BASIC entirely by typing NEW and then typing in the following BASIC program:

```
1 RANDOMIZE USR 26672
2 STOP
9999 LOAD "" CODE
```

which may then be saved using SAVE "MUSIC" LINE 9999 then SAVE "MUSIC" CODE 26624,102.

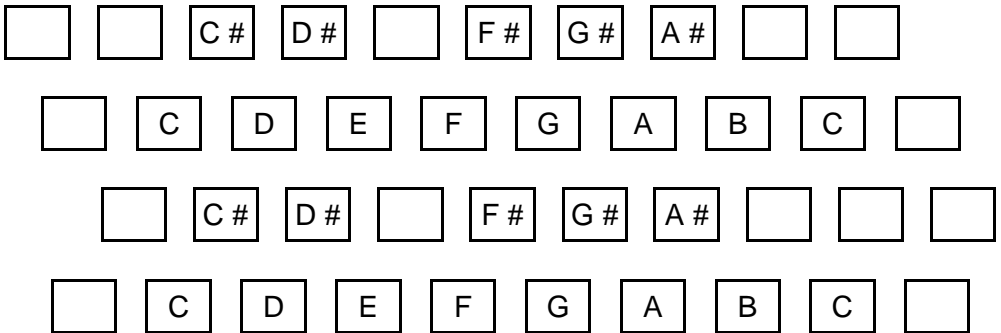


Figure 13.

You now have two octaves at your disposal – the keyboard of figure 13.1 shows where the notes are. A fair number of tunes may be played quite successfully.

If you get one of those little adaptors with a Spectrum-sized jack plug at one end and whatever your stereo takes at the other, then the Spectrum sound can be radically improved. You can plug this jack into either the EAR or the MIC socket of the Spectrum. The Sinclair manual does say you can do this (for BEEP) but doesn't seem to stress the point very strongly – I do! I think the Spectrum sound via an amplifier is really good, and I do urge you to try it out if you can.

You can re-tune the individual keys if you want. The program listing also indicates which keys correspond to which bytes (in the block of data labelled NOTES). To change the note on a key you merely change the value of that byte. A zero means that the key will give nothing but silence. Otherwise, the larger the byte, the lower the note and the smaller the byte, the higher the note.

Once the program is running any key which is programmed with a note will play that note until your finger is removed. You may *not* play more than one note simultaneously. Note by the way that the key “*symbol shift*” by itself has a note on it (the note C). You could not possibly detect this key in BASIC as you can in machine code. To break out of the program you must press “*caps shift*” together with “*space*”.

How the program actually works is in principle quite simple. The machine code instruction OUT (FE),A (and incidentally also the BASIC instruction OUT 254,A) is the all important one. Bits two, one and zero of the byte you're outputting must contain the BORDER colour, but it is bit four that does the exciting part. This is the “sound bit”, and when you output it makes a noise. In actual fact a noise is only produced when bit four *changes*, so if you want to produce a continuous note you have to keep changing bit four from one to zero and back again. The speed at which you do this determines the frequency (tone) of the note.

Spectrum music is a fascinating subject, and it is possible to write a program which reads data not from the keyboard but from a long string of bytes listing the note value and the length of note for each note in a tune. I'll

leave that one to you though, because the only real way to learn is by experiment. If you were wondering about the DI instruction in the program by the way, it's actually there to make the speed of operation a bit more precise (and a whole lot faster). Try omitting it, if you like, to see what happens.

We'll leave the subject of music altogether now and turn our attention to something slightly different – pictures...

## **PICTURES**

This is yet another program which relies on the artistic ability of the human operator. The problem is that because the Spectrum uses up so much memory on storing the display file this program is strictly for 48K users only.

The program stores in memory up to five different complete screen pictures, and cycles through them one at a time, displaying each on the screen for as long as you want. A “picture” can be anything you can create on the screen – you can use full colour, high resolution graphics, the lot. You can use the entire character set including the user-defined ones, PLOT, DRAW and CIRCLE. In fact, any image that it is possible to create on the screen of the Spectrum.

The first thing you do is to reserve some memory in which to store these pictures. To do this you must type CLEAR 26623 which will free all memory from address 6800 upwards.

Now you're ready. Write this machine code to address EF05:

		ORG EF05	
210040	STORE	LD HL,D_FILE	Point to start of screen.

110068	LD DE,PICTURE1	Point to address at which to store first picture.
01001 B	LD BC,1 B00	Number of bytes in screen.
EDB0	LDIR	Copy picture.
3A485C	LD A,(BORDCR)	
1F	RRA	
1F	RRA	
1F	RRA	A: = BORDER colour.
12	LD (DE),A	Store BORDER colour.
C9	RET	

Now write a program – BASIC or otherwise – which prints a picture on the screen. You *can* type NEW if you want to because the machine code is above (RAMTOP) and so won't be erased. Typing NEW is not necessary, of course – you can always add some more BASIC to HEXLD 3 (say at line 1000) or even use it to write machine code. The last line of your program should be RANDOMIZE USR 61189.

A useful fact to know is that if you type (as a line of program – not a direct command) POKE 23659,0 then you can print to all twenty-four lines of the screen. Even PRINT AT 23,0;"X" works! If you do this, however, then it is *imperative* that you POKE 23659,2 *before* the program goes back to command mode (either by running out of program, stopping or getting an error message) or the system will crash.

Now, having done all that, change the LD DE,PICTURE1 instruction to LD DE,PICTURE2, come up with a different picture (and RANDOMIZE USR 61189), etc., etc.

The addresses you need for this are:

*Mastering Machine Code on your ZX Spectrum*

PICTURE1	6800
PICTURE2	8301
PICTURE3	9E02
PICTURE4	B903
PICTURE5	D404

Type RANDOMIZE FN R(“FFF6”,“6800”) to set the BEGIN variable of HEXLD 3 to 6800 – the first address to SAVE – and then RUN 400 should SAVE the whole lot; be warned, it may take some time.

For the first time in this book we are going to make use of the PAUSE facility. The instruction CALL PAUSE\_BC will display the TV picture for BC frames, or until a key is pressed. Enter this machine code program to address EF18:

0605	PICTURES	LD B,number of pictures.	
210068		LD HL,PICTURE1	
C5	NEXT_PIC	PUSH BC	
01001B		LD BC,1 B00	
110040		LD DE,D_FILE	Point DE to start of screen.
EDB0		LDIR	Copy next picture.
7E		LD A,(HL)	A: = BORDER colour.
D3FE		OUT (FE),A	Change BORDER colour.
23		INC HL	Point to next picture.
E5		PUSH HL	
016400		LD BC, length of pause.	
CD3D1F		CALL PAUSE_BC	Pause for required number of frames.
E1		POP HL	
C1		POP BC	

```

10E8          DJNZ NEXT_PIC
C9           RET

```

Did you notice the new instruction OUT (FE),A by the way. This instruction will change the BORDER colour to the value of A. If A is greater than seven then only the three rightmost bits will be used. Incidentally, the instruction OUT (C),r will do exactly the same job providing that C contains FE. When OUT is used to change the BORDER colour the change is not permanent. To change the BORDER colour permanently to A you must use instructions to the effect of:

```

D3FE          OUT (FE),A          Change BORDER
                                         Colour
                                         immediately.

17           RLA
17           RLA
17           RLA          Multiply by eight.
E638          AND 38          Ignore bits 7 and 6,
                                         and set ink black.

CB6F          BIT 5,A          Does colour have
                                         a green
                                         component?

2002          JR Z,N_BORD      Jump if so.
F607          OR 07           Set ink white.
32485C        N_BORD         LD (BORDCR),A          Store new
                                         BORDER colour.

```

## LIFE

In the last program in this chapter, we turn the tables slightly. We humans have had the monopoly on art for long enough – now it's time to let the computers take their turn...

This program is called LIFE – it is supposed to represent the birth/growth/death cycle of a colony of cells living on a square grid. It produces rather fascinating results. Before your very eyes you see a

constantly evolving pattern – starting off totally random and finishing sometimes with the ultimate death of the cell colony, sometimes with a fixed and unmoving cell structure which has reached equilibrium, and sometimes with a continuing moving repeating sequence, called dynamic equilibrium. It really is amazing to watch.

LIFE was invented in 1970 by a man called John Conway of Cambridge University, and it's rather surprising that the Tate Gallery hasn't yet got a copy of it. Although it is in fact about the growth of cells which follow hard and fast mathematical rules it, in reality, becomes a rather effective pattern-generating algorithm.

The principle of LIFE is very simple. A grid – usually a square – is dotted with about a quarter of its available squares filled with cells. In this program these positions are chosen entirely at random. This configuration of the grid is called *generation zero*.

Successive generations are then worked out by a fairly simple to understand principle. Every square on the grid has eight surrounding squares. Each of these surrounding squares will either contain another cell or will be empty. Every cell which has precisely either two or three neighbouring cells will survive to the next generation, and every other cell will die. A new cell, however, will be born in any empty square which has precisely three neighbouring cells. With these fairly simple rules, it is rather surprising that the program should produce the rather elegant results that it does.

In this version of LIFE our grid is sixteen by sixteen because, of course, sixteen is a fairly easy number to work with in hexadecimal. Further, our grid is rather strangely constructed in some curved space continuum, so that every square on the left-hand edge is connected to a square in the right-hand edge, and *vice versa*, and also every square on the top edge is connected to the corresponding square on the bottom edge, and *vice versa*.

The surrounding BASIC is:

```
10 RAND USR START
20 RAND USR NEXT_GEN
30 GO TO 20
9999 LOAD "LIFE" CODE
```

which should be saved using SAVE "LIFE" LINE 9999. *START* and *NEXT\_GEN* are addresses in the machine code program. In this listing we assume that the very first address used is 6800, and that the address of the label *START* is hex 6908. You can easily change these if you wish.

	DUMP	ORG 6800 DEFS 100h	A working space in which to calculate the next generation.
EF010110	TABLE	DEFB EF 01 01 10	Data representing the displacements to the neighbouring squares.
10FFFFFF0 AF 323C5C	START	DEFB 10 FF FF F0 XOR A LD (TVFLAG),A	Send print output to upper part of screen.
328F5C		LD (ATTR_T),A	Temporarily set PAPER and INK both to black.
0E10 0610 3E4F D7	BLACK_1 BLACK_2	LD C,10 LD B,10 LD A,"O" RST 10	Print next cell position.
10FB		DJNZ BLACK_2	



*Mastering Machine Code on your ZX Spectrum*

3E0D		LD A, "enter"	
D7		RST 10	End of row.
0D		DEC C	
20F3		JR NZ, BLACK_1	
210058		LD HL, ATTRS	Point HL to attributes file.
0E10		LD C, 10	
0610	CELL_1	LD B, 10	
E5	CELL_2	PUSH HL	
2A765C		LD HL, (SEED)	HL: = random number seed.
54		LD D, H	
5D		LD E, L	DE: = random number seed.
29		ADD HL, HL	
29		ADD HL, HL	
19		ADD HL, DE	
29		ADD HL, HL	
29		ADD HL, HL	
29		ADD HL, HL	
19		ADD HL, DE	HL: = new random number.
22765C		LD (SEED), HL	
7C		LD A, H	
FEC4		CP C4	
3804		JR C, BLACK	Decide whether or not to place cell.
3E07		LD A, 07	
1801		JR C_1	
AF	BLACK	XOR A	
E1	C_1	POP HL	HL: = points to attribute position of cell.
77		LD (HL), A	Print cell if it exists.

23		INC HL	Points to next attribute position.
10E1		DJNZ CELL_2	
111000		LD DE,0010	
19		ADD HL,DE	Point to first cell of next row.
0D		DEC C	
20D8		JR NZ,CELL_1	
C9		RET	
210058	NEXT_GEN	LD HL,ATTRS	Point HL to attributes file.
110068		LD DE,DUMP	Point DE to grid copy.
E5		PUSH HL	Stack the constant 580 0.
D5		PUSH DE	Stack the address of DUMP.
011000		LD BC,0010	Number of squares in each row.
79		LD A,C	Number of rows.
C5	COPY_G	PUSH BC	
EDB0		LDIR	Copy one row into dump area.
C1		POP BC	BC: = 0010.
09		ADD HL,BC	Point HL to next row in attributes file.
3D		DEC A	
20F8		JR NZ,COPY_G	Copy whole grid into dump area.
E1		POP HL	HL: = address of dump
D1		POP DE	DE: = 5800.
D5		PUSH DE	
E5		PUSH HL	

*Mastering Machine Code on your ZX Spectrum*

0E00	NXT_CELL	LD C,00	C counts number of neighbouring cells.
110069		LD DE, TABLE	DE: = points to displacement table
0608		LD B,08	
1A	NXT_DIS	LDA,(DE)	A . = next displacement.
13		INC DE	Point to next displacement in table.
85		ADD A,L	
6F		LD L,A	HL: = points to next neighbouring square.
7E		LD A,(HL)	A: = contents of this square.
A7		AND A	Does this square contain a cell?
2801		JR Z,COUNTED	
0C		INC C	Increment count if so.
10F5	COUNTED	DJNZ NXT_DIS	Repeat for all neighbours.
E1		POP HL	HL: points to current square within dump.
79		LD A,C	A: = number of neighbouring cells.
FE02		CP 02	
380F		JR C,NO_CELL	Cell dies if one or no neighbours.
FE04		CP 04	
300B		JR NC,NO_CELL	Cell dies if four or more neighbours.

FE03 2803		CP 03 JR Z,CELL	Cell is created if exactly three neighbours.
7E		LD A,(HL)	A: = cell or no cell at present.
1805 3E07	CELL	JR PUT LD A,07	07 means "white on black" as an attribute.
1801 AF	NO_CELL	JR PUT XOR A	00 means "black on black" as an attribute.
E3	PUT	EX (SP),HL	HL points into attributes file.
77		LD (HL),A	Print cell or otherwise.
23		INC HL	Point to next square.
7D E61F FE10 2004 111000 19		LD A,L AND 1F CP 10 JR NZ,ROW LD DE,0010 ADD HL,DE	Point HL to start of next row if needed.
E3	ROW	EX (SP),HL	HL points into dump.
23		INC HL	Point to next square.
E5 7D A7		PUSH HL LD A,L AND A	

*Mastering Machine Code on your ZX Spectrum*

20C3	JR NZ,NXT_CELL	Repeat until every cell considered.
E1	POP HL	
E1	POP HL	Balance the stack.
C9	RET	Return to BASIC.

If you used the same addresses as in the listing then START is 26888 and NEXT\_GEN is 26956.

The first thing you should do is type RANDOMIZE, because the program makes use of the random number seed. You may now type RUN.

An interesting point about this program is that it is capable of producing its own random numbers. The part just after the label CELL 2 does this. You should study how this is achieved, and by all means you are quite welcome to use the same principle in your own programs.

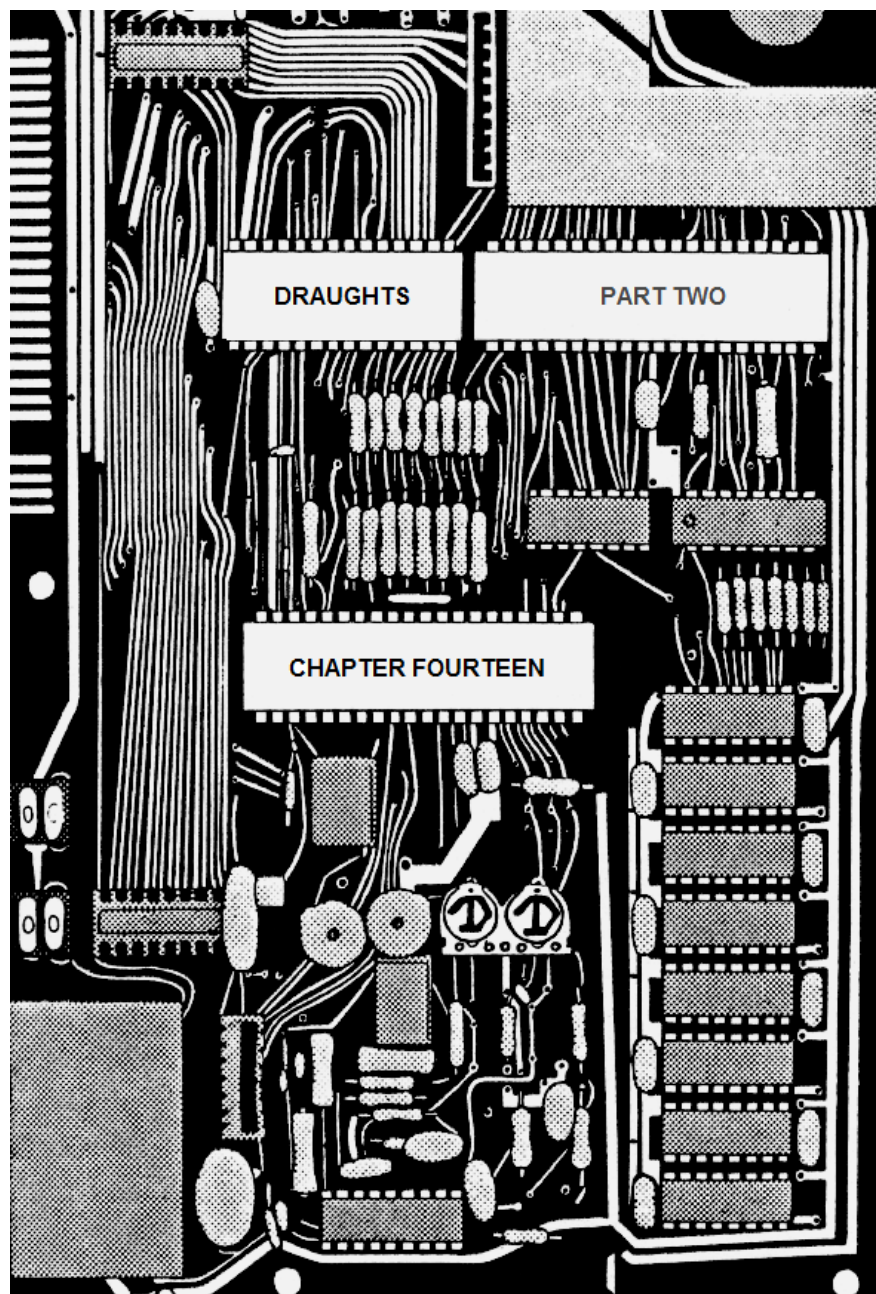
LIFE will print out a randomly constructed generation zero more or less instantly, and the successive generations are produced at the staggering rate of fifteen generations per second! You can slow this down by adding a BASIC PAUSE statement if you wish. Another way is to add a LET X = 0/LET X = X + 1/PRINT AT 17,0; "Generation ";X loop – this has the added advantage of telling you how many generations have been displayed.

Finally, you should follow the manner in which this program, unlike some other LIFE programs I have seen, calculates each new generation entirely on the basis of the previous one. It does not work out the first row and then calculate the second row by counting the neighbours in the now changed new first row, the second row is determined by the *previous* status of the first row. (This is what the block of memory called DUMP in the machine code listing is for.) Each new generation then is *correctly* set up.

There are many other pattern-generating programs, some much simpler, but none with the mathematical elegance of LIFE. If you feel like a challenge you might try writing a twenty-four by twenty-four LIFE, or even a

twenty-four by thirty-two version. You can use RST 10 on the bottom two lines provided that you load the system variable DF\_SZ with zero beforehand, and with two as soon as the printing is done.

The biggest LIFE you could possibly hope to achieve on a Spectrum is 256d by 192d using white pixels for cells and black pixels for empty squares, but that would be quite a complicated program. If you feel really enthusiastic you might like to have a bash at this monumental task. I will let that decision rest with your sanity.



DRAUGHTS

PART TWO

CHAPTER FOURTEEN

## CHAPTER FOURTEEN

### DRAUGHTS PART TWO

This is the routine which chooses a move for the computer to make, after the human's move.

You may have to follow this that we are about to embark upon very carefully. Here in brief is a systematic breakdown of the way in which the move is chosen.

We scan the board one (black) square at a time and whenever we find a computer's piece, we sit and think about it for a bit. To each move we find possible we assign a numerical value, such that the bigger the number, the better we think the move is. It then follows that to select a move we merely have to choose the one with the highest possible value.

Of course, this idea won't let the computer plan ahead – it can only think one move at a time. In order to construct this list of moves and accompanying numerical values, we don't actually have to store every single move we find. Having located a possible move, and worked out its score, what happens is this:

If the score is *lower* than those on the list then it is ignored.

If the score is *equal* to those on the list then it is added to the list.

If the score is *higher* than those on the list, then the list is abolished and a new one started.

In this way the list is always as short as it possibly can be. When the final decision time actually arrives the computer now merely has to select one of these moves at random. Next question – where will the list be stored? Answer – *the stack*. This simplifies things, but it does mean that we must keep a record of where the start of the list is. We shall store this at address



5CB0 (the system variable SPARE2 – two bytes not used). This quantity we shall call L\_BASE (the base of the list). It is always quite safe to store things in this system variable since, as the manual quite rightly says, this is not used by the ROM.

The decision making of the computer starts at address 6AE7. The first instruction is LD (L\_BASE),SP. The start of the list is now preserved. We can play around with the stack now as much as we like, as long as we remember to restore its value before we return to BASIC. The second and third instructions stack the number zero onto our list, which will indicate to the program that the list is currently empty.

The checking loop looks like this. Notice that two new variables: L\_COUNT (which counts the number of entries in the list) and P\_COUNT (which counts the number of pieces which have been considered) are used. These are both stored amongst the system variable MEMBOT – ordinarily the Spectrum's calculator memories. Again, this is usually quite a good place to store numbers. It is quite safe to use in any machine code program which does not require the use of floating point numbers. The routine, and in fact all subsequent routines in this chapter, is to be *inserted*, because the board printing routine must be at the end. The address 6AE7 to which this routine must be inserted, in fact, places the routine immediately before this board print.

ED73	BD_SCAN.	LD (L_BASE),SP	Store the start of the list.
B05C			
210000		LD HL,0000	
E5		PUSH HL	Initialise the list.
22925C		LD (P_COUNT), HL	Initialise count of number of pieces, and count of number of entries in list.

210668		LD HL,BOARD _2 + 6	Point HL to first black square on board.
7E	NXT_CHK	LD A,(HL)	A: = contents of square.
E67F		AND 7F	Disregard king status.
FE07		CP 07	Have we found a computer's piece?
CC5200		CALL Z,EVALUATE	(This line is temporary).
23		INC HL	Point to next square.
7D		LD A,L	
FE30		CP 30	Repeat for all squares on board.
20F2		JR NZ,NXT_CHK	Empty the list to prevent crashing.
ED7BB05C		LD SP,(L_BASE)	

As you can see, this particular bit is quite straightforward, except that it calls a subroutine which we haven't written yet – EVALUATE – whose job it will be to compute all possible moves from that square. Because we haven't written it yet I've changed the Hex of that instruction to translate as CALL Z,0052 which is an address in the ROM at which a RET instruction lies – thus making the instruction temporarily pointless, although it does remind us what is supposed to be going on. The final instruction is not really a part of things, but is just there to prevent crashing. Can you see how loading SP with (L\_BASE) eliminates the need to POP everything from the stack before returning? Loading SP will fool the machine into thinking that the stack hasn't changed since the first instruction in the routine.

Notice also that the instruction LD (P\_COUNT),HL will assign both P\_COUNT and L\_COUNT, since these are one byte variables, and HL

contains two bytes. This, of course, would only work where L\_COUNT and P\_COUNT are stored at adjacent addresses.

Now we need to think about what form we want the list to take. Let's examine the problem in reverse. What form would we *like* the list to take in order to make examining the stack easier?

The first item on the stack should be the number of steps involved in the move – that is one for a single move/jump, two for a double jump, three for a triple jump, and so on. The second item should be the numerical value which the items in the list have been assigned – the *priority*, as we shall call it. Following these items of information should be the list itself, starting with the square to be moved from, followed by a sequence of one or more directions in which to be moved. Immediately after this the second item in the list in the same form, then the third, and so on...

You'll notice that each thing we need on the stack will only need to be one byte in length. The number of steps cannot possibly be more than FF. The priority can be chosen however we like – we can always make it one byte if we wish. The initial square can be stored by stacking only the *low* part of its address in BOARD\_2. The directions to be moved can be stored in the same manner as before – 05, 06, FA or FB for plus or minus five or six. In order to make the program as space efficient as we can, it makes sense to do just that.

Imagine that the list is now complete, and we are about to select an item from it. The stack now looks like the diagram, Figure 14.1(A). If we now use the instruction POP BC, B will contain the priority and C the number of steps. The priority is now a redundant piece of information, since it was only needed to construct the list in the first place. C, however, is very important. In the diagram of Figure 14.1(A), C would be one but in other cases it may not be. The stack would now look as in Figure 14.2(B).

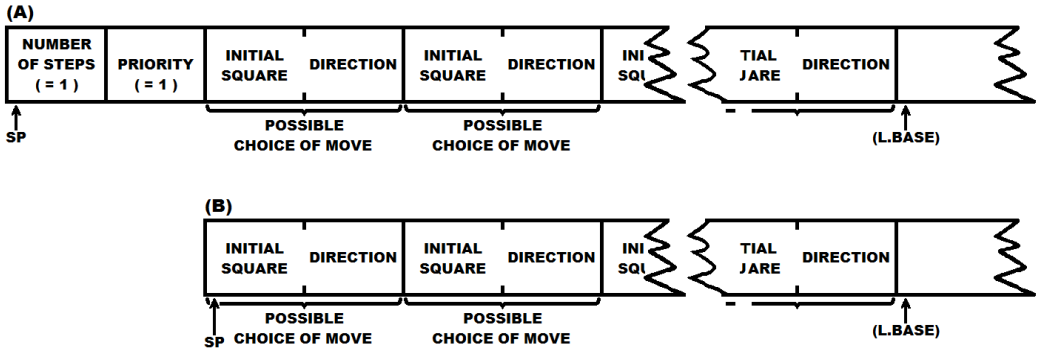


Figure 14.1

Suppose instead that there were *two* steps to each move in the list, not one. In this case the stack would be as in Figure 14.2.

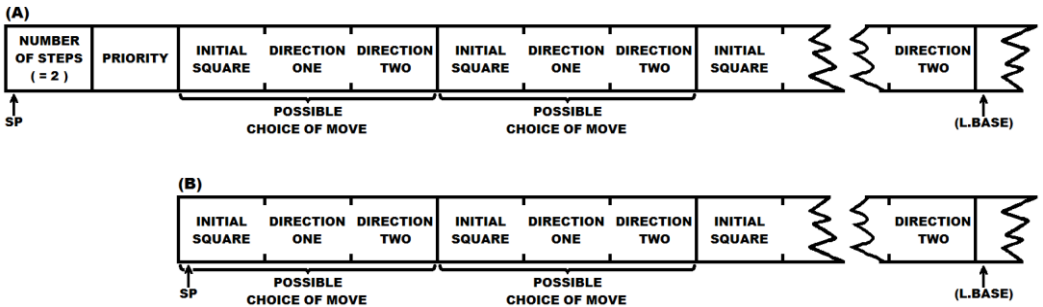


Figure 14.2

What we must do then is to remove a random number of items from the stack, so that the move we finally choose is left at the top. The following routine will do just that. It should be inserted to address 6B03.

C1	CHOOSE	POP BC	C: = number of steps in move sequence.
3A935C		LD A,(L_COUNT)	A: = number of

*Mastering Machine Code on your ZX Spectrum*

A7		AND A	possible choices.
2812		JR Z,CHOICE	Jump if there are no choices.
5F		LD E,A	
3A785C		LD A,(FRAMES)low	A: = random numbers between 00 and FF.
93	RANDOM	SUB A,E	
30FD		JR NC,RANDOM	
83		ADD A,E	A: = random number between 00 and E - 1.
2808		JR Z,CHOICE	If zero then do nothing.
41	NSQ_OFF	LD B,C	B: = number of steps in move sequence.
33		INC SP	Delete "initial square" from stack.
33	NDR_OFF	INC SP	Delete "next direction" from stack.
10FD		DJNZ NDR_OFF	Delete whole of move sequence.
3D		DEC A	
20F8		JR NZ,NSQ_OFF	Delete 'A' items in all.
	CHOICE	\$	

The selected move is now at the top of the stack. To carry it out let's take a look at what the stack is like now. Figure 14.3 shows this.

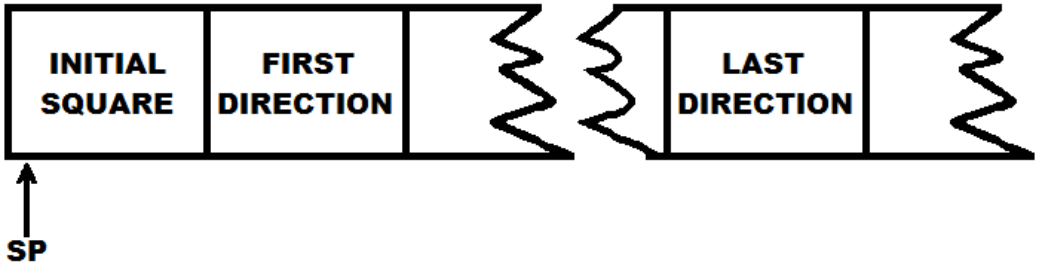


Figure 14.3

To find the initial square the sequence is DEC SP/ POP AF/LD L,A/LD H,68. You see “initial square” is the *low* part of the address – the high part being 68. It is very important that the stack pointer (SP) never points any higher in memory than the last item on the stack. In the diagram Figure 14.3, this means that SP is allowed no further to the right of the picture than the very topmost item (ie where it is in the diagram). We can move it to the left (with DEC SP or PUSH) quite happily, but we may not move it to the right – not even temporarily – except by using POP. This is because whenever the Spectrum receives an *interrupt* (which happens fifty times a second) several items at once will instantaneously be PUSHed onto the stack and then POPped back again, so that anything to the left of the immediate position of SP will be erased. An interrupt may occur between any two instructions. Of course, there’s nothing to stop us “cheating” by simply disabling the interrupts (and in fact if we did this our program would run faster) but this is not considered to be “good” programming. Don’t ask why – I don’t know! Anyway, I’ll stick with professional opinion and be very careful with the stack pointer.

DEC SP/POP AF then will load the topmost byte on the stack into the A register without endangering the list. All that is required now is for the move to be printed and carried out. Insert this next part to address 6B1C. Note that there are no checks this time round, because we are assuming that the computer cannot cheat.

```
CD3768
```

```
CALL S_PRINT
```

*Mastering Machine Code on your ZX Spectrum*

16030B		DEFM AT 3, 11d	
4D79206D6F766520		DEFM My move space	
00		DEFB 00	
3B		DEC SP	
F1		POP AF	A: = initial square.
F5		PUSH AF	
87		ADDA,A	Multiply by two to take white squares into account.
3C		INC A	Take into account the fact that the first black square is 12 and not 11.
11FF0B		LD DE,0BFF	D: = number of squares across board map (see Figure 12.2),
1C	MOD	INC E	E: = - 1.
92		SUB D	E counts rows.
30FC		JR NC,MOD	E: = first co-ordinate of square.
82		ADD A,D	A: = second co-ordinate of given square.
F5		PUSH AF	
7B		LD A,E	
C630		ADD A,30	Convert to ASCII character.
D7		RST 10	Print first co-ordinate.
F1		POP AF	A: = second co-ordinate.

C630		ADD A,30	Convert to ASCII character.
D7		RST 10	Print second digit.
F1		POP AF	A: = square number (Figure 12.2).
6F		LD L,A	
2668		LD H,68	HL: = points to initial square in BOARD_2.
41		LD B,C	B:= number of steps in move sequence.
4E	NXT_STEP	LD C,(HL)	C: = computer's piece.
3600		LD (HL),00	Erase piece from board.
3B		DEC SP	
F1		POP AF	A: = next direction: FA, FB, 06 or 05.
F5		PUSH AF	FA for direction "A".
1647		LD D,47	FB for direction "B", etc.
CB7F		BIT 7,A	
2004		JRNZ,DIR_CHR	Jump if direction FA or FB.
1649		LD D,49	
ED44		NEG	A: = FA for direction "C", FB for direction "D".
82	DIR_CHR	ADD A,D	A: = 41, 42, 43 or 44.
D7		RST 10	Print direction.
F1		POP AF	A: = direction: FA, FB, 06 or 05.



*Mastering Machine Code on your ZX Spectrum*

57		LD D,A	
85		ADD A,L	
6F		LD L,A	HL: = destination square.
7E		LD A,(HL)	A: = contents of square.
A7		AND A	
2805		JR Z,SQUARE	Is this a move or a jump?
3600		LD (HL),00	Erase "taken" piece.
7D		LD A,L	
82		ADD A,D	
6F		LD L,A	HL: = new destination square.
71	SQUARE	LD (HL),C	Place computer's piece at new position.
10DE		DJNZ NXT_STEP	
	M_DONE	\$	

Now, whatever you do, don't try to run the program just yet, for it now has one little bug which we must cure. In the section labelled CHOOSE (the last but one section) control at present jumps to the label CHOICE if (L\_COUNT) is zero. (This came about because at the time we wrote it in, its true destination did not exist.) It should instead jump to the label M\_DONE in the last section. The following sequence of inputs will cure this:

RUN 100/6B09/5E

As things stand you can't actually test this routine, or any routine in this chapter come to that, because the subroutine EVALUATE doesn't exist. So what I suggest is to write a "mock" EVALUATE subroutine, which sets up the stack as we want it set up, but which is not a generally working

subroutine. Using WRITE (RUN 100) place this “pretend” subroutine at address 7CBA (16K) or FCBA (48K). This is a rather inconsistent place for it – it is in fact the address of user defined graphic B. (Remember that HEXLD 3 saves the user defined graphics as well.) Graphic A is already used up (a draughts piece) but from “B” onwards we’re OK. Ready when you are:

C1	EVALUATE POP BC	BC: = subroutine return address.
	_MOCK	
111106	LD DE,0611	
D5	PUSH DE	Stack move 32C.
15	DEC D	
D5	PUSH DE	Stack move 32D.
1C	INC E	
D5	PUSH DE	Stack move 34D.
14	INC D	
D5	PUSH DE	Stack move 34C.
1C	INC E	
D5	PUSH DE	Stack move 36C.
15	DEC D	
D5	PUSH DE	Stack move 36D.
1C	INC E	
D5	PUSH DE	Stack move 38D.
110101	LD DE,0101	
D5	PUSH DE	Stack priority and number of steps per move sequence.
110C07	LD DE,070C	
ED53925C	LD (P_COUNT),DE	(P_COUNT) = 0D (Twelve pieces on board). (L_COUNT) = 07 (Choice of seven moves).

Mastering Machine Code on your ZX Spectrum

C5	PUSH BC	Restack return Address.
C9	RET	

Now type RANDOMIZE FN R (“FFFE”, “6BA8”) (48K) or RANDOMIZE FN R (“7FFE”, “6BA8”) (16K) to reset LIMIT.

Typing RUN 100/6AFB/BA7C changes the address in the CALL Z, EVALUATE instruction, so you can now see the program working in all its glory. Note that only the *first* move of the computer is at present programmed correctly, so you can only “play” the game for one move. The *real* EVALUATE subroutine is something we shall concentrate on in DRAUGHTS Part Three. In the meantime, we still have that end of game check to worry about. I’ll give you this routine without any additional explanation. See if you can work it out for yourself. Write this to address 6BA7 – the very end of the program:

3A925C	GAME-OVER	LDA,(P_COUNT)	A:= number of
A7		AND A	computer’s pieces.
2036		JR NZ,GO_2	
CD3768		CALL S_PRINT	
16020B		DEFM AT 2, 11d	
1107		DEFM <i>paper white</i>	
1201		DEFM <i>flash on</i>	
20434F4E4752			
4154554C4154			
494F4E53202D20		DEFM <i>space</i> CONGRATULATIONS <i>space – space</i>	
16030B		DEFM AT 3, 11d	
20594F5520		DEFM <i>space</i> YOU <i>space</i>	
4841564520		DEFM HAVE <i>space</i>	
574F4E212121202020		DEFM WON!!! <i>three spaces</i>	
00		DEFB 00	
CF		RST 08	

FF		DEFB FF	Return to BASIC command mode.
210668	GO_2	LD HL,BOARD_2 + 6	Point HL to first square.
062A 7E	GO_LOOP	LD B,2A LD, A,(HL)	A: = contents of square.
23		INC HL	Point to next square.
E67F		AND 7F	Disregard king Status.
FE02		CP 02	Is this a human piece?
C8		RET Z	Return to BASIC if so.
10F7 CD3768 16020B 1107 1201 2042414420 4C55434B20 2D20 16030B 204920 4841564520 574F4E2120 00 CF FF		DJNZ GO_LOOP CALL S_PRINT DEFM AT 2, 11d DEFM <i>paper white</i> DEFM <i>flash on</i> DEFM <i>space BAD space</i> DEFM LUCK <i>space</i> DEFM – <i>space</i> DEFMAT 3, 11d DEFM <i>space   space</i> DEFM HAVE <i>space</i> DEFM WON! <i>space</i> DEFB 00 RST 08 DEFB FF	Return to BASIC command mode.

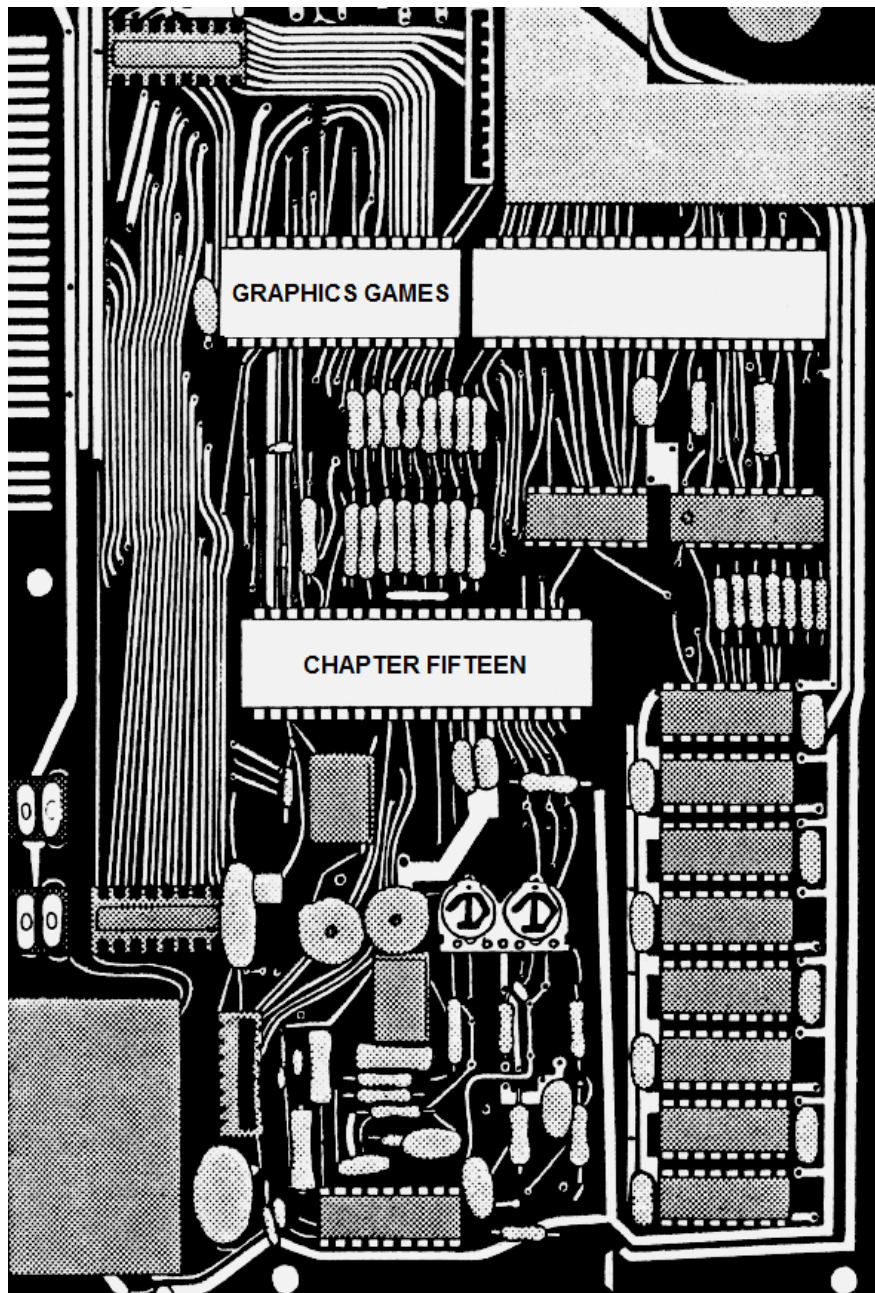
We ended Part One with a routine which checked whether any human piece became a king. I'd like to continue in the same tradition now with a

*Mastering Machine Code on your ZX Spectrum*

routine which does the same for the computer's piece. The routine must be inserted directly before the board printing routine – at address 6B6C:

212C68		LD HL,BOARD_2+2C	Point to first square of last row.
0604		LD B,04	
7E	W_KING	LD A,(HL)	A: = contents of square.
FE07		CP 07	Have we found a computer's piece?
2002		JR NZ,WK_2	
3687		LD (HL),87	Change piece to a king.
23	WK_2	INC HL	
10F6		DJNZ W_KING	

In the next chapter we'll look at some complete (and short) games designed to demonstrate something of what machine code can achieve in terms of speed, and in actually very few bytes compared with BASIC.



# CHAPTER FIFTEEN – GRAPHICS GAMES

## SPIRALS

In this fast moving real-time graphics game you are placed at the start of a square spiral and must reach the end of it in the minimum possible time. Your score is constantly displayed – it starts off at 99900 and decrements continuously, but you can't cheat by breaking out early with a high score as the program won't allow that. Now and again, the score will reach zero before you reach the end of the spiral. If that happens you obviously need more practice!

This is a fascinating game to watch – witnessing the score decrease before your very eyes is surprisingly effective. You can make the game as difficult as you like by altering the initial value of the “timing” – held in BC. I've given it 0800, but you could use 1000 for a slower game, 0400 for a faster game, and so on.

There is one difficulty built in, though – if you hit a wall you don't just bounce off, you actually become embedded in it, and the only way you can get out is to actually *reverse* your direction. It can be quite tricky.

Well good luck on your race – keep a record of the high scores (no cheating) and see if you can master it.

The keys will move you as follows: “I” to move upwards, “J” to move leftwards, “K” to move rightwards, and “M” to move downwards. These keys form a kind of “cross” on the keyboard, and so it is very easy to remember which key does what. I prefer using these keys as directional keys rather than the cursor keys

“5”, “6”, “7” and “8” for this reason. It is sometimes awfully hard to remember which of “6” and “7” is up and which is down.

The program lists as follows, and is designed to sit at address 6800. It may, however, be relocated to any desired address providing that every absolute address used in the program is also changed accordingly. The program should be called from the point labelled START.

E0FF	SPIRAL	DEFW 1111 1111 1110 0000 b	This data represents the shape of the square spiral, with ones representing walls, and zeros representing spaces.
20A0		DEFW 1010 0000 0010 0000 b	
A0AF		DEFW 1010 1111 1010 0000 b	
A0A8		DEFW 1010 1000 1010 0000	
A0AA		DEFW 1010 1010 1010 0000 b	
A0AE		DEFW 1010 1110 1010 0000 b	
A0A0		DEFW 1010 0000 1010 0000 b	
A0BF		DEFW 1011 1111 1010 0000 b	
2080		DEFW 1000 0000 0010 0000 b	
E0FF		DEFW 1111 1111 1110 0000 b	
0D	SCORE_I	DEFM <i>enter</i>	This data represents the message to be printing giving your current score.
596F757220		DEFM <i>Your space</i>	
73636F726520		DEFM <i>score space</i>	
6E6F7720		DEFM <i>now space</i>	
3939393030		DEFM 99900	
160B0F	SCORE_C	DEFM AT 11d,15d;	This data will be used to represent the current score.
303030		DEFM 000	
call here:			
AF	START	XOR A	Print to upper part of screen.
323C5C		LD (TVFLAG),A	
060A		LD B,0A	Point to spiral data.
210068		LD HL,SPIRAL	



*Mastering Machine Code on your ZX Spectrum*

5E	S_LOOP	LD E,(HL)	
23		INC HL	
56		LD D,(HL)	DE: = next word of data.
23		INC HL	
EB		EX DE,HL	HL: = next word of data.
29	R_LOOP	ADD HL,HL	
3806		JR C,WALL	
FD36553F		LD (ATTR_T),3F	Set colours white on white.
1804		JR PR_SQ	
FD365500	WALL	LD (ATTR_T),00	Set colours black on black.
3E2B	PR_SQ	LD A, " + "	
D7		RST 10	Print as required.
7C		LD A,H	
B5		OR L	
20EC		JR NZ,R_LOOP	Print whole row.
3E0D		LD A, "enter"	
D7		RST 10	Prepare for next row.
EB		EX DE,HL	HL: = points to next word of data.
10E1		DJNZ S_LOOP	Print whole structure.
3E38		LD A,38	A represents the colour scheme black on white.
322158		LD (ATTRS + 21),A	Print cross at starting position.
FD365530		LD (ATTR_T),30	Attribute for black on yellow.
211468		LD HL,SCORE_I	Point HL to initial score data.

0615		LD B,15	
7E	SC_LOOP	LD A,(HL)	
23		INC HL	
D7		RST 10	Print initial score.
10FB		DJNZ SC_LOOP	
213939		LD HL,3939	
222C68		LD (SCORE_C + 3),HL	
222D68		LD (SCORE_C + 4),HL	Set current score to 999(00).
212158		LD HL,ATTRS + 21	Point HL to current position.
22925C		LD (POSITION),HL	Store current position.
210000		LD HL,0000	
22945C		LD (LAST_MOVE),HL	
212E68	LOOP	LD HL,SCORE_C + 5	Point HL to hundreds digit of score.
7E	DECIMAL	LD A,(HL)	
FE0F		CP 0F	
2008		JR NZ,POSITIVE	
0603		LD B,03	
23	RESET	INC HL	
3630		LD (HL),"0"	
10FB		DJNZ RESET	
C9		RET	
3D	POSITIVE	DEC A	Decrement the score.
FE2F		CP 2F	
2005		JR NZ,OK	
3639		LD (HL),"9"	
2B		DEC HL	
18E9		JR DECIMAL	
77	OK	LD (HL),A	

*Mastering Machine Code on your ZX Spectrum*

212968		LD HL,SCORE_C	Point HL to current score data.
0606		LD B,06	
7E	CS_LOOP	LD A,(HL)	
23		INC HL	
D7		RST 10	Print the current score.
10FB		DJNZ CS_LOOP	
010008		LD BC,0800	A timed delay. Altering the initial value of BC changes the speed of the game.
0B	DELAY	DEC BC	
78		LD A, B	
B1		OR C	
20FB		JR NZ,DELAY	
CD8E02		CALL KEY_SCAN	Scan keyboard. A: = which key has been pressed.
7B		LD A,E	
FE09		CP 09	Check for "J" key.
2811		JR Z,LEFT	
FE10		CP 10	Check for "M" key.
2812		JR Z,DOWN	
FE11		CP 11	Check for "K" key.
2813		JR Z,RIGHT	
FE12		CP12	Check for "I" KEY.
20BF		JR NZ,LOOP	
11E0FF	UP	LD DE,FFE0	
180D		JR MOVE	
11FFFF	LEFT	LD DE,FFFF	
1808		JR MOVE	

112000	DOWN	LD DE,0020	
1803		JR MOVE	
110100	RIGHT	LD DE,0001	
2A945C	MOVE	LD HL,(LAST_MOVE)	Is the player embedded in the wall?
7C		LD A,H	
B5		OR L	
2805		JR Z,MOVE_OK	
19		ADD HL,DE	If so, is the player reversing?
7C		LD A,H	
B5		OR L	
20A1		JR NZ,LOOP	
2A925C	MOVE_OK	LD HL,(POSITION)	
7E		LD A,(HL)	
EE07		XOR 07	Reassign square with black or white square as required.
77		LD (HL),A	Find new position.
19		ADD HL,DE	
7E		LD A,(HL)	
EE07		XOR 07	Draw black or white cross as appropriate.
77		LD (HL),A	Store new position.
22925C		LD (POSITION),HL	
210000		LD HL,0000	
FE38		CP 38	Have we hit a wall?
2802		JR Z,FINISH	Jump if not.
62		LD H,D	
6B		LD L,E	
22945C	FINISH	LD (LAST_MOVE),HL	
2A925C		LD HL,(POSITION)	

### Mastering Machine Code on your ZX Spectrum

118558	LD DE,ATTRS + 85	
ED52	SBC HL,DE	Check whether or
C8	RET Z	not the finishing
		square has been
		reached.
C37F68	JP LOOP	

Note that the last but two instruction SBC HL,DE relies upon the fact that the carry flag is already reset from the CP 38 instruction earlier. It is, of course, essential that the carry flag is reset before an SBC instruction is used if we want to carry out a true subtraction.

## BREAKOUT

In this version of BREAKOUT you move the bat to the left with the *caps shift* key, and right with the *space* key. In many ways it uses the same techniques that we have been using until now – filling the screen with the same character but then hiding that character by clever manipulation of the attributes file. In this game, however, there are two symbols needed – that for a bat and that for a ball, and also we need to show different coloured bricks and a current score. In this program the trick is that the entire screen is filled with “ball” symbols which are then “hidden” by making PAPER and INK the same colour at each square – white for an empty square, black for a wall, or any other colour for a brick.

Each different coloured brick will give you a different number of points – one for blue, two for red, and so on. The points value of each colour is the key above which the name of that colour appears.

BREAKOUT is a game in four parts: (i) initialise everything for a new game, (ii) re-start the game for a new ball, (iii) move the ball, and (iv) move the bat, if needed. We shall now go over each of these steps in scrutinous detail.

Firstly to initialise everything. This involves (a) printing the playing board, (b) defining the initial ball position, and (c) setting the initial speed of the game and the number of lives left, etc. First, we must look at how to print the board.

We have a subroutine whose job it is to print a row of bricks in two different colours:

		ORG 6800	
0608	BRICKS	LD B,08	
77	BR_LOOP	LD (HL),A	
23		INC HL	
77		LD (HL),A	Print brick in first colour.
23		INC HL	
C609		ADD A,09	Change colour.
77		LD (HL),A	
23		INC HL	
77		LD (HL),A	Print brick in second colour.
23		INC HL	
D609		SUB 09	Revert to original colour.
10F2		DJNZ BR_LOOP	
C9		RET	

And then we have some data for the use of the program.

16001A	SCORE	DEFM AT 0,26d;	Data for PRINT SCORE routine later on.
1007		DEFM <i>ink white</i>	
1100		DEFM <i>paper black</i>	
0000000000		DEFS 05	Space in which to store current score.

*Mastering Machine Code on your ZX Spectrum*

And one last subroutine – this is called AT\_15\_A which simulates PRINT AT 15h,A;

F5	AT_15_A	PUSH AF	
F5		PUSH AF	
3E16		LD A,“AT control”	
D7		RST 10	
3E15		LD A,15	First AT co-ordinate.
D7		RST 10	
F1		POP AF	Second AT co-ordinate.
D7		RST 10	
F1		POP AF	
C9		RET	

OK – this is where we start – this is the BREAKOUT program itself, and the code required to set up the board:

AF	START	XOR AF	A: = zero, CARRY reset.
F5		PUSH AF	Stack NO CARRY.
323C5C		LD (TVFLAG),A	Sent PRINT to upper part of screen.
326B5C		LD (DF_SZ),A	Extend upper part of screen to 24d lines.
01DF02		LD BC,02DF	
3E0D		LD A,“enter”	
D7		RST 10	Print from start of second line.
3E90	SET_UP	LD A,“graphic_A”	
D7		RST 10	PRINT “ball” symbol.
0B		DEC BC	
78		LD A,B	
B1		OR C	
20F8		JR NZ,SET_UP	Fill whole screen.

212158		LD HL,ATTRS+21	Point HL to first white square.
363F		LD (HL),3F	Set colours white on white.
112258		LD DE,ATTRS+22	Point to second white square.
01DE02		LD BC,02DE	
EDB0		LDIR	Whiten all required squares.
218158		LD HL,ATTRS+81	Point HL to first brick.
3E09		LD A,09	
CD0068		CALL BRICKS	Print row of blue and red bricks.
3E2D		LD A,2D	
CD0068		CALL BRICKS	Print row of cyan and yellow bricks.
3E1B		LD A,1B	
CD0068		CALL BRICKS	Print row of magenta and green bricks.
3E09		LD A,09	
CD0068		CALL BRICKS	Print row of blue and red bricks.
AF		XOR A	
210058		LD HL,ATTRS	Point to top left-hand corner of wall.
77		LD (HL),A	Black this square.
110158		LD DE,ATTRS+01	Point to second square of wall.
011F00		LD BC,001F	
C5		PUSH BC	
EDB0		LDIR	Blacken remainder of top wall.
D1		POP DE	DE: = 001F
0617		LD B,17	
23	SIDES	INC HL	Point to next square in



*Mastering Machine Code on your ZX Spectrum*

77		LD (HL),A	left wall.
19		ADD HL,DE	Blacken this square.
			Point to next square in
			right wall.
77		LD (HL),A	Blacken this square.
10FA		DJNZ SIDES	
211868		LD HL,SCORE+07	Point to SCORE space
			reserved.
0604		LD B,04	
3620	SC_LOOP	LD (HL), "space"	Blank out any existing
			data.
23		INC HL	
10FB		DJNZ SC_LOOP	
3630		LD (HL), "0"	Reset score to zero.
2E11		LD L,11	Point HL to label
			SCORE.
060C		LD B,0C	
7E	PR_SC	LD A,(HL)	
23		INC HL	
D7		RST 10	Print the score onto the
			screen.
10FB		DJNZ PR_SC	

Notice how in this game the score is stored independently of the screen, and yet it is still stored in decimal, digit by digit. This is because it is easier to read a single character from a fixed memory location than it is to decipher a pixel pattern from eight different fixed memory locations. Finally, to set the ball position, speed, and number of lives, the procedure is:

21605A		LD HL,ATTRS+0260	
22B05C		LD (BALL_INIT),HL	Store the ball's starting
			position.
21000B		LD HL,0B00	

22AE5C	LD (SPEED),HL	Store the initial speed of play.
FD363102	LD (DF_SZ),02	Restore system variable to avoid crashing.
3E09	LD A,09	
32A85C	LD (LIVES),A	Store number of lives plus one.

This is actually all the initialisation we need. You'll notice several things missing – for example, although the ball is located it is not actually printed. The bat is not mentioned at all! The reason is that the bat is re-drawn every time the game is re-started, and so is the ball. Why bother to find the initial position then? Well, in this version, the ball starts off in a slightly different position each time. This ensures that it is possible to wipe out all of the bricks.

The variable SPEED determines how fast the game runs. It controls a delay loop – needed to slow things down because machine code is so fast. In short, the bigger the number, the slower the game and the smaller the number, the faster the game. The speed, however, does not stay constant throughout the game, unlike SPIRALS. In this game, things get faster as time goes on!

Section two of the game does the following tasks: (a) change the initial ball position and set the ball currently at this position, (b) set the initial direction of movement to up-right, (c) print the bat, and at the same time delete any previous bat symbol which may have been there, (d) decrease the number of remaining lives, and exit if this number reaches zero, and (e) give the human player a chance to recover from the last session, since presumably she won't want one ball to leap into the game immediately the last one vanishes. The section is this. Look at the manner in which the bat is printed and the previous bat overwritten.

Mastering Machine Code on your ZX Spectrum

3AA85C	RESTART	LD A,(LIVES)	
3D		DEC A	
2002		JR NZ,L_OK	
F1		POP AF	Balance the stack.
C9		RET	Return to BASIC if no lives left.
32A85C	L_OK	LD (LIVES),A	
2AB05C		LD HL,(BALL_INIT)	
23		INC HL	Change the starting position of the ball.
22B05C		LD (BALL_INIT),HL	
22AC5C		LD (BALL_POS),HL	Set current ball position.
3639		LD ( HL),39	Print the ball (blue on white).
21E1FF		LD HL,FFE1	
22AA5C		LD (DIRECTION),HL	Set initial direction to up-right.
3E03		LD A,03	
32A95C		LD (BAT_POS),A	Set centre of bat to column three.
3E01		LD A,01	
CD1D68		CALL AT_15_A	Print AT 15h,1;.
FD365507		LD (ATTR_T),07	Set colours to white on black.
0605		LD B,05	
3E8C	BAT_LOOP	LD A," <i>graphic shift 3</i> "	
D7		RST 10	Print bat symbol.
10FB		DJNZ BAT_LOOP	
FD36553F		LD (ATTR_T),3F	Set colours to white on white.
0619		LD B,19	
3E90	ERASE	LD A, " <i>graphic A</i> "	
D7		RST 10	Erase former bat symbol.

10FB		DJNZ ERASE	
0604		LD B,04	
210000	RDL_1	LD HL,0000	
2B	RDL_2	DEC HL	
7C		LD A,H	
B5		OR L	
20FB		JR NZ,RDL_2	Set a very long delay,
10F6		DJNZ RDL_1	for the player to recover for the next ball.

To print the ball we first of all go through a timed delay loop (controlled by SPEED – the speed of the game) and then *unprint* the previous position of the ball. The contents of the next square in the direction the ball is travelling are examined, and if a bat symbol is at that location then the *direction* of travel is reversed, but the ball is not moved; otherwise, the ball will move in the direction given. The ball will not be printed at its new position.

Then a couple of conditions are checked. If the former contents of the current ball position was a coloured brick then the score is increased, the vertical component of the ball's direction is reversed (although the horizontal component is unaltered) and the speed of the game is increased.

If the next square above or below (depending on the ball's direction of travel) is a wall or a bat then the vertical component of the ball's direction is reversed.

If the next square to the left or right (depending on the ball's direction of travel) is a wall or a bat then the horizontal component of the ball's direction is reversed.

These steps ensure that the next square in the direction of travel of the ball is empty, and that the ball will always bounce off the walls and the bat at the right angles. The part of the program which will achieve all of this is as follows:

Mastering Machine Code on your ZX Spectrum

2AAE5C	LOOP	LD HL,(SPEED)	
2B	DELAY	DEC HL	This is a short delay loop which controls the speed of the game.
7C		LD A,H	
B5		OR L	
20FB		JR NZ,DELAY	
F1		POP AF	Retrieve the carry flag.
3F		CCF	Complement it.
F5		PUSH AF	And re-store it.
DA9369		JP C,MOVE_BAT	The ball is only moved every <i>other</i> time round the loop, so that the bat moves twice as fast as the ball.
2AAC5C		LD HL,(BALL_POS)	HL points to ball.
363F		LD (HL),3F	Erase former ball symbol.
ED5BAA5C		LD DE,(DIRECTION)	
19	NEW_POS	ADD HL,DE	Compute new ball position.
2AAC5C		LD (BALL_POS),HL	Store this position.
7E		LD A,(HL)	
FE07		CP 07	Is a bat at this new position?
200D		JR NZ,SQ_FREE	Jump if not.
7B		LD A,E	
ED44		NEG	
5F		LD E,A	
7A		LD A,D	
2F		CPL	Negate DE (the direction of travel).
57		LD D,A	
ED53AA5C		LD (DIRECTION),DE	Store new direction.
18EA		JR NEW_POS	

3639	SQ_FREE	LD (HL),39	Print ball (blue on white) at new position.
FE3F		CP 3F	Was a brick formerly at that square?
282A		JR Z,VC_H	Jump if not.
E607		AND 07	A: = score attained for hitting brick.
2AAE5C		LD HL,(SPEED)	
01F0FF		LD BC,FFF0	BC: = - 10.
09		ADD HL,BC	
22AE5C		LD (SPEED),HL	Increase speed of game.
211C68		LD HL,SCORE+0B	HL points to last digit of score.
86		ADD A,(HL)	A: = new units digit.
FE3A	CARRY	CP 3A	Is new digit greater than nine?
380A		JR C,INC	Jump if not.
D60A		SUB 0A	Find proper digit.
77		LD (HL),A	Store this digit.
2B		DEC HL	Point to next left digit.
7E		LD A,(HL)	A: = next digit.
F610		OR 10	Change from "space" to "zero" if needed.
3C		INC A	Add the carry from the last digit.
18F2		JR CARRY	
77	INC	LD (HL),A	Store digit.
2E11		LD L,11	Point HL to the label SCORE.
060C		LD B,0C	
7E	PR_SC_2	LD A,(HL)	
23		INC HL	
D7		RST 10	Print score.
10FB		DJNZ PR_SC_2	

*Mastering Machine Code on your ZX Spectrum*

180F		JR VERT	
D5	V_CH	PUSH DE	Stack direction.
7A		LD A,D	
E6C0		AND C0	
F620		OR 20	
5F		LD E,A	DE: = vertical component of direction.
2AAC5C		LD HL,(BALL_POS)	HL: = current ball position.
19		ADD HL,DE	
7E		LD A,(HL)	A: = attribute of computed square.
E6F8		AND F8	Disregard ink colour.
D1		POP DE	DE: = true direction.
200B		JR NZ,H_CH	Jump unless paper colour is black (ie bat or wall).
7B	VERT	LD A,E	
EEC0		XOR C0	
5F		LD E,A	
7A		LD A,D	
2F		CPL	
57		LD D,A	Reverse vertical component of direction.
ED53AA5C		LD (DIRECTION),DE	
D5	H_CH	PUSH DE	Stack direction.
CB1B		RR E	
CB1B		RR E	
9F		SBC A,A	A: = FF (leftward) or 00 (rightward).
57		LD D,A	
F601		OR 01	A: = FF (leftward) or 01 (rightward).
5F		LD E,A	DE: = horizontal component of direction.

2AAC5C		LD HL,(BALL_POS)	HL: = current position of ball.
19		ADD HL,DE	
7E		LD A,(HL)	A: = contents of computed square.
E6F8		AND F8	Disregard ink colour.
D1		POP DE	DE: = true direction.
2008		JR NZ,L_CH	Jump unless wall or bat at square.
7B		LD A,E	
EE3E		XOR 3E	
5F		LD E,A	Reverse horizontal
ED53AA5C		LD (DIRECTION),DE	component of direction.
2AAC5C	L_CH	LD HL,(BALL_POS)	HL: = current ball position.
11E05A		LD DE,ATTRS + 02E0	
A7		AND A	
ED52		SBC HL,DE	
3806		JR C,MOVE_BAT	
19		ADD HL,DE	If ball is on bottom row
363F		LD (HL),3F	of screen then erase it,
C3A468		JP RESTART	and restart game with new ball.

An interesting point to watch for is the way in which the score is increased. Compare the mechanism to that used in SPIRALS to decrease the score. There are one or two differences between this and the last. Firstly, of course, the BREAKOUT score increases instead of decreases but secondly, the score may be increased by any amount (less than ten) in one foul swoop.

If you couldn't quite fathom the direction changing bits, try following through with a few examples. Note that the only directions ever used will be FFDF (up-left), FFE1 (up-right), 001F (down-left) and 0021



(down-right). All of these examples will be converted as required by the routines above.

To move the bat, first of all the keyboard is scanned. If more than one key is pressed at once then the bat remains stationary. If *caps shift* is pressed then the bat moves to the left, and if *space* is pressed the bat moves to the right. Note that the bat will only move if the next square in line is free. If it is blocked, either by a wall or by (in extremely rare circumstances, but which must be considered nonetheless) a ball, then the bat is not moved.

Study this, the final part of the program, and watch the way the bat is actually moved. Remember that the variable BAT\_POS stores the column number of the *middle* of the bat.

CD8E02	MOVE_BAT	CALL KEY_SCAN	
14		INC D	
2042		JR NZ,CYCLE	Don't move bat if more than one key pressed.
7B		LDA,E	A: = key code of key pressed (FF if none).
FE20		CP 20	Check for <i>space</i> .
2808		JR Z,RIGHT	
FE27		CP 27	Check for <i>caps shift</i> .
2039		JR NZ,CYCLE	
1601	LEFT	LD D,01	
1802		JR M_BAT_1	
16FF	RIGHT	LD D,FF	
3AA95C	M_BAT_1	LD A,(BAT_POS)	A: = column number of bat centre.
92		SUB D	
92		SUB D	
92		SUB D	A: = column number of next square in line.

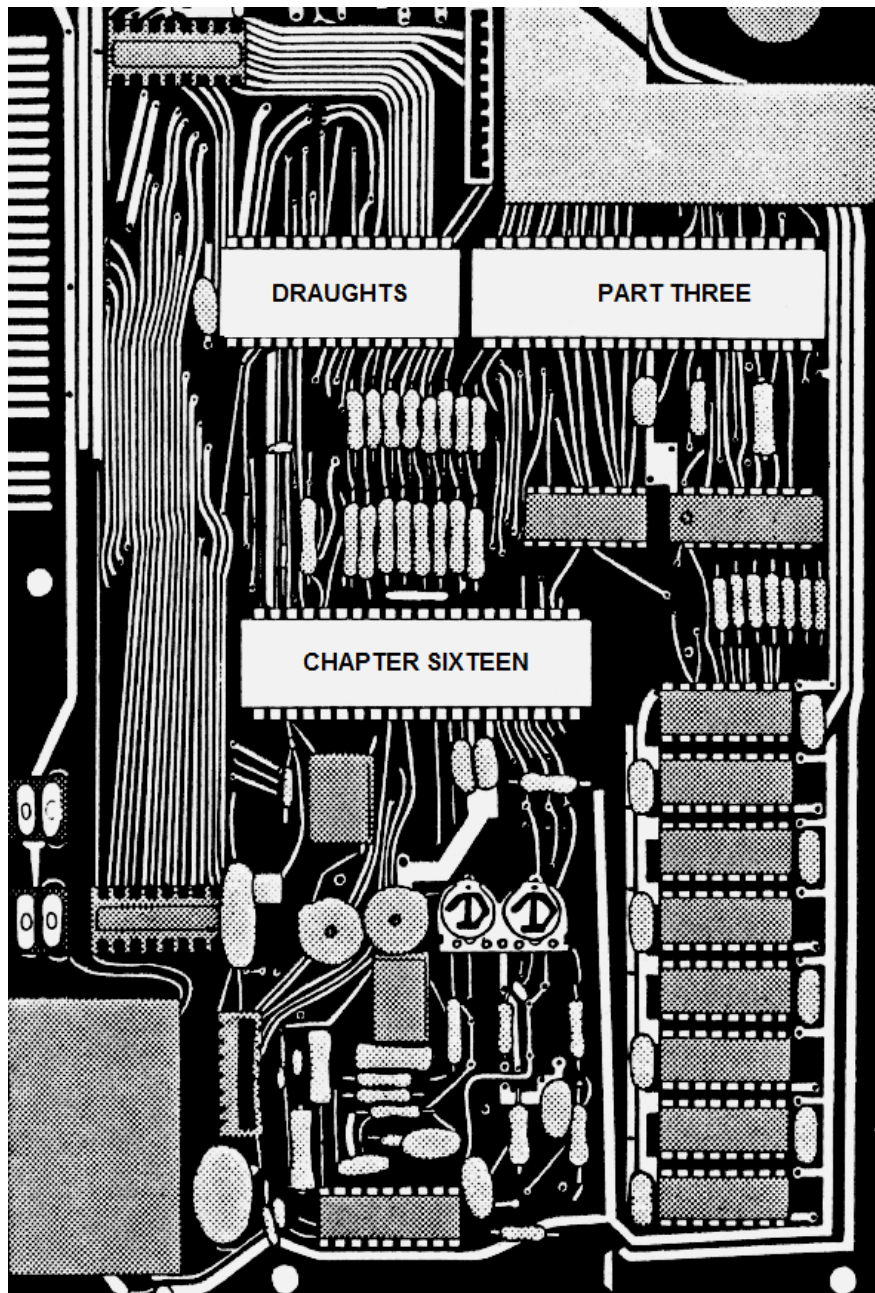
F5		PUSH AF	
C6A0		ADD A,A0	
6F		LD L,A	
265A		LD H,5A	HL: = address of this square in attributes.
7E		LD A,(HL)	
FE3F		CP 3F	Is this square empty?
2803		JR Z,M_BAT_2	Jump if so.
F1		POP AF	Balance the stack.
181F		JR CYCLE	Bat doesn't move.
F1	M_BAT_2	POP AF	
CD1D68		CALL AT_15_A	Move print position to this square.
FD365507		LD (ATTR_T),07	Set colours to white on black.
F5		PUSH AF	
3E8C		LD A," <i>graphic shift</i> <i>3</i> "	
D7		RST 10	Extend bat in required direction.
F1		POP AF	
82		ADD A,D	
82		ADD A,D	
32A95C		LD (BAT_POS),A	Store new bat position.
82		ADD A,D	
82		ADD A,D	
82		ADD A,D	
CD1D68		CALL AT_15_A	Move print position to trailing end of bat.
FD36553F		LD (ATTR_T),3F	Set colours to white on white.
3E90		LD A," <i>graphic A</i> "	

*Mastering Machine Code on your ZX Spectrum*

D7  
C3ED68      CYCLE      JP LOOP

RST 10

Erase trailing edge of  
bat.  
Same for next time  
round.



DRAUGHTS

PART THREE

CHAPTER SIXTEEN

# CHAPTER SIXTEEN

## DRAUGHTS PART THREE

The story so far... Once upon a time a human being input a move to a ZX Spectrum. The computer checked this move to make sure that no cheating was going on, and cast a wicked spell on the poor human if there was (which meant that the whole move had to be typed in all over again). The move was made. The computer started to search through the board for pieces that it could move, it then miraculously found itself at a subroutine called EVALUATE. Where do we go from here?

Well, common sense dictates that the first bit is quite straightforward. It is to answer the question “*Can* the piece move at all?”, and if not, to return to the previous bit of program. If the piece can, in fact, do something... well – that’s a problem we shall think about later on.

This is the start of the EVALUATE routine. As I have said, its job is simply to decide whether or not a move is possible. Write this subroutine to address 6C1A:

E5	EVALUATE PUSH HL	Stack the address which we are considering.
FD3458	INC (P_COUNT)	Count the number of computer pieces on the board.
0604	LD B,04	
7E	LD A,(HL)	A: = 07 for a piece, 87 for a king.
17	RLA	
3802	JR C,EV_LOOP	Jump if a king.
05	DEC B	

05		DEC B	B: = number of directions allowed.
48	EV_ LOOP	LD C,B	C: = number of next direction considered.
0D		DEC C	C: = 00, 01, 02 or 03
CB39		SRL C	C: = 00 (no CARRY), 00 (CARRY), 01 (no CARRY) or 01 CARRY).
3E05		LD A,05	
3001		JR NC,EV_1	Jump if move one or three considered.
3C		INC A	A: = 05, 06, 05 or 06.
OD	EV_1	DEC C	C: = FF, FF, 00 or 00.
2002		JR NZ,EV_2	Jump if move one or two considered.
ED44		NEG	A: = 05, 06, FB or FA.
4F	EV_2	LD C,A	C: = direction displacement.
85		ADD A,L	
6F		LD L,A	HL: = destination square.
7E		LD A,(HL)	A: = contents of that square.
E67F		AND 7F	Disregard king status.
2811		JR Z,MV_FOUND	Jump if square empty.
FE02		CP 02	
2007		JR NZ,EV_3	Jump unless human piece or king.
7D		LD A,L	
81		ADD A,C	
6F		LD L,A	HL: = new destination square.
7E		LD A,(HL)	A: = contents of that square.

*Mastering Machine Code on your ZX Spectrum*

A7		AND A	
2806		JR Z,MV_FOUND	Jump if this square empty.
E1	EV_3	POP HL	HL: = original square considered.
E5		PUSH HL	
10DB	EV_DJ	DJ NZ EV_LOOP	Repeat for all directions allowed.
E1		POP HL	Restore original HL.
C9		RET	End of subroutine.
	MV_FOUND	\$	

How it works is that B represents the “direction number” being considered – this is either one (direction D), two (direction C), three (direction B) or four (direction A). Note that if a king is being considered then B cycles through all of these, but if an ordinary piece is being considered then B will only go through two and one. The actual direction displacement involved is calculated from B in the A register and then stored eventually in the C register. From then on, everything becomes obvious. A move is considered to be possible if either (i) the next square in this direction is empty, or (ii) the next square in this direction is occupied by a human piece or king but the next square after that is empty. If a move is found then we jump to the label MV\_FOUND, and a part of program we have yet to write, otherwise the loop is re-executed for each direction.

So there we have it. The next part is the more breathtaking bit, because its job is to assign a numerical value – a priority – to this choice of move (with the higher the number, the better the move), and to manipulate the list accordingly.

The best way of doing this is to make a copy of the board (yet another copy!) so that we can play around with BOARD\_2 as much as we like without totally corrupting it. Let’s make some space for this copy – which shall be called BOARD\_3. Insert forty two (decimal) spaces to address

6C1A, so that the subroutine EVALUATE now begins at address 6C44. These spaces will be used as a place to store our copy. This, then, is the first part of the MV\_FOUND routine – write it to address 6C77:

210668	MV_FOUND	LD HL,BOARD_2 + 6	Point HL to first playable square.
111A6C		LD DE,BOARD_3	Point DE to first space reserved.
C5		PUSH BC	
012A00		LD BC,002A	
EDB0		LDIR	Copy the board.

And then to make the move (write to 6C83):

C1		POP BC	C: = direction Displacement.
E1		POP HL	HL: = square from which to move.
E5		PUSH HL	
C5		PUSH BC	
1E00		LD E,00	Initial “score” starts at zero.
46		LD B,(HL)	B: = computer’s piece.
73		LD (HL),E	Erase computer’s piece from current square.
7D		LD A,L	
81		ADD A,C	
6F		LD L,A	HL: = destination square.
7E		LD A,(HL)	A: = contents of this square.
A7		AND A	



*Mastering Machine Code on your ZX Spectrum*

280D		JR Z,MV_1	Jump unless this is a “take”.
73		LD (HL),E	Erase piece to be taken.
1E81		LD E,81	
17		RLA	CARRY: = reset for a piece, set for a king.
CB13		RL E	E: = 02 (piece) or 03 (king).
CB13		RL E	E: = 05 (piece) or 07 (king).
CB 13		RL E	E: = 0A (piece) or 0E (king).
7D		LD A,L	
81		ADD A,C	
6F		LD L,A	HL: = new destination square.
70	MV_1	LD (HL),B	Place piece at new position

A subroutine now – yes, you’ve got it – a subroutine within a subroutine. The job of this one is to determine whether or not a computer’s piece is in danger of being taken from a given direction. The subroutine requires HL to point to the square in question, and C to contain the direction – 05 or 06.

The subroutine needs to be inserted (since we have to know its absolute address in order to call it) at address 6C44.

E5	DANGER	PUSH HL
7D		LD A,L
91		SUB C
47		LD B,A
7D		LD A,L
81		ADD A,C

6F		LD L,A	HL: = points to next square downwards.
7E E67F 2006		LD A,(HL) AND 7F JR NZ,DN_1	Disregard king status. Jump if square not empty.
68		LD L,B	HL: = points to next square upwards.
7E FE82		LD A,(HL) CP 82	Does this square contain a human king?
280C FE02	DN_1	JR Z,DN_3 CP 02	Jump if so. Does square contain a human piece or king?
2005 68		JR NZ,DN_2 LD L,B	Jump if not. HL: = points to next square upwards.
7E A7 2803		LD A,(HL) AND A JR Z,DN_3	Jump if this square empty.
E1 A7 C9	DN_2	POP HL AND A RET	Restore original HL. Reset the carry flag.
E1 37	DN_3	POP HL SCF	Restore original HL. Set CARRY to indicate danger.
C9		RET	

The EVALUATE routine will now begin at address 6C65 because of this insertion. Next, the piece of machine code which uses this subroutine. Take a look at Figure 16.1. It gives each square a “weighting”, so that some squares are considered more important than other squares. This next routine scans the board and adjusts the “score so far” (E) according to both the weighting and whether or not any piece is in

danger of being taken. Here we go then – write this routine to address 6CC1.

	06		07		08		09
	0		0		0		0
0B		0C		0D		0E	
0		1		0		1	
	11		12		13		14
	0		0		0		0
16		17		18		19	
1		2		2		0	
	1C		1D		1E		1F
	0		2		2		1
21		22		23		24	
0		0		0		0	
	27		28		29		2A
	0		0		0		0
2C		2D		2E		2F	
0		0		0		0	

**Figure 16.1**

210668		LD HL,BOARD__2	Point HL to first
		+ 06	playable square.
7E	DN_LOOP	LD A,(HL)	A: = contents of
			next square to
			scan.
E67F		AND 7F	Disregard king
			status.
FE07		CP 07	Is this a
			computer's
			piece.
203B		JR NZ,DN_NS	Jump if not.

7D D60C 281 B		LD A,L SUB 0C JR Z,P_1	Square 0C has weighting 01.
D602 2817		SUB 02 JR Z,P_1	Square 0E has weighting 01.
D608 2813		SUB 08 JR Z,P_1	Square 16 has weighting 01.
3D 280F		DEC A JR Z,P_2	Square 17 has weighting 02.
3D 280C		DEC A JR Z,P_2	Square 18 has weighting 02.
D605 2808		SUB 05 JR Z,P_2	Square 1D has weighting 02.
3D 2805		DEC A JR Z,P_2	Square 1E has weighting 02.
3D 2803		DEC A JR NZ,P_1	Square 1F has weighting 01.
1802		JR P_0	All other squares have weighting 00.
1C 1C 7E	P_2 P_1 P_0	INC E INC E LD A,(HL)	A: = computer's piece.
17 1681		RLA LD D,81	

*Mastering Machine Code on your ZX Spectrum*

CB12		RL D	D: = 02 (piece) or 03 (king).
CB12		RL D	D: = 05 (piece) or 07 (king).
CB12		RL D	D: = 0A (piece) or 0E (king).
0E05 CD446C		LD C,05 CALL DANGER	Check danger in direction 05.
3806		JR C, DN_FOUND	Jump if piece in danger.
0C CD446C		INC C CALL DANGER	Check danger in direction 06.
3003		JR NC, DN_NS	Jump if piece safe.
7B	DN_ FOUND	LD A,E	A: = score so far.
92 5F		SUB D LD E,A	Decrease score as required.
7D 23	DN_NS	LD A,L INC HL	Point HL to next square to scan.
FE2F		CP 2F	Have we scanned the whole board?
20B8		JR NZ, DN_LOOP	Jump back if not.

The next change we must make to the score is to assign a bonus of seven points to any piece which reaches the back row and becomes a king. This machine code, to be written to address 6D0C, will do just that.

0604 2B	KCH_LP	LD B,04 DEC HL	Point to next square on back row.
7E FE07		LD A,(HL) CP 07	Is there a computer's piece (not yet king) on back row?
2004 7B C607 5F		JR NZ,KCH_N LD A,E ADD A,07 LD E,A	Jump if not.  Increase score if so.
10F4	KCH_N	DJNZ KCH_LP	

And now to restore the original board (write to address 6D1A):

D5 211A6C 110668 012A00 EDB0 D1 7B C680		PUSH DE LD HL,BOARD_3 LD DE,BOARD_2 + 6 LD BC,002A LDIR POP DE LD A,E ADD A,80	E: = score. A: = score. A: = priority assigned to move.
57		LD D,A	D: = priority assigned to move.
1E01		LD E,01	E: = number of steps in move.

*Mastering Machine Code on your ZX Spectrum*

Notice that we have added 80 to the final score in order to prevent the priority from being negative. This is simply so that our arithmetic is made easier.

And finally, to adjust the list as we require, one last piece of code – write to address 6D2D:

D9 C1	EXX POP BC	B: = direction number. C: = direction displacement.
E1	POP HL	HL: = address of square to move from.
D1	POP DE	DE: = subroutine return address.
D9 E1	EXX POP HL	HL: = priority of items on list.
A7 ED52 19 280A	AND A SBC HL,DE ADD HL,DE JR Z,L_ADD	Add to list if priority the same.
3014	JR NC,L_OK	Forget move if not good enough.
ED7BB05C	LD SP,(L_BASE)	Abolish previous List.
FD365900	LD (L_COUNT),00	Set number of items on list to zero.
D9 L_ADD	EXX	

61 E5		LD H,C PUSH HL	Stack initial square and direction.
2668		LD H,68	HL: = address of square to move from.
D9 D5		EXX PUSH DE	Stack number of steps and priority.
FD3459		INC (L_COUNT)	Count number of items on list.
1801 E5	L_OK	JR L_EXIT PUSH HL	Stack former number of steps and priority.
215827 D9	L_EXIT	LD HL,2758 EXX	HL': 2758 to prevent crashing. B: = direction number. C: = direction displacement. DE: = subroutine return address. HL: = square to move from.
D5		PUSH DE	Re-stack subroutine return address.



E5	PUSH HL	Re-stack address to move from.
C3946C	JP EV_DJ	Re-enter checking loop.

To change the address which the CALL Z,EVALUATE instruction refers to you should type RUN 100/6AFB/656C. Your DRAUGHTS program is now complete – or at least, as complete as I intend to take it. As things stand the instruction RUN 700 will set things in motion and will play quite a reasonable game, but note the following:

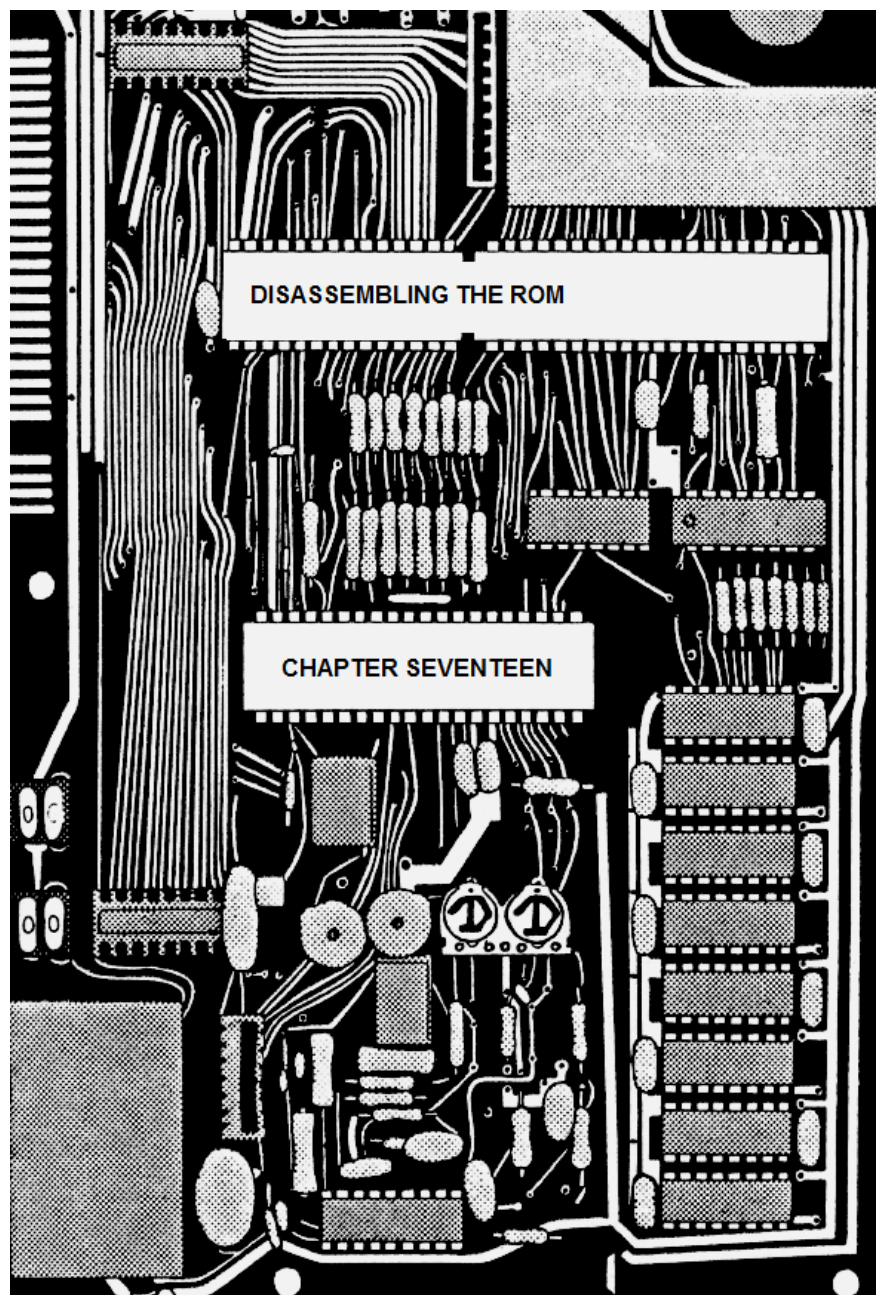
- (i) compulsory taking is *not* in force and therefore, a piece will not be “huffed” (removed from the board) simply because it refuses to take.
- (ii) with the program as it stands I’ve left a deliberate area of improvement for you. Although you are allowed multiple jumps, and although the computer will play a fairly reasonable game, the computer is not yet clever enough to think up multiple jumps for itself. The computer will jump if it wants to, but it will only jump once in any go, even where it may have been possible to jump three or four pieces at once. This then, is your challenge – I’ve done all the work so far – now it’s your turn. The *principle* is the same as for single jumps, but the *procedure* needs to be slightly altered because there is more than one direction to store during the evaluation (you should store these in RAM rather than in registers) and more than one direction to stack onto the list.
- (iii) It is not at present possible to offer a draw. The stalemate condition (where one player has some of her pieces left but is nonetheless unable to move any of them) is not checked for, or allowed for, for either the human or the computer. However, to adapt the program so that the case is otherwise is actually quite simple – you must

check for some special input meaning “a draw” (for the human) and simply look at the variable L\_COUNT (for the computer).

Just a couple of tidying up jobs to do now. RUN 800/3E10/3EAF/7CBA (16K – 48K users should change the last address to FCBA) will restore *graphic B* to *graphic U*, but will leave *graphic A* unchanged. RUN 100/694F/1600 will change the implicit “GO TO 710” instruction in the machine code to an implicit “GO TO 20”. Now you need to type the following:

```
NEW
10 RANDOMIZE USR 26706
20 INPUT LINE A$
30 RANDOMIZE USR 26881
40 LOAD "" CODE
50 LOAD "" CODE USR "A"
SAVE "Draughts" LINE 40
SAVE "Dr mc" CODE 26624,1370
SAVE "Dr gr" CODE USR "A",8
```

You can, of course, change the colour scheme of things if you wish. A straightforward RUN instruction is now all that is needed to start things going. Of course, if you want to improve DRAUGHTS as I have suggested then you'll need to leave HEXLD 3 floating around with the playing part still at lines 700. Still – as I've said before – that's your problem!



# CHAPTER SEVENTEEN

## DISASSEMBLING THE ROM

There are three “levels” at which we may disassemble, each slightly more sophisticated than the previous. The first two levels are not all that satisfactory, but they are very easy to program.

The first “level” we have already achieved – the USR routine H\_LIST in HEXLD 3 which we saw earlier in the book will do this for us. That is, given an address such as 15F2 it will produce an output like this:

```
15F2 D9
15F3 E5
15F4 2A *
15F5 51 Q
15F6 5C\
```

and so on. This is not really disassembly, although you can, of course, look these bytes up in the tables at the back of the book, but it’s quite a time-consuming task, and you’re also very likely to get lost halfway through. The second “level” is not much better, but again is quite easy to program. What I’m talking about is an output something like this:

```
15F2 D9
15F3 E5
15F4 2A515C
15F7 5E
....
```

and so on. As you can see, each instruction has its component bytes listed out to exactly the right length. This produces a very pleasing display, and there is little or no chance of getting lost when actually

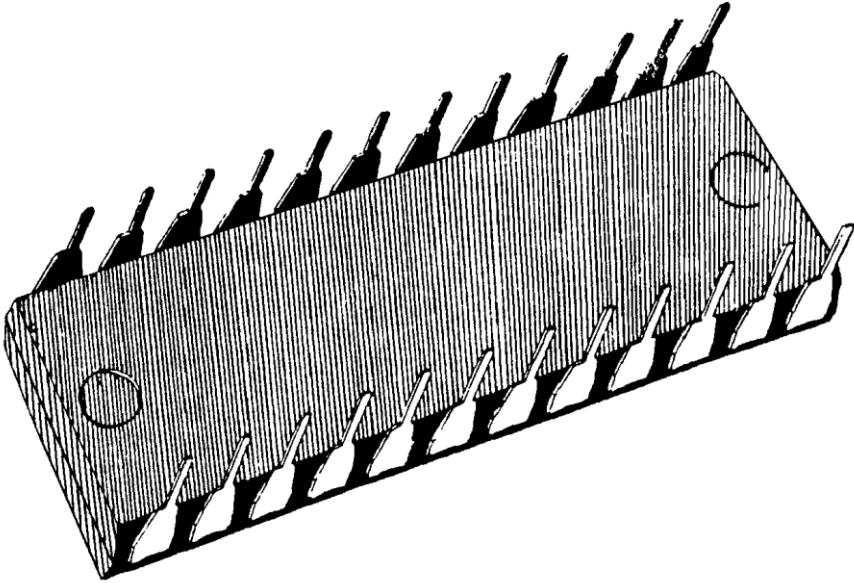
looking these bytes up in tables. The third level is the one we are actually aiming at – the one everybody wants. What we'd like is an output like this:

```
15F2 EXX
15F3 PUSH HL
15F4 LD HL,(5C51)
15F7 LD E,(HL)
....
```

and so on. This *can* be quite easy to program – simply make the computer look up the appropriate words from a table instead of doing it ourselves – however, this would take up rather a large amount of space just to store the table (around four or five K, in fact). The method I shall describe to you will allow such a program to fit in just less than one and a quarter K, but be warned – it is rather difficult. There is actually a fourth level of disassembly, which I won't even attempt to touch, but which you may like to think about. Imagine an output something like this:

```
D9          PRINT_A    EXX
E5          PUSH HL
2A515C      LD HL,(CURCHL)
5E          LD E,(HL)
```

As I've said, I'm not even going to touch this one. The only extra it involves is storing yet another table, this time containing all of the labels used. Let's go back a bit now to something relatively simple. Let's consider a *slightly* improved version of H\_LIST which reaches the "second level" of disassembly, and works out the length of each instruction before printing it.



**Figure 17.1**

All we need is a table containing just two pieces of information for each byte. These are (i) the number of bytes in any instruction which begins with this byte, and (ii) the number of bytes in any instruction which begins with DD or FD followed by this byte. As you know, some confusion may arise over those instructions beginning with CD or ED, but we don't actually need any tables or anything to cope with these, provided we remember the following rules:

All instructions beginning in CB are two bytes in length.

All instructions beginning DDCB or FDCB are four bytes in length.

All instructions beginning ED are two bytes in length, except for LD BC,(pq), LD DE,(pq), LD SP,(pq), LD (pq), BC, LD (pq), DE and LD (pq),

SP. The byte immediately after the ED for these six instructions is 4B, 5B, 7B, 43, 53 or 73 respectively. In binary, these six numbers have the form 01\*\* \*011. No other instructions have this form.

There are *no* instructions beginning DDED or FDED.

Thus, we need a table containing a very small amount of information relating to each byte. Firstly, those instructions which do *not* begin DD, ED or FD can be only one, two or three bytes in length. This means that to store the required information we need just *two* bits. Secondly, those instructions which begin DD or FD can only be two, three or four bytes in length, so ignoring the DD or FD itself this leaves one, two or three bytes. Again, we need only two bits. This makes four bits altogether, and we can thus represent the appropriate lengths for each byte by a single hexadecimal digit (or *nibble* to use the jargon – I’ve been avoiding the use of that word for the whole of the book so far because I can’t type it without rolling around the floor in fits of hysterics). Yes folks, a *nibble* is the same as a *byte*, only smaller – four bits instead of eight. A nibble can be written in just one hexadecimal digit whereas a byte needs two. Our program then will make use of the following table, called LENS, which in my program sits at address 6800. You may, however, store it anywhere in RAM you wish provided it is stored such that each element of the table has the same high part for its address.

LENS DEFB	5F	55	55	A5	55	55	55	A5
	AF	55	55	A5	A5	55	55	A5
	AF	F5	55	A5	A5	F5	55	A5
	AF	F5	99	E5	A5	F5	55	A5
	55	55	55	95	55	55	55	95
	55	55	55	95	55	55	55	95
	55	55	55	95	55	55	55	95
	99	99	99	59	55	55	55	95
	55	55	55	95	55	55	55	95
	55	55	55	95	55	55	55	95
	55	55	55	95	55	55	55	95

55	55	55	95	55	55	55	95
55	FF	F5	A5	55	FE	FF	A5
55	FA	F5	A5	55	FA	F5	A5
55	F5	F5	A5	55	F5	FA	A5
55	F5	F5	A5	55	F5	F5	A5

As you can see, there are sixteen rows, with sixteen hex digits in each row. Those byte sequences beginning with DD or FD which do not represent a valid opcode, such as DD00, have simply been assigned the same number of bytes as if the DD or FD were not there.

The following program will “disassemble” to a sequence of bytes of the correct length. It assumes that the table LENS exists (and resides at 6800), and it assumes that a subroutine H\_PRINT is available which prints out the contents of the A register in hexadecimal without corrupting any of the other registers. This, in fact, is the subroutine given earlier in the book as part of H\_LIST in HEXLD 3. This new program is called H\_LIST\_2, and my version of it lives at 6880.

AF 323C5C	H_LIST_2	XOR A LD (TVFLAG),A	Direct print output to upper part of screen.
2AF8FF (or 2AF87F)		LD HL,(ADDRESS)	HL: = address from which to list.
3E0D D7	NEXT	LD A,“enter” RST 10	Print from start of new line.
CD7DFF (or CD7D7F) 0E00		CALL PR_HL LD C,00	Print HL in hex. C is just a flag to tell us whether the instruction begins with DD or FD or not.



*Mastering Machine Code on your ZX Spectrum*

7E		LDA,(HL)	A: = start of next Instruction.
FEDD		CP DD	
2804		JR Z,DDFD	
FEFD		CP FD	Does the instruction begin in either DD or FD?
2006		JR NZ,NORM	
	DDFD	\$	
CDD0FF (or CDD07F)		CALL H_PRINT	Print the "DD" or "FD".
23		INC HL	Point to next byte.
0C		INC C	Change the C flag accordingly.
7E		LD A,(HL)	A: = next byte.
FEED	NORM	CP ED	Does the byte begin "ED"?
2011		JR NZ,SIMPLE	Print the byte "ED".
CDD0FF (or CDD07F)		CALL H_PRINT	
23		INC HL	
7E		LD A,(HL)	
E6C7		AND C7	Is it of the form 01** *011?
FE43		CP 43	
0601		LD B,01	
201 E		JR	
		NZ,NXT_BYT	
0603		LD B,03	
181A		JR NXT_BYT	
E5	SIMPLE	PUSH HL	
CB3F		SRL A	Divide A by two.

F5		PUSH AF	Stack the carry flag.
C600		ADD A,LENS <i>low</i>	
6F		LD L,A	
2668		LD H,LENS <i>high</i>	HL: = points to appropriate point in table.
F1		POP AF	
7E		LD A,(HL)	A: = element of Table.
3804		JR C,NIBBLE	Use the carry flag to decide on whether the first or the second hex digit will be used.
1F		RRA	
1F		RRA	
1F		RRA	
1F		RRA	
0D	NIBBLE	DEC C	Use C to decide which two bits to use.
2002		JR NZ,OK	
1F		RRA	
1F		RRA	
E603	OK	AND 03	Put this number into B to use as a count.
47		LD B,A	
E1		POP HL	Retrieve the address of the byte to be "disassembled".
7E	NXT_BYT	LD A,(HL)	
23		INC HL	
CDD0FF (or CDD07F)		CALL H_PRINT	Print the next byte in Hex.
10F9		DJNZ NXT_BYT	
18B1		JR NEXT	

Now we ascend to the “third level” – *real* disassembly in other words. However, I’m not going to explain in precise detail exactly how each bit of the program works – instead I’ll leave you to try and work all that out for yourself. I will, however, as well as a listing of the program, tell you the *algorithm* by which the program works.

## **DISASSEMBLING**

This then, is an algorithm which will enable you to disassemble any machine code into assembler. That is – to change (for example) 69 to LD L,C, or from FDCB2A76 to BIT 6,(IY + 2A). One way of doing this would be to store a very large table – such as I have included in the appendices – but while this may be alright for human beings it lacks the elegance of a well thought-out computer program. The data alone would occupy around 4K. This algorithm will enable us to write our own machine language program occupying significantly less – just under one and a quarter K, in fact.

In this algorithm the following conventions will be used:

r(0) means “B”, r(1) means “C”, r(2) means “D”, r(3) means “E”, r(4) means “H”, r(5) means “L”, r(6) means “x” and r(7) means “A”.

s(0) means “BC”, s(1) means “DE”, s(2) means “y” and s(3) means “SP”.

q(0) means “BC”, q(1) means “DE”, q(2) means “y” and q(3) means “AF”.

n(0) means “0”, n(1) means “1”, n(2) means “2”, n(3) means “3”, n(4) means “4”, n(5) means “5”, n(6) means “6” and n(7) means “7”.

c(0) means “NZ”, c(1) means “Z”, c(2) means “NC”, c(3) means “C”, c(4) means “PO”, c(5) means “PE”, c(6) means “P” and c(7) means “M”.

x(0) means “ADD A,” x(1) means “ADC A,” x(2) means “SUB *space*”, x(3) means “SBC A,” x(4) means “AND *space*”, x(5) means “XOR *space*”, x(6) means “OR *space*” and x(7) means “CP *space*”.

Define two variables, CLASS and INDEX, and initially let both of them be zero.

Write the byte being disassembled in binary, and split it into three parts: F, G and H. F consists of bits 7 and 6, G of bits 5, 4 and 3, and H of bits 2, 1 and 0. Thus, to disassemble the byte 69 (binary 0110 1001) you’d start off by splitting it into three parts thus: 01 / 101 / 001. In this particular case F is one, G is five and H is one.

Next, split G into two parts, J and K, with J consisting of bits 2 and 1, and K just bit 0. If G then were binary 101 as above then split it like this: 10/1. In this case, we would define J to be two and K to be one.

Set aside an area of memory called DIS (the start of spare RAM is a good place). This is to contain a *string* of unknown length. To do this we merely make the first byte of DIS count the number of bytes in the string, and begin the text of the string itself from the second byte of DIS. (You only need one byte since DIS will never be more than FF characters in length). DIS should initially be an *empty* string, ie containing no text at all.

The algorithm begins here...

- 1) If the byte is CB then...
  - let CLASS equal one, and
  - if INDEX equals zero then continue from the next byte,
  - if INDEX is non-zero then interpret the next two bytes in reverse order.
- 2) If the byte is ED then let CLASS equal two and continue from the next byte.

- 3) If the byte is DD then let INDEX equal one and continue from the next byte.
- 4) If the byte is FD then let INDEX equal two and continue from the next byte.

**If CLASS equals zero then the following applies:**

If the byte is 76 then the complete disassembled instruction is HALT.

If F equals zero then...

If H equals zero then...

If G greater than 3 then let DIS equal JR c(G - 4),v.

If G less than 4 then choose the (G + 1)th item in this list:

NOP/EX AF,AF'/DJNZ v/JR v.

If H equals one then...

If K is zero then let DIS equal LD s(J),w.

If K is one then let DIS equal ADD y,s(J).

If H equals two then let DIS equal LD followed by the (G + 1)th item in this list:

(BC),A/A,(BC)/(DE),A/A,(DE)/(w),y/y,(w)/(w),A/A,(w).

If H equals three then...

If K equals zero then let DIS equal INC s(J).

If K equals one then let DIS equal DEC s(J).

If H equals four then let DIS equal INC r(G).

If H equals five then let DIS equal DEC r(G).

If H equals six then let DIS equal LD r(G),v.

If H equals seven then choose the  $(G + 1)$ th item in this list:  
RLCA/RRCA/RLA/RRRA/DAA/CPL/SCF/CCF

If F equals one then let DIS equal LD  $r(G), r(H)$ .

If F equals two then let DIS equal  $x(G)$  followed immediately by  $r(H)$ .

If F equals three then...

If H equals zero then let DIS equal RET  $c(G)$ .

If H equals one then...

If K equals zero then let DIS equal POP  $q(J)$ .

If K equals one then let DIS equal the  $(J + 1)$ th item in this list:  
RET/EXX/JP  $(y)$ /LD SP,  $y$ .

If H equals two then let DIS equal JP  $c(G), w$ .

If H equals three then choose the  $(G + 1)$ th item in this list:  
JP  $w/^*/OUT (v), A/IN A, (v)/EX (SP), y/EX DE, HL/DI/EI$ .

If H equals four then let DIS equal CALL  $c(G), w$ .

If H equals five then...

If K is zero then let DIS equal PUSH  $q(J)$ .

If K is one then let DIS equal CALL  $w$ .

If H equals six then let DIS equal  $x(G)$  followed by  $v$ .

If H equals seven then let DIS equal RST followed by the  $(G + 1)$ th item in this list:

00/08  $v/10/18/20/28/30/38$ .

**If CLASS equals one then the following applies:**

If F equals zero then choose the (G + 1)th item in this list:  
RLC/RRC/RL/RR/SLA/SRA\*/SRL and follow it by r(H).

If F equals one then let DIS equal BIT n(G),r(H).

If F equals two then let DIS equal RES n(G),r(H).

If F equals three then let DIS equal SET n(G),r(H).

**If CLASS equals two then the following applies:**

F cannot possibly equal zero.

If F equals one then...

If H equals zero then let DIS equal IN r(G),(C).

If H equals one then let DIS equal OUT (C),r(G).

If H equals two then...

If K equals zero then let DIS equal SBC HL,s(J).

If K equals one then let DIS equal ADC HL,s(J).

If H equals three then...

If K equals zero then let DIS equal LD (w),s(J).

If K equals one then let DIS equal LD s(J),(w).

If H equals four then let DIS equal NEG.

If H equals five then...

If K equals zero then let DIS equal RETN.

If K equals one then let DIS equal RETI.

If H equals six then choose the  $(G + 1)$ th item in this list:  
IM 0/\*/IM 1/IM 2.

If H equals seven then choose the  $(G + 1)$ th item in this list:  
LD I,A/LD R,A/LD A,I/LD A,R/RRD/RLD/\*/\*.

If F equals two then choose the  $(H + 1)$ th item in the following list:  
LD/CP/IN/OT and follow it by the  $(G - 3)$ th item in this list:  
I/D/IR/DR.

F cannot possibly equal three.

### **To compute the final output**

Replace every x by:  
(HL) if INDEX equals zero.  
(IX + v) if INDEX equals one.  
(IY + v) if INDEX equals two.

Replace every y by:  
HL if INDEX equals zero.  
IX if INDEX equals one.  
IY if INDEX equals two.

Replace every v by the next byte to be disassembled, in hex.

Replace every w by the next two bytes to be disassembled, in hex, and in reverse order.

The string DIS now contains the correctly disassembled instruction. This should now be printed to the screen.

We shall in due course write a machine language program which disassembles code using this algorithm. This program will occupy just under one and a quarter K. Surprising as this may sound I should add



that although it is possible, the program itself is rather complicated and involves a number of completely new programming techniques.

The program revolves around eight different subroutines, which are linked together by one MASTER subroutine which calls each of the others in any required order. This is achieved by the following means.

Somewhere in the program there should be a table called SUBRTS which contains eight addresses – the addresses of the eight “baby” subroutines which run the program. The register pair HL’ (note the dash) will be pointing to a sequence of data which the parent subroutine MASTER interprets as the order in which its offspring must be called. The data in this sequence is terminated by an item with bit seven set. Bits two, one and zero direct exactly which subroutine to call so for instance, any byte of the binary form \*\*\*\* \*110 will call subroutine six and any byte of the binary form \*\*\*\* \*011 will call subroutine three. A byte of the binary form 1\*\*\* \*011 will not only call subroutine three but will also indicate to the MASTER subroutine that this is the last subroutine to be called.

So any item of data in this sequence looks like this: 0\*\*\* \*nnn for all but the last item, and 1\*\*\* \*nnn for the last item itself. (The part I’ve written as “nnn” means the appropriate number zero to seven as described.) Some of these subroutines will also require *data* (which will never need to be more than three bits). If we then write this data as “ddd” (binary) then it makes sense to save space by storing this number amongst some of the so far unused bits of the controlling byte, thus making the controlling byte look now like this: 0\*dd dnnn for all but the last byte, and 1 \*dd dnnn for the last byte. Bit six, so far unused, has a separate meaning – that DIS is to have a comma placed at the end after the current subroutine has been carried out.

I hope that didn’t confuse you. To make things clear, suppose HL’ points to an address at which is stored the sequence of data 04 44 25 85. This means that first of all, subroutine four is to be called with data 000b, then

subroutine four is to be called again (also with data 000b) but this time the present state of the string DIS is to be followed by a comma, then subroutine five is to be called with data 100b, and finally (because bit seven is set), subroutine five is to be called again, this time with data 000b.

The MASTER subroutine which will achieve all this is as follows:

D9	MASTER	EXX	
7E		LD A,(HL)	Find byte of data and increment pointer.
23		INC HL	
D9		EXX	
5F		LD E,A	E: = this byte (bits 7 to 3 will be needed later on).
E607		AND 07	A: = subroutine number to call.
17		RLA	Multiply by two.
4F		LD C,A	
0600		LD B,00	BC: = subroutine number times two.
21????		LD HL,RETURN	HL: = return address from minor subroutines.
E5		PUSH HL	Stack this address.
21????		LD HL,SUBRTS	
09		ADD HL,BC	Point HL to address of subroutine to call.
4E		LD C,(HL)	
23		INC HL	
46		LD B,(HL)	BC: = address to call.

C5	PUSH BC	
C9	RET	Call this subroutine.

You can learn a lot from studying this MASTER subroutine. Can you see how the appropriate subroutine (one of eight) is called? First of all, the label RETURN is pushed onto the stack. This means that if each of the eight subroutines ends with RET then control at that point will jump to the label RETURN. Think carefully about how this works. The required address is pushed onto the stack, on top of the label RETURN. Then a RET instruction is executed. RET has the effect of popping the first number off the stack and then jumping to that address. The item now left at the top of the stack is the label RETURN, which means that when the subroutine ends, control will return correctly. All of this is necessary because there is no such instruction as CALL (BC): – in BASIC, the statement GO SUB VARIABLE is allowed, but not in machine code. Another way we could have achieved the same as PUSH BC/RET is LD H,B/LD L,C/JP (HL). Can you see why this does the same thing?

Each of the *alternative* registers (except A') in this program have a dedicated purpose. These are:

BC' The address of the next byte to be disassembled.  
D' The variable INDEX.  
E' The variable CLASS.  
HL' Points to subroutine data.

The byte to be disassembled is located and stored in the D register by the means EXX/LD A,(BC)/INC BC/EXX/LD D,A. From this, the quantities I've called F, G and H may be generated. Somewhere in the program there must be a table called DATADS containing twelve different addresses which correspond to the cases CLASS = 0/ F = 0, 1, 2 or 3, CLASS = 1/F = 0, 1, 2 or 3 and CLASS = 2/F = 0, 1, 2 or 3. The initial value of HL' is simply read from this table.

HL' may also be altered by the subroutines themselves. For instance, it is the job of subroutine zero to re-position HL' to the (H + 1)th address in a list of eight addresses.

Several other subroutines will also be needed in addition to the eight controlling ones. One such routine of quite major importance is a subroutine to append a single character to the end of the string DIS. Using the convention that the string starts at (STKEND) (the start of spare RAM) and the first byte is the length of the string, the string may be emptied by the sequence LD HL,(STKEND)/LD (HL),00. To add a character (held in the A register) the subroutine is this:

F5	CHR	PUSH AF	Store the
C5		PUSH BC	registers A, B, C,
E5		PUSH HL	H and L so that
			they won't be
			corrupted by this
			subroutine.
E67F		AND 7F	Disregard bit
			seven.
2A655C		LD HL,(STKEND)	Point HL to string
			DIS.
4E		LD C,(HL)	C: = former
			length of DIS.
0C		INC C	C: = new length
			of DIS.
71		LD (HL),C	Store new length
			of DIS.
0600		LD B,00	BC: = length of
			DIS.
09		ADD HL,BC	Point HL to new
			last byte of DIS.
77		LD (HL),A	Store the given
			character.

E1	POP HL	Restore all of the
C1	POP BC	registers.
F1	POP AF	
C9	RET	End of subroutine.

Figure 17.2 gives a diagrammatical breakdown of more or less what goes on. The eight subroutines needed for this disassembly program are as follow:

### **SUBROUTINE ZERO – SPLIT**

This is the subroutine called by the byte 00. If it is used at all then it will be the first byte in the sequence. Following the byte 00 are eight new addresses. Located at these addresses are eight different sequences of data. The subroutine selects the  $(H + 1)$ th of these addresses and reads all subsequent data from the new location.

### **SUBROUTINE ONE – LITERAL**

The control byte is followed by a series of characters, such as “N”, “O” and “P”, which represent part or all of the disassembled instruction. The last character of this string must have bit seven set. Subroutine one must detect this bit and use it as a check for the end of the string. The data part of the control byte should be either 000 or 001. If the data is 001 it means that the string given must be followed by a space. The string which is specified by this means is to be appended to the end of DIS.

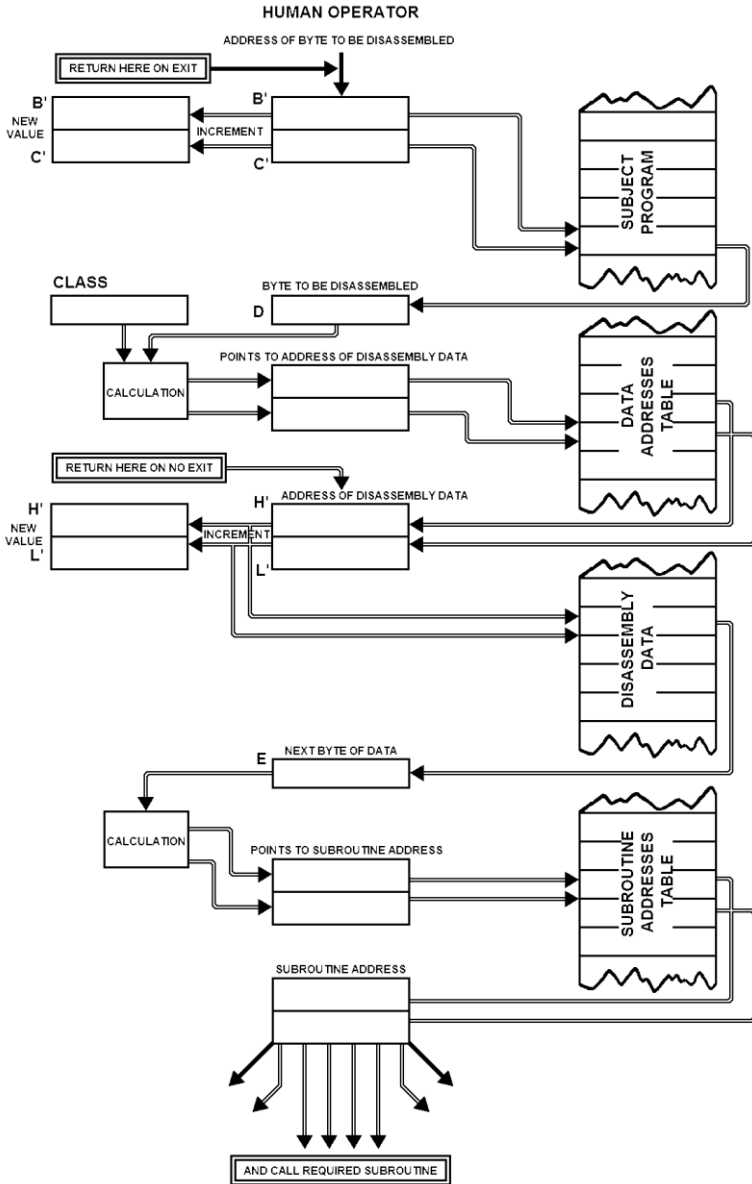


Figure 17.2

## **SUBROUTINE TWO – LIST-G**

This means select the (G + 1)th item in the following list. The data for this subroutine should be 011 if there are four items in the list, or 111 if there are eight items in the list. For example, the control byte 3A (binary 0/0/111/010) means call Subroutine Two which will select the (G + 1)th item from the following list of eight. The list could, for instance, be RLCA/RRCA/RLA/RRR/DAA/CPL/SCF/CCF. The last byte in each word must have bit seven set so that the subroutine will know where one word stops and the next one starts. Thus, in the example given, if G were five, then the literal CPL would be appended to the string DIS. The next byte to be interpreted by the MASTER subroutine should be the first byte after the “F” in “CCF”.

## **SUBROUTINE 3 – LIST-H**

As Subroutine Two but with H instead of G.

## **SUBROUTINE 4 – SELECT-G**

Interpret this one using the data supplied:

000 means select r(G).

001 means select s(G).

010 means select q(G).

011 means select n(G).

100 means select c(G).

101 is not used.

110 means select x(G).

111 is not used.

The item selected is to be appended to the end of DIS.

## **SUBROUTINE 5 – SELECT-H**

As Subroutine Four but with H instead of G.

## **SUBROUTINE 6 – SKIP**

Resets bit five of E (the byte to be disassembled). If *the previous* value

of bit five was one then skip over n bytes of data. The number “n” is the value of the immediately following byte. If bit five was formerly zero then this immediately following byte, which only exists to specify n, is ignored and the byte following this is interpreted as the next control byte.

### **SUBROUTINE 7 – KSKIP**

Replace bit three of E by bit four; replace four by bit five; reset bit five. (So far this has achieved LET G = J.) Then, if the previous value of bit three was one, n bytes are skipped over, as in Subroutine Six. This subroutine can be interpreted as IF K = 0 THEN..../IF K = 1 THEN ....

### **THE PROGRAM AS A WHOLE**

The entire disassembly program consists of initialising the variables CLASS and INDEX, assigning BC' (input by the human operator), finding the address HL' from tables, and then going into the *MASTER* routine. On exiting this, it must then replace all x's and y's, and then all v's and w's as defined earlier in the chapter. The computed result may now be printed and the next byte to be disassembled will be treated in exactly the same way. The entire program, including all subroutines needed and all data, occupies just less than one and a quarter K.

However well or badly I may have explained this procedure, you will undoubtedly find trying to interpret how it performs its task a challenge. I've deliberately listed the program this time without comments, so that if you want to get full benefit from the listing you'll have to work it all out for yourself.

Just to prove that the program works (in case you don't believe me.), I have actually used the program to list itself. You can see then, a sample of its output. The BASIC which must accompany the machine code consists only of POKEing the address (ADDRESS) ((7FF8) in my listing) with the address from which you want to disassemble and then calling the machine code from address 69B5 (27061d). It might even be an idea to append this program permanently to HEXLD 3. Anyway – I've



done all the work this time – understanding what I've done is your problem...

Good luck.

6800 TYPES	Data for subroutines four and five.
6854 SUBRTS	Addresses of eight control subroutines.
6864 DATADS	Addresses of data sequences.
687C REPLACE	Subroutine to replace x, y, v and w as required.
68B4 CHR	Subroutine to append a character to the string DIS.
68C7 HP_BC	Prints BC in hex.
68CC HP_A	Prints A in hex.
68D5 HP_AL	Prints the low part of A in hex.
68DF PRINT_A	Prints an ASCII character to the screen.
68E9 INS	Inserts next byte of subject program into DIS.
6903 RETURN	Return address from control subroutines.
6911 DECODE	Decode DIS and print the final output.
69B5 START	Call from this address.
69BF RESTART	Re-initialise for next instruction to disassemble.
69C9 MAIN	Main routine.
6A33 MASTER	Select and call appropriate control routine.
6A4B SPLIT	Subroutine Zero.
6A5C LITERAL	Subroutine One.
6A6E FIND	Selects the (A + 1)th item from the following list.
6A72 SELECT	Selects the (A + 1)th item from any list.
6A7E LIST-G	Subroutine Two.
6A84 LIST-H	Subroutine Three.
6AA1 SELECT-G	Subroutine Four.
6AA7 SELECT-H	Subroutine Five.

6ABE KSKIP	Subroutine Seven.
6ACE SKIP	Subroutine Six.
6AE2 DATA	All data accessed by control subroutines.

The following addresses contain data rather than machine code:

6800 – 687B  
6925 – 6929  
6932 – 6938  
693E – 6944  
6950 – 6952  
695B – 695D  
6963 – 6965  
6975 – 6977  
6997 – 699B  
69F1 – 69F4  
6AE2 – 6CE6

687C POP IX  
687E LD DE, (5065)  
6882 AND A  
6883 SBC HL,DE  
6885 LD A, (DE)  
6886 SUB L  
6887 INC A  
6888 PUSH AF  
6889 LD A, (DE)  
688A LD L,A  
688B ADD A, (IX+00)  
688E DEC A  
688F LD (DE), A  
6890 ADD HL, DE  
6891 LD E, (IX+00)  
6894 LD D, 00  
6896 PUSH HL

*Mastering Machine Code on your ZX Spectrum*

6897 ADD HL, DE  
6898 DEC HL  
6899 EX DE, HL  
689A POP HL  
689B POP AF  
689C PUSH BC  
689D LD C, A  
689E LD B, 00  
68A0 LDDR  
68A2 EX DE, HL  
68A3 INC DE  
68A4 LD C, (IX+00)  
68A7 PUSH IX  
68A9 POP HL  
68AA INC HL  
68AB LD B, 00  
68AD LDIR  
68AF EX DE, HL  
68B0 DEC HL  
68B1 POP BC  
68B2 PUSH DE  
68B3 RET  
68B4 PUSH AF  
68B5 PUSH BC  
68B6 PUSH HL  
68B7 AND 7F  
B8B9 LD HL, (5C65)  
68BC LD C, (HL)  
68BD INC C  
68BE LD (HL), C  
68BF LD B, 00  
68C1 ADD HL, BC  
68C2 LD (HL), A  
68C3 POP HL  
68C4 POP BC

68C5 POP AF  
68C6 RET  
68C7 LD A, B  
68C8 CALL 68CC  
68CB LD A, C  
68CC PUSH AF  
68CD RRA  
68CE RRA  
68CF RRA  
68D0 RRA  
68D1 CALL 68D5  
68D4 POP AF  
68D5 AND 0F  
68D7 OR 00  
68D9 OP 0A  
68DB JR C, 02  
68DD ADD A, 07  
68DF PUSH BC  
68E0 PUSH DE  
68E1 PUSH HL  
68E2 EXX  
68E3 RST 10  
68E4 EXX  
68E5 POP HL  
68E6 POP DE  
68E7 POP BC  
68E8 RET  
68E9 EXX  
68EA LD A, (BC)  
68EB INC BC  
68EC EXX  
68ED PUSH AF  
68EE CALL 68F6  
68F1 POP AF  
68F2 RRA

*Mastering Machine Code on your ZX Spectrum*

68F3 RRA  
68F4 RRA  
68F5 RRA  
68F6 AND 0F  
68F8 OR 30  
68FA OP 3A  
68FC JR C, 02  
68FE ADD A, 07  
6900 LD (HL),A  
6901 DEC HL  
6902 RET  
6903 BIT 6, E  
6905 JR Z, 05  
6907 LD A, 2C  
6909 CALL 68B4  
690C BIT 7, E  
690E JP Z, 6A33  
6911 EXX  
6912 LD, AD  
6913 EXX  
6914 LD C, A  
6915 LD HL, (5C65)  
6918 LD B, (HL)  
6919 INC HL  
691A LD A, (HL)  
691B AND A  
691C JR NZ, 27  
691E LD A, C  
691F AND A  
6920 JR NZ, 0A  
6922 CALL 687C  
692A JR 19  
692C DEC A  
692D JR NZ, 0C  
692F CALL 687C

6939 JR 0A  
693B CALL 687C  
6945 LD A, (HL)  
6946 DEC A  
6947 JR NZ, 1D  
6949 LD A, C  
694A AND A  
694B JR NZ, 08  
694D CALL 687C  
6950 JR 11  
6955 DEC A  
6956 JR NZ, 08  
6958 CALL 687C  
695E JR 06  
6960 CALL 687C  
6966 DJNZ B1  
6968 LD HL, (5065)  
696B LD B, (HL)  
696C INC HL  
696D LD A, (HL)  
696E SUB 02  
6970 JR NZ, 1F  
6972 CALL 687C  
6978 EXX  
6979 LD A, E  
697A RRA  
697B SBS A, A  
697C AND D  
697D EXX  
697E PUSH AF  
697F JR Z, 04  
6981 EXX  
6982 DEC BC  
6983 DEC BO  
6984 EXX

*Mastering Machine Code on your ZX Spectrum*

6985 CALL 68E9  
6988 POP AF  
6989 JR Z, 19  
698B EXX  
698C INC BC  
698D NOP  
698E EXX  
698F JR 13  
6991 DEC A  
6992 JR NZ, 12  
6994 CALL 687C  
699C CALL 68E9  
699F CALL 68E9  
69A2 INC HL  
69A3 INC HL  
69A4 INC HL  
69A5 INC HL  
69A6 DJNZ C4  
69A8 LD HL, (5C65)  
69AB LD B, (HL)  
69AC INC HL  
69AD LD A, (HL)  
69AE CALL 68DF  
69B1 DJNZ F9  
69B3 JR 0A  
69B5 XOR A  
69B6 LD (5C3C), A  
69B9 EXX  
69BA LD BC, (7FF8)  
69BE EXX  
69BF LD HL, (5C65)  
69C2 LD (HL), 00  
69C4 EXX  
69C5 LD DE, 0000  
69C8 EXX

69C9 LD A, 0D  
69CB CALL 68DF  
69CE EXX  
69CF CALL 6807  
69D2 LD A, 20  
69D4 CALL 68DF  
69D7 LD A, (BC)  
69D8 INC BC  
69D9 PUSH DE  
69DA EXX  
69DB POP BC  
69DC OP 76  
69DE JR NZ, 16  
69E0 DEC O  
69E1 JR Z, 12  
69E3 LD HL, 69F1  
69E6 LD B, 04  
69E8 LD A, (HL)  
69E9 INC HL  
69EA CALL 68DF  
69ED DJNZ F9  
69EF JR D8  
69F5 INC C  
69F6 CP OB  
69F8 JR Z, 18  
69FA CP ED  
69FC JR Z, 10  
69FE CP DD  
6A00 JR Z, 08  
6A02 CP FD  
6A04 JR NZ, 1A  
6A06 LD B, 02  
6A08 JR 11  
6A0A LD B, 01  
6A0C JR 0D



*Mastering Machine Code on your ZX Spectrum*

6A0E LD C, 02  
6A10 JR 09  
6A12 LD C, 01  
6A14 INC B  
6A15 DEC B  
6A16 JR Z, 03  
6A18 EXX  
6A19 INC BC  
6A1A EXX  
6A1B PUSH BC  
6A1C EXX  
6A1D POP DE  
6A1E JR B7  
6A20 LD D, A  
6A21 AND 00  
6A23 OR C  
6A24 RLCA  
6A25 RLCA  
6A26 RLCA  
6A27 ADD A, 64  
6A29 LD L, A  
6A2A LD H, 68  
6A2C LD C, (HL)  
6A2D INC HL  
6A2E LD B, (HL)  
6A2F PUSH BC  
6A30 EXX  
6A31 POP HL  
6A32 EXX  
6A33 EXX  
6A34 LD A, (HL)  
6A35 INC HL  
6A36 EXX

6A37 LD E, A  
6A38 AND 07  
6A3A RLA  
6A3B LD O, A  
6A3C LD B, 00  
6A3E LD HL, 6903  
6A41 PUSH HL  
6A42 LD HL, 6854  
6A45 ADD HL, BC  
6A46 LD C, (HL)  
6A47 INC HL  
6A48 LD B, (HL)  
6A49 PUSH BC  
6A4A RET  
6A4B LD A, D  
6A4C EXX  
6A4D AND 07  
6A4F RLA  
6A50 PUSH DE  
6A51 LD E, A  
6A52 LD D, 00  
6A54 ADD HL, DE  
6A55 LD E, (HL)  
6A56 INC HL  
6A57 LD D, (HL)  
6A58 EX DE, HL  
6A59 POP DE  
6A5A EXX  
6A5B RET  
6A5C EXX  
6A5D LD A, (HL)  
6A5E INC HL  
6A5F CALL 68B4  
6A62 RLA  
6A63 JR NC, F8

*Mastering Machine Code on your ZX Spectrum*

6A65 EXX  
6A66 BIT 3, E  
6A68 RET Z  
6A69 LD A, 20  
6A6B JP 68B4  
6A6E EXX  
6A6F PUSH HL  
6A70 EXX  
6A71 POP HL  
6A72 LD B, A  
6A73 INC B  
6A74 DEC B  
6A75 RET Z  
6A76 LD A, (HL)  
6A77 INC HL  
6A78 RLA  
6A79 JR NC, FB  
6A7B DJNZ F9  
6A7D RET  
6A7E LD A, D  
6A7F RRA  
6A80 RRA  
6A81 RRA  
6A82 JR 01  
6A84 LD A, D  
6A85 AND 07  
6A87 CALL 6A6E  
6A8A LD A, (HL)  
6A8B INC HL  
6A8C CALL 68B4  
6A8F RLA  
6A90 JR NC, F8  
6A92 LD A, E  
6A93 RRA  
6A94 RRA

6A95 RRA  
6A96 AND 07  
6A98 INC A  
6A99 CALL 6A6E  
6A9C PUSH HL  
6A9D EXX  
6A9E POP HL  
6A9F EXX  
6AA0 RET  
6AA1 LD A, D  
6AA2 RRA  
6AA3 RRA  
6AA4 RRA  
6AA5 JR 01  
6AA7 LD A, D  
6AA8 AND 07  
6AAA PUSH AF  
6AAB LD A, E  
6AAC AND 38  
6AAE LD L, A  
6AAF LD A, 68  
6AB1 POP AF  
6AB2 CALL 6A72  
6AB5 LD A, (HL)  
6AB6 INC HL  
6AB7 CALL 68B4  
6ABA RLA  
6ABB JR NC, F8  
6ABD RET  
6ABE LD A, D  
6ABF RRA  
6AC0 AND 18  
6AC2 LD B, A  
6AC3 LD A, D  
6AC4 RLA

*Mastering Machine Code on your ZX Spectrum*

6AC5 RLA  
6AC6 AND 20  
6AC8 OR B  
6AC9 XOR D  
6ACA AND 38  
6ACC XOR D  
6ACD LD D, A  
6ACE BIT 5, D  
6AD0 JR NZ, 04  
6AD2 EXX  
6AD3 INC HL  
6AD4 EXX  
6AD5 RET  
6AD6 RES 5, D  
6AD8 EXX  
6AD9 PUSH BC  
6ADA LD C, (HL)  
6ADB INC HL  
6ADC LD B, 00  
6ADE ADD HL, BC  
6ADF POP BC  
6AE0 EXX  
6AE1 RET

6800	C2	C3	C4	C5	C8	CC	B0	C1
6808	42	C3	44	C5	81	53	D0	00
6810	42	03	44	C5	81	41	C6	00
6818	B0	B1	B2	B3	B4	B5	B6	B7
6820	4E	DA	DA	4E	C3	C3	50	CF
6828	50	C5	D0	CD	00	00	00	00
6830	41	44	44	20	41	AC	41	44
6838	43	20	41	AC	53	55	42	A0
6840	53	42	43	20	41	AC	41	4E
6848	44	A0	58	4F	52	A0	4F	52
6850	A0	43	50	A0	4B	6A	5C	6A
6858	7E	6A	84	6A	A1	6A	A7	6A
6860	CE	6A	BE	6A	E2	6A	7F	6B
6868	84	6B	86	6B	1A	6C	32	6C
6870	38	6C	3E	6C	00	00	44	6C
6878	C9	6C	00	00				
6920	20	0A	CD	7C	68	04	28	48
6928	4C	29	18	19	3D	20	0C	CD
6930	7C	68	06	28	49	58	2B	02
6938	29	18	0A	CD	7C	6B	05	28
6940	49	59	2B	02	29	7E	3D	20
6948	1D	79	A7	20	08	CD	7C	68
6950	02	48	4C	18	11	3D	20	08
6958	CD	7C	68	02	49	58	18	06
6960	CD	7C	68	02	49	59	10	B1
6968	2A	65	5C	46	23	7E	D6	02
6970	20	1F	CD	7C	68	02	00	00
6978	D9	7B	1F	9F	A2	D9	F5	28
6980	04	D9	0B	0B	D9	CD	E9	68
6988	F1	28	19	D9	03	00	D9	18
6990	13	3D	20	12	CD	7C	68	04
6998	00	00	00	00	CD	E9	68	CD
69A0	E9	68	23	23	23	23	10	C4
69A8	2A	65	5C	46	23	7E	CD	DF
69B0	68	10	F9	18	0A	AF	32	3C

*Mastering Machine Code on your ZX Spectrum*

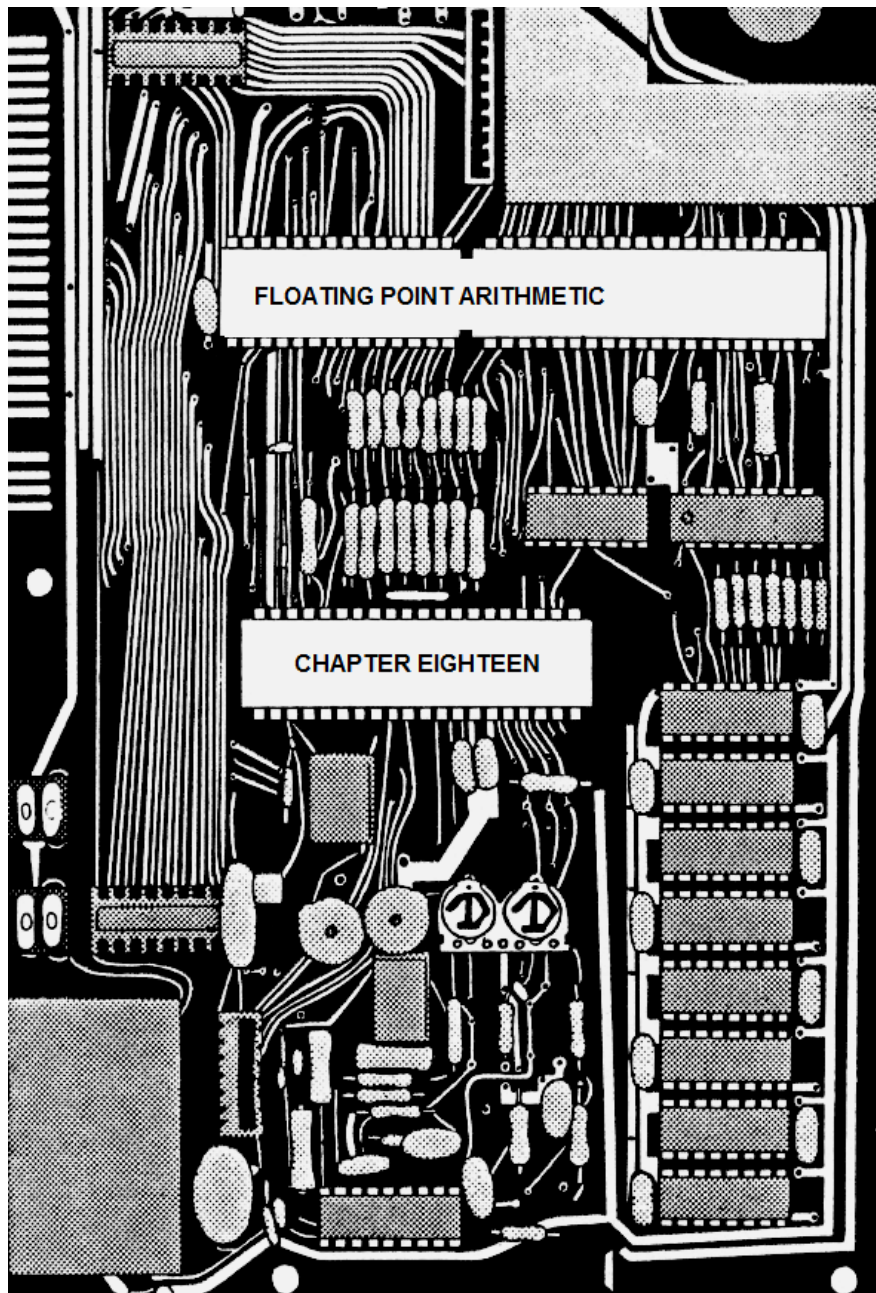
69B8	5C	D9	ED	4B	F8	7F	D9	2A
69C0	65	50	36	00	D9	11	00	00
69C8	D9	3E	0D	CD	DF	68	D9	CD
69D0	C7	68	3E	20	CD	DF	68	0A
69D8	03	D5	D9	C1	FE	76	20	16
69E0	0D	28	12	21	F1	69	06	04
69E8	7E	23	CD	DF	68	10	F9	18
69F0	D8	48	41	40	54	0C	FE	CB
6AE0	D9	C9	00	F3	6A	12	6B	22
6AE8	6B	52	6B	DD	6C	E2	6C	5E
6AF0	6B	64	6B	06	17	9A	4E	4F
6AF8	D0	45	58	20	41	46	20	41
6B00	46	A7	44	4A	4E	5A	20	82
6B08	4A	52	20	82	09	4A	D2	64
6B10	81	82	07	07	09	4C	C4	4C
6B18	81	83	00	41	41	44	44	20
6B20	81	8C	09	4C	C4	BA	28	42
6B28	43	29	2C	C1	41	2C	28	42
6B30	43	A9	28	44	45	22	2C	C1
6B38	41	2C	28	44	45	A9	28	03
6B40	29	2C	81	01	2C	28	03	A9
6B48	28	03	29	2C	C1	41	2C	28
6B50	03	A9	07	05	09	49	4E	C3
6B58	8C	09	44	45	C0	6C	09	4C
6B60	C4	44	81	82	BA	52	4C	43
6B68	C1	52	52	43	C1	52	4C	C1
6B70	52	52	C1	44	41	C1	43	50
6B78	CC	53	43	C6	43	43	C6	09
6B80	4C	C4	44	85	34	85	00	97
6B88	6B	9C	6B	B7	6B	BD	6B	E9
6B90	6B	F1	6B	00	6C	03	6C	09
6B98	52	45	D4	A4	07	05	09	50
6BA0	4F	D0	94	9A	52	45	D4	45
6BA8	58	D8	4A	50	20	28	01	A9
6BB0	4C	44	20	53	50	2C	81	09

6BB8	4A	D0	64	81	83	BA	4A	50
6BC0	20	83	A0	4F	55	54	20	28
6BC8	02	29	2C	C1	49	4E	20	41
6BD0	2C	28	02	A9	45	58	20	28
6BD8	53	50	29	2C	81	45	58	20
6BE0	44	45	20	48	CC	44	C9	45
6BE8	C9	09	43	41	4C	CC	64	81
6BF0	83	07	06	09	50	55	53	C8
6BF8	94	81	43	41	4C	4C	20	83
6C00	34	81	82	09	52	53	D4	BA
6C08	30	B0	30	38	20	82	31	B0
6C10	31	B8	32	B0	32	B8	33	B0
6C18	33	B8	3A	52	4C	C3	52	52
6C20	C3	52	CC	52	D2	53	4C	C1
6C28	53	52	C1	A0	53	52	CC	01
6C30	A0	85	09	42	49	D4	5C	85
6C38	09	52	45	D3	5C	65	09	53
6C40	45	D4	5C	65	00	55	6C	5D
6C48	6C	66	6C	78	6C	6A	6C	8E
6C50	6C	9A	6C	A8	5C	09	49	CE
6C58	44	81	28	43	A9	41	4F	55
6C60	54	20	28	43	A9	84	07	08
6C68	41	53	42	43	20	48	CC	8C
6C70	41	41	44	43	20	48	CC	6C
6C78	07	08	41	40	44	20	28	03
6C80	A9	8C	09	4C	C4	4C	81	28
6C88	03	A9	81	4E	45	C7	07	05
6C90	81	52	45	54	CE	61	52	45
6C98	54	C9	9A	49	4D	20	B0	A0
6CA0	49	4D	20	B1	49	4D	20	B2
6CA8	BA	4C	44	20	49	2C	C1	4C
6CB0	44	20	52	2C	C1	4C	44	20
6CB8	41	2C	C9	4C	44	20	41	2C
6CC0	D2	52	52	C4	52	4C	C4	A0
6CC8	A0	1B	4C	C4	43	D0	49	CE



*Mastering Machine Code on your ZX Spectrum*

6CD0	4F	D4	BA	A0	A0	A0	A0	C9
6CD8	C4	49	D2	44	D2	09	49	4E
6CE0	C3	84	09	44	45	C3	84	1C
6CE8	00	00	00	00	00	00	00	00
6CF0	00	00	00	00	00	00	00	00
6CF8	00	00	00	00	00	00	00	00
6D00	00	00	00	00	00	00	00	00
6D08	00	00	00	00	00	00	00	00
6D10	00	00	00	00	00	00	00	00
6D18	00	00	00	00	00	00	00	00



FLOATING POINT ARITHMETIC

CHAPTER EIGHTEEN

# CHAPTER EIGHTEEN – FLOATING POINT ARITHMETIC

Arithmetic in the Spectrum ROM falls into two classes: *integer* and *floating point*. Both of these have one thing in common – numbers in both classes are represented in five bytes.

The five byte forms of the integers are quite simple to follow. Here is a list of the five byte forms of the integers zero to four – you should be able to spot a pattern quite easily:

<i>Decimal</i>	<i>Five byte form</i>				
0	00	00	00	00	00
1	00	00	01	00	00
2	00	00	02	00	00
3	00	00	03	00	00
4	00	00	04	00	00

And the negatives?

<i>Decimal</i>	<i>Five byte form</i>				
-1	00	FF	FF	FF	00
-2	00	FF	FE	FF	00
-3	00	FF	FD	FF	00
-4	00	FF	FC	FF	00

OK – I hope you've all worked out for yourselves what seems to be going on. Let's try some bigger numbers now and see what happens to them.

<i>Decimal</i>	<i>Five byte form</i>	
1000	00 00 E8 03 00	(Note that 1000d = 03E8h).
-1000	00 FF 18 FC 00	(Note that -1000d = FC18h).

In other words – the rule is as follows. For a positive number which may be written in two bytes in hex as “mmnn”, the five byte form is 00 00 nn mm 00. For a negative number you must add 65536d, and then the five byte form is 00 FF nn mm 00.

Decimals, on the other hand, are an entirely different kettle of fish. Numbers like 0.5 or PI – how are they stored? Because I think that knowing how the five byte form is built up will slightly help your understanding of the ROM, I shall give a brief explanation of how to turn decimal numbers into Sinclair numbers. It only takes a few simple steps.

First of all you need to check the size of such a number. The largest number which may be stored in five byte form is 1.7014118E38. (That is to say – 170,141,180,000,000,000,000,000,000,000, 000,000.) The smallest number is, of course, minus that quantity. No number outside this range may be handled by the Spectrum.

Now that we’ve established that the number we wish to convert is within the required range, we need to follow through a sequence of steps. Let’s give this number a name – let’s call it X.

Step one is to ignore the sign of the number (ie LET X1 = ABS X) and then compute a quantity called the EXPONENT by the following formula:

$$\text{LET EXPONENT} = 1 + \text{INT}(\text{LOG X1}/\text{LOG 2}).$$

The first byte of the five byte form is 80 plus this exponent. Let’s demonstrate this with a few examples:

<i>Number</i>	<i>Absolute Value</i>	<i>Exponent</i>	<i>Five Byte Form</i>
X	X1 = ABS X	1 + INT(LOG X1/LOG 2)	
0.375	0.375	-1	7F

*Mastering Machine Code on your ZX Spectrum*

PI	3.1415927	2	82
-1.5E20	1.5E20	68	C4

Step two is to compute a quantity called the MANTISSA. The following formula will work it out for you:

LET MANTISSA=X1 / 2 ↑ EXPONENT.

<i>Number</i>	<i>Exponent</i>	<i>Mantissa</i>
0.375	-1	0.75
PI	2	0.7853982
-1.5E20	68	0.5082198

Concealed somewhere in this mantissa are the last four bytes of the Sinclair form, so somehow we have to get them out. The best way for me to explain how to do this is to suggest you imagine a BASIC routine which does this:

```
LET A = MANTISSA
FOR I = 1 TO 4
LET A = 256*A
Next byte of five byte form is INT A written in hex.
LET A = A - INT A
NEXT I
```

The “variable” I’ve written in the above routine as A doesn’t really come into it – nor does the loop variable, I – they are just dummy variables, existing for illustrative purposes only. However, the *final value* of A is something we *shall* use in a moment, so this we ought to hang on to. Let’s see how using the above procedure affects our examples:

<i>Number</i>	<i>Five byte form</i>	<i>Final “A”</i>
0.375	7F C0 00 00 00	0
PI	82 C9 0F DA A1	0.6314368
-1.5E20	C4 82 1A B0 D4	0.2842112

The next stage is called “rounding” and works as follows. If the final value of A is greater than or equal to zero point five then you must *increment*, as a whole, the last four bytes of the five byte form (eg 00 00 00 29 increments to 00 00 00 2A, 23 29 FF FF increments to 23 2A 00 00, and so on). There is one point you do have to watch out for though. If the four bytes started their life as FF FF FF FF then you *don’t* “increment” them to 00 00 00 00 – what in fact happens is that these four bytes actually become 80 00 00 00, but the *first* byte of the five byte form is incremented. Our examples become rounded as follows:

<i>Number</i>	<i>Five byte form</i>
0.375	7F C0 00 00 00
PI	82 C9 0F DA A2
-1.5E20	C4 82 1A B0 D4

The final step is to re-introduce the sign of the original number X. The rule is very simple: if X was positive then subtract 80 from the second byte and if X was negative then do nothing.

<i>Number</i>	<i>Five byte form</i>
0.375	7F 40 00 00 00
PI	82 49 0F DA A2
-1.5E20	C4 82 1A B0 D4

The process is now complete. This is the final form of the five byte representation. For obvious reasons, the largest exponent you may have is FF, so the largest number which can be stored is FF 7F FF FF FF. This, as has already been stated, is 1.7014118E38. The “E” in the notation means “with the decimal point moved (in the above case) 38d places to the right”. Of course, only the first nine decimal digits will be stored accurately though. There is also a restriction on the smallest positive number you can have too, because if the first byte of the five byte form is 00 then the number is assumed to be an *integer* (ie of the form 00 ss nn mm 00). Thus, the smallest exponent you can get away with is 01, and so the smallest positive number you can have is 01 00 00 00 00, or in decimal 2.9387359E

## Mastering Machine Code on your ZX Spectrum

– 39 (zero is not a positive number). To you and me that's 0.000,000,000,000,000,000,000,000,000,000,000,000,002, 938,735,9 – which I'd say was pretty microscopic.

You can check all of this with the following BASIC program:

```
10 LET A = 0
20 LET B = PEEK 23627 + 256*PEEK 23628
30 FOR I = 1 TO 5
40 INPUT A$
50 PRINT A$;" ";
60 POKE B + I, 16*FN K(A$(1)) + FN K(A$(2))
70 NEXT I
80 PRINT ,A
90 GO TO 30
100 DEF FN K(X$) = CODE X$ - 48 - (7 AND X$>"9") - (32 AND X$>"F")
```

The program sets up a variable A, and then overwrites its previous value by POKEing into the variables area one byte at a time. If you run it and input "7F" / "40" / "00" / "00" / "00" you should get the number 0.375 as your result. Similarly, if you input "00" / "00" / "03" / "00" / "00" you'll find the number three printed. And so on.

An interesting little quirk is that if you input "00" as the first byte, and any *odd* number other than FF as the second byte, you'll get some rather unexpected results – for instance, "00" / "01" / "01" / "00" / "00" actually prints 1023. Another little curiosity is that inputting "02" / "D9" / "C7" / "DC" / "EC" gives the same result as "00" / "FF" / "00" / "00" / "00" (in theory this is minus zero). Don't panic! This doesn't really happen in the ROM. We made it happen by POKEing something which doesn't make sense. The ROM does at the best of times behave slightly more sensibly than human beings (although INT – 65536 leaves a lot to be desired).

## HOW TO USE FLOATING POINT NUMBERS PROPERLY

A number which takes five bytes to store cannot, of course, be stored in a register, or a register pair. We can, if we wish, store such a floating point number in a register *quintuplet* (such as AEDCB, for instance) but if we actually want to do any arithmetic then we'll certainly need to remember more than just one number at a time. So the question is, if we can't store these numbers in the registers, then where *can* we store them? Answer – There is an area of RAM set aside for just that purpose – the *calculator stack*.

The calculator stack is a stack just like the familiar stack we've been working with all the way through the book. It has just one or two differences: (i) it is stored in a different area of memory (see the diagram on page 165 of the Spectrum manual); (ii) the calculator stack “grows” *upwards*, not *downwards*, so that the topmost item on the calculator stack has a higher address than the second topmost; (iii) each item on the calculator stack occupies *five* bytes, not two; (iv) it can store both numbers and strings; and (v) CALL and RET do not in any way alter it.

There are three different ROM subroutines which are useful for pushing a number onto the calculator stack. The first of these is called STACK\_A which takes the integer stored in the A register, converts it to five byte form and pushes this form onto the calculator stack. Secondly, And very similar, there is a subroutine called STACK\_BC which takes the integer contained in the BC register pair and converts it to five byte form, and then stacks this form. The third one is just a bit more powerful as it gives us the full floating point potential. It is called STACK\_AEDCB and its job is to push the contents of the register *quintuplet* AEDCB onto the calculator stack. This last subroutine does no conversion at all because it assumes that the number is already in five byte form, with the first byte being in A, the second in E, and so on. The addresses are 2D28 (STACK\_A), 2D2B (STACK\_BC) and 2AB6 (STACK\_AEDCB). In hex:



*Mastering Machine Code on your ZX Spectrum*

CD282D	CALL STACK_A
CD2B2D	CALL STACK_BC
CDB62A	CALL STACK_AEDCB

Incidentally the first two instructions in the STACK\_A routine are LD C,A and LD B,00. It then leads straight into the STACK\_BC routine. This, in turn, loads A and E both with zero, moves BC into D and C (low part first), loads B with zero and then leaps straight into STACK\_AEDCB. Quite clever, methinks.

Conversely, there are subroutines which remove numbers from the calculator stack. These are called FP\_TO\_A, FP\_TO\_BC, and FP\_TO\_AEDCB. These subroutines are quite general and will never halt the program with an error report even if the number at the top of the stack won't fit into the designated register. Here then, is a breakdown of what happens when either FP\_TO\_A or FP\_TO\_BC is called. Suppose the number at the top of the calculator stack is called Y. This number is then POPped from the stack, irrespective of whether it will fit in its destination register. If it will fit with no problems at all, then the zero flag will be set and the carry flag will be reset. If problems occur then the flags will tell you what's going on...

If the carry flag is set, it means that the absolute value of the number (ABS Y) was too big to fit in the register. For instance, if FP\_TO\_A was used and the number was 1000d, then since 1000d cannot be stored in the A register alone this case would result in the carry flag being set.

If the zero flag is reset it means that the number was negative, however, (unless the carry flag is set) the absolute value of the number (ie ABS Y) will still be stored in the designated register.

The subroutine FP\_TO\_AEDCB, of course, can never cause any problems at all, since any five byte number can always fit into five registers.

The addresses are 2DD5 (FP\_TO\_A), 2DA2 (FP\_TO\_BC) and 2BF1 (FP\_TO\_AEDCB). In hex:

CDD52D	CALL FP_TO_A
CDA22D	CALL FP_TO_BC
CDF12B	CALL FP_TO_AEDCB

Arithmetic with five byte numbers is quite straightforward. The addresses are:

3014	ADD	Addition.
300F	SUB	Subtraction.
30CA	MULT	Multiplication.
31AF	DIV	Division.

They work like this. The five byte number stored at the address specified by HL (ie the number whose five bytes are stored in RAM at addresses (HL), (HL + 1), (HL + 2), (HL + 3) and (HL + 4)) is added by, multiplied by, divided by, or has a specified number subtracted from it. The second number is stored in RAM at the address specified by DE. The result of the calculation is stored at the address specified by HL.

To multiply together the two numbers at the top of the calculator stack, one method would be as follows:

2A655C	LD HL,(STKEND)
11FBFF	LD DE,FFFB
19	ADD HL,DE
E5	PUSH HL
22655C	LD (STKEND),HL
19	ADD HL,DE
D1	POP DE
CDCA30	CALL MULT

## Mastering Machine Code on your ZX Spectrum

Can you follow exactly what is going on? HL is loaded with the contents of the system variable STKEND – which gives the address of the first byte after the end of the calculator stack. DE is loaded with minus five and added to HL, so HL is decreased by five. This new value is loaded back into STKEND because we start off with two items on the stack and we want to end up with only one. This is the address of one of the numbers to be multiplied. If you follow the listing through carefully you'll see that DE ends up with this value. HL is decreased by five *again*, to find the start of the other number to be multiplied.

To check that it really works, run this program:

```
3E06      START      LD A,06
CD282D                    CALL STACK_A
3E07                    LD A,07
CD282D                    CALL STACK_A
2A655C                    LD HL,(STKEND)
11FBFF                    LD DE,FFFB
19                      ADD HL,DE
E5                      PUSH HL
22655C                    LD (STKEND),HL
19                      ADD HL,DE
D1                      POP DE
CDCA30                    CALL MULT
CDA22D                    CALL FP_TO_BC
C9                      RET
```

Run it by typing PRINT USR START. What do you get?

But surely there must be easier ways to multiply six by seven. After all, the above program does look very complicated and not something you'd easily remember. Well it's here that we start making full use of the ROM. The following program does exactly the same job, and I shall explain why:

```

3E06          LD A,06
CD282D       CALL STACK_A
3E07          LD A,07
CD282D       CALL STACK_A
EF           RST 28
04           DEFB 04
38           DEFB 38
CDA22D       CALL FP_TO_BC
C9           RET

```

In the Spectrum, RST 28 means perform five byte arithmetic. The data which follows tells it precisely what calculations it's supposed to do. The byte 04 means "multiply" – all of the shuffling around of the calculator stack is done automatically. The byte 38 is used after a RST 28 instruction to indicate that there is no more data to come and the next byte will be a machine code instruction.

The RST 28 data codes are: add = 0F, sub = 03, mult = 04 and div = 05. Don't forget you'll need a byte 38 as well though to end the data.

RST 28 allows you to do much, much more than simple arithmetic. All of the unary functions of the Spectrum are available to you. The data code for any particular function is just the character code of that function minus 93. For instance, the character code of SIN is B2. B2 minus 93 is 1F. This means we can find the SIN of the number at the top of the calculator stack using the sequence:

```

EF           RST 28
1F           DEFB 1F (sin)
38           DEFB 38 (end-calc).

```

To multiply two numbers (at the top of the calculator stack) together and then find the square root of the result we can use the sequence:

## *Mastering Machine Code on your ZX Spectrum*

EF	RST 28
04	DEFB 04 (mult)
28	DEFB 28 (sqr)
38	DEFB 38 (end-calc).

If you're not absolutely convinced yet, run this program, which multiplies five by twenty, and then takes the square root:

3E05	LD A,05
CD282D	CALL STACK_A
3E14	LD A,14
CD282D	CALL STACK_A
EF	RST 28
04	DEFB 04 (mult)
28	DEFB 28 (sqr)
38	DEFB 38 (end_calc)
CDA22D	CALL FP_TO_BC
C9	RET

You'll notice that this is the first time we've used more than one code in the RST 28 data. In fact, you can use as many as you like, provided you end the list with 38.

To save you working it out for yourselves, Appendix Six contains a list of available RST 28 codes. Most of these you will be able to use straight off without any additional explanation from me.

Some of the entries in the list may surprise you. For instance, we have two functions – one called “usr\_sr” and the other called “usr\_n”. “usr\_s” is the function USR (string) in BASIC which gives the address of the user-defined graphic pixel breakdown for any character. “usr-n” is the function USR (number), and this can be very confusing. What happens when the ROM comes across the byte 2D (usr\_n) in RST 28 data is that the topmost item on the calculator stack is loaded into BC. Then a machine code subroutine at address BC is called. On returning from this subroutine,

the current value of BC is pushed onto the calculator stack (in five byte form, of course) and then the *next* byte of RST 28 data (ie after the 2D) is interpreted.

PEEK (byte 2B) works the same way, POPping an address from the calculator stack, PEEKing there, and PUSHing this value back onto the stack.

All of the functions when used in this way will remove the number currently at the top of the calculator stack and replace it by a new one. For instance, if the number at the top of the stack is 3.5 and the function INT is called, the 3.5 will be removed and replaced by the number three.

The string functions CODE, VAL and LEN, also VAL\$, STR\$ and CHR\$ need a small amount of explanation. You see, as well as storing numbers, the calculator stack can also store *strings*, so if you start off with the number 2000d on the top of the calculator stack and then call STR\$ (by using code 2E in RST 28 data) then the item at the top of the calculator stack will now be the string "2000". You can demonstrate this with the following:

```

01D007          LD BC,2000d
CD2B2D          CALL STACK_BC
EF              RST 28
2E              DEFB 2E (str$)
1C              DEFB 1C (code)
38              DEFB 38 (exit)
CDA22D          CALL FP_TO_BC
C9              RET

```

This should produce the result of CODE STR\$ 2000. Does it?

## USING THE CALCULATOR'S MEMORY

If you take a quick glance at the manual you'll see that one of the system variables, MEMBOT, is thirty bytes long. This is the calculator's memory area. A quick calculation involving dividing by five, if you're up to it, shows that this leaves enough room to store six different five byte numbers. The six different areas of memory may each be used by RST 28 to store or retrieve either numbers or strings from the top of the calculator stack. There are twelve different codes to achieve this – these are:

C0	Stores in memory zero.
C1	Stores in memory one.
C2	Stores in memory two.
C3	Stores in memory three.
C4	Stores in memory four.
C5	Stores in memory five.
E0	Recalls memory zero.
E1	Recalls memory one.
E2	Recalls memory two.
E3	Recalls memory three.
E4	Recalls memory four.
E5	Recalls memory five.

Storing a number copies it from the top of the calculator stack (without physically removing it) and recalling a number extends the stack by one item by appending the contents of the specified memory to the top of the stack. (It doesn't overwrite the previous top item.)

There are some warnings which go with these memories, however. Firstly, certain functions (SIN, COS and STR\$ to name but a few) will erase the former contents of memories zero, one and two, so to be really safe you should only ever use memories three, four and five. In addition, there is the added complication that whenever you use RST 10 to print a graphics character from the number keys (ie those characters with codes 80 to 8F)

then memories zero and one will be erased. Bearing all this in mind, let's see what we can do from here.

Suppose we want to find  $\text{SIN } X + \text{COS } X$ . We may use the following technique. Assume that  $X$  is at the top of the stack:

EF	RST 28
C5	DEFB C5 (store_5)
1F	DEFB 1F (sin)
E5	DEFB E5 (recall 5)
20	DEFB 20 (cos)
0F	DEFB 0F (add)
38	DEFB 38 (end_calc)

Notice that  $X$  does not need to be recalled immediately after it is stored because the act of storing it does not at the same time remove it from the stack. Storing a number in a memory leaves the length of the calculator stack unchanged. Recalling a number, on the other hand, actually increases the length of the calculator stack because one more item has to be PUSHed onto it. The above routine works by changing  $X$  to  $\text{SIN } X$ , re-PUSHing  $X$  onto the stack and changing this  $X$  into  $\text{COS } X$ . At this point there are two items on the stack –  $\text{SIN } X$  and  $\text{COS } X$ . The ADD routine will then replace both of these by one single result – the result we were after –  $\text{SIN } X + \text{COS } X$ .

We have now performed a fairly complex trigonometric function in just *seven bytes!*

There are one or two other things you can do with RST 28. We can use the logic functions AND and OR (that is BASIC AND and BASIC OR). You can swap the two topmost items, you can delete or duplicate the topmost item, and so on. What other interesting tricks can we play?



## RANDOM NUMBERS

The following routine will place a random decimal between zero and one (strictly less than one) at the top of the calculator stack. See if you can deduce how it works:

```
3EA5          LD A,"RND"  
CD282D       CALL STACK_A  
061D         LD B,1D  
EF           RST 28  
2F           DEFB 2F (chr$)  
1D           DEFB 1D (val)  
38           DEFB 38 (end_calc)
```

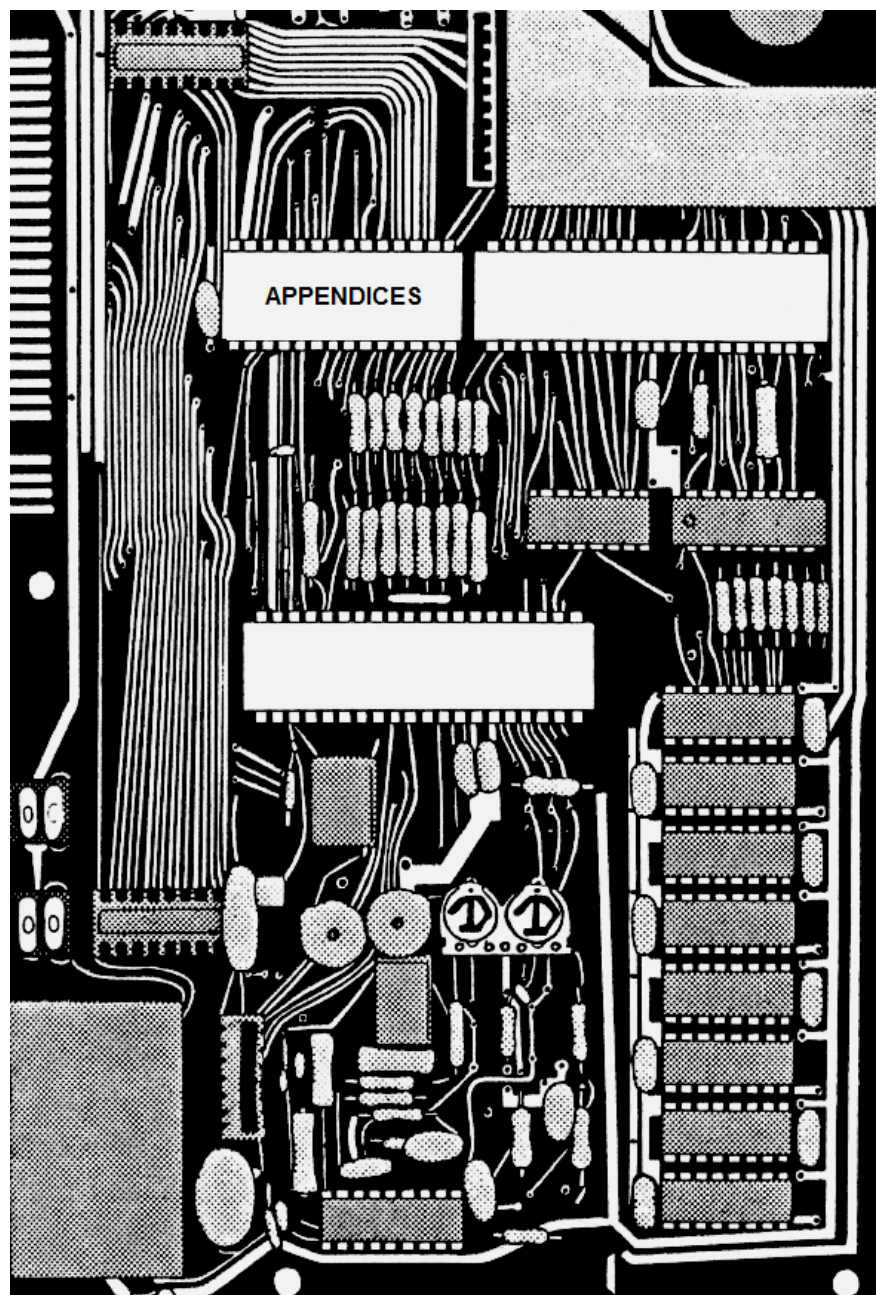
And for completeness, the following routine will leave in the BC register pair a random integer between one and the initial value of BC:

```
CD2B2D       CALL STACK_BC  
3EA5          LD A,"RND"  
CD282D       CALL STACK_A  
061 D        LD B,1D  
EF           RST 28  
2F           DEFB 2F (chr$)  
1D           DEFB 1D (val)  
04           DEFB 04 (mult)  
27           DEFB 27 (init)  
38           DEFB 38 (end_calc)  
CDA22D       CALL FP_TO_BC  
03           INC BC
```

The following sequence will raise one number to the power of another. Can you see why? After RST 28: 01 25 04 26 34. (Note that data byte 06 achieves more or less the same result all on its own.)

We can do far, far more mathematics with RST 28 than we can in a single BASIC expression – evaluate Taylor series for a start. However, that's something I shall leave for another day.

To be continued...



# APPENDICES

These appendices are designed to give you quick and easy reference to most of the things you'd want to look up.

A detailed list of the precise effect of each Z80 instruction may be found in Chapter Eight. This should be treated as a separate appendix.

The appendices are as follows:

- APPENDIX ONE – A conversion table from hex to decimal and *vice versa*, and from hex to binary and *vice versa*.
- APPENDIX TWO – The system variables.
- APPENDIX THREE – A conversion table from hex to assembly.
- APPENDIX FOUR – A conversion table from assembly to hex, including the effect of each instruction on the flags.
- APPENDIX FIVE – The Spectrum character set.
- APPENDIX SIX – Meanings of some of the data codes used after RST 28.

# APPENDIX ONE

	<u>0</u>	<u>1</u>	<u>2</u>	<u>3</u>	<u>4</u>	<u>5</u>	<u>6</u>	<u>7</u>	<u>8</u>	<u>9</u>	<u>A</u>	<u>B</u>	<u>C</u>	<u>D</u>	<u>E</u>	<u>F</u>
<u>0</u>	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
<u>1</u>	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
<u>2</u>	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
<u>3</u>	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
<u>4</u>	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
<u>5</u>	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
<u>6</u>	96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111
<u>7</u>	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127
<u>8</u>	128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143
<u>9</u>	144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159
<u>A</u>	160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175
<u>B</u>	176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191
<u>C</u>	192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207
<u>D</u>	208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223
<u>E</u>	224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239
<u>F</u>	240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
8	-128	-127	-126	-125	-124	-123	-122	-121	-120	-119	-118	-117	-116	-115	-114	-113
9	-112	-111	-110	-109	-108	-107	-106	-105	-104	-103	-102	-101	-100	-99	-98	-97
A	-96	-95	-94	-93	-92	-91	-90	-89	-88	-87	-86	-85	-84	-83	-82	-81
B	-80	-79	-78	-77	-76	-75	-74	-73	-72	-71	-70	-69	-68	-67	-66	-65
C	-64	-63	-62	-61	-60	-59	-58	-57	-56	-55	-54	-53	-52	-51	-50	-49
D	-48	-47	-46	-45	-44	-43	-42	-41	-40	-39	-38	-37	-36	-35	-34	-33
E	-32	-31	-30	-29	-28	-27	-26	-25	-24	-23	-22	-21	-20	-19	-18	-17
F	-16	-15	-14	-13	-12	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
A	1010
B	1011
C	1100
D	1101
E	1110
F	1111

## APPENDIX TWO

Decimal	Hex		Name
23552	5C00	IY+C6	KSTATE
23560	5C08	IY+CE	LAST_K
23561	5C09	IY+CF	REPDEL
23562	5C0A	IY+D0	REPPER
23563	5C0B	IY+D1	DEFADD
23565	5C0D	IY+D3	K_DATA
23566	5C0E	IY+D4	TVDATA
23568	5C10	IY+D6	STRMS
23606	5C36	IY+FC	CHARS
23608	5C38	IY+FE	RASP
23609	5C39	IY+FF	PIP
23610	5C3A	IY+00	ERR_NR
23611	5C3B	IY+01	FLAGS
23612	5C3C	IY+02	TVFLAG
23613	5C3D	IY+03	ERR_SP

Decimal	Hex		Name
23659	5C6B	IY+31	DF_SZ
23660	5C6C	IY+32	S_TOP
23662	5C6E	IY+34	OLDPPC
23664	5C70	IY+36	OSPPC
23665	5C71	IY+37	FLAGX
23666	5C72	IY+38	STRLEN
23668	5C74	IY+3A	T_ADDR
23670	5C76	IY+3C	SEED
23672	5C78	IY+3E	FRAMES
23675	5C7B	IY+41	UDG
23677	5C7D	IY+43	COORDS
23679	5C7F	IY+45	P_POSN
23680	5C80	IY+46	PR_CC
23681	5C81	IY+47	SPARE1
23682	5C82	IY+48	ECHO_E

Mastering Machine Code on your ZX Spectrum

Decimal	Hex		Name
23615	5C3F	IY+05	LISTSP
23617	5C41	IY+07	MODE
23618	5C42	IY+08	NEWPPC
23620	5C44	IY+0A	NSPPC
23621	5C45	IY+0B	PPC
23623	5C47	IY+0D	SUBPPC
23624	5C48	IY+0E	BORDCR
23625	5C49	IY+0F	E_PPC
23627	5C4B	IY+11	VARS
23629	5C4D	IY+13	DEST
23631	5C4F	IY+15	CHANS
23633	5C51	IY+17	CURCHL
23635	5C53	IY+19	PROG
23637	5C55	IY+1B	NXTLIN
23639	5C57	IY+1D	DATADD
23641	5C59	IY+1F	E_LINE
23643	5C5B	IY+21	K_CUR
23645	5C5D	IY+23	CH_ADD
23647	5C5F	IY+25	X_PTR
23649	5C61	IY+27	WORKSP
23651	5C63	IY+29	STKBOT
23653	5C65	IY+2B	STKEND
23655	5C67	IY+2D	B_REG
23656	5C68	IY+2E	MEM
23658	5C6A	IY+30	FLAGS2

Decimal	Hex		Name
23684	5C84	IY+4A	DF_CC
23686	5C86	IY+4C	DFCCL
23688	5C88	IY+4E	S_POSN
23690	5C8A	IY+50	SPOSNL
23692	5C8C	IY+52	SCR_CT
23693	5C8D	IY+53	ATTR_P
23694	5C8E	IY+54	MASK_P
23695	5C8F	IY+55	ATTR_T
23696	5C90	IY+56	MASK_T
23697	5C91	IY+57	P_FLAG
23698	5C92	IY+58	MEMBOT
23728	5CB0	IY+76	SPARE2
23730	5CB2	IY+78	RAMTOP
23732	5CB4	IY+7A	P_RAMT

SYSTEM VAR.	ADDR	IY DISP	NO. BYTES	PURPOSE
ATTR_P	5C8D	53	1	Permanent current colours (as used by CLS etc).
ATTR_T	5C8F	55	1	Temporary current colours (as used by RST 10).
BORDCR	5C48	0E	1	Temporary current colours for lower part of screen. Also doubles as store for border colour *8.
B_REG	5C67	2D	1	Used by five-byte calculator.
CHANS	5C4F	15	2	Address of start of channel information area.
CHARS	5C36	FC	2	Address of standard character set minus 0100.
CH_ADD	5C5D	23	2	Address of next character to interpret.
COORDS	5C7D	43	2	Coordinates of last point PLOTted.
CURCHL	5C51	17	2	Address within channel information area of current input/output data.
DATADD	5C57	1D	2	Address within program of character after last DATA item read.
DEFADD	5C0B	D1	2	Address of user defined function arguments.
DEST	5C4D	13	2	Address of variable being assigned.
DF_CC	5C84	4A	2	Address of print position in upper part of screen.
DFCCL	5C86	4C	2	Address of print position in lower part of screen.
DF_SZ	5C6B	31	1	Number of lines in lower part of screen.



*Mastering Machine Code on your ZX Spectrum*

SYSTEM VAR.	ADDR	IY DISP	NO. BYTES	PURPOSE
ECHO_E	5C82	48	2	Coordinates in lower part of screen beyond which "cursor-right" won't work.*
ERR_NR	5C3A	00	1	Current report code minus one.
ERR_SP	5C3D	03	2	Address of return address from RST 08.
E_LINE	5C59	1F	2	Address of start of edit line.
E_PPC	5C49	0F	2	Line number of line with cursor.
FLAGS	5C3B	01	1	Various flags.
FLAGS2	5C6A	30	1	Various flags.
FLAGX	5C71	37	1	Various flags.
FRAMES	5C78	3E	3	Incremented once for each TV frame displayed.
KSTATE	5C00	C6	8	Keyboard information used by repeat facility.
K_CUR	5C5B	21	2	Address of cursor.
K_DATA	5C0D	D3	1	Stores second byte of colour control codes entered from the keyboard. (eg both-shifts 4 stores byte 04).
LAST_K	5C08	CE	1	Character code of last key registered.
LISTSP	5C3F	05	2	Address of return address from automatic listing.
MASK_P	5C8E	54	1	Permanent attribute mask.
MASK_T	5C90	56	1	Temporary attribute mask.
MEM	5C68	2E	2	Address of start of calculator's memory area.

SYSTEM VAR.	ADDR	IY DISP	NO. BYTES	PURPOSE
MEMBOT	5C92	58	1E	Area used (i) as calculator memory, (ii) to compute decimal form of floating point numbers, and (iii) to compute graphics characters 80 to 8F, when printed.
MODE	5C41	07	1	Current cursor mode. (00=K/L/C, 01=E, 02=G).
NEWPPC	5C42	08	2	Line number of line to be jumped to.
NSPPC	5C44	0A	1	Statement number within line to be jumped to.
NXTLIN	5C55	1B	2	Address of next program line to be executed.
OLDPPC	5C6E	34	2	Line number to which CONTINUE jumps.
OSPPC	5C70	36	1	Statement number within line to which CONTINUE jumps.
PIP	5C39	FF	1	Length of keyboard click.
PPC	5C45	0B	2	Line number of line being executed.
PROG	5C53	19	2	Address of start of BASIC program area.
PR_CC	5C80	46	1	LPRINT position (high part assumed 5B).
P_FLAG	5C91	57	1	Various flags.
P_POSN	5C7F	45	1	x coordinate of LPRINT position.*
P_RAMT	5CB4	7A	2	Address of last byte of physical RAM.
RAMTOP	5CB2	78	2	Address of last byte erased by NEW.
RASP	5C38	FE	1	Length of warning buzz.

*Mastering Machine Code on your ZX Spectrum*

SYSTEM VAR.	ADDR	IY DISP	NO. BYTES	PURPOSE
REPDEL	5C09	CF	1	Delay before key repeats (in 1/50ths of a second).
REPPER	5C0A	D0	1	Delay between successive repeats (in 1/50th secs).
SCR_CT	5C8C	52	1	Number of scrolls to be performed plus one.
SEED	5C76	3C	2	Seed for random number generator.
SPARE1	5C81	47	1	One spare byte.
SPARE2	5CB0	76	2	Two spare bytes.
STKBOT	5C63	29	2	Address of start of calculator stack.
STKEND	5C65	2B	2	Address of start of spare RAM.
STRLEN	5C72	38	2	Length of string being assigned procrusteally.
STRMS	5C10	D6	26	Addresses of channels attached to streams.
SUBPPC	5C47	0D	1	Statement within line being executed.
S_POSN	5C88	4E	2	Coordinates of PRINT position in upper part of screen.*
SPOSNL	5C8A	50	2	Coordinates of PRINT position in lower part of screen.*
S_TOP	5C6C	32	2	Line number of line at top of screen.
TVDATA	5C0E	D4	2	2nd and 3rd bytes of control codes going to TV.
TVFLAG	5C3C	02	1	Various flags.
T_ADDR	5C74	3A	2	Address of next item in syntax table.

SYSTEM VAR.	ADDR	IY DISP	NO. BYTES	PURPOSE
UDG	5C7B	41	2	Address of start of user defined graphics area.
VAR5	5C4B	11	2	Address of start of variables area.
WORKSP	5C61	27	2	Address of start of temporary workspace.
X_PTR	5C3F	25	2	Address of first syntactically incorrect character.

\*coordinates are considered in the “-y,x” convention, in which the first coordinate is 28 minus the row number, and the second coordinate is 21 minus the column number. These two coordinates are of course reversed when stored in the system variables as are all two byte quantities.

# APPENDIX THREE

	0	1	2	3	4	5	6	7
0	NOP	LD BC,mn	LD (BC),a	INC BC	INC B	DEC B	LD B,n	RLCA
1	DJNZ e	LD DE,mn	LD (DE),a	INC DE	INC D	DEC D	LD D,n	RLA
2	JR NZ,e	LD HL,mn	Ld (pq),HL	INC HL	INC H	DEC H	LD H,n	DAA
3	JR NC,e	LD SP,mn	LD (pq),A	INC SP	INC (HL)	DEC (HL)	LD (HL),n	SCF
4	LD B,B	LD B,C	LD B,D	LD B,E	LD B,H	LD B,L	LD B,(HL)	LD B,A
5	LD D,B	LD D,C	LD D,D	LD D,E	LD D,H	LD D,L	LD D,(HL)	LD D,A
6	LD H,B	LD H,C	LD H,D	LD H,E	LD H,H	LD H,L	LD H,(HL)	LD H,A
7	LD (HL),B	LD (HL),C	LD (HL),D	LD (HL),E	LD (HL),H	LD (HL),L	HALT	LD (HL),A
8	ADD A,B	ADD A,C	ADD A,D	ADD A,E	ADD A,H	AND A,L	ADD A,(HL)	ADD A,A
9	SUB B	SUB C	SUB D	SUB E	SUB H	SUB L	SUB (HL)	SUB A
A	AND B	AND C	AND D	AND E	AND H	AND L	AND (HL)	AND A
B	OR B	OR C	OR D	OR E	OR H	OR L	OR (HL)	OR A
C	RET NZ	POP BC	JP NZ,pq	JP pq	CALL NZ,pq	PUSH BC	ADD A,n	RST 00
D	RET NC	POP DE	JP NC,pq	OUT (n),A	CALL NC,pq	PUSH DE	SUB n	RST 10
E	RET PO	POP HL	JP PO,pq	EX (SP),HL	CALL PO,pq	PUSH HL	AND n	RST 20
F	RET P	POP AF	JP P,pq	DI	CALL P,pq	PUSH AF	OR n	RST 30

	8	9	A	B	C	D	E	F
0	EX AF,AF	ADD HL,BC	LD A,(BC)	DEC BC	INC C	DECC	LD C,n	RRCA
1	JR e	ADD HL,DE	LD A,(DE)	DEC DE	INC E	DEC E	LD E,n	RRA
2	JR Z,e	ADD HL,HL	LD HL,(pq)	DEC HL	INC L	DECL	LD L,n	CPL
3	JR C,e	ADD HL,SP	LD A,(pq)	DEC SP	INC A	DECA	LD A,n	CCF
4	LD C,B	LD C,C	LD C,D	LD C,E	LD C,H	LD C,L	LD C,(HL)	LD C,A
5	LD E,B	LD E,C	LD E,D	LD E,E	LD E,H	LD E,L	LD E,(HL)	LD E,A
6	LD L,B	LD L,C	LD L,D	LD L,E	LD L,H	LD L,L	LD L,(HL)	LD L,A
7	LD A,B	LD A,C	LD A,D	LD A,E	LD A,H	LD A,L	LD A,(HL)	LD A,A
8	ADC A,B	ADC A,C	ADC A,D	ADC A,E	ADC A,H	ADC A,L	ADC A,(HL)	ADC A,A
9	SBC A,B	SBC A,C	SBC A,D	SBC A,E	SBC A,H	SBC A,L	SBC A,(HL)	SBC A,A
A	XOR B	XOR C	XOR D	XOR E	XOR H	XOR L	XOR (HL)	XOR A
B	CP B	CP C	CP D	CP E	CP H	CP L	CP (HL)	CP A
C	RET Z	RET	JP Z,pq	•	CALL Z,pq	CALL pq	ADC A,n	RST 08
D	RET C	EXX	JP C,pq	IN A,(n)	CALL C,pq	•	SBC A,n	RST 18
E	RET PR	JP (HL)	JP PE,pq	EX DE,HL	CALL PE,pq	•	XOR n	RST 28
F	RET M	LD SP,HL	JP M,pq	EI	CALL M,pq	•	CP n	RST 38

AFTER B											
0	RLCB	RLCC	RLCD	RLCE	RLCH	RLCL	RLCHL	RLCA	RLCB	RLCC	RLCD
1	RLB	RLC	RLD	RL E	RLH	RL L	RLHL	RLA	RLB	RLC	RLD
2	SLAB	SLAC	SLAD	SLAE	SLAH	SLAL	SLAHL	SLAA	SLAB	SLAC	SLAD
3											
4	BT0B	BT0C	BT0D	BT0E	BT0H	BT0L	BT0HL	BT0A	BT0B	BT0C	BT0D
5	BT2B	BT2C	BT2D	BT2E	BT2H	BT2L	BT2HL	BT2A	BT2B	BT2C	BT2D
6	BT4B	BT4C	BT4D	BT4E	BT4H	BT4L	BT4HL	BT4A	BT4B	BT4C	BT4D
7	BT6B	BT6C	BT6D	BT6E	BT6H	BT6L	BT6HL	BT6A	BT6B	BT6C	BT6D
8	RES0B	RES0C	RES0D	RES0E	RES0H	RES0L	RES0HL	RES0A	RES0B	RES0C	RES0D
9	RES2B	RES2C	RES2D	RES2E	RES2H	RES2L	RES2HL	RES2A	RES2B	RES2C	RES2D
A	RES4B	RES4C	RES4D	RES4E	RES4H	RES4L	RES4HL	RES4A	RES4B	RES4C	RES4D
B	RES6B	RES6C	RES6D	RES6E	RES6H	RES6L	RES6HL	RES6A	RES6B	RES6C	RES6D
C	RES8B	RES8C	RES8D	RES8E	RES8H	RES8L	RES8HL	RES8A	RES8B	RES8C	RES8D
D	SET2B	SET2C	SET2D	SET2E	SET2H	SET2L	SET2HL	SET2A	SET2B	SET2C	SET2D
E	SET4B	SET4C	SET4D	SET4E	SET4H	SET4L	SET4HL	SET4A	SET4B	SET4C	SET4D
F	SET6B	SET6C	SET6D	SET6E	SET6H	SET6L	SET6HL	SET6A	SET6B	SET6C	SET6D
AFTER D											
0											
1											
2											
3											
4	INB(C)	OUT C(B)	SBC HL(B)	LD HL(B,C)	NEG	RETR	M 0	LD IA	INB(C)	OUT C(B)	SBC HL(B)
5	INB(C)	OUT C(D)	SBC HL(D)	LD HL(D,C)			M 1	LD IA	INB(C)	OUT C(D)	SBC HL(D)
6								RRD			
7											
8											
9	LD	CH	IN	OUT					LD	CH	IN
A	LD	CH	INR	OUTR					LD	CH	INR
B											
C											
D											
E											
AFTER ID											
0											
1											
2											
3											
4											
5											
6											
7											
8											
9											
A											
B											
C											
D											
E											
AFTER (H)											
0											
1											
2											
3											
4											
5											
6											
7											
8											
9											
A											
B											
C											
D											
E											
AFTER (L)											
0											
1											
2											
3											
4											
5											
6											
7											
8											
9											
A											
B											
C											
D											
E											



AFTER DDCBdd		AFTER FDCBdd	
6	E	6	E
0 RLC (IX+d)	RRC (IX+d)	0 RLC (IY+d)	RRC (IY+d)
1 RL (IX+d)	RR (IX+d)	1 RL (IY+d)	RR (IY+d)
2 SLA (IX+d)	SRA (IX+d)	2 SLA (IY+d)	SRA (IY+d)
3 –	SRL (IX+d)	3 –	SRL (IY+d)
4 BIT 0,(IX+d)	BIT 1,(IX+d)	4 BIT 0,(IY+d)	BIT 1,(IY+d)
5 BIT 2,(IX+d)	BIT 3,(IX+d)	5 BIT 2,(IY+d)	BIT 3,(IY+d)
6 BIT 4,(IX+d)	BIT 5,(IX+d)	6 BIT 4,(IY+d)	BIT 5,(IY+d)
7 BIT 6,(IX+d)	BIT 7,(IX+d)	7 BIT 6,(IY+d)	BIT 7,(IY+d)
8 RES 0,(IX+d)	RES 1,(IX+d)	8 RES 0,(IY+d)	RES 1,(IY+d)
9 RES 2,(IX+d)	RES 3,(IX+d)	9 RES 2,(IY+d)	RES 3,(IY+d)
A RES 4,(IX+d)	RES 5,(IX+d)	A RES 4,(IY+d)	RES 5,(IY+d)
B RES 6,(IX+d)	RES 7,(IX+d)	B RES 6,(IY+d)	RES 7,(IY+d)
C SET 0,(IX+d)	SET 1,(IX+d)	C SET 0,(IY+d)	SET 1,(IY+d)
D SET 2,(IX+d)	SET 3,(IX+d)	D SET 2,(IY+d)	SET 3,(IY+d)
E SET 4,(IX+d)	SET 5,(IX+d)	E SET 4,(IY+d)	SET 5,(IY+d)
F SET 6,(IX+d)	SET 7,(IX+d)	F SET 6,(IY+d)	SET 7,(IY+d)



## APPENDIX FOUR

This table shows, in more or less alphabetical order, each Z80 instruction and either its hexadecimal code or the words “table 1”, “table 2” or “table 3”. In such a case, the appropriate code will be found in the table given. This appendix also lists the flags which are altered by each instruction. The symbols used for this have the following meanings:

- The flag is not altered by the instruction.
- @ The flag is altered as you would expect it to be.
- 0 The flag becomes zero.
- 1 The flag becomes one.
- ? The flag is altered unpredictably.
- x Special case – an explanation will be given.

<u>INSTRUCTIONS</u>		<u>FLAGS</u>								
<u>Opcode</u>	<u>Hexcode</u>	<u>S</u>	<u>Z</u>	<u>–</u>	<u>H</u>	<u>–</u>	<u>P</u>	<u>N</u>	<u>C</u>	
ADC A,r	table 1	@	@	–	@	–	@	0	@	
ADC HL,s	table 2	@	@	–	@	–	@	0	@	
ADD A,r	table 1	@	@	–	@	–	@	0	@	
ADD HL,s	table 2	–	–	–	@	–	–	0	@	
ADD IX,s	table 2	–	–	–	@	–	–	0	@	
ADD IY,s	table 2	–	–	–	@	–	–	0	@	
AND r	table 1	@	@	–	1	–	@	0	0	
BIT b,r	table 1	?	@	–	1	–	?	0	0	
CALL pq	CDqqpp	–	–	–	–	–	–	–	–	
CALL c,pq	table 3	–	–	–	–	–	–	–	–	
CCF	3F	–	–	–	x	–	–	0	@	
	(the H flag becomes the previous value of the C flag)									
CP r	table 1	@	@	–	@	–	@	1	@	
CPI	EDA1	@	x	–	@	–	x	1	–	
CPD	EDA9	@	x	–	@	–	x	1	–	

<u>INSTRUCTIONS</u>		<u>FLAGS</u>							
CPIR	EDB1	@	x	-	@	-	x	1	-
CPDR	EDB9	@	x	-	@	-	x	1	-
(P becomes 0 if BC becomes zero, S becomes 1 if A = (HL±1))									
CPL	2F	-	-	-	1	-	-	1	-
DAA	27	@	@	-	@	-	@	-	@
DEC r	table 1	@	@	-	@	-	@	1	-
DEC s	table 2	-	-	-	-	-	-	-	-
DI	F3	-	-	-	-	-	-	-	-
DJNZ e	10ee	-	-	-	-	-	-	-	-
EI	FB	-	-	-	-	-	-	-	-
EX AF,AF'	08	-	-	-	-	-	-	-	-
EX DE,HL	EB	-	-	-	-	-	-	-	-
EX (SP),HL	E3	-	-	-	-	-	-	-	-
EX (SP),IX	DDE3	-	-	-	-	-	-	-	-
EX (SP),IY	FDE3	-	-	-	-	-	-	-	-
EXX	D9	-	-	-	-	-	-	-	-
HALT	76	-	-	-	-	-	-	-	-
IM 0	ED46	-	-	-	-	-	-	-	-
IM 1	ED56	-	-	-	-	-	-	-	-
IM 2	ED5E	-	-	-	-	-	-	-	-
INC r	table 1	@	@	-	@	-	@	0	-
INC s	table 2	-	-	-	-	-	-	-	-
IN A,(n)	DBnn	-	-	-	-	-	-	-	-
IN r,(C)	table 1	@	@	-	@	-	@	0	-
INI	EDA2	?	X	-	?	-	?	1	-
IND	EDAA	?	X	-	?	-	?	1	-
(S becomes 1 if B becomes zero)									
INIR	EDB2	?	1	-	?	-	?	1	-
INDR	EDBA	?	1	-	?	-	?	1	-

<u>INSTRUCTIONS</u>		<u>FLAGS</u>							
JP pq	C3qqpp	-	-	-	-	-	-	-	-
JP c,pq	table 3	-	-	-	-	-	-	-	-
JP (HL)	E9	-	-	-	-	-	-	-	-
JP (IX)	DDE9	-	-	-	-	-	-	-	-
JP (IY)	FDE9	-	-	-	-	-	-	-	-
JR e	18ee	-	-	-	-	-	-	-	-
JR c,e	table 3	-	-	-	-	-	-	-	-
LD (BC),A	02	-	-	-	-	-	-	-	-
LD A,(BC)	0A	-	-	-	-	-	-	-	-
LD (DE),A	12	-	-	-	-	-	-	-	-
LD A,(DE)	1A	-	-	-	-	-	-	-	-
LD I,A	ED47	-	-	-	-	-	-	-	-
LD R,A	ED4F	-	-	-	-	-	-	-	-
LD A,I	ED57	@	@	-	0	-	x	0	-
LD A,R	ED5F	@	@	-	0	-	x	0	-
	(P/V is set to interrupt storage flag)								
LD SP,HL	F9	-	-	-	-	-	-	-	-
LD SP,IX	DDF9	-	-	-	-	-	-	-	-
LD SP,IY	FDF9	-	-	-	-	-	-	-	-
LD r,r	table 1	-	-	-	-	-	-	-	-
LD s,mn	table 2	-	-	-	-	-	-	-	-
LD A,(pq)	3Aqqpp	-	-	-	-	-	-	-	-
LD s,(pq)	table 2	-	-	-	-	-	-	-	-
LD (pq),A	32qqpp	-	-	-	-	-	-	-	-
LD (pq),s	table 2	-	-	-	-	-	-	-	-
LDI	EDA0	-	-	-	0	-	x	0	-
LDD	EDA8	-	-	-	0	-	x	0	-
	(P/V becomes 0 if BC becomes 0)								
LDIR	EDB0	-	-	-	0	-	0	0	-
LDDR	EDB8	-	-	-	0	-	0	0	-

<u>INSTRUCTIONS</u>		<u>FLAGS</u>							
NEG	ED44	@	@	-	@	-	@	1	@
NOP	00	-	-	-	-	-	-	-	-
OR r	table 1	@	@	-	0	-	@	0	0
OUT (n),A	D3nn	-	-	-	-	-	-	-	-
OUT (C),r	table 1	-	-	-	-	-	-	-	-
OUTI	EDA3	?	x	-	?	-	?	1	-
OUTD	EDAB	?	x	-	?	-	?	1	-
	(Z becomes 1 if B becomes zero)								
OTIR	EDB3	?	1	-	?	-	?	1	-
OTDR	EDBB	?	1	-	?	-	?	1	-
POP AF	F1	x	x	x	x	x	x	X	x
	(Flags are determined by the byte at the top of the stack)								
POP s	table 2	-	-	-	-	-	-	-	-
PUSH AF	F5	-	-	-	-	-	-	-	-
PUSH s	table 2	-	-	-	-	-	-	-	-
RES b,r	table 1	-	-	-	-	-	-	-	-
RET	C9	-	-	-	-	-	-	-	-
RET c	table 3	-	-	-	-	-	-	-	-
RETN	ED45	-	-	-	-	-	-	-	-
RETI	ED4D	-	-	-	-	-	-	-	-
RES b,r	table 1	-	-	-	-	-	-	-	-
RET	C9	-	-	-	-	-	-	-	-
RET c	table 3	-	-	-	-	-	-	-	-
RETN	ED45	-	-	-	-	-	-	-	-
RETI	ED4D	-	-	-	-	-	-	-	-
RLCA	07	-	-	-	0	-	-	0	@
RRCA	0F	-	-	-	0	-	-	0	@
RLA	17	-	-	-	0	-	-	0	@
RRA	1F	-	-	-	0	-	-	0	@

Mastering Machine Code on your ZX Spectrum

<u>INSTRUCTIONS</u>		<u>FLAGS</u>								
RLC r	table 1	@	@	-	0	-	@	0	@	
RRC r	table 1	@	@	-	0	-	@	0	@	
RL r	table 1	@	@	-	0	-	@	0	@	
RR r	table 1	@	@	-	0	-	@	0	@	
RRD	ED67	@	@	-	0	-	@	0	-	
RLD	ED6F	@	@	-	0	-	@	0	-	
RST 00	C7	-	-	-	-	-	-	-	-	
RST 08	CF	-	-	-	-	-	-	-	-	
RST 10	D7	-	-	-	-	-	-	-	-	
RST 18	DF	-	-	-	-	-	-	-	-	
RST 20	E7	-	-	-	-	-	-	-	-	
RST 28	EF	-	-	-	-	-	-	-	-	
RST 30	F7	-	-	-	-	-	-	-	-	
RST 38	FF	-	-	-	-	-	-	-	-	
SBC A,r	table 1	@	@	-	@	-	@	1	@	
SBC HL,s	table 2	@	@	-	@	-	@	1	@	
SCF	37	-	-	-	0	-	-	0	1	
SET b,r	table 1	-	-	-	-	-	-	-	-	
SLA r	table 1	@	@	-	0	-	@	0	@	
SRA r	table 1	@	@	-	0	-	@	0	@	
SRL r	table 1	@	@	-	0	-	@	0	@	
SUB r	table 1	@	@	-	@	-	@	1	@	
XDR r	table 1	@	@	-	0	-	@	0	0	

TABLE ONE

r	B	C	D	E	H	L	(HL)	A	(IX + d)	(IY + d)	n
ADD A,r	80	81	82	83	84	85	86	87	DD86dd	FD86dd	C6nn
ADC A,r	88	89	8A	8B	8C	8D	8E	8F	DD8Edd	FD8Edd	CEnn
AND r	A0	A1	A2	A3	A4	A5	A6	A7	DDA6dd	FDA6dd	E6nn
BIT 0,r	CB40	CB41	CB42	CB43	CB44	CB45	CB46	CB47	DDCBdd46	FDCBdd46	-
BIT 1,r	CB48	CB49	CB4A	CB4B	CB4C	CB4D	CB4E	CB4F	DDCBdd4E	FDCBdd4E	-
BIT 2,r	CB50	CB51	CB52	CB53	CB54	CB55	CB56	CB57	DDCBdd56	FDCBdd56	-
BIT 3,r	CB58	CB59	CB5A	CB5B	CB5C	CB5D	CB5E	CB5F	DDCBdd5E	FDCBdd5E	-
BIT 4,r	CB60	CB61	CB62	CB63	CB64	CB65	CB66	CB67	DDCBdd66	FDCBdd66	-
BIT 5,r	CB68	CB69	CB6A	CB6B	CB6C	CB6D	CB6E	CB6F	DDCBdd6E	FDCBdd6E	-
BIT 6,r	CB70	CB71	CB72	CB73	CB74	CB75	CB76	CB77	DDCBdd76	FDCBdd76	-
BIT 7,r	CB78	CB79	CB7A	CB7B	CB7C	CB7D	CB7E	CB7F	DDCBdd7E	FDCBdd7E	-
CP r	88	89	8A	8B	8C	8D	8E	8F	DDBEdd	FDBEdd	FEnn
DEC r	05	0D	15	1D	25	2D	35	3D	DD35dd	FD35dd	-
IN r,(C)	ED40	ED48	ED50	ED58	ED60	ED68	-	ED78	-	-	-
INC r	04	0C	14	1C	24	2C	34	3C	DD34dd	FD34dd	-
LD B,r	40	41	42	43	44	45	46	47	DD46dd	FD46dd	06nn
LD C,r	48	49	4A	4B	4C	4D	4E	4F	DD4Edd	FD4Edd	0Enn
LD D,r	50	51	52	53	54	55	56	57	DD56dd	FD56dd	16nn
LD E,r	58	59	5A	5B	5C	5D	5E	5F	DD5Edd	FD5Edd	1Enn
LD H,r	60	61	62	63	64	65	66	67	DD66dd	FD66dd	26nn
LD L,r	68	69	6A	6B	6C	6D	6E	6F	DD6Edd	FD6Edd	2Enn
LD (HL),r	70	71	72	73	74	75	-	77	-	-	36nn
LD A,r	78	79	7A	7B	7C	7D	7E	7F	DD7Edd	FD7Edd	3Enn
LD (IX + d),r	DD70	DD71	DD72	DD73	DD74	DD75	-	DD77	-	-	DD36
LD (IY + d),r	FD70	FD71	FD72	FD73	FD74	FD75	-	FD77	-	-	FD36
LD (HL),r	dd	dd	dd	dd	Dd	dd	-	dd	-	-	ddnn
OR r	B0	B1	B2	B3	B4	B5	B6	B7	DDB6dd	FDB6dd	F6nn
OUT (C),r	ED41	ED49	ED51	ED59	ED61	ED69	-	ED79	-	-	-
RES 0,r	CB80	CB81	CB82	CB83	CB84	CB85	CB86	CB87	DDCBdd86	FDCB86	-
RES 1,r	CB88	CB89	CB8A	CB8B	CB8C	CB8D	CB8E	CB8F	DDCB8E	FDCBdd8E	-
RES 2,r	CB90	CB91	CB92	CB93	CB94	CB95	CB96	CB97	DDCBdd96	FDCBdd96	-

## Mastering Machine Code on your ZX Spectrum

RES 3,r	CB98	CB99	CB9A	CB9B	CB9C	CB9D	CB9E	CB9F	DDCBdd9E	FDCBdd9E	-
RES 4,r	CBA0	CBA1	CBA2	CBA3	CBA4	CBA5	CBA6	CBA7	DDCBddA6	FDCBddA6	-
RES 5,r	CBA8	CBA9	CBAA	CBAB	CBAC	CBAD	CBAE	CBAF	DDCBddAE	FDCBddAE	-
RES 6,r	CBB0	CBB1	CBB2	CBB3	CBB4	CBB5	CBB6	CBB7	DDCBddB6	FDCBddB6	-
RES 7,r	CBB8	CBB9	CBBA	CBBB	CBBC	CBBD	CBBE	CBBF	DDCBddBE	FDCBddBE	-
RLC r	CB00	CB01	CB02	CB03	CB04	CB05	CB06	CB07	DDCBdd06	FDCBdd06	-
RRC r	CB08	CB09	CB0A	CB0B	CB0C	CB0D	CB0E	CB0F	DDCBdd0E	FDCBdd0E	-
RL r	CB10	CB11	CB12	CB13	CB14	CB15	CB16	CB17	DDCBdd16	FDCBdd16	-
RR r	CB18	CB19	CB1A	CB1B	CB1C	CB1D	CB1E	CB1F	DDCBdd1E	FDCBdd1E	-
SET 0,r	CBC0	CBC1	CBC2	CBC3	CBC4	CBC5	CBC6	CBC7	DDCBddC6	FDCBddC6	-
SET 1,r	CBC8	CBC9	CBCA	CBCB	CBCC	CBCD	CBCE	CBCF	DDCBddCE	FDCBddCE	-
SET 2,r	CBD0	CBD1	CBD2	CBD3	CBD4	CBD5	CBD6	CBD7	DDCBddD6	FDCBddD6	-
SET 3,r	CBD8	CBD9	CBDA	CBDB	CBDC	CBDD	CBDE	CBDF	DDCBddDE	FDCBddDE	-
SET 4,r	CBE0	CBE1	CBE2	CBE3	CBE4	CBE5	CBE6	CBE7	DDCBddE6	FDCBddE6	-
SET 5,r	CBE8	CBE9	CBEA	CBEB	CBEC	CBED	CBEE	CBEF	DDCBddEE	FDCBddEE	-
SET 6,r	CBF0	CBF1	CBF2	CBF3	CBF4	CBF5	CBF6	CBF7	DDCBddF6	FDCBddF6	-
SET 7,r	CBF8	CBF9	CBFA	CBFB	CBFC	CBFD	CBFE	CBFF	DDCBddFE	FDCBddFE	-
SUB A,r	90	91	92	93	94	95	96	97	DD96dd	FD96dd	D6nn
SBC A,r	98	99	9A	9B	9C	9D	9E	9F	DD9Edd	FD9Edd	DEnn
SLA r	CB20	CB21	CB22	CB23	CB24	CB25	CB26	CB27	DDCBdd26	FDCBdd26	-
SRA r	CB28	CB29	CB2A	CB2B	CB2C	CB2D	CB2E	CB2F	DDCBdd2E	FDCBdd2E	-
SRL r	CB38	CB39	CB3A	CB3B	CB3C	CB3D	CB3E	CB3F	DDCBdd3E	FDCBdd3E	-
XOR r	A8	A9	AA	AB	AC	AD	AE	AF	DDAEdd	FDAEdd	EEnn

TABLE TWO

s	BC	DE	HL	SP	IX	IY
ADC HL,s	ED4A	ED5A	ED6A	ED7A	–	–
ADD HL,s	09	19	29	39	–	–
ADD IX,s	DD09	DD19	–	DD39	DD29	–
ADD IY,s	FD09	FD19	–	FD39	–	FD29
DEC s	0B	1B	2B	3B	DD2B	FD2B
INC s	03	13	23	33	DD23	FD23
LD s,mn	01nnmm	11nnmm	21nnmm	31nnmm	DD21nnmm	FD21nnmm
LD s,(pq)	ED4Bqppp	ED5Bqppp	2Aqppp	ED7Bqppp	DD2Aqppp	DD2Aqppp
LD (pq),s	ED43qppp	ED53qppp	22qppp	ED73qppp	DD22qppp	FD22qppp
POP s	C1	D1	E1	–	DDE1	FDE1
PUSH s	C5	D5	E5	–	DDE5	FDE5
SBC HL,s	ED42	ED52	ED62	ED72	–	–

TABLE THREE

c	NZ	Z	NC	C	PO	PE	P	M
CALL c,pq	C4qppp	CCqppp	D4qppp	DCqppp	E4qppp	ECqppp	F4qppp	FCqppp
JP c,pq	C2qppp	CAqppp	D2qppp	DAqppp	E2qppp	EAqppp	F2qppp	FAqppp
JRc,e	20ee	28ee	30ee	38ee	–	–	–	–
RET c	C0	C8	D0	D8	E0	E8	F0	F8



## APPENDIX FIVE

Every *control character* – that is, every character whose code is between 00 and 1F, has two different meanings – the INPUT control (as read by INKEY\$) and the OUTPUT control (as given by PRINT). Sometimes these two coincide (eg 0D) but this is not usually the case. This table then lists both the INPUT and the OUTPUT uses of each control character.

Abbreviations: bs means “both shifts together, followed by...”  
 cs means “caps shift simultaneous with...”  
 ss means “symbol shift simultaneous with...”

CODE	INPUT CONTROL		OUTPUT CONTROL
	<u>HOW OBTAINED</u>	<u>MEANING</u>	<u>PRINTS AS</u>
00	bs cs 8	flash off	?
01	bs cs 9	flash on	?
02	bs 8	bright off	?
03	bs 9	bright on	?
04	cs 3	true video	?
05	cs 4	inverse video	?
06	cs 2	caps lock	comma control††
07	cs 1	edit	?
08	cs 5	cursor left	backspace
09	cs 8	cursor right	current colours
0A	cs 6	cursor down	?
0B	cs 7	cursor up	?
0C	cs 0	delete	?
0D	enter	enter	newline

CODE	INPUT CONTROL		OUTPUT CONTROL
	<u>HOW OBTAINED</u>	<u>MEANING</u>	<u>PRINTS AS</u>
0E†	bs	E-control	?
0F	cs 9	graphics	?
10	bs 0	paper black	ink control*
11	bs 1	paper blue	paper control*
12	bs 2	paper red	flash control*
13	bs 3	paper magenta	bright control*
14	bs 4	paper green	inverse control*
15	bs 5	paper cyan	over control*
16	bs 6	paper yellow	at control**
17	bs 7	paper white	tab control**
18	bs cs 0	ink black	?
19	bs cs 1	ink blue	?
1A	bs cs 2	ink red	?
1B	bs cs 3	ink magenta	?
1C	bs cs 4	ink green	?
1D	bs cs 5	ink cyan	?
1E	bs cs 6	ink yellow	?
1F	bs cs 7	ink white	?

† The byte 0E, where it occurs in a BASIC program listing, is taken to mean “data in five byte form now follows”. The byte 0E and the first five bytes immediately following it will be rendered invisible in a listing.

†† The byte 06 (comma control) may be entered direct from the keyboard by the means bs 6 cs. 0 (paper yellow followed by delete).

\* Requires one byte of data to follow.

















\*\* Requires two bytes of data to follow.

*Mastering Machine Code on your ZX Spectrum*

The characters whose codes lie between 20 and 7F are the standard ASCII character set.\* These are listed in tabular form below. Note that the symbol corresponding to code 20 is "space".

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
2		!	"	#	\$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	↑	_
6	£	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{		}	~	©

The remaining codes are all either graphics codes or tokens which print as whole words. These are as follows:

80	graphic 8		CD	STEP
81	graphic 1		CE	DEF FN
82	graphic 2		CF	CAT
83	graphic 3		D0	FORMAT
84	graphic 4		D1	MOVE
85	graphic 5		D2	ERASE
86	graphic 6		D3	OPEN
87	graphic 7		D4	CLOSE
88	graphic shift 7		D5	MERGE
89	graphic shift 6		D6	VERIFY
8A	graphic shift 5		D7	BEEP
8B	graphic shift 4		D8	CIRCLE
8C	graphic shift 3		D9	INK
8D	graphic shift 2		DA	PAPER
8E	graphic shift 1		DB	FLASH
8F	graphic shift 8		DC	BRIGHT
			DD	INVERSE

90	graphic A	DE	OVER
91	graphic B	DF	OUT
92	graphic C	E0	LPRINT
93	graphic D	E1	LLIST
94	graphic E	E2	STOP
95	graphic F	E3	READ
96	graphic G	E4	DATA
97	graphic H	E5	RESTORE
98	graphic I	E6	NEW
99	graphic J	E7	BORDER
9A	graphic K	E8	CONTINUE
9B	graphic L	E9	DIM
9C	graphic M	EA	REM
9D	graphic N	EB	FOR
9E	graphic O	EC	GO TO
9F	graphic P	ED	GO SUB
A0	graphic Q	EE	INPUT
A1	graphic R	EF	LOAD
A2	graphic S	F0	LIST
A3	graphic T	F1	LET
A4	graphic U	F2	PAUSE
A5	RND	F3	NEXT
A6	INKEY\$	F4	POKE
A7	PI	F5	PRINT
A8	FN	F6	PLOT
A9	POINT	F7	RUN
AA	SCREEN\$	F8	SAVE
AB	ATTR	F9	RANDOMIZE
AC	AT	FA	IF
AD	TAB	FB	CLS
AE	VAL\$	FC	DRAW
AF	CODE	FD	CLEAR
B0	VAL	FE	RETURN
B1	LEN	FF	COPY
B2	SIN		

*Mastering Machine Code on your ZX Spectrum*

B3	COS	key A	
B4	TAN	key B	
B5	ASN	key C	A
B6	ACS	key D	B
B7	ATN	key E	C
B8	LN	key F	D
B9	EXP	key G	E
BA	INT	key H	F
BB	SQR	key I	G
BC	SGN	key J	H
BD	ABS	key K	I
BE	PEEK	key L	J
BF	IN	key M	K
C0	USR	key N	L
C1	STR\$	key O	M
C2	CHR\$	key P	N
C3	NOT	key Q	O
C4	BIN	key R	P
C5	OR	key S	Q
C6	AND	key T	R
C7	<=	key U	S
C8	>=	key V	T
C9	<>	key W	U
CA	LINE	key X	V
CB	THEN	key Y	W
CC	TO	key Z	X
			Y
			Z

\* Note that codes 60 and 7F are non-standard.

## APPENDIX SIX

The following table gives the data codes to be used after RST 28 which are used to manipulate the calculator stack. Only those codes which have been mentioned in this book are listed as some of the others would require rather extensive explanations.

01	exchange	Swap the two topmost items on the stack.
02	delete	Delete the topmost item on the stack.
03	subtract	Delete the two topmost items and push the result of their subtraction.
04	multiply	Delete the two topmost items and push their product.
05	divide	Delete the two topmost items and push the result of their division.
06	power	Delete the two topmost items and push the result of raising one to the power of the other.
0F	add	Delete the two topmost items and then stack their sum.
17	s_add	As 0F but adds strings, not numbers.
18	val\$	Replace the topmost item by VAL\$ of that item.
19	usr_s	Replace the topmost item (a string) by USR of that string.
1B	negate	Replace the topmost item (a number) by minus that number.
1C	code	) Replace the topmost item (a string) by the
1D	val	) – result of applying the appropriate function to that
1E	len	) string.
1F	sin	) Replace the topmost item (a number) by
20	cos	) – the result of applying the appropriate function to
21	tan	) that number.
22	asn	)
23	acs	)
24	atn	)

*Mastering Machine Code on your ZX Spectrum*

25	ln	)	
26	exp	)	
27	int	)	
28	sqr	)	
29	sgn	)	
2A	abs	)	
2B	peek	)	
2C	in	)	
2D	usr n	)	
2E	str\$	)	
2F	chr\$	)	
31	duplicate		Stack an extra copy of the topmost item on the stack.
38	end_calc		No more data follows – return to normal machine code.
A0	const_0		Stack the number zero.
A1	const_1		Stack the number one.
A2	const_2		Stack the number zero point five.
A3	const_π/2		Stack the number half-PI (1.5707963).
A4	const_10		Stack the number ten.
On	store_n		Let memory n equal the topmost item on the stack.
En	recall_n		Stack the contents of memory n.

This book is designed to teach the reader the essential elements of programming in machine code. It assumes absolutely no knowledge of the subject whatsoever, and yet still promises to take you to a level of proficiency beyond your wildest dreams.

Machine Code is a computer language – just as BASIC is a computer language. If you can understand BASIC instructions like `LET A = 5` then you will surely understand machine code instructions like `LD A,5`. There's no mystery about it, just a myth of misunderstanding. This book will explode that myth, and enable you to bring your programming skills to their full potential.

Starting with simple addition and subtraction you are slowly guided through printing and inputting in machine code. You are shown how to use the screen to its utmost. The book explores and utilises the incredible speed of machine code, giving you real time graphics games like `BREAKOUT`, and leads you up to a full working `DRAUGHTS` program. At each stage there are exercises which test your understanding of the subject.

This book will teach you how to make music in real time – something which is **IMPOSSIBLE** in BASIC; to plot in high resolution graphics faster than you would have believed possible. This is not a book full of useless information to file away – **YOU** will acquire these skills.

The last two chapters are rather ambitious. They concern the ROM, and will lead to a better understanding about the way that BASIC variables are treated. Go for it – if you will. You'll find the rewards are well worth the effort. The Spectrum can do more than just shoot down space invaders. Master it, and always remember the most important thing in life is tea.

**Another great book from  
INTERFACE PUBLICATIONS**





# Mastering Machine Code

## Errata Version 1.1

This is errata sheet 1.1, which means the first errata sheet produced applicable to the first edition of the book. Any errata sheets which come after this will be called Version 1.2, Version 1.3, etc. It is hoped that edition two of the book will appear with none of the errors listed herebelow, but that's something beyond my control. The numbers down the left hand edge of the page refer to the number of the page on which the amendment is to be made.

- 13 ~~13~~ In line 64 of BASIC program: "=" should read "+".
- 44) At end of third paragraph: "286672" should read "28672".
- 45) 2nd line of 1st paragraph: "LD B,(5CB2)" should read "LD BC,(5CBF)".
- 59) "IF A>B HEX GO 0..." should read "IF A<B THEN GO 0...".
- 67) 8th line of 1st paragraph: "AND A" should read "AND B".
- 67) 10th line of 1st paragraph: "1 AND 1 is 0," should read "1 AND 1 is 1, 0 AND 1 is 0, and".
- 68) The binary number 0101112 is missing from the 4th column. It should appear between "OR" and "equals".
- 74) Last line of 1st paragraph: "semicolons" should read "colons".
- 91) Insert an extra BASIC line: 169 INPUT "" (see below).
- 98) In definition of CPU and CPDR: The PV flag is reset if BC becomes zero, and is set otherwise. The zero flag is set if A was equal to (HL), and is reset otherwise.
- 106) The definition of SCP is missing: It means "set the carry flag, leaving everything else unchanged".
- 125) 5th line of program: "78 LD A,C" should read "79 LD A,C".
- 125) Penultimate line of program: "CALL C\_PRINT" should read "CALL C,C\_PRINT".
- 129) In penultimate line on page: "21" should read "0".
- 131) 14th line of program: "3897 JR NC,HL\_NEXT" should read "5897 JR C,HL\_NEXT".
- 139) Line 522 of BASIC program: "A%" should read "A\$(I)".
- 142) 6th machine code instruction: "1D" should read "10".
- 149) Line 936 of BASIC program: "32631" should read "65399".
- 161) 4th line of table: "0" should read "0".
- 162) 7th line of 2nd paragraph: "KEY SCAN" should read "KEY TEST".
- 163) 3rd line: "P mode" should read "K mode".
- 167) 18th line of program: "54484120" should read "5448415420".
- 197) Penultimate line: "caps shift" should read "caps shift space".
- 198) 3rd line: "caps shift" should read "caps shift apsc".
- 221) 1st line of text: "6B69" should read "6BA6".
- 221) 2nd line of text: "6B69" should read "6BA6".
- 221) Last line of 2nd paragraph: "6B98" should read "6BA7".
- 226) 8th line of program: "A+" should read "+".
- 231) 19th line of program: "AFFS+56" should read "AFFS+61".
- 237) 16th line of program: "LD D,A" should read "LD E,A".
- 248) Opcode "DEC A" is missing from 3rd line of program.
- 273) 8th line: "6912" should read "6A22".
- 298) There is a line of program missing in the RANDOM NUMBERS section. Between "C1282D CALL STACK\_A" and "EF RSP 28" insert the line "051D LD B,10".
- 299) As 298. See below.

The following errors occur in the Appendices:

Appendix Two: System variables 5C~~00~~ to 5C39 can be accessed using (IY+d) but note that the displacement 'd' here is negative (as in Appendix One Table Two).

Appendix Four: Main table: P flag for instruction BIT b,r: "P" should read "Z".  
Z and P flags for CPL/CPDR/CPI/CPDR: See error 98.  
Last column flags heading: "S Z - H P N C" should read  
"S Z - H - P N C".

Appendix Four: Table One: The following entries in the table are wrong. The correct codes for them are listed here:

ADC A,(IX+d)	DD8Edd
ADC A,(IY+d)	FD8Edd
CP (IY+d)	FD8Rdd
LD (IY+d),B	FD7Fdd
R/S 5,H	CBAC
RLC (IY+d)	FD8Bdd <del>6</del>

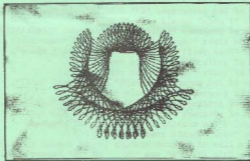
Finally, some notes on errors 91, 298 and 299.

The up and down scrolling (machine code) program is not dedicated to the BASIC part, therefore it uses all twenty four lines of the screen. The BASIC part, though, cannot print on the bottom two lines, and so it is written to take into account the top twenty two lines only. However, the failure to remember this fact when using the machine code "down" routine results in the twenty second line being scrolled down to the twenty third - this is an error since we should really consider it off the screen. To cure the bug an additional line of BASIC is necessary: Line 169 which should read INPUT "". This has the effect of clearing the lower two lines of the screen after the machine code has been called - hence removing the problem.

The machine code RSP 28 routines for VAL (code 1D) and VAL~~2~~ (code 1B) are exactly identical. That is to say "1B" and "1D" will both call the same subroutine address in the ROM. In order to distinguish between the two it needs some way of telling the difference, and the way it does this is to check the B register, which is assumed to contain the appropriate code (1B or 1D). In practice if B contains any number in the range ~~00~~ to 1C then VAL~~2~~ will be evaluated, and if B contains a number in the range 1D to FF then VAL will be executed. Most of the RSP 28 routines leave the B register unaltered - in particular 2F (chr~~2~~) does not change it. In the random number routines given the instruction LD B,1D should be inserted immediately before RSP 28 in order to ensure that VAL is carried out and not VAL~~2~~ (the CHR~~2~~ instruction in between will not affect things) - if this is not done then report code C: Nonsense in BASIC may be generated since VAL~~2~~("RND") does not make sense.

Love & Peace

Tom Baker



INFORMATION ON  
MACHINE CODE MAY  
BE OBTAINED FROM  
TOM BAKER,  
THE ZX MACHINE  
CODE USERS' CLUB,  
37 STRATFORD ROAD  
WOLVERTON  
MILTON KEYNES  
MK12 5LW.