Spectrum Machine Code Made Easy
Volume Two – For Advanced Programmers

At last, the book you've been waiting for, to take you through machine code on the Spectrum from first principles. Contents include jumping; jumping relatively; calling and returning; restarts; jumping with decisions; flags; loops; double-height characters; ANDing; ORing; XORing; looping with double byte registers; rotating and shifting bytes; interrupts; the ROM interrupt routine; and much more. Appendices include Z80 instructions and mnemonics.

If you want to increase your knowledge of machine code then this is the book for you.

Another great book from
INTERFACE PUBLICATIONS

£5-95

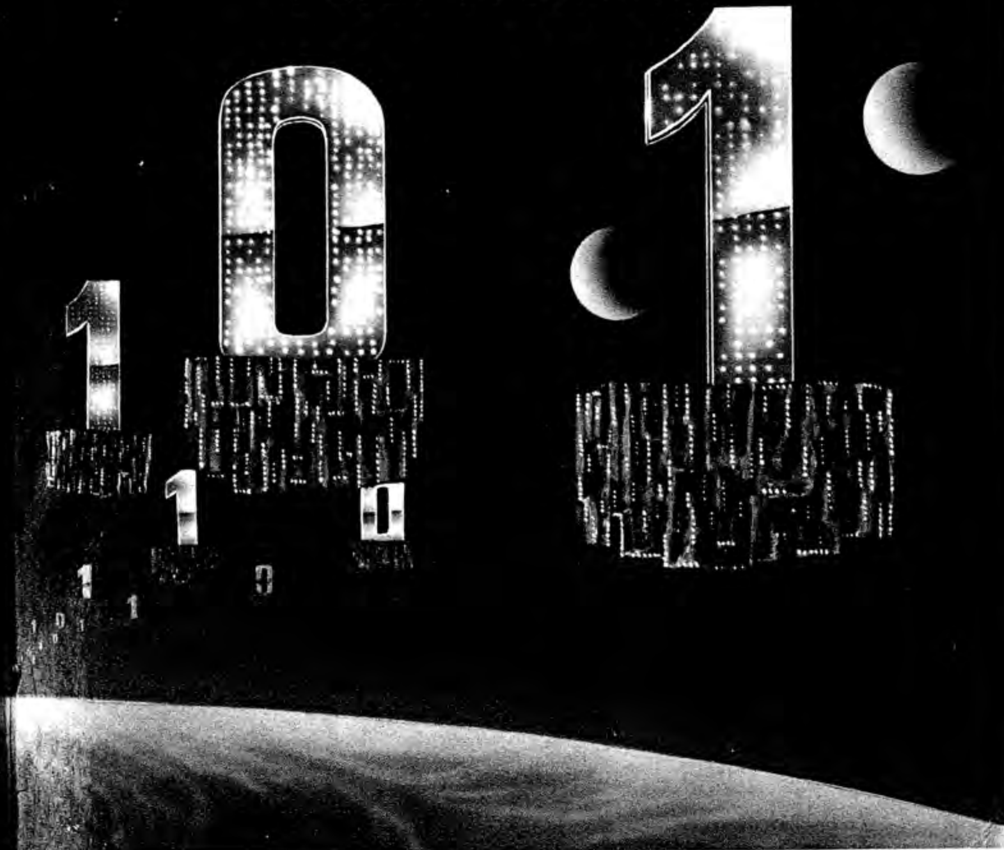
SPECTRUM MACHINE CODE MADE EASY VOL2 – Paul Holmes INTERFACE

SPECTRUM MACHINE CODE MADE EASY

Volume Two

For Advanced Programmers

Paul Holmes



SPECTRUM MACHINE CODE MADE EASY

Volume Two

For Advanced Programmers

Paul Holmes

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

© Copyright 1983, Paul Holmes

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 44 9

Cover Illustrator: Stuart Hughes

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

Features

Easy to understand step by step approach.
Lots of examples to try using the 'hands on' approach.
Dictionary of Z8Ø machine code.
A further look at system variables.
Printing double height characters.
How to make sound effects.
ROM routines including BEEP, character printing, etc.
Takes the reader from a basic knowledge to a full understanding of the Z8Ø instruction set and the ZX Spectrum.

Contents

1. **Going places**: Jumping, Jumping relatively. Negatives. Hex loader program. Calling and Returning. Restarts. Printing. 13
2. **To go or not to go?**: Jumping with decisions. Flags. Calling with decisions. Returning with decisions, loops. Comparing. Double height characters. 35
3. **A chapter of bits**: Setting and re-setting bits, testing bits. Mini-word processor/Intelligent typewriter program. 53
4. **Logical Approach**: ANDing, ORing, XORing, looping with double byte registers. 71
5. **Rotating**: Rotating and shifting bytes. Decimals. 83
6. **Ports**: Making sound, changing the border. BEEP routine. 89
7. **May I interrupt?**: Interrupts, the ROM interrupt routine, interrupt routines. 99

Appendices

A: DECIMAL, HEX, ASCII CHARACTER SET, Z8Ø MNEMONICS. 109

B: Z8Ø INSTRUCTIONS FLAG ADJUSTMENTS. 115

C: SYSTEM VARIABLES EXPLANATIONS. 121

D: DEFINITIONS OF ALL Z8Ø INSTRUCTIONS. 131

About the author

Paul Holmes, at the time of writing, was a seventeen year old student who first started computing 3 years ago with a ZX80. Since then he wrote two ZX81 software packages, toolkit and graphics toolkit, that were marketed by JRS Software in the UK and Softsync. in the US. He has for a period been a software reviewer for ZX Computing and also wrote software for Jupiter Cantab before their recent downfall. More recently he has become the owner-manager of Timescape Software and has published several of his own games, worked for Atari for a small period, and has gained much praise from the technical press about his games which reflect his excellent programming and design experience.

Acknowledgements and Dedications

Acknowledgements: Special thanks to:

Tim Hartnell who first suggested the idea for the book.

Richard Lawrence who first aroused my interest in computing.

Liz North for the egg and toast for lunch.

Mr. Coulson, my physics teacher for his continuous support throughout the writing of this book.

Dedicated to: My friends in Sutton Coldfield.

INTRODUCTION

So you thought you would like to master machine code? So what's the story so far? Oh I see! You know the basics but want to learn the heavy stuff? Well, you have just picked up the right book because in this book I assume a very basic knowledge of machine code, but even if you haven't a clue read on — I am not finished yet! Is machine code really heavy stuff? Or is it as easy as riding a bike? Well, in this book it is my job to make it as easy as riding a bike, but until the day of the flat screen, I don't think I can teach you to ride a bike at the same time. Now a quick note for those who are absolute beginners. As I have said, this is a book for those who know the basics but, even if you don't, try reading the first part of Chapter One and if this inspires the mind without tying it in knots then buy this book. If, on the other hand, your brain gives you 'Nonsense in English' messages then you're going to need some extra tuition which can be found in the form of 'First Steps In Machine Code' by James Walsh. His book starts from the very beginning and moves at a very light pace to give the reader a basic knowledge of machine code on your ZX Spectrum. Now, if you already own this copy of the book then take a deep breath and dive into Chapter One. On the other hand, if you're just browsing through in a shop then either buy this book or put it back — after all, what do you think this is, a library?

Chapter One
GOING PLACES

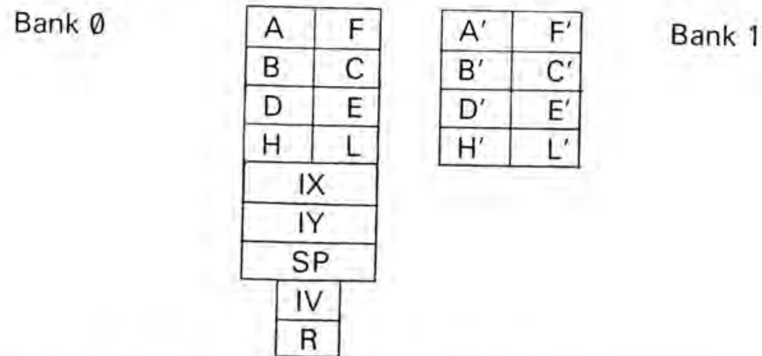
Before we get stuck into the business of writing machine code programs, let us review what we shall aim to do in the following chapters and how to use the book.

The book, as I mentioned in the introduction does not cater for the absolute beginner but for someone who knows the bare essentials of machine code. Just in case you need reminding about the bare essentials, have a brief summary of what you should know before starting.

Machine code is a programming language that the CPU (Central Processing Unit) understands. It is represented by a series of numbers in the memory that the CPU understands and will execute. An example of this is the ROM, a giant program in a Read Only Memory that organizes the Spectrum. This ROM understands our BASIC instructions, interprets them and executes machine code routines to do jobs like PRINT and GOTO.

Machine code is much closer to the actual machine, much faster and has nothing to do with the ROM; the Z80 chip in the Spectrum is the only part that deals with machine code. Z80 machine code, though stored in the memory, has appropriate names for each instruction, so we have an idea what it does. These names are called 'mnemonics'. The Z80 has a number of registers and these are the foundation of the language. We use these registers like we would use variables in BASIC. There are two sets of registers, bank one and bank zero and some special

registers, the equivalent of the system variables. This is how we think of the registers.



Each box contains one register. The shorter boxes can hold numbers between 0 and 255, and the long boxes can store numbers between 0 and 65535. If we join two adjacent short boxes we can make a long box. For instance, if we join B and C we get a 'double byte register' called BC. We can only use registers A, B, C, D, H and L. Their equivalents in bank one and IX and IY. The other registers are special purpose registers for use by the Z80 chip itself.

In addition we can only use one bank of registers at a time. The instruction:

EXX

switches to the other bank so we may use the other registers. To put a value in a variable in BASIC we say:

LET A=20

In machine code we use the word 'load' instead, which gets abbreviated to LD, so:

LD A, 14

is the equivalent. We say fourteen not as decimal, our everyday numbers, but as hexadecimal. This is base sixteen. The

number fourteen is the Hex (an abbreviation for hexadecimal) for decimal 20. This is because 1 × 16 + 4 makes 20. The right hand column is the units, the next the sixteens, the next (if we use a double byte number) is the 255s and the fourth is the 4096s. Because we, in base sixteen, need the numbers 0 to 15 we have to use letters when we get past nine. Here are the first 20 numbers.

Hex	Decimal
00	0
01	1
02	2
03	3
04	4
05	5
06	6
07	7
08	8
09	9
0A	10
0B	11
0C	12
0D	13
0E	14
0F	15
10	16
11	17
12	18
13	19
14	20

Two Hex digits can be stored in a memory byte and so this allows to use decimal numbers 0 to 255.

With registers we can perform lots of operations. We have instructions like LD to set a value and others such as ADD to add two registers. We store the program, as I have already mentioned, in the memory as a series of codes. For instance,

the instruction 'Load A with Data' has two Hex numbers:

LD A,d   3E d

The first number is 3E Hex to show that we are going to 'load' A with data' and the 'd' (that can be any one byte number) is the data to be put in A. In the case of loading a double byte register such as BC we need to name two data bytes, but unusually these are always the opposite way round. So this is how we would load BC with 156A Hex:

LD BC, 156A   Ø1 6A 15

The first byte is Ø1 to show that we are doing a load instruction and next is the 'least significant byte' of the data, that is the two digits on the right-hand side of 156A. The next number, 15 Hex is the 'most significant byte' of 156A. We always put the least significant byte first – this is convention and must always be followed.

There are many more instructions and these are all summarised in Appendix D. The instructions you should know are the load instructions, arithmetic such as ADD and SUB, RET and you'll also need to know about the stack though it is explained in this the first chapter.

If you find an instruction in a program that you don't understand and isn't explained in the chapter then you will find definitions of all the instructions in Appendix A.

Throughout the book I use a BASIC hexadecimal loader program with which the programs can be entered. If you have one of the machine code editors that are available then by all means use it. All the listings in the book show the address of the code and the program code itself. Note also we use hexadecimal in all the listings but in the text after a number there usually appears 'Hex' or 'decimal' to clarify what sort of number it is. If you have an assembler then you can enter the programs from the given mnemonics, but make sure your assembler accepts standard Zilog Z8Ø mnemonics and can assemble the machine code for the addresses shown.

In Chapter Two, we start by looking at the various methods of jumping in a program. Then we go on to see how the computer can make decisions and in this chapter is a program to allow double height characters to be used on the Spectrum. Chapter Three is a chapter of bits and looks at the bits in a byte, how to set, reset and test these bits and look at the attribute file as an example.

In Chapter Three we conclude with a program that allows the Spectrum to become an intelligent typewriter. Chapter Four deals with the logical instructions XOR, AND and OR again using the attribute file for demonstrating examples. The next chapter looks at the Rotate instructions and their uses; also in Chapter Five is a brief look at binary Coded Decimal and how to use it with Z8Ø instructions.

Chapter Six details 'ports' and discusses how a computer communicates with the outside world and how to use these communications. Much time is spent looking at how sound effects can be generated on a Spectrum. The final chapter 'May I interrupt?' looks at a topic which links with hardware – the interrupt system. It is explained with the aid of material from the ROM and describes the two types of interrupts and the three interrupt modes.

At the end of the book are four appendices which list: Hex, decimal, ASCII and Z8Ø mnemonics; the way Z8Ø instructions adjust flags; the system variables with extra explanations; and the definitions of what each and every Z8Ø instruction does.

So now we have dealt with all that, we can now start our first topic, jumping in machine code, a key part of any machine code program.

When programming in BASIC we often want to move to different parts of one program. We can do this quite easily using the GOTO statement. Rather conveniently it doesn't have to be followed by a number, it can have an 'expression'

after it. Because it can have an expression, such as:

GOTO 10*A+100

We have a very versatile system of *jumping*. The word jumping, is really associated with moving around in machine code rather than BASIC. But it is a quite appropriate word for any language that allows you to 'move' from one part of the program to another.

In machine code things are a little different. 'Expressions' don't exist, but we do have two different types of 'instruction' to tell the microprocessor in the Spectrum (a Z80-A, if you didn't know that then you ought to!) to carry on elsewhere in the program.

But as usual, machine code holds more complicated solutions to everything it seems. So first we'll have a look at our dear friend the Program Counter (PC) to make some later explanations clearer.

Inside one Z80 CPU (Central Processing Unit) there are lots of registers, more than most CPUs, that help us go about programming. There are a few which really belong to the CPU for its own reference. A bit like the system variables in BASIC. One of these is the Program Counter.

The Program Counter tells the CPU whereabouts it's up to in your program. Consider the following program.

```
8000    LD A,10     3E 10
8002    LD B,08     06 08
8004    ADD A,B     80
8005    RET         C9
```

On the left we have the *addresses* starting at 8000. That's where the code is stored. Looking down the actual program we can see that A is being loaded with 10 Hex. Make sure that is fixed in your mind that we're working in Hex otherwise you will get confused! Next, 08 Hex is loaded into register B. A and

B are then added together. The RET instruction sends us back to BASIC.

Now to the Program Counter. As one CPU 'collects' each instruction, its Program Counter keeps a check on where it is up to. Firstly, as we 'execute' this program, the Program Counter, PC, holds Hex value 8000 (where to find the first 'instruction'). The CPU recognises that it is to load the Accumulator with a 'direct value' and so moves PC up to the next address 8001, where it finds the value 10 Hex. That is loaded into the Accumulator, then PC is moved onto address 8002 to 'collect' the next instruction. It carries on like that until it is told to go elsewhere. In this case by one RET instruction, but we'll look at that in more detail later.

You are bound to be asking 'how do I send it elsewhere?', so now I shall move quickly to the point and tell you. We use the instruction JP followed by an address (the place to go). It's quite like GOTO really, except that you can't use expressions; you can use some registers but we'll cover that later. Look at this rather pointless program.

```
8000    LD A,10
8002    LD B,01
8004    ADD A,B
8005    JP 8004
```

What it actually does is set the Accumulator up with 10 Hex and register B up with 01 Hex. Then it adds B to A. Now the JP instruction causes it to go to address 8004 where it again adds A to B. This carries *ad infinitum*, or when you trip over the mains lead!

What actually happens is that when the CPU finds the JP instruction, it collects the next two following bytes and loads them into PC. Now PC has a different value the CPU starts 'executing' from a different place, in this case 8004. Notice how 8004 is stored in the program, the 04 first *then* the 80. Always keep to this 'backwards' way of storing numbers when

it consists of two bytes. In the case of JP, it is always followed by *two* numbers.

For instance, if you wish to jump to address 0001 you cannot miss off the first two zeros:

```
6000    JP 0001 C3 01 00
```

Have a look at the following program:

```
8000    JP 8006    C3 06 80
8003    JP 8009    C3 09 80
8006    JP 8003    C3 03 80
8009    JP 8003    C3 03 80
800C    RET
800D    JP 800C    C3 0C 80
```

Will it ever reach the RET to return to BASIC?

As you should have realised by now, the code for JP is C3 and it is followed by two bytes which tell the CPU where to jump to.

There are quite a few more forms of JP, most of them we shall look at in Chapter Two, but there are a few we'll have a look at now.

JP(HL)
JP(IX)
JP(IY)

In BASIC we can say:

GOTO A

And in machine code, with a few limitations we can do this as well. Using HL or the index registers. The IY register cannot be used otherwise it will mess up the Spectrum's 'works', though not permanently, it may cause the loss of your program.

JP(HL)

The above instruction causes a jump to be made to the address in HL. For instance, if HL contained 700C (Hex) a jump would be made to 700C.


```
6FF9 01 02 00      LD   BC,0002
6FFC 21 00 70      LD   HL,7000
6FFF E9            JP    (HL)
7000 09            ADD  HL,BC
7001 E9            JP    (HL)
7002 C9            RET
```

Again, a pointless program but see if you can follow its operation.

6FF9 — BC is loaded with 2.
6FFC — HL is loaded with 7000.
6FFF — A jump is made to the value in HL, currently 7000.

7000 — BC is added to HL, so now HL has (2 + 7000) 7002 held it in.
7001 — Another jump is made to HL, this time 7002 (pointless because it is the next instruction).
7002 — RET, the return to BASIC.

Now we shall try putting a few short programs into your Spectrum. The first one will be the one we have just covered so you can prove to yourself that it works.

To get a program into the Spectrum, we shall need a small BASIC program to 'POKE' it all above RAMTOP. Type in the following:

```
  10 LET address=28665
  15 READ a$
  20 LET byte=0
  30 FOR i=1 TO 0 STEP -1
  40 LET a=CODE a$-55: IF a$<":"
 THEN LET a=a+7
  50 LET byte=byte+a*16↑i: LET a
$=a$(2 TO ): NEXT i
  60 POKE address,byte: LET addr
ess=address+1: IF LEN a$=0 THEN
GO TO 15
  70 GO TO 20
  80 DATA "010200","210070","E9"
,"09","E9","C9"
```

The above program will translate hexadecimal into decimal, then POKE it to any point in RAM we choose. In line 10 we see:

LET ADDRESS = 28665

This is because the first instruction is at 6FF9 (28665 in decimal). The op-codes in Hex, are stored in a DATA statement in line 80. We have moved RAMTOP down using the CLEAR command, to 28600; this is lower than necessary but you should always leave room for movement. You may want to alter or change a program, for instance.

Now type:

RUN

The program will stop with:

Out of DATA Line 80

That's fine, it is just that we haven't put a check in for one end of the DATA – no harm done.

The program is loaded into ten bytes of RAM starting at 6FF9 or 28665 decimal. We can give it a run now – type:

PRINT USR 28665

On the screen appears the number two.

The first thing we know is that it does manage to reach the RET instruction at 7002 Hex. This we can be sure of (otherwise it would have got stuck in an endless loop, so you would have had to pull the plug out!). The second thing we notice is that a two is printed.

What happens is that whenever the Spectrum finishes 'doing' our machine code, (by meeting a RET) it returns the last value held in BC on the screen. If we look at the program we see:

LD BC,0002

which, as you know, makes the BC register pair hold the value two. This is why when we use PRINT, with a USR statement, a number appears on the screen. We do not always want to have a figure appear on the screen, in part of a game for example, but we can get round this using the LET statement. We use a 'dummy' variable. This is a variable which serves no useful purpose, except to match up with the LET statement. Try:

LET dummy = USR 28665

Nothing appears on the screen this time, but it has worked. Just to prove it, type:

PRINT dummy

Surprise, surprise! Up comes the number two on the screen.

Sometimes we will want to move a program around. For instance, we have a program which we have written to fit at the very top end of RAM on a 16K Spectrum. Then a friend wants to use the program, but this time at the top end of RAM on a *48K* Spectrum. Using our BASIC loader program, the obvious thing to do would be change the value assigned to 'address' in line 10. But if we had used JP in the program, that would not be all that would need changing.

```
7FF0   LD  B,08
7FF2   LD  A,4C
7FF4   ADD  A,B
7FF5   JP   7FFF
             :
             :
7FFF   RET
```

The above program is the one written for the 16K Spectrum. It tucks right up to 7FFF, the last byte of RAM. It just adds 08 to 4C (Hex) then jumps to 7FFF where it 'meets' a RET instruction, to return to BASIC.

Now let's have a look what happens if we simply move it up in RAM for the 48K Spectrum:

```
FFF0   LD B,08
FFF2   LD A,4C
FFF4   ADD A,B
FFF5   JP  7FFF
         :
         :
FFFF   RET
```

Now again it succeeds in adding 08 and 4C together but it jumps to 7FFF. This byte could be filled with any old rubbish on your friend's Spectrum, so we have to alter the JP to FFFF — then it would work properly. That is quite straightforward, but imagine a large program with 50 JPs in it. That would take some time to alter all of those. What we need is a program that is 're-locatable'. That is one that can be put almost anywhere without alteration.

In machine code we have a special instruction to make this possible. It has a few more limitations compared with JP, but still is a very versatile instruction. To use it we must understand the *two's complement* convention.

The jump itself is called *Jump Relative* (JR), and is followed by a number, specifying how many bytes to skip, and whether forwards or backwards. If it is to jump backwards, then a negative number is used, and this is where we need to learn about the two's complement convention.

Normally the CPU just deals in positive numbers but sometimes, as in the case of JR, it can recognise certain types of single byte numbers as being negative.

To form a negative number, let's say negative 10 (Hex), you subtract it from 100 Hex.

100 Hex – 10 Hex = F0 (or –10 HEX)
256 dec – 16 dec = 240 (or –16 dec)

The bottom calculation is the same, but in decimal. It is probably easier for you to work the subtraction out in decimal, so a little conversion is required. You can use Appendix A in the Sinclair Manual for this or Appendix A in this book, to convert the values.

In one Z80 instruction set there is an instruction which will do this calculation for you:

NEG

It makes a positive number in the Accumulator into a negative one like we did manually before. To show this in action we shall try a small program.

Type in the BASIC program listed earlier (forget that if you still have it in your Spectrum) and make sure you type:

CLEAR 28600

Now, here is the machine code program we shall use:

```
7000 3E 10      LD   A,10
7002 ED 44      NEG
7004 4F         LD   C,A
7005 06 00      LD   B,00
7007 C9         RET
```

Notice that NEG is a two byte instruction. ED Hex is the prefix then 44 is the op-code.

To load it into the Spectrum change line 80, the DATA statement, to:

```
  80>DATA "3E10","ED44","4F","06
00","C9"
```

Change line 10 to:

```
10 LET address=28672
```

Now type:

RUN

Now type:

PRINT USR 28672

As we expect, up on the screen comes 240, that is the decimal answer, the same as we worked out before.

Looking at the program:

7000 A is loaded with 10 Hex.
7002 It is made into negative 10.
7004 We put A into BC so we can see the answer on the screen.
7005 The high byte, B, is cleared because it is not needed with numbers smaller than 256.
7007 RET returns us to BASIC.

Have a look at this program:

```
7000 0E 00          LD    C,00
7002 06 01          LD    B,01
7004 18 04          JR     700A
7006 0E 10          LD    C,10
7008 06 00          LD    B,00
700A C9             RET
```

First it clears C and loads B with 01 Hex. So BC contains 0100. Then it Jumps Relative, the 04 means skip the following four bytes, so it misses out the LD C,10 and LDB,00 thus reading one RET instruction. The number following the JR instruction, in this case 04, always operates from that byte when positive.

So if 08 followed, it would skip the next eight bytes.

Now look at this program:

```
7000 C9             RET
7001 3E 10          LD    A,10
7003 06 06          LD    B,06
7005 80             ADD   A,B
7006 18 F8          JR     7000
7008 00             NOP
```


If we started it at 7001, then A is loaded with 10, B with 06, the two added together and we come to Jump Relative. This time the following number is negative. It jumps *backwards* by eight bytes. If we start at the NOP, and move eight bytes backwards, we 'land' on RET telling the CPU to return to BASIC. Let's check this out:

256 – 8 = 242 ; convert to Hex:
242 = F8 Hex

F8 follows the JR instruction, so that seems fine.

Using two's complement we can have positive numbers 0–127 decimal and –1 to –127 decimal. (Yes, zero is counted as being positive.) In a long program we would not be able to use JR all the time because we would not be able to jump further than 127 bytes forwards or backwards. In this way, we cannot make all programs relocatable.

Study this:

```
7000 18 FE          JR       7000
7002 C9             RET
```

You should recognise that the FE Hex is interpreted as –2. So it would just loop back to the JR instruction forever, or until you unplug the Spectrum.

If you do not mind typing the BASIC loader in again, you can try the above program.

Just change line 80 to:

```
80 DATA "18FE","C9"
```

Type RUN then:

PRINT USR 28672

Nothing appears to happen!

The computer is stuck in an endless loop! Try pressing BREAK. Nothing happens again. Remember BREAK does not work with machine code. You can test it for yourself in the program, and we shall learn how to do that later on in the book. For now you will just have to unplug and start again.

Sometimes in a program, you will want to use the same routine lots of times. To save you having to enter it every time it is needed you can use subroutines. They're quite like the ones we use in BASIC using GOSUB and RETURN. In machine code we do have RETURN written as RET but instead of GOSUB we have CALL.

When we use machine code on the Spectrum, our routine is actually a subroutine CALLed by the ROM. So when we wish it to end and return to the ROM (back to BASIC) we use the RET instruction.

CALL uses a direct address, eg. it is simply loaded into PC, not calculated relatively like JR. There is no form of CALL relative which also inhibits the use of relocatable programs.

The following program adds the two double byte numbers held in BC and DE and returns the result in HL.

```
7000 60             LD    H,B
7001 69             LD    L,C
7002 19             ADD   HL,DE
7003 C9             RET
```

It is written as a subroutine, here is a short program to use it.

```
7004 01 4C 2A       LD    BC,2A4C
7007 11 7E 4D       LD    DE,4D7E
700A CD 00 70       CALL  7000
700D 44             LD    B,H
700E 4D             LD    C,L
700F C9             RET
```

It loads BC with 2A4C and DE with 4D7E then CALLs the 'adding' subroutine. The result is loaded back into BC so we

can print the result in decimal when we return to BASIC. Now let's give it a run through ensuring line 10 is:

```
10>LET address=28672
```

Change line 80 to:

```
80>DATA "60","69","19","C9","0
14C2A","117E4D","CD0070","44","4
D"
```

Now type RUN. Then:

PRINT USR 28672

Up comes 30666 on the screen. That's the answer in decimal.

There are some subroutines which are useful to use in many programs. For instance, there is one in Chapter Seven which tests the BREAK key. The ROM itself holds many very useful subroutines, two of these we shall look at later.

As you should have noticed the Hex code for an *unconditional* CALL is CD. Following it is the direct address, two bytes:

eg. 'CALL 70C0 — CDC0 70'

There are other forms of CALL called conditional CALLs but we shall look at those in Chapter Two. Another form of CALL is RST. It is different in that you can only access subroutines at:

0000
0008
0010
0018
0020
0028
0030
0038

Those addresses are all in the ROM. It is faster than the CALL and only occupies one byte because a different code is used for each RST address. RST is short for RESTART.

RST ØØ — C7
RST Ø8 — CF
RST 1Ø — D7
RST 18 — DF
RST 2Ø — E7
RST 28 — EF
RST 3Ø — F7
RST 38 — FF

You may be wondering why I'm telling you this, because if they're addresses in the ROM you can't use them for your own subroutines.

BUT!! This is where one of the most useful routines in the ROM lies — at RST, 1Ø. It is the print routine and from here all the FLASHy, BRIGHT and colourful characters can be printed.

The subroutine is a minor goldmine for us machine coders, and saves a lot of the trouble of printing. First, the A register must be loaded with the character you want to print, then you can use the instruction RST 1Ø to have it printed on the screen. For instance, say you want to print 'HI':

```
LD A,48
RST 1Ø
LD A,49
RST 1Ø
RET
```

First, one Accumulator is loaded with 48 Hex. Look at the table in Appendix A of the Sinclair manual of this book to check this is the code for the character 'H'. Then RST 1Ø makes the CPU execute a subroutine starting at ØØ1Ø Hex, to put it technically. And then it goes on to print 'I' in a similar manner. I did mention colour earlier, and I shall explain how to 'colour' your words

and more! Have a look again at Appendix A, you will see the INK control code is 1Ø Hex. Now, if we send this code to the print routine, then send a number between ØØ and Ø7, the appropriate INK colour will be selected! Let's try a Blue Peter.

```
7000 3E 10        LD   A,10
7002 D7           RST  10
7003 3E 01        LD   A,01
7005 D7           RST  10
7006 3E 50        LD   A,50
7008 D7           RST  10
7009 3E 65        LD   A,65
700B D7           RST  10
700C 3E 74        LD   A,74
700E D7           RST  10
700F 3E 65        LD   A,65
7011 D7           RST  10
7012 3E 72        LD   A,72
7014 D7           RST  10
7015 C9           RET
```

To begin with it sends 1Ø in the Accumulator to the print routine, so the next code to be sent must be a colour code, in this case it is Ø1 for Blue. If it is not a valid code (valid being between ØØ and Ø9 inclusive) then an error message is given:

K Invalid colour

Looking further down the program you should realise, using Appendix A, that the codes are for 'Peter'. Note particularly that the code for RST 1Ø is D7. You may be thinking that the program is very long-winded just to print 'Peter' in Blue but that is the slow way of doing it (slow to put in, very fast to execute) but sooner or later you'll figure out a quicker method. But if you don't then you can use my routine described a little further on. Now we can give Blue Peter a whirl!

Change line 1Ø of the loader to:

```
10>LET address=28672
```

(it may already be that!)

and line 8Ø to:

```
  8Ø>DATA "3E1Ø","D7","3EØ1","D7
","3E5Ø","D7","3E65","D7","3E74"
,"D7","3E65","D7","3E72","D7","C
9"
```

Now after you've checked it (make sure it's all correct) then type RUN then:

PRINT AT Ø,Ø;:RANDOMIZE USR 28672

Hopefully a 'Blue Peter' will appear on the screen (as you may have guessed this is an adapted BBC program!). Have a look at the third block of data in line 8Ø of the loader, "3EØ1". Try changing the Ø1 to another colour code to check if it works properly. If you do, remember to run the loader again, before typing the USR line. I can't claim to know everything and one thing I admittedly haven't figured a definite reason for is that when the PRINT AT is left out and the USR statement is executed alone, 'Blue Peter' merely appears and disappears. Try it for yourself:

RANDOMIZE USR 28672

See what happens? Now find out what happens when you type:

i CLS : RANDOMIZE USR 28672
ii PRINT TAB 1Ø : RANDOMIZE USR 28672
iii PRINT' : RANDOMIZE USR 28672
iv PRINT; : RANDOMIZE USR 28672
v PRINT, : RANDOMIZE USR 28672

There are plenty more things that can be done by using one print routine. You can set it to FLASH (how rude!) or make it BRIGHT. Once again, have a glance at that kernel of information, the infamous Appendix A and look up the code for FLASH. With any luck you'll find it is 12 Hex. If you didn't come to that then there is either a printing error, or you need glasses! Now, to turn FLASH on we simply follow 12 with Ø1

and to turn it off we use ØØ, and similarly with BRIGHT whose code is 13 Hex we use ØØ and Ø1 to turn it off and on respectively. Let's try making 'Blue Peter' BRIGHTer and FLASHier than before! Insert another section of Hex data at the beginning of line 8Ø (just before the "3E1Ø"):

"3E12", "D7", "3EØ1", "D7", "3E13", "D7", "3EØ1"

You should understand that the 12 is for FLASH and the 13 is for BRIGHT. Remember to check it, then RUN the loader, then type:

PRINT AT Ø,Ø;: RANDOMIZE USR 28672

There are many other control codes we can use — here is a full list:

Ø6	This moves the print position to column ØØ or 16 whichever is next, just like using a comma in a PRINT statement. Just use Ø6 on its own.
Ø8 LEFT Ø9 RIGHT ØA DOWN ØB UP	These four codes you would expect to move the cursor up, down etc. but only left (Ø8) works, the others print a '?' Ø8 (LEFT) can be used for overprinting as described in the Sinclair manual.
1Ø INK	We have used this code already but just to recap it must be followed by a valid INK code (ØØ–Ø9), to change the INK colour.
11 PAPER	This, in the same manner as 1Ø (INK), sets the PAPER colour. Again a valid code must follow.
12 FLASH	Code 12 turns FLASH on when followed by Ø1, and off when followed by ØØ. Code Ø8, can be used for 'transparent' printing. That is printing that is just the same as what was underneath. (A full explanation is given in Chapter 16 of the Sinclair manual.)

13 BRIGHT	Changes the luminosity (for me, you and the window cleaner, that's brightness!) in the same manner as described for FLASH (12).
14 INVERSE	Turns inverse printing on and off in the same way as BRIGHT.
15 OVER	This code turns the OVER printing mode off or on. Used in conjunction with code Ø8 (cursor left) can be used to put characters 'on top' of each other. Follow it with a '1' or 'Ø' as required.

Chapter Two
TO GO OR NOT TO GO

We now know how to jump, Jump Relative and how to call a subroutine and return from it. The next thing for us to look at are decisions; after all, a computer is defined as a machine capable of making logical decisions.

DECISIONS, DECISIONS

The CPU makes decisions based on a single register, the 'F' register or 'Flags'. Let us have a look at the bits of the 'F' register.

BIT			
	Ø	C	Carry Flag
	1	N	Add/Subtract Flag
	2	P/V	Parity/Overflow Flag
	3	–	NOT USED ALWAYS ZERO
	4	H	Half carry Flag
	5	–	NOT USED ALWAYS ZERO
	6	Z	Zero Flag
	7	S	Sign Flag

As you can see there are only five flags, two bits being unused. Each bit has a letter or two letters as an abbreviation for its meaning. We can ignore the Add/Subtract Flag (S) and H Flag because the CPU cannot make decisions based on these, and has little need to.

The flags change after certain instructions and in certain ways. They usually change when an operation is done with one Accumulator such as subtraction or addition; they also change when a register is INCremented or DECremented and in many

other situations. For instance, if we add 10 Hex to the Accumulator, the flags will tell us if the Accumulator is now zero, if it is negative (according to two's complement convention) or if it had overflowed and caused a carry.

THE FLAGS THEMSELVES

The Carry Flag tells us if an overflow occurred. For instance, if we add 10 Hex and F3 Hex, we get 103 Hex which is too large for a single byte register and so the first digit (one) gets removed, leaving 03 as the result. But F3 + 10 does not equal 03 so the Carry Flag is set to one to warn us that an overflow has taken place. Also, if there is an underflow it will be set. For instance, if we take 05 from 03 the result is a minus number in two's complement convention. But we do not know if that number is a positive Hex number, or a two's complement number. So we can check the Carry Flag to see if there has been an underflow and if there has, the Carry Flag will be set.

Parity/Overflow Flag The use of this flag and the conditions of when it is set or reset are a little complicated and would not hold much relevance if I explained them in this section of the book.

Zero Flag The use of this flag is quite simple: it tells us if the result of the last operation was zero. For instance, if we first load register B with 02 Hex. Now, if we DEC B, the Zero Flag will be rest to zero because B did not end up as 00Hex. If we now DEC B again it does end up as zero so the CPU sets the Zero Flag to one. This is a *VERY* useful flag and we will see more of this later on.

Sign Flag If we are working by two's complement convention, the Sign Flag will tell us if the result of the last operation was negative. If it was negative by two's complement convention then the Sign Flag will be set, otherwise it is reset.

The other two flags, the H and N flags are for the CPU's reference only — so we won't bother to examine them.

Consider this example:

```
LD A,10
LD B,10
SUB A,B
```

10 is subtracted from 10 which leaves zero, so one Z flag is set. There has been no carry so the C flag is reset. The Sign Flag is reset to indicate that the Accumulator had a final value that was positive:

```
Z : 1
C : 0
S : 0
```

Have a look at the following examples; below each are the resulting flag conditions:

```
1.            LD A,00
              LD B,B5
              ADD A,B
  Z : 0 (Zero)
  C : 0 (Carry)
  S : 1 (Sign)

2.            LD A,00
              SUB 20
  Z : 0 (Zero)
  C : 1 (Carry)
  S : 1 (Sign)

3.            LD A,00
              SUB B5
  Z : 0 (Zero)
  C : 1 (Carry)
  S : 0 (Sign)

4.            LD A,00
              ADD 20
  Z : 0 (Zero)
  C : 0 (Carry)
  S : 0 (Sign)
```

What would be the resulting value if we added 10 Hex to 70 Hex?

What would the Zero, Carry and Sign Flags hold?

Well, firstly, if we add 70 Hex to 10 Hex we get 80 Hex or −80 in two's complement form. Neither 80 or −80 are zero, so the Z flag is reset. There hasn't been an underflow or overflow so the Carry Flag is reset. But bit 7 is set indicating that by two's complement convention the number is negative so the sign bit is set to one.

'Where is all this Flags business leading to?' you may well ask. Well, using JR, JP, Call and RET in their *conditional forms*, we can make *conditional jumps*:

```
  JP   cond1, nnnn
CALL   cond1, nnnn
  JR   cond2, dis
RET    cond1
```

cond[1] = Z/NZ/NC/C/PO/PE/M/P   dis = ± Hex number
cond[2] = Z/NZ/NC/C   nnnn = two byte Hex address

As you can see, each of the jumps and the return instruction now has a letter or pair of letters suffixing them. This is what the letters mean:

Z	=	If Zero Flag is set.
NZ	=	If Zero Flag is not set.
C	=	If Carry Flag is set.
NC	=	If Carry Flag is not set.
PO	=	If parity odd (if P/V flag is reset).
PE	=	If parity even (if P/V flag is set).
M	=	If minus (if S flag is set).
P	=	If positive (if S flag is reset).

So we can interpret:

JP Z, 705C

as 'if the result of the last operation was zero then jump directly to 705C'. Also:

CALL NC, 700A

can be interpreted as 'if the last operation didn't cause a carry then call the subroutine at 700A.

By 'the last instruction' we mean the last instruction which alters the flags. Instructions such as 'LD' and many others do not adjust the flags at all.

```
LD A, 10
ADD A, 05
LD B, 03
LD HL, 0735
RET Z
```

When, in the above program, 'RET Z' is reached the flags are still set to reflect the value of A after 'ADD A, 05'. The various 'LD' instructions can be ignored when looking at the flags. Here is a list of the instructions which do adjust the flags that interest us, that is the C,Z,S and P/V flags.

ADC
ADD
AND
BIT
CCF
CP
CPJ
CPJR
CPDR
CPL
DAA
DEC (single byte registers only)
INC (single byte registers only)
IN
INI
IND
INIR

INDR
LD A,I (note, these are the only basic 'LD'
LD A,R instructions that adjust the flags)
LDI
LDD
LDIR
LDDR
NEG
OR
OUTI
OUTD
OTIR
OTDR
POP AF (flags are set by top byte of stack)
RLA
RL
RLCA
RLC
RLD
RRA
RR
RRCA
RRC
RRD
SBC
SCF
SCF
SLA
SRA
SRL
SUB
XOR

We have not looked at some of these instructions yet so do not start worrying.

As you can see, most of these instructions are mathematical in some form or another. If you wish to check the exact flags

altered for each instruction check Appendix C at the back of the book.

There are two very common uses for the flags these use firstly the 'DEC' instruction and secondly the 'CP' instruction.

In BASIC we can quite easily form loops using a FOR... TO...(STEP)... NEXT sequence. In machine code things are quite different although the principle is the same. Firstly, to form a very basic type of loop, we load a register with the amount of times we want a particular thing to be done. Then, where we would put a 'NEXT' in BASIC we decrement the register, and if it is not yet zero the loop is done again. Have a look at this program to make things clearer. It prints 5 'A's using 'RST 1Ø' as we described in Chapter One.

```
7000 06 05       LD   B,05
7002 3E 41       LD   A,41
7004 D7          RST  10
7005 05          DEC  B
7006 20 FA       JR    NZ,7002
7008 C9          RET
```

7ØØØ : B is set up with Ø5 Hex.
7ØØ2 : A is loaded with 41 Hex because this is the code for 'A'.
7ØØ4 : the character is printed.
7ØØ5 : B is decremented because it is the counter.
7ØØ6 : If B hasn't reached zero then we need to go back to 7ØØ2 so another 'A' can be printed, — we use JRNZ meaning 'Jump Relative if not zero'.

The B register is very commonly used as a counter. This is because we have a useful instruction exclusively for register B.

DJNZ (dis) 1Ø*xx*

This instruction combines 'DEC B' and 'JR NZ' into one instruction and has two advantages:

(a) It uses up less memory — one byte for the instruction and another for the displacement.

(b) It is quicker than doing 'DEC B', 'JR NZ' and this can sometimes be critical in precision timing loops.

This means we can save a byte and re-write the program as:

```
7000 06 05        LD    B,05
7002 3E 41        LD    A,41
7004 D7           RST   10
7005 10 FA        DJNZ  7001
7007 C9           RET
```

Type the modified version in — the data is:

```
  60>DATA "0605","3E41","D7","10
FA","C9"
```

Ensure line 10 is:

```
  10>LET address=28672
```

Now once you have done that and checked it, type:

RUN, then:

PRINT AT 0,0;:RAND USR 28672

Remember that we need the 'PRINT AT 0,0;' because otherwise the Spectrum thinks it is printing on the Edit line (or just above, in fact) and so when it has finished it clears the lot. Now, if your typing is accurate you will find five beautiful capital 'A's on your screen.

This system of looping is fine when a simple count is needed, but what if you need a loop using values 10 to 20 Hex? Now this system will not do because it relies on the counting register ending up with 00 Hex. So here is where we introduce a very useful instruction:

CP

Its literal meaning is 'Compare the following with the Accumulator'. The results of the comparison are reflected in the flags. What it actually does is subtract the following data from A without actually putting the result in A — it just sets the flags to that result.

The following data can be a single byte register, direct data or the memory location pointed to by HL, IX + (dis) or IY + (dis).

CP nn	FE nn
CP A	BF
CP B	B8
CP C	B9
CP D	BA
CP E	BB
CP H	BC
CP L	BD
CP(HL)	BE
CP(IX + (dis))	DD BE
CP(IY + (dis))	FD BE

Above are the codes for each of the CP instructions.

The compare instruction is useful in emulating IF...THEN situations; for instance, if we 'CP20' then if A contains 20 the Z flag will be set (20 − 20 = 0), if A is less than 20 then the C flag will be set and if it is greater than 20, then the C flag will be reset.

The following situations can be interpreted into almost BASIC meanings:

1. CP 5F
   JP Z, 7000

As you can see, 5F is theoretically subtracted from A. In this case, if the calculation results in zero, a jump is made. The only way it can result in zero is for the Accumulator to hold 5F to begin with (5F − 5F = 00), so we could say:

If A = 5F (Hex)THEN GOTO 7000(Hex)

2. CP 2C
   JP C, 7000

2C, in this case, is theoretically subtracted from A. (Remember we say theoretically because the result is never put into A.)

Now if A is larger than 2C then 2C taken from it will give a positive result. But if A is smaller than 2C then it will give a negative result, an *underflow*, so the Carry Flag is set. We could write this as:

IF A< 2C(Hex)THEN GOTO 7000(Hex)

3. CP 10
   JP NC, 7000

Now if we take 10 from A and A contained a number greater than 10 then the result is positive. This means that there will be no overflow and so the Carry Flag will be reset. So for this we could write:

```
IF A>10(Hex)THEN GOTO 7000 (Hex)
```

Now getting back to our problem of a loop that doesn't end in 0. Have a look at the following, it loops from 10 Hex to 1F Hex.

```
7000 3E 10          LD    A,10
7002 3C             INC   A
7003 FE 20          CP    20
7005 20 FB          JR     NZ,7002
7007 C9             RET
```

Firstly A is loaded with 10, the value we start the loop at. Then, at 7002 the Accumulator is incremented, then 'CP 20' compares 20 and if the Accumulator does not hold 20 then the Z flag is reset so a jump is made back to 7002. We compare 20 because it is one more than the last value we want it to do. If we compared 1F then it would increment A at 1E, make it 1F but a return would be made and 1F would never be run.

Expanding on this program, here is a program that prints 16 'A's in columns, 16 to 31 decimal or 10 to 1F Hex.

```
7000 06 10          LD    B,10
7002 3E 17          LD    A,17
7004 D7             RST   10
7005 78             LD    A,B
7006 D7             RST   10
```



```
7007 3E 41          LD    A,41
7009 D7             RST   10
700A 78             LD    A,B
700B FE 20          CP    20
700D 20 F3          JR     NZ,7002
700F C9             RET
```

7000 B is set up with 10 instead of A because we will need A for the printing.
7002 A is loaded with 17 Hex, the control code for TAB.
7004 The TAB control is 'sent'.
7005 Now we get the column number, in B, into the Accumulator.
7006 The column number is 'sent'.
7007 A is then loaded with the code for 'A'.
7009 That is then printed.
700A We now get the count, in B, into A so we can test it.
700B 20 is compared because the last value that should be used is 1F (31 decimal).
700D If the Accumulator is not yet 20 then another A is printed.
700F Return to BASIC.

With loops using single byte registers everything is quite straightforward, but when we come to need double byte counting there is an added complication. The decrementing of a double byte register affects the flags so that they will not work. So you can *not* do this, for instance:

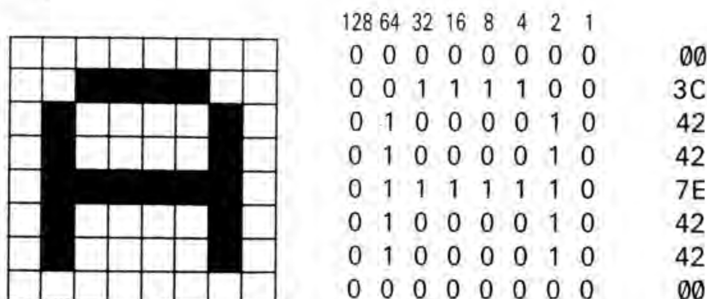
LD BC, 062D
DEC BC
JR NZ. . . .

We shall discuss how to get around this problem in Chapter Four where we shall learn all about instructions that operate 'logically' on a register.

THE CHARACTER SET

Before I give you a neat little routine which provides double height characters, we shall revise the character set.

It is quite convenient that there are user-defined graphics on the ZX Spectrum because this means less explaining is needed from me! You should all know that each character is made up of 64 dots in an eight by eight square. Many of you will know that each 'slice' of eight dots is stored in memory as eight bits or a byte.



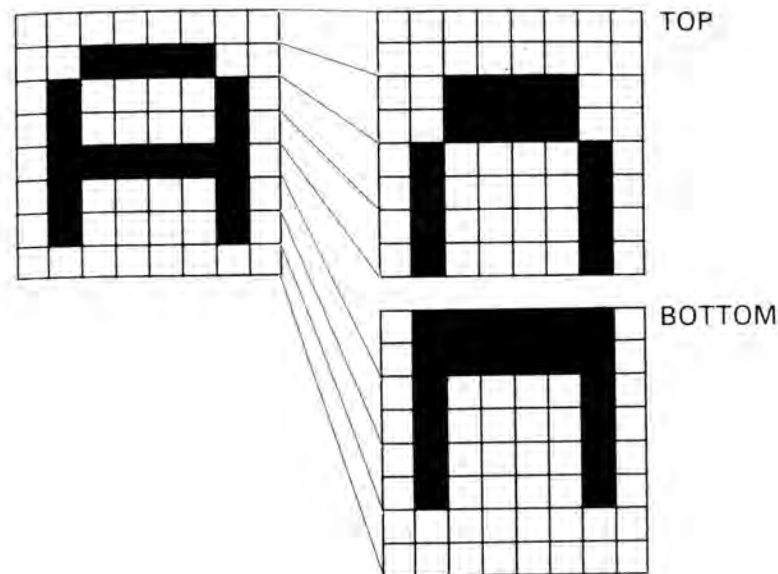
128	64	32	16	8	4	2	1	
0	0	0	0	0	0	0	0	ØØ
0	0	1	1	1	1	0	0	3C
0	1	0	0	0	0	1	0	42
0	1	0	0	0	0	1	0	42
0	1	1	1	1	1	1	0	7E
0	1	0	0	0	0	1	0	42
0	1	0	0	0	0	1	0	42
0	0	0	0	0	0	0	0	ØØ

The usual Sinclair character set is stored at 3DØØ onwards. Here, there are the definitions of the characters from a space to the copyright symbol. The user-defined graphics are stored separately and the graphics are calculated by the ROM. If we re-define the character set elsewhere we must change the value in CHARS (5C36) in the system variables. It is normally 3CØØ, which is 1ØØ Hex less than the address of the character set. You must remember that when you define a new character set, this must be changed to 1ØØ Hex *less* than the actual address so that you can use the character set.

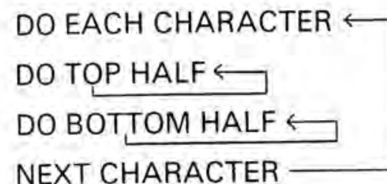
DOUBLE HEIGHT CHARACTERS

To create a double height character set we must in fact create two character sets. One for the top half of the characters and one for the bottom half. To generate these two character sets we get one original character and 'stretch' them into two halves. For example, with A. (see over).

So, for the top four bytes we copy each one in double into the first character set for the top half. Then we must copy the bottom four bytes in double into the second character set for the bottom half.



By using machine code we can have this laborious job done in a blink of an eye. Firstly, we need to have an outer loop which cycles through each of the 96 characters. Then we want one inner loop to the top half of each character and another loop to do the bottom.



Now we must decide where to put these character sets. The best place is above RAMTOP and this is where we will store them. If you have a disassembler, editor or alterer which sits above RAMTOP then you will have to change the addresses. Where it says PUT1 and PUT2 in this first bit of the program use the following addresses:

48K : (PUT1); FA(PUT2); FD
16K : (PUT1); 7A(PUT2); 7D

Our first task is to set up a loop to process each of the characters. We will need to have two addresses on hand, which will tell us whereabouts we're up to in the two different character sets.

```
LD HL,BOTSET     21 00 (PUT2)
PUSH HL          E5
LD HL,TOPSET     21 00 (PUT1)
LD DE,SNCSET     11 00 3D
LD C,60          0E 60
```

So that is our first piece. We set the address for the bottom half (BOTSET) and stack it. We then set the address for the top half (TOPSET) and leave this in HL.DE is loaded with SNCSET (3D00) which is the beginning of the Sinclair character set. Finally, C is loaded with 60 Hex because there are 60 Hex (96 decimal) characters to be done. Next, we want the loop for the top half of the character.

```
TPHALF  LD B,04        06 04
TSLICE  LD A,(DE)      1A
        LD (HL),A      77
        INC HL         23
        LD (HL),A      77
        INC HL         23
        INC DE         13
        DJNZ TSLICE    10 F8
```

Notice how the first instruction loads B with four. This sets B up as a counter and since we are to do four 'slices', it is loaded with four. The Accumulator is then loaded with the byte pointed to be DE; remember that DE points to the Sinclair character set. This byte is now transferred into the top character set. HL is then incremented and the byte, once again, is stored in (HL) (if you remember we have to put each in double to get the stretch effect). Next, DE is incremented so that it points to the next 'slice' of the character.

This process is repeated four times using the DJNZ instruction we learnt about earlier.

Our final step before closing the outer loop, is to make the bottom half of the character. This is done in a very similar way to the process of making the top half as described above.


```
        EX HL,(SP)       E3
        LD B,04          06 04
BSLICE  LD A,(DE)        1A
        LD (HL),A        77
        INC HL           23
        LD (HL),A        77
        INC HL           23
        INC DE           13
        DJNZ BSLICE      10 F8
        EX HL,(SP)       E3
        DEC C            0D
        JR NZ,TPHALF     20 E6
        POP HL           E1
        RET              C9
```

This bit starts quite differently with EX HL,(SP). For the uninitiated this quite useful instruction simply stops the value in HL and the top value on the stack over. This is used here to switch from HL pointing to the upper character set. This is so the lower character set it set up at a different address, otherwise things would be a mess! After DJNZ BSLICE there is another 'EX HL,(SP)'. This switches the values again so HL is set up for the top character set. C is then decremented, and if it isn't zero then it Jumps Relative to do the next character. If there are no more characters left to do then the last two instructions will be executed, 'POP HL' and 'RET'. The 'POP HL' is there so that the top value on the stack is removed, that value being the pointer in the 'top' character set.

```
7000  21 00 7D     LD    HL,7D00
7003  E5           PUSH  HL
7004  21 00 7A     LD    HL,7A00
7007  11 00 3D     LD    DE,3D00
700A  0E 60        LD    C,60
700C  06 04        LD    B,04
700E  1A           LD    A,(DE)
700F  77           LD    (HL),A
7010  23           INC   HL
7011  77           LD    (HL),A
7012  23           INC   HL
7013  13           INC   DE
7014  10 F8        DJNZ  700E
7016  E3           EX    (SP),HL
7017  06 04        LD    B,04
7019  1A           LD    A,(DE)
701A  77           LD    (HL),A
701B  23           INC   HL
701C  77           LD    (HL),A
701D  23           INC   HL
```

```
701E 13          INC  DE
701F 10 F8       DJNZ 7019
7021 E3          EX    (SP),HL
7022 0D          DEC  C
7023 20 E7       JR    NZ,700C
7025 E1          POP  HL
7026 C9          RET
```

To get this routine into action put the Hex codes into data statements at end of the Hex loader. Remember to put in the appropriate numbers for 16 or 48K (whichever machine you have). I have given the 16K ones in the program but if 48K users look back they will see where I have explained how to change them. If you are using a machine code monitor that uses addresses 7A00–7FFF on the 16K or FA00 FFFF on the 48K then some adjustments will have to be made to the addresses otherwise the program will set up the character sets in your monitor!

Now down to hows and whys of using your double height character generator. Assuming you have entered the data and RUN the loader then you now want to know how to use the new letters.

Firstly, we shall set up two variables:

For the 16K – LET TOP = 121
LET BOT = 124

For the 48K – LET TOP = 249
LET BOT = 252

Now, let's imagine we want to print 'Hello'.

```
10 POKE 23607, TOP: PRINT "Hello"
20 POKE 23607, BOT: PRINT "Hello": POKE 23607, 60
```

In the first line we point CHARS (A system variable, see Appendix C) to our top half character set. So then it uses our top halves of characters. Then to add our bottom half directly below we use POKE once again to point CHARS this time to our lower halves. The last POKE at the end of line 20 is what needs to be done to reset printing to normal.

See what happens when you:

POKE 23607, TOP

in direct mode. Also:

POKE 23607, 0

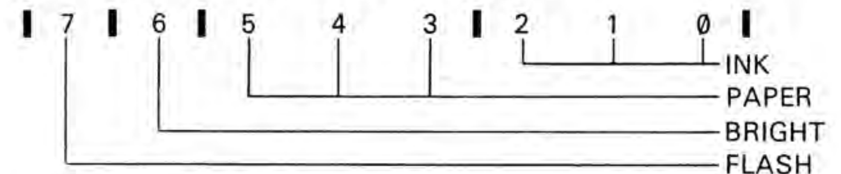
Why does it do that?

Can you think of a use for it?

Chapter Three
A CHAPTER OF BITS

As you should know a byte consists of eight bits. Each of these bits is binary digit which represents a number, if set. All these numbers added together form a byte. Sometimes we use a byte, not all together as an 'eight bit word', but as separate bits for flags or smaller numbers. We have already looked at the F register and its bits, and how each bit tells us something about the last operation or comparison. The colour attributes file (starting at 5800 Hex) has a byte for each character square on the screen. Each attribute byte tells the logic chip the INK colour, PAPER colour and whether that square is flashing or not, bright or normal.

This is how an attribute byte is arranged:



If we want to make a character square flash, we set bit 7, but none of the other bits must be adjusted. To do this we have the instruction:

SET

The following program makes the first character on the screen flash:

```
7000 21 00 58     LD    HL,5800
7003 CB FE        SET   7,(HL)
7005 C9           RET
```

7000 HL is loaded with 5800 Hex. This is the address of the first attribute byte, which refers to the first character on the screen.
7003 The SET instruction sets bit 7 of location (HL) to 1, and since HL points to the first attribute and bit 7 is the flash bit, it makes the first character flash.
7004 Return to BASIC.

The SET instruction only sets the specified bit to one — it leaves all the other bits as they were. It can be used to set any bit (0 to 7) of any single byte register (A,B,C,D,E,H or L) or any location pointed to by (HL), (IX + dis) or (IY + dis). Here is a list of the SET instructions ((x) is any number 0-7):

SET (x), A
SET (x), B
SET (x), C
SET (x), D
SET (x), E
SET (x), H
SET (x), L
SET (x), (HL)
SET (x), (IY + dis)
SET (x), (IX + dis)

To complement the SET instruction we have RES. It operates in a very similar manner except that it resets the appropriate bit to zero.

RES (x), A
RES (x), B
RES (x), C
RES (x), D
RES (x), E
RES (x), H
RES (x), L
RES (x), (HL)
RES (x), (IY + dis)
RES (x), (IX + dis)

So, if we want to stop that first character flashing we can set bit 7 to 0 using the RES instruction.

```
7000 21 00 58    LD    HL,5800
7003 CB BE       RES   7,(HL)
7005 C9          RET
```

Because there are so many different SET and RES instructions it would be a waste of space to list them here but you can find them all in Appendix A in either this book or your Sinclair manual.

The following program demonstrates the SET instruction by making the top eight lines of the screen flash.

```
7000 21 00 58    LD    HL,5800
7003 06 00       LD    B,00
7005 CB FE       SET   7,(HL)
7007 23          INC   HL
7008 10 FB       DJNZ  7005
700A C9          RET
```

7000	:	HL is loaded with the address of the first byte, 5800.
7003	:	B, being used as a counter, is loaded with 00 so the first 256 bytes are filled.
7005	:	The flash bit is set in (HL).
7007	:	HL is moved on to the next attribute.
7008	:	The DJNZ causes 256 operations.
700A	:	Return to BASIC.

Lets try this out, using one BASIC loader program change line 80 to

```
80>DATA "210058","0600","CBFE"
,"23","10FB","C9"
```

And check line 10 is:

```
10>LET address=28672
```

Now type RUN and when you get the error message type:

RANDOMIZE USR 28672

As you can see, the top eight lines are flashing. (If they're not, you've mis-typed something — if you've lost the loader, load it in again and try again, otherwise check line 80.) Now just to prove to yourself it won't affect anything else in those character squares, type:

LIST: RANDOMIZE USR 28672

And hey presto! You have got a program which is flashing its top!! As we responsible citizens cannot allow further disgusting behaviour, we must BRIGHTen the outlook with a slight modification.

```
7000 21 00 58     LD    HL,5800
7003 06 00        LD    B,00
7005 CB F6        SET   6,(HL)
7007 23           INC   HL
7008 10 FB        DJNZ  7005
700A C9           RET
```

Instead of setting bit seven we now are setting bit six, the BRIGHT bit. So change "CBFE" in line 80 to "CBF6".

Now type RUN, then:

LIST: RANDOMIZE USR 28672

Now if you are using white paper, the bit washed in machine code is whiter than white!

There is one more instruction concerning bits which we have to look at:

BIT

The BIT instruction tests a certain bit and if the bit is zero the Z flag is set, or if the bit is one the Z flag is reset. The different forms of the BIT instruction are similar to RES and SET, and like RES and SET they all have the prefix code CB. The codes can be found in Appendix A of this book or the Sinclair manual.

BIT (x), A
BIT (x), B

BIT (x), C
BIT (x), D
BIT (x), E
BIT (x), H
BIT (x), L
BIT (x), (HL)
BIT (x), (IX + dis)
BIT (x), (IY + dis)

The following program changes all the flashing characters to none flashing and *vice versa*.

```
7000 21 00 58     LD    HL,5800
7003 CB 7E        BIT   7,(HL)
7005 28 04        JR      Z,700B
7007 CB BE        RES   7,(HL)
7009 18 02        JR      700D
700B CB FE        SET   7,(HL)
700D 23           INC   HL
700E 7C           LD    A,H
700F FE 5B        CP    5B
7011 20 F0        JR      NZ,7003
7013 C9           RET
```

7000	:	is loaded with the start of the attribute file.
7003	:	The flash flag of (HL) is tested.
7005	:	If the character is not flashing it goes to 700B.
7007	:	Else stop it flashing.
7009	:	JR to 700D.
700B	:	Make the character flash.
700D	:	Move HL to the next attribute.
700E	:	Get H into A and test to see if it is at the end of attribute file.
7011	:	If not open GOTO 7003 to do the next byte.
7013	:	Return to BASIC.

Let's give it a try.

Change line 80 of the loader to:

```
  80>DATA "210058","CB7E","2804"
,"CBBE","1802","CBFE","23","7C",
"FE5B","20F0","C9"
```

Now type RUN, then:

RANDOMIZE USR 28672

The screen is flashing! Now type:

RANDOMIZE USR 28672

Now it has stopped. Try typing:

PRINT AT 10,0; FLASH 1; "Testing FLASH 2 routine".

Now we can make everything else flash *but* the test message:

RANDOMIZE USR 28672

We shall now collect all we know so far and get cracking on a new program.

TYPEWRITER

The program we shall write is Typewriter. It is a simple program allowing the use of the screen as a page on which to do some typing — you could use it for making printed program instructions if you have a printer or just to have fun if not. The main functions of Typewriter are:

a) Cursor controls; up, down left and right.
b) Auto repeat.
c) Cursor Home.
d) Delete.
e) Carriage return (or Enter).

Firstly we want to have two bytes of data telling us where the cursor is on the screen, and two bytes of data telling us where to print in the attribute file. The fifth data byte will tell us what key was pressed. We shall give all these data bytes names:

PRINT — 6D00
ATTRB — 6D02
KEY — 6D04

We will start writing the program at 6D05 immediately following the data.

So far I have not explained how to get a keypress in machine code. Well, it is very simple indeed. In the system variables there is a location called LAST K, this contains the key that was last pressed. This is updated every 50th of a second (in the UK) or every 60th of a second (in the U.S.A.). The procedure to wait for a keypress and put it in the accumulator is:

1. Make LAST K zero.
2. Get LAST K into the Accumulator.
3. If the Accumulator is zero go back to step 2.

The good thing about this is that the system will generate an auto repeat for you.

We can use our knowledge of RST10 to print on the screen using the AT control code, but the flashing cursor cannot be done by RST10 because it would destroy what would be underneath our cursor. This is because our cursor will be different from the Spectrum's. Instead of going in between the characters it will go on top of them, a flashing blob. This saves the trouble of shifting everything around each time the cursor is moved.

The program will do one of two things when a key is pressed, print the character or do a cursor operation such as delete or cursor up. It will have to know what codes it should print and what codes are cursor controls. What we shall do is make the program look through a table of codes, each code having a two byte address after it. If the keypress code matches a code in the table it jumps to the address following it. If it doesn't find a matching code it prints the keypress code. Have a look at the table:

```
DEFB 07  —  EDIT      07 05 6D
DEFB 08  —  LEFT      08 83 6D
DEFB 09  —  RIGHT     09 5A 6D
DEFB 0A  —  DOWN      0A 69 6D
DEFB 0B  —  UP        0B 76 6D
DEFB 0C  —  DELETE    0C 94 6D
DEFB 0D  —  ENTER     0D A6 6D
```

```
DEFB ØF    —    GRAPHICS     ØFA5 6D
DEFB FF    —    TABLE END    FF
```

In the table there are the codes for a number of the control keys — each of the codes has two byte numbers following it. This two byte number points to the start of the routine which handles that control key:

Ø7	—	EDIT	—	Move cursor to Ø,Ø.
Ø8	—	LEFT	—	Move cursor to left.
Ø9	—	RIGHT	—	Move cursor right.
ØA	—	DOWN	—	Move cursor down.
ØB	—	UP	—	Move cursor up.
ØC	—	DELETE	—	Delete character.
ØD	—	ENTER	—	Carriage return.
ØF	—	GRAPHICS	—	Exit 'Typewriter'.

'Delete' deletes the character underneath the cursor and moves the cursor one square to the left. 'Enter' moves the cursor to the left-hand edge of the line below. 'Graphics' exits to BASIC leaving the screen intact, available for COPY.

Now we have cleared up a few problems we can draw ourselves a flow chart. On the next page you will see a complete flowchart for 'Typewriter'.

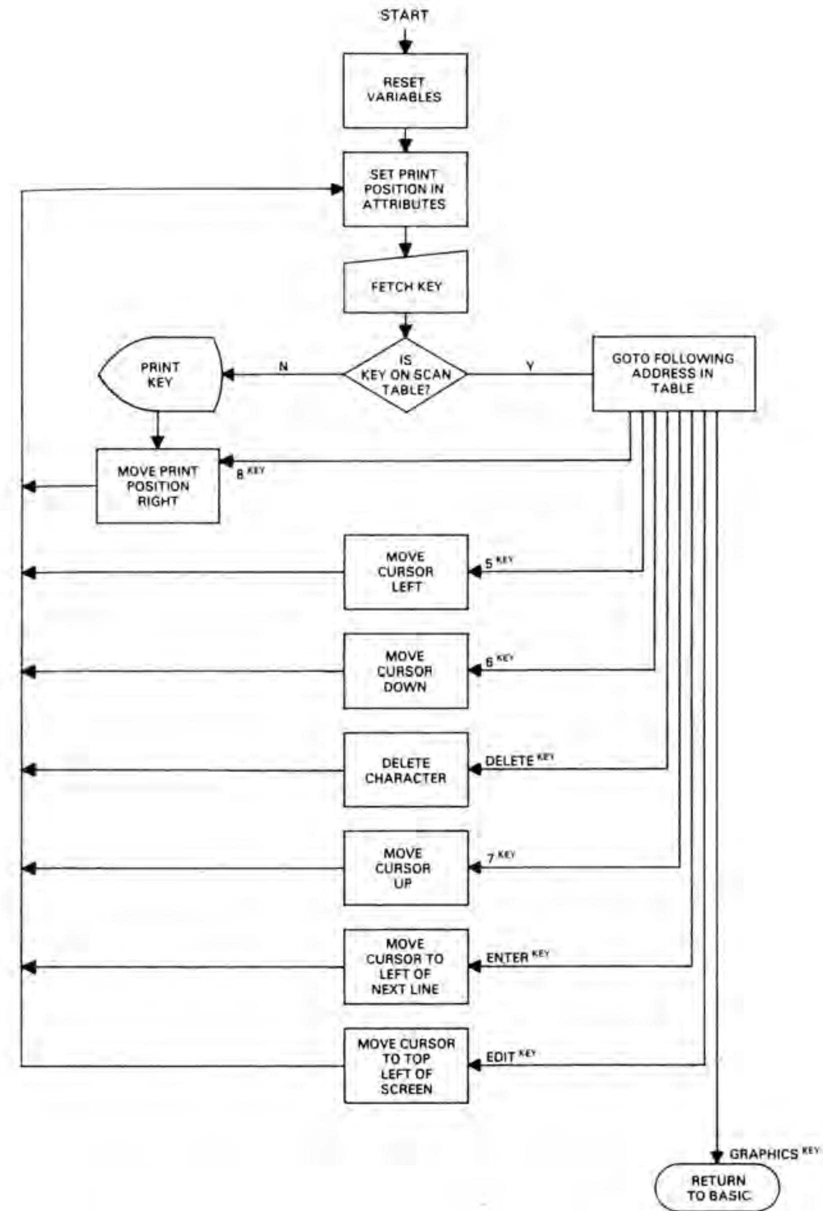
From this flowchart we can write the program systematically, writing the bit for each box in turn. So let's have a look at that first box.

START: SET PRINT AT Ø,Ø;

Now as we have decided, our cursor position as AT co-ordinates will be stored in data bytes called PRINT (6DØØ Hex). So quite simply for this first box PRINT (column number) and PRINT + 1 (line number) must be loaded with ØØ.

```
6DØ5  RESET     LD HL,ØØØØ        21 ØØ ØØ
                LD (PRINT), HL    22 ØØ 6D
```

SET ATTRIBUTE PRINT POSITION



TYPEWRITER PROGRAM FLOWCHART

What this entails is to find the address of the corresponding byte in the attribute file given the AT co-ordinates. To do this we multiply the line number by 32 add this to the column number and then add the result of that to ATTRBS, the beginning of the attribute file.

```
6D0B ATRSET    LD DE, (PRINT)     ED 5B 00 6D
               LD C,E             4B
               LD E,D             5A
               LD D,00            16 00
               LD HL,ATTRBS       21 00 58
               LD B,20            06 20
MULT 32        ADD HL,DE          19
               DJNZ MULT32        10 FD
               LD E,C             59
               ADD HL,DE          19
               LD(ATTRB),HL       22 02 6D
```

The last instruction refers to 6002 which as we said is ATTRB, the print position is the attribute file. There is no need to attempt to enter any of this yet — just understand each bit in turn and at the end we shall go about entering and running it all.

PUT CURSOR ON SCREEN

This is very simple indeed. HL contains the print position in the attribute file because that was calculated in the last box. So all we need do is set bit 7, the flash bit, in (HL).

```
6020  CURPUT     SET 7, (HL)     CB FE
```

GET KEYPRESS

As I described before, reset LAST K (5C08) to 0 and wait for a change.

```
6D22  KEYGET   LD A,00          3E 00
               LD HL,LAST K     21 08 5C
               LD(HL),00        36 00
      KEYTST   CP(HL)           BE
               JR Z,KEYTST      28 FD
```



```
               LD A,(HL)        7E
               LD (KEY),A       32 04 6D
```

The key value that was received is stored in 'KEY' (6D04).

REMOVE CURSOR

We now quite simply remove the cursor using the RES instruction.

```
6D30 NOCURS    LD HL,(ATTRB)    2A 02 6D
               RES 7,(HL)       CB BE
```

IS KEY ON TABLE?

Now we must find out if a special key is being pressed such as ENTER or DELETE or whether the code is simply to be printed. To do this we shall use the table I listed earlier which will be located at 'TABLE'. Operation will be as follows:

1. Point DE to the start of the table, 'TABLE' (6DAC).
2. See if (DE), the code in the table is the same as the keypress, 'KEY'.
3. If it is, move DE up by one byte.
4. Collect the following two bytes in the table and put them in HL. The JP (HL) to the routine for the key.
5. If the table code isn't the same as the keypress KEY, then move DE up three bytes to the next code in the table.
6. If that byte is FF then print the KEY, else go back to step 2.

```
6D35  SEARCH   LD DE,TABLE      11 AC 6D
               LD HL, KEY       21 04 6D
      TSTKEY   LD A,(DE)        1A
               CP(HL)           BE
               JR NZ,NEXBYT     20 07
               INC DE           13
               EX DE, HL        EB
               LD E,(HL)        5E
               INC HL           23
               LD D,(HL)        56
```

```
            EX DE,HL          EB
            JP (HL)           E9
NEXBYT      INC DE            13
            INC DE            13
            INC DE            13
            CP FF             FE FF
            JR NZ,TSTKEY      20 EE
```

We shall deal with the boxes for special keys later but now.

PRINT A CHARACTER

We looked in Chapter One at RST 10 and how it facilitates easy (very easy, in fact) printing on screen. We shall use control code 16 and RST 10 to get our keypress onto the screen.

```
6D4D PUT KEY   LD DE, (PRINT)    ED 5B 00 6D
               LD A,16           3E 16
               RST 10            D7
               LD A,D            7A
               RST 10            D7
               LD A, E           7B
               RST 10            D7
               LD A, (HL)        7E
               RST 10            D7
```

MOVE CURSOR RIGHT

After a character has been printed the cursor is moved one square right. If, before it was moved, it was on column 32, then it is moved to the first column of the next line by loading 6D00 (column number) with 00 (the jumping to DOWN where the cursor is moved down). This routine is jumped to when the 'cursor right' key is pressed. Once the cursor has been moved right a jump is made to ATRSET, the second box described.

```
6D5A RIGHT     LD HL,6D00        21 00 6D
               INC (HL)          34
               LD A, 20          3E 20
               CP (HL)           BE
               JR NZ, ATRSET     20 A8
6D63 NEXLIN    LD (HL),00        36 00
```


MOVE CURSOR DOWN

The cursor is moved down one square unless it is on the 22nd line of the screen. A jump is then made to ATRSET.

```
6D69 DOWN      LD HL,6D01        21 01 60
               LD A,15           3E 15
               CP (HL)           BE
               JP Z,ATRSET       CA 0B 6D
               INC (HL)          34
               JP ATRSET         C3 0B 6D
```

MOVE CURSOR UP

The cursor is moved up one square unless it is on the first line of the screen. A jump is then made to ATRSET.

```
6D76 CUR UP    LD HL,6D01        21 01 6D
               LD A,00           3E 00
               CP (HL)           BE
               JP Z, ATRSET      CA 0B 6D
               DEC (HL)          35
               JP ATRSET         C3 0B 6D
```

CURSOR LEFT

The cursor is moved left, unless it is on column zero in which case 6D00 (column number) is loaded with 1F Hex or 31 decimal, the last column and jump made to CUR UP. If the cursor was moved left successfully then a jump is made to ATRSET.

```
6D 83 LEFT     LD HL,6D00        21 00 6D
               DEC (HL)          35
               LD A,FF           3E FF
               CP (HL)           BE
               JP NZ, ATRSET     C2 0B 6D
               LD (HL) 1F        36 1F
               JP CUR UP         C3 76 6D
```

PRINT A SPACE

This is the delete routine. Using RST 10 a space is printed at

the current co-ordinates, then the cursor is moved back one square by jumping to LEFT.

```
6D94 DELETE    LD DE,(6DØØ)    ED 5B ØØ 6D
               LD A,16         3E 16
               RST 1Ø          D7
               LD A, D         7A
               RST 1Ø          D7
               LD A, E         7B
               RST 1Ø          D7
               LD A,2Ø         3E 2Ø
               RST 1Ø          D7
               JP LEFT         C3 83 6D
```

RETURN TO BASIC

When shift-9 is pressed (GRAPHICS) then a return to BASIC is made:

```
6DA5 EXIT RET    C9
```

'PUT CURSOR AT LEFT EDGE'

It doesn't even do that in fact. That is left up to the end of RIGHT called NEXLIN (6D63). This routine is jumped to when Enter is pressed.

```
6DA6 ENTER     LD HL,6DØØ      21 ØØ 6D
               JP NEXLIN       C3 63 6D
```

'KEY TABLE'

This is the last bit of listing before we go about entering it. It is the table of jump addresses for special keys.

```
6DAC TABLE     DEFB Ø7 'EDIT'         Ø7
               DEFB Ø56D              Ø5 6D
               DEFB Ø8 'LEFT'         Ø8
               DEFB 836D              83 6D
               DEFB Ø9 'RIGHT'        Ø9
               DEFB 5A6D              5A 6D
               DEFB ØA 'DOWN'         ØA
               DEFB 696D              69 6D
```



```
               DEFB ØB 'UP'           ØB
               DEFB 766D              76 6D
               DEFB ØC 'DELETE'       ØC
               DEFB 946D              94 6D
               DEFB ØD 'ENTER'        ØD
               DEFB A66D              A6 6D
               DEFB ØF 'GRAPHICS'     ØF
               DEFB FF 'END'          FF
```

Now we have gone through the complete listing, I shall list it in full so you have a complete reference to look at.

```
6D05 21 00 00    LD   HL,0000
6D08 22 00 6D    LD   (6D00),HL
6D0B ED 5B 00 6D LD   DE,(6D00)
6D0F 4B          LD   C,E
6D10 5A          LD   E,D
6D11 16 00       LD   D,00
6D13 21 00 58    LD   HL,5800
6D16 06 20       LD   B,20
6D18 19          ADD  HL,DE
6D19 10 FD       DJNZ 6D18
6D1B 59          LD   E,C
6D1C 19          ADD  HL,DE
6D1D 22 02 6D    LD   (6D02),HL
6D20 CB FE       SET  7,(HL)
6D22 3E 00       LD   A,00
6D24 21 08 5C    LD   HL,5C08
6D27 36 00       LD   (HL),00
6D29 BE          CP   (HL)
6D2A 28 FD       JR   Z,6D29
6D2C 7E          LD   A,(HL)
6D2D 32 04 6D    LD   (6D04),A
6D30 2A 02 6D    LD   HL,(6D02)
6D33 CB BE       RES  7,(HL)
6D35 11 AC 6D    LD   DE,6DAC
6D38 21 04 6D    LD   HL,6D04
6D3B 1A          LD   A,(DE)
6D3C BE          CP   (HL)
6D3D 20 07       JR   NZ,6D46
6D3F 13          INC  DE
6D40 EB          EX   DE,HL
6D41 5E          LD   E,(HL)
6D42 23          INC  HL
6D43 56          LD   D,(HL)
6D44 EB          EX   DE,HL
6D45 E9          JP   (HL)
6D46 13          INC  DE
6D47 13          INC  DE
6D48 13          INC  DE
```

```
6D49 FE FF        CP   FF
6D4B 20 EE        JR    NZ,6D3B
6D4D ED 5B 00 6D  LD   DE,(6D00)
6D51 3E 16        LD   A,16
6D53 D7           RST  10
6D54 7A           LD   A,D
6D55 D7           RST  10
6D56 7B           LD   A,E
6D57 D7           RST  10
6D58 7E           LD   A,(HL)
6D59 D7           RST  10
6D5A 21 00 6D     LD   HL,6D00
6D5D 34           INC  (HL)
6D5E 3E 20        LD   A,20
6D60 BE           CP   (HL)
6D61 20 A8        JR    NZ,6D0B
6D63 36 00        LD   (HL),00
6D65 00           NOP
6D66 00           NOP
6D67 00           NOP
6D68 00           NOP
6D69 21 01 6D     LD   HL,6D01
6D6C 3E 15        LD   A,15
6D6E BE           CP   (HL)
6D6F CA 0B 6D     JP    Z,6D0B
6D72 34           INC  (HL)
6D73 C3 0B 6D     JP   6D0B
6D76 21 01 6D     LD   HL,6D01
6D79 3E 00        LD   A,00
6D7B BE           CP   (HL)
6D7C CA 0B 6D     JP    Z,6D0B
6D7F 35           DEC  (HL)
6D80 C3 0B 6D     JP   6D0B
6D83 21 00 6D     LD   HL,6D00
6D86 35           DEC  (HL)
6D87 3E FF        LD   A,FF
6D89 BE           CP   (HL)
6D8A C2 0B 6D     JP   NZ,6D0B
6D8D 36 1F        LD   (HL),1F
6D8F 00           NOP
6D90 00           NOP
6D91 C3 76 6D     JP   6D76
6D94 ED 5B 00 6D  LD   DE,(6D00)
6D98 3E 16        LD   A,16
6D9A D7           RST  10
6D9B 7A           LD   A,D
6D9C D7           RST  10
6D9D 7B           LD   A,E
6D9E D7           RST  10
6D9F 3E 20        LD   A,20
6DA1 D7           RST  10
6DA2 C3 83 6D     JP   6D83
6DA5 C9           RET
6DA6 21 00 6D     LD   HL,6D00
6DA9 C3 63 6D     JP   6D63
```


Put the Hex codes into data statements in the BASIC Hex loader. Then immediately following that data add the following Hex code into data (not the addresses in the first column though!):

```
6DAC 07 05 6D 08 83 6D 09 5A
6DB4 6D 0A 69 6D 0B 76 6D 0C
6DBC 94 6D 0D A6 6D 0F A5 6D
6DC4 FF 00 00 00 00 00 00 00
6DCC 00 00 00 00 00 00 00 00
6DD4 00 00 00 00 00 00 00 00
6DDC 00 00 00 00 00 00 00 00
6DE4 00 00 00 00 00 00 00 00
```

Now because we started using addresses 6DØØ onwards this time instead of 7ØØØ as we usually do, we shall have to change line 1Ø to:

10 LET Address = 279Ø9

Just in case you have made a mistake with the data (like governments do with statistics) then it is best to SAVE the loader and its data on tape before RUNning it. Once all that is done, you can RUN the loader and to start typewriting enter:

PRINT AT Ø,Ø;: RANDOMIZE USR 279Ø9

As we have gone through the program in detail you should know how to use the program but if not, here are the details.

SHIFT 5 to SHIFT 8	—	Cursor controls.
DELETE	—	Deletes character.
EDIT	—	Returns cursor to Ø,Ø.
GRAPHICS	—	Exits to BASIC.

Happy Typing!!

Chapter Four
LOGICAL OPERATION

We have learnt about bits and the instructions for manipulating them individually. In this chapter I shall look at some instructions that also alter bits, but alter each one at a time. These instructions are called *logical* instructions. This is because they use 'boolean' logic. At the moment you probably are clueless to what 'boolean' logic is, but it is merely a term covering logical instructions, just like 'structured language' describes a language with structure like Pascal or FORTH.

In BASIC we have the words OR, AND and NOT. We use these in IF — THEN statements for making decisions. These too are 'boolean' logic but in a much more flexible system. We cannot make up IF — THEN equivalents in machine code, but we can still produce a similar effect by using what we know about conditional jumps (such as JP Z, nnnn) along with these 'logical instructions'.

Let us have a look at the principal behind the BASIC statements OR and AND. These statements, as we know, work as:

1. x AND y: If condition x is true and condition y is true then the result is true. For instance, 1 = 1 AND 10 = 10 is true and "H" = "H" and a$ = a$ is true, whereas 2 = 3 and 1 = 1 is false and "H" = "H" and "B" = "C" is false.
2. x OR y: If condition x is true, condition y is true or if they both are true, then the result is

true. For instance, 2 = 2 OR 3 = 1 is true and a$ = a$ OR b$ = c$ is true, whereas 1 = 2 OR 5 = Ø is false and "C" = "D" OR "B" = "A" is false.

In machine code we do not work in strings or variables, but in this case 'bits' or '1's and 'Ø's. We can get BASIC to do this as well. The computer can tell us if something is true or false. Try typing:

PRINT 1 = 1

This may seem silly but the computer prints up a '1'. Why? Because by giving us a '1' it is telling us "1 = 1" is true. Now try:

PRINT 1 = Ø

We know this isn't true and so does the Spectrum. It says so by printing up a 'Ø'. So we can conclude, when using '1's and 'Ø's that:

Ø = false
1 = true

Now try typing:

LET false = Ø: LET true = 1

PRINT true AND true

Now since true = 1 the computer has printed a '1'. This is because the variable 'true' is equivalent to a true expression. Bearing in mind the variable 'false' = 'Ø' and that 'Ø' does mean 'false', see if you can predict what will happen when you type:

PRINT false AND true.

A 'Ø' is printed (meaning 'false') because not both conditions are 'true'. We can make up a table for the 'AND' instruction:

AND		RESULT
false	false	false
true	true	true
false	true	false
true	false	false

Now we can convert this to '1's and 'Ø's remembering '1' means true and 'Ø' means false:

AND:			
	Ø	Ø	Ø
	Ø	1	Ø
	1	Ø	Ø
	1	1	1

If we went through the same experiments with OR we would find:

OR:			
	Ø	Ø	Ø
	Ø	1	1
	1	Ø	1
	1	1	1

These tables correspond exactly to the operation of the machine code 'OR' and 'AND' instructions, except that they don't just work on one set of bits, they work on a set of eight, or two bytes.

If we 'AND' the Accumulator with another register, the CPU goes through each bit a time, comparing the two. The result is

worked out to be a '1' or 'Ø'. The table we have just formed tells how it decides upon a '1' or a 'Ø'. For instance:

```
AND:     1 1 Ø 1 Ø 1 1 Ø
         Ø 1 1 Ø 1 1 Ø 1
         ---------------
result   Ø 1 Ø Ø Ø 1 Ø Ø
```

Notice that the only bits in the result that are set to one are the bits with two corresponding '1's above. Have a look at another example and follow it through bit by bit understanding the result.

```
AND:     Ø Ø 1 1 1 1 Ø 1
         1 1 1 1 Ø 1 Ø Ø
         ---------------
result   Ø Ø 1 1 Ø 1 Ø Ø
```

Get it? Now see if you can manage one on your own!

```
AND:     1 Ø 1 1 Ø 1 1 1
result   1 Ø 1 Ø 1 1 Ø 1
         ---------------
               ?
```

Remembering that only '1' and '1' make a '1' you should get the result:

1 Ø 1 Ø Ø 1 Ø 1

Here is the OR table again:

OR:

Ø	Ø	Ø
1	Ø	1
Ø	1	1
1	1	1

So if we OR:

Ø 1 Ø 1 1 Ø Ø 1
Ø 1 1 1 Ø 1 Ø 1

we get:

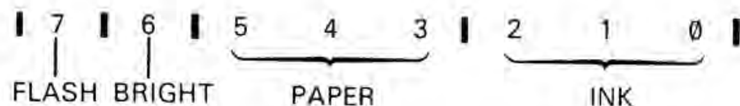
Ø 1 1 1 1 1 Ø 1

Notice that the only bits in the result that are 'Ø' have two bits above that are 'Ø'.

The AND instruction take the form 'AND r' (where r is a register). It logically 'AND's r with the Accumulator and then stores the result in the Accumulator. Similarly 'OR r' logically 'OR's r with the Accumulator. Here is a list of AND and OR instructions with their codes.

AND A	A7
AND B	AØ
AND C	A1
AND D	A2
AND E	A3
AND H	A4
AND L	A5
AND (HL)	A6
AND (IX + dis)	DD A6 dis
AND (IY + dis)	FD A6 dis
AND nn	E6 nn
OR A	B7
OR B	BØ
OR C	B1
OR D	B2
OR E	B3
OR H	B4
OR L	B5
OR (HL)	B6
OR (IX + dis)	DD B6 dis
OR (IY + dis)	ED B6 dis
OR nn	F6 nn

One use for AND is for testing certain bits of a byte. Let us take the example of an attribute byte as we did in Chapter Three.



Our task is to change all the squares with INK 0 to INK 2 (i.e. black to red). Obviously we will need a loop to process each attribute byte. But the crucial code will lie within the loop. Firstly, we will want to get hold of the three least significant bits (that it, bits 0,1 and 2) for testing. This where AND comes in. Look at this:

```
ATTRIBUTE    1 1 0 1 0 1 1 1
07 (ANDed)   0 0 0 0 0 1 1 1
             ---------------
RESULT       0 0 0 0 0 1 1 1
```

You will notice only bits 0,1 and 2 of the attribute appear in the result. This is because the byte it has ANDed with contained 07 Hex (meaning bits 0,1 and 2 were set). Have a look at another example:

```
ATTRIBUTE    1 1 1 1 0 1 1 0
07 (ANDed)   0 0 0 0 0 1 1 1
             ---------------
RESULT       0 0 0 0 0 1 1 0
```

The result is 06 Hex which is the what bits 0,1 and 2 of the attribute come to. So as we can see when we AND a number with a byte, the bits that are zero 'mask' any bits that are one in the byte. This process is not surprisingly called 'masking', effectively describing the way we can hide certain bits of a byte.

Back to our task, if we AND 07 Hex with the attribute, we are left with only the INK colour, the other bits being reset by the AND instruction. We can test this 'INK' number using the flags and see if it is to be changed.

```
7000 21 00 58      LD    HL,5800
7003 7E            LD    A,(HL)
7004 E6 07         AND   07
```



```
7006 20 02         JR    NZ,700A
7008 CB CF         SET   1,A
700A 77            LD    (HL),A
700B 23            INC   HL
700C 7C            LD    A,H
700D FE 5B         CP    5B
700F 20 F2         JR    NZ,7003
7011 C9            RET
```

If we look at the program step by step first we see HL is loaded with the address of the beginning of the attribute file. Then the Accumulator is loaded with the attribute pointed to by HL. Then we AND 07. This zeros all the bits except those for the 'INK' colour. If the byte is not zero then a Jump Relative is made to 700A, because if the byte is not zero then the INK cannot be black. If the INK is black the bit 1 of the Accumulator is set to one making the INK red.

Let's check the program out :

Make sure the BASIC loader is ready. Then change line 80 for the codes in the program listing.

Then, type RUN followed by:

INK 2:LIST:RANDOMIZE USR address

You should find (providing PAPER is not red) that your black listing changes instantly to red!

To prove it only changes black INK, type:

INK 0:LIST;INK 2: LIST:RANDOMIZE USR Address

A common use for OR is setting particular bits to one in a byte. For instance we have an attribute byte and we want to set bits six and seven to one to make the character BRIGHT and FLASH. We can do this by making a byte with only bits six and seven set then ORing this with our attribute. If we put this method inside a loop we can make the whole screen flash and be bright.

```
ATTRIBUTE Ø Ø 1 Ø 1 Ø Ø 1
EØ(ORred)  1 1 Ø Ø Ø Ø Ø Ø
           ---------------
           1 1 1 Ø 1 Ø Ø 1
```

Note that the only bits changed are bits six and seven which is just what we wanted. So now let us now examine the program to make the screen flash and be bright.

```
7000  21 00 58    LD    HL,5800
7003  7E          LD    A,(HL)
7004  F6 E0       OR    E0
7006  77          LD    (HL),A
7007  23          INC   HL
7008  7C          LD    A,H
7009  FE 5B       CP    5B
700B  20 F6       JR     NZ,7003
700D  C9          RET
```

The operation is quite simple, HL is pointed to the beginning of the attribute file, then the attribute is collected in A. Bits six and seven are set (by OR EØ) and the attribute is replaced. Finally, HL is incremented, tested to see if it has reached the end and if not, processes the next attribute.

Now let's test the program. Use the following line 8Ø in the loader:

80 DATA "21ØØ587EF6EØ77237CFE5B2ØF6C9"

Next, after typing RUN, type:

LIST:RANDOMIZE USR Address

You will notice the instant change from normal to BRIGHT and FLASHing.

THINGS TO TRY

1. Try other values than EØ with the OR instruction and work out the reasons for their effects.
2. Try using AND instead of OR with different values. Work out the reasons for their effects.

3. Instead of using OR to set bits six and seven, try modifying the program to use SET which we looked at in the last chapter.

EXCLUSIVE!

There is one more logical instruction which we have to look at. It is called Exclusive-OR and is a close relation of OR. It is used by the Spectrum when OVER is selected. The following should make sense to you.

```
PAPER + PAPER = PAPER
INK   + PAPER = INK
PAPER + INK   = INK
INK   + INK   = PAPER
```

That is what the results are when OVER is set. It uses XOR when plotting or printing on the screen. If we think of INK being 'one' and PAPER being 'zero' we notice:

Ø	Ø	Ø
1	Ø	1
Ø	1	1
1	1	Ø

This is the logic table for XOR, the only difference from OR being at the bottom where two 'ones' make a zero instead of a one. Here is a list of the op-codes for all the XOR instructions:

XOR A	AF
XOR B	A8
XOR C	A9
XOR D	AA
XOR E	AB
XOR H	AC
XOR L	AD
XOR (HL)	AE
XOR (IX + dis)	DD AE dis
XOR (IY + dis)	FD AE dis
XOR nn	EE nn

Try using XOR in the last program and see its effects. Also make sure you try it when something is already FLASHing or BRIGHT on the screen. If you do not understand its effects, work out these 'sums' and see if you then understand. If in difficulty refer to the table given below:

XOR	Ø	Ø	Ø
	Ø	1	1
	1	Ø	1
	1	1	1

1. 1 XOR Ø = ?

2. 1 XOR 1 = ?

```
3.    1 Ø 1 1
XOR   Ø 1 1 Ø
      -------
         ?
```

```
4.    1 1 Ø Ø
      1 Ø 1 Ø
      -------
         ?
```

```
5.    1 Ø 1 1 1 Ø 1 1
XOR   Ø 1 1 Ø 1 Ø 1 Ø
      ---------------
            ?
```

The last principle we look at in this chapter is a method for 'double byte zero testing'. In Chapter Two we looked at looping using single byte registers. When we wish to do an operation more than 256 (decimal) times we need to use a double byte register as our counter. One way we could do that is by using CP.

```
        LD HL,1ØØØ
PROG          :
              :
        LD A,H
        CP ØØ
        JR NZ,PROG
```



```
        LD A,L
        CP ØØ
        JR NZ,PROG
        RET
```

As you should notice H and L are tested separately to check if they are both zero. The 'CP nn' instruction uses two bytes, but we can save space here by using AND. If we use 'AND A' the Accumulator is not altered but the important thing that happens is that the flags are set to represent A. Let us see why A is not altered. Remember 'AND A' simply ANDs the Accumulator with itself so the two numbers will be the same.

```
1 1 Ø Ø 1 Ø 1 Ø
1 1 Ø Ø 1 Ø 1 Ø
---------------
1 1 Ø Ø 1 Ø 1 Ø
```

Bear in mind the AND table:

Ø	Ø	Ø
Ø	1	Ø
1	Ø	Ø
1	1	1

So as we can see, A is left unaltered and the flags are set. This allows us to make a valid decision, so we can replace the CPØØs with AND A.

```
        LD HL,1ØØØ
PROG          :
        INC HL
        LD A,H
        AND A
        JR NZ,PROG
        LD A,L
        AND A
        JR NZ,PROG
        RET
```

But as yet we have not made our most significant discovery. We can compress this zero test even further by using the OR instruction. If we load A with H and OR L, any bit set to one will show through and A will not be zero. Whereas if all the bits in HL are zero, A will also turn out to be zero. Here is the modified loop:

```
PROG      LD HL,1000       21 00 10
START          :
          INC:HL           23
          LD A,H           7C
          OR L             B5
          JR NZ START      20?
          RET              C9
```

Let us try it out by using it to print 200(Hex) asterisks on the screen.

```
7000 21 00 02    LD   HL,0200
7003 3E 2A       LD   A,2A
7005 D7          RST  10
7006 23          INC  HL
7007 7C          LD   A,H
7008 B5          OR   L
7009 20 FA       JR   NZ,7005
700B C9          RET
```

Use the Hex on the right in the DATA statement of the loader (remember to use quotes). Then type:

RUN and after that:

PRINT AT 0,0;0: RANDOMIZE USR Address

And exactly 200 (Hex) or 512 (Decimal) asterisks are printed.

Chapter Five
ROTATING

Rotating is what this chapter is all about. What most of these instructions basically do is move the bits in a register to the left or right, with various combinations involving the Carry Flag. They are best represented graphically (see appendix 'D' for diagrams), but here I have given a written explanation for each as well as the instruction codes for each type.

Rotate instructions have no obvious uses, but can be used mainly for applications where we need to build up a byte, bit by bit, or to test a byte bit by bit. Rotate instructions can also be used for multiplication by powers of two (2,4,6,8, etc) and there are a couple of instructions for use with BCD (Binary Coded Decimal) which I shall explain later on. Most of the instructions have a variation which operates specifically for the Accumulator as well as instruction for all the usual eight bit registers including the Accumulator. The latter are two byte op-codes and so for the Accumulator there are usually two duplicate instructions. The only difference being that one is two bytes long and the other a single byte. It is obvious that in situations where the Accumulator is to be rotated then the single byte version will be chosen in the interests of saving memory.

RLC r (r = A,B,C,D,E,H,L,(HL),(IX + dis),(IY + dis))

This rotates register or byte 'r' (see above) to the left; the most significant bit (bit 7) is brought round to the least significant bit. The carry bit or flag is set to register the contents of bit 7 at the start of the operation. This feature allows the programmer

to determine what bit seven was prior to the shifting operation by testing the Carry Flag after the rotate instruction.

RLCA	Ø7	RLC H	CB Ø4
1		RLC L	CB Ø5
RLC A	CB Ø7	RLC (HL)	CB Ø6
RLC B	CB ØØ	RLC (IX + dis)	DD CB dis Ø6
RLC C	CB Ø1	RLC (IY + dis)	FD CB dis Ø6
RLC D	CB Ø2		
RLC E	CB Ø3		

RL r

This instruction rotates register or byte 'r' to the left through the carry bit. Bit seven is moved into the carry bit and the old contents of the carry bit are moved into bit zero.

RLA	17	RL E	CB 13
		RL H	CB 14
RLA	CB 17	RL L	CB 15
RL B	CB 1Ø	RL (HL)	CB 16
RL C	CB 11	RL (IX + dis)	DD CB dis 16
RL D	CB 12	RL (IY + dis)	FD CB dis 16

RRC r

This operation is similar to the RLC r instruction except that rotation is done to the right. This time bit zero will be placed in bit seven or 'r' and also into the Carry Flag.

RRCA	ØF	RRC E	CB ØB
		RRC H	CB ØC
RRC A	CB ØF	RRC L	CB ØD
RRC B	CB Ø8	RRC (HL)	CB ØE
RRC C	CB Ø9	RRC (IX + dis)	DD CB dis ØE
RRC D	CB ØA	RRC (IY + dis)	FD CB dis ØE

RR r

This is the reverse operation of RL 'r'. Here bit zero of 'r' is brought around to the Carry Flag and the initial value of the Carry Flag is put in bit seven of 'r'.

RRA	1F	RRE	CB 1B
		RR H	CB 1C
RR A	CB 1F	RR L	CB 1D
RR B	CB 18	RR (HL)	CB 1E
RR C	CB 19	RR (IX + dis)	DD CB dis 1E
RR D	CB 1A	RR (IY + dis)	FD CB dis 1E

Sometimes it is preferable to 'shift' a register rather than 'rotate' it. Instead of 'rotating' bit zero around to bit seven or *vice versa*, bit seven can be cleared or left unchanged as its original value is shifted over to the next bit position. These bit shifting instructions do not have a duplicate one byte instruction for the Accumulator. They are all made up of two byte codes starting with CB (Hex).

SLA r

This instruction simply shifts all one bits to the left. Bit seven is moved into the Carry Flag and a zero is put into bit zero.

SLA A	CB 27	SLA H	CB 24
SLA B	CB 2Ø	SLA L	CB 25
SLA C	CB 21	SLA (HL)	CB 26
SLA D	CB 22	SLA (IX + dis)	DD CB dis 26
SLA E	CB 23	SLA (IY + dis)	FD CB dis 26

SRA r

This instruction shifts all the bits of the register or byte to the right. Bit zero is moved into the carry bit. Bit seven is copied into bit six but is left unchanged.

SRA A	CB 2F	SRA H	CB 2C
SRA B	CB 28	SRA L	CB 2D
SRA C	CB 29	SRA (HL)	CB 2E
SRA D	CB 2A	SRA (IX + dis)	DD CB dis 2E
SRA E	CB 2B	SRA (IY + dis)	FD CB dis 2E

SRL r

This instruction is a right-shifting operation but unusually does not have a left shifting equivalent, eg 'SLL'. There is even a gap

in the codes for rotates where SLL should fit which indicates that this function is a part of the Z80 chip that Zilog, the manufacturers, could not get to work. SRL shifts the bits to the right putting a zero into bit seven and moving bit zero into the Carry Flag.

SRL A	CB 3F	SRL H	CB 3C
SRL B	CB 38	SRL L	CB 3D
SRL C	CB 39	SRL (HL)	CB 3E
SRL D	CB 3A	SRL (IX + dis)	DD CB dis 3E
SRL E	CB 3B	SRL (IY + dis)	FD CB dis 3E

DECIMALS

Before we look at RLD and RRD, we must understand BCD, Binary Coded Decimal. BCD is an important part of computers. It is a sort of 'interface' between humans and binary, trying to link the two. Most seven-sequent LED displays take the digits in the form of BCD digits. A single BCD digit is quite straightforward. It is four bits whose combined value does not exceed nine (otherwise it wouldn't be decimal).

When we want to store numbers in a decimal form we can use BCD. For instance, if we wish to store 99 as decimal 99 it would have to be converted to Hex then converted back if, for instance, we wished to display it as decimal on the screen. This is how decimal 56 would be represented:

Higher four bits 0 1 0 1   0 1 1 0 lower four bits
5   6

If we were to think of that 56 as Hex and then convert it (look 56 Hex up in the appendix of the manual) we would find it would be 86 decimal, but we are thinking of it as 56 decimal not Hex.

If an ordinary Hex number has been loaded into the Accumulator it can be converted into BCD with a DAA instruction — Decimal Adjust Accumulator.

DAA 27

So let us imagine we have just loaded the Accumulator with 43 Hex. Now if we check in the manual we find that 43 Hex is 67 in decimal. If we now perform a DAA instruction the Accumulator changes the 43 to 67, a BCD number.

DAA can only change decimals up to 99 because we only have two digits in a register.

If we add two BCD numbers together, then the result will need to be adjusted and DAA can be used to correct the result to BCD. Similarly if two BCD numbers are subtracted then DAA will correct the result to BCD.

So now let us see how the RLD and RRD instructions work.

RLD
7 4 | 3 0   7 4 | 3 0
Accumulator   contents of (HL)

The RLD instructions rotates the three blocks of four bits in the lower half of the Accumulator and both halves of the memory location pointed to by HL all to the left. These four bit blocks we imagine are BCD digits but another less well known name for half a byte is a 'nibble' — no, seriously, that is quite true and technical buffs will confirm that fact. (While we're on the subject of strange technical terms do you know that Commodore's manuals refer to a 1/50th of a second technically as a 'jiffy'?) Back to the subject in question and here is a diagram of RRD:

RRD
7 4 | 3 0   7 4 | 3 0
Accumulator   contents of (HL)

The RRD instruction operates in the opposite direction to RLD, moving the most significant BCD digit of (HL) into the least significant half, moving this lower half into the lower half

of the Accumulator and the lower half of the Accumulator moves to the upper half of (HL).

So now we draw Chapter Five to an end and in the next chapter things are going to sound good, ships will be shipping and ports will be alive with the sound of music. Confused? Read on...

Chapter Six
PORTS

Computers must be able to communicate with the outside world. That usually includes you, me, printers, tape recorders, keyboards and displays in the ZX computers world. Without the ability to communicate a computer would be useless, a bit like buying a video recorder without having a TV. We divide the communication into two separate classes, input and output. Input being information received by the computer such as that from the keyboard on which your merry fingers dance! And conversely, output being information sent by the computer such as the television display. To help it communicate with various things the usual computer system, be it in a programmable washing machine or the ZX Spectrum, has what are technically named *ports*. This name follows a sort of logic — if you can imagine a ship carrying information into the port for the computer to collect and then taking some information from the computer and shipping it elsewhere. Obviously if you look inside your Spectrum you won't see ships whizzing around inside but the principle is much the same. If you are a hardware freak (there are tablets available) then you can use some of the spare ports on the Spectrum — but you must read Chapter 23 of the Sinclair manual to save me repeating Steven Vickers' wise and wonderful words.

But ports are not just for electronics buffs. We needn't buy extra equipment to see what they do since we have some useful ports on our doorstep — the ports that the Spectrum uses to communicate with the outside world. Hopefully you will know how to access ports from BASIC, but we will start there anyway since it makes a logical introduction to the

machine code side of the business. First let me remind you about ports. Normally there can be up to 256 of these ports on a Z80 system but due to a quirk of the Zilog Z80 it can use up to 65536 ports sometimes, but more about that later in the chapter.

The Spectrum only uses a few of these ports and the one we are going to deal with is the port for the BORDER colour and for making sound.

Type in this little program and RUN it.

```
   10 FOR i=0 TO 7: OUT 254,i: PA
USE 3: NEXT i: GO TO 10
```

Now as you can see it changes the BORDER through all the eight different colours on the Spectrum. The first point to note is that we are using port 254. This port is associated with the BORDER and with the speaker, EAR and MIC socket. The OUT instruction in BASIC takes the form:

OUT port, data

In our example, the port is number 254 and the data is the BORDER colour. The port itself is split up into various parts. Each part is controlled by a different bit or bits of our eight bit data byte. This is what the bits do in port 254:

Bit		Function
0	}	BORDER colour
1		
2		
3	–	MIC socket
4	–	Loudspeaker
5	}	Not used
6		
7		

So now we know what does what, we should be able to make some sort of sound. Firstly we must realize that a one at bit four makes the speaker go out and a zero makes the speaker go in.

So if we alternate between a one and zero we should get a noise. Type in the following program which does this.

```
   10 OUT 254,2↑4*1: OUT 254,2↑4*
0: GO TO 10
```

There are two OUT commands, one to turn the speaker on and one to turn it off. If you look at the listing you will notice that the first data is 2↑4 * 1. The two is because we are working in binary, base two. The four is to show it is the data for bit four, and the one is to show it that we want the speaker turned on. Conveniently, in the second OUT command, there is a zero to show we want the speaker turned off. As the program stands it is very slow and the sound consists of feeble clicks. To speed this up we shall save the computer having to work this out each time by replacing 2↑4 * 1 and 2↑4 * 0 with their actual values, 16 and zero respectively so alter the program so it looks like this:

```
   10 OUT 254,16: OUT 254,0: GO T
O 10
```

But even speeded up, that buzz is still feeble and a long shot from the zaps, pows and booms of the local 'Pacman' or 'Defender' machine. So to get the pace a little hotter we shall transfer our efforts to machine code now that we have a foundation knowledge of ports.

In Z80 machine language we have instructions that relate directly to the BASIC ones. They even have the same names, IN and OUT. For instance, if we wish to send the data held in the Accumulator to port 254, we can simply says:

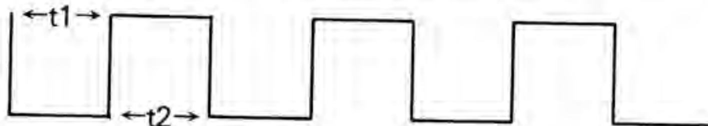
OUT (FE), A

Unlike BASIC we have brackets round the port number FE (the Hex equivalent of decimal 254); this is because we are thinking of the port number as an address and so to follow standard Z80 format, we use brackets to show that the information, in this case A, is being sent to a location, a port address. So now let us translate our BASIC program into machine code. Now

because machine code is so fast we shall have to put delays in between the OUT instructions, otherwise no sooner than the speaker manages to start moving one way it is told by the computer to move back again, and the net result is that it dithers in between not making a sound at all. For these delays we shall use the 'B' register as it is the most convenient. We shall set up C with the number of clicks to be made. The real name for these clicks is 'cycles'. Here is the program. To use it put the Hex codes into the BASIC loader and follow the usual procedure. Those two strange instructions at the beginning and end are not mistakes, they are two instructions that are explained in the next chapter. Basically what they do is to ensure that the computer devotes all its energy to the BEEPing and not rushing off elsewhere every 1/50th of a second to look at things like the keyboard.

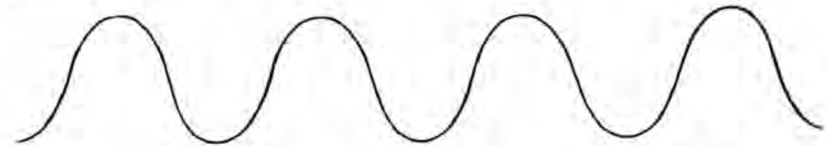
```
7000 F3          DI
7001 0E FF       LD    C,FF
7003 3E 00       LD    A,00
7005 D3 FE       OUT    (FE),A
7007 06 C0       LD    B,C0
7009 10 FE       DJNZ  7009
700B 3E 18       LD    A,18
700D D3 FE       OUT    (FE),A
700F 06 C0       LD    B,C0
7011 10 FE       DJNZ  7011
7013 0D          DEC   C
7014 20 ED       JR     NZ,7003
7016 FB          EI
7017 C9          RET
```

In the program we have C set up with FF so it produces 255 (decimal) cycles. This relates to the length of the note. The two loops using B decide on how long the speaker is left on and how long it is left off. By adjusting these values for B we can adjust the pitch. This is how it looks graphically:



The length of t1 is decided by the first B loop, and the length of t2 is decided by the second B loop. The number of cycles is

decided by C. As you can see the cycles are square and so the name we give them is 'square waves'. Different sounds have different shaped waves. A flute, for instance, tends to produce a fine wave like this:



This has a much rounder, smoother sound. Unfortunately on the Spectrum, we are totally restricted to square waves unless we buy a musical add-on so the sound we make is quite harsh and not very musical. But even with this limitation some good effects can be generated.

One final point concerning our last program, you will have noticed that the BORDER turns black. This is because we have let bits zero, one and 2 remain zero, resulting in a BORDER colour zero, hence black!

Our program as it stands is fine, but really quite unnecessary as there is a routine in the ROM that can do all this for us and is quite convenient to use. It resides at Ø3B5 and to use it we must provide it with the number of cycles in DE and the length of the cycles, or the pitch in HL. For instance, try this program:

```
7000 21 00 05    LD    HL,0500
7003 11 00 02    LD    DE,0200
7006 CD B5 03    CALL  03B5
7009 C9          RET
```

There we are! A much quicker way of making a simple BEEP! But we are still looking for something more than a simple BEEP since we can do this in BASIC, so let's now see how we can make some more interesting sounds.

The following program uses the ROM routine to produce a rising BEEP:

```
7000 21 00 10    LD    HL,1000
7003 11 20 00    LD    DE,0020
```

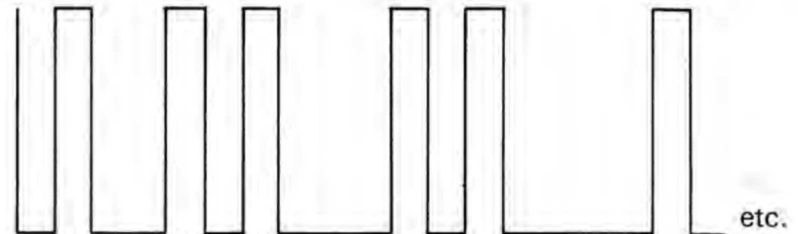
```
7006 ED 52        SBC  HL,DE
7008 11 01 00     LD   DE,0001
700B E5           PUSH HL
700C CD B5 03     CALL 03B5
700F E1           POP  HL
7010 7C           LD   A,H
7011 A7           AND  A
7012 20 EF        JR    NZ,7003
7014 C9           RET
```

The program starts by loading HL with 1000 (Hex); this is the starting value for the BEEP. Next DE is loaded with 0020 (Hex) which is how much HL will decrease by each time; we'll call this the 'step'. Next DE is subtracted from HL to reduce HL by the 'step'. DE is then loaded with a value to determine how long the computer spends on each tone before moving onto a higher one. Before we call the ROM routine, HL is pushed on to the stack to save the value. After the ROM has done its bit we pop HL then check the H register to see if it is zero. If it is, then the program finishes otherwise we move onto the next tone. All these tones together result in a rising tone which could be used for laser shots. This routine is quite versatile, so try using other values for the step, initial value of HL and length of tone. By experimenting a variety of sounds can be obtained, from short zaps to long rising tones. If we try these effects in a loop in BASIC they can be very effective.

Now how about ship's engines? Well that sound has a special name, 'noise'. 'Noise' may not sound very technical but it is the term for sounds like the hiss on a tape recorder which is high pitched, compared to the low rumble of 'space ship engines' which is low pitched. People into synthesizers will know about the different types of noise pink, white, heavy metal, etc, but here we shall keep things simple by sticking to a good old 'noise'.

Noise is really a collection of randomly spaced pulses and so we can simulate this quite easily. We first need a source of random data and instead of writing some complicated random number generator, we shall actually use all the codes in the ROM as random data. This will mean we will repeat ourselves

when we run out of data but since noise is not the 'catchiest' of sounds it is unlikely you will notice the repeating. A ships engine is a low pitched noise so we shall have to space the pulses out with a B loop. We shall use HL to point to which byte we are up to in the ROM and DE to count the number of cycles required. This is what our noise would look like graphically:



Here is our rumblin' program:

```
7000 F3           DI
7001 21 00 00     LD   HL,0000
7004 11 00 20     LD   DE,2000
7007 7E           LD   A,(HL)
7008 D3 FE        OUT   (FE),A
700A 06 FF        LD   B,FF
700C 10 FE        DJNZ 700C
700E 1B           DEC  DE
700F 23           INC  HL
7010 7B           LD   A,E
7011 B2           OR   D
7012 20 F3        JR    NZ,7007
7014 FB           EI
7015 C9           RET
```

Try the program out using the BASIC loader. You will notice that the program also makes all sorts of colours on the BORDER which can be quite effective. How would you remove this effect? *Clue: use AND.*

Now how about explosions? Well, the technique is quite the same, we use the same noise but we make it go from high to low pitch. To do this we lengthen the delay between each

noise to producing noise that decays in pitch. Here is a program to do this:

```
7000 21 00 00      LD    HL,0000
7003 0E 00         LD    C,00
7005 16 20         LD    D,20
7007 7E            LD    A,(HL)
7008 E6 18         AND   18
700A D3 FE         OUT    (FE),A
700C 41            LD    B,C
700D 10 FE         DJNZ  700D
700F 23            INC   HL
7010 15            DEC   D
7011 20 F4         JR     NZ,7007
7013 0C            INC   C
7014 20 EF         JR     NZ,7005
7016 C9            RET
```

In this program we use C as a counter which makes the B loop longer each time. We can slow the whole effect down by increasing the initial value of D. If you wish to speed it up you can reduce the initial value of D. With a short length of explosion, called from a basic loop randomly, it can give the effect of thunder and lightning. If you change the INC C to a DEC C (code 0D) then you will be making a rising noise sound which is very effective in a game for when something appears on the screen. (Arcade aces will know this from the 'Defender' machine.) In this program is also the solution to my last question, how to get rid of a multi-coloured BORDER. We have used AND 18 to ensure only bits three and four are set.

Well that is the end of Chapter Six and I hope that has given you some ideas for making sound effects and also an idea of how computers communicate. A full list of the various IN and OUT instructions are given in Appendix D along with definitions. If you would like to experiment with ports further than the existing ZX Spectrum allows, you'll need an add-on port. There are various ones available and it doesn't really matter which one you buy as long as you realise that they do vary in type. Some have separate input and output lines while others use more complicated chips and are programmable. In general, for home use, the more I/0 lines it has the better it will be. Also, make sure you can use the IN and OUT instructions in

BASIC because if you can then you will certainly be able to use machine code equivalents.

Chapter Seven
MAY I INTERRUPT?

In this chapter we shall look at interrupts before we go on to look at the ROM. Interrupts are of little direct use to you on the Spectrum but they do help us understand a little about the machine's workings.

At regular intervals in a computer there are certain jobs the ROM must do. These jobs vary from machine to machine but a basic need is to update some sort of timer or counter. A counter allows for precise timing for things such as a clock. The way the computer does these is by use of interrupts.

An interrupt is a signal sent to the CPU from an external piece of equipment such as the logic chip. This tells the CPU to stop what it is doing and do something else. When it finishes doing the job or more technically, servicing the interrupt, it carries on from where it was interrupted.

There are two types of interrupt. A maskable interrupt and a non-maskable interrupt. A maskable interrupt is one that the CPU can be set to ignore when desired. In the Sinclair ZX Spectrum a maskable interrupt is generated every 5Øth of a second. This is done by the logic chip sending a pulse to a pin called INT on the Z8Ø–A CPU.

When this is received the subroutine at ØØ38 Hex is called. This is the program:

Interrupt routine

```
003A 2A 78 5C   LD   HL,(5C78)
003D 23         INC  HL
003E 22 78 5C   LD   (5C78),HL
0041 7C         LD   A,H
0042 B5         OR   L
0043 20 03      JR    NZ,0048
0045 FD 34 40   INC  (IY+40)
0048 C5         PUSH BC
0049 D5         PUSH DE
004A CD BF 02   CALL 02BF
004D D1         POP  DE
004E C1         POP  BC
004F E1         POP  HL
0050 F1         POP  AF
0051 FB         EI
0052 C9         RET
```

Firstly the AF and HL registers are put on the stack. This is so that any values held in them are not corrupted because remember, a routine has just been interrupted and when we have finished 'servicing' the interrupt that routine will be continued as if nothing had happened. The next move is to load HL with the lower two bytes of FRAMES. This double byte value is then incremented and a test is made to see if it has reached zero. This is done by the 'OR' technique described in Chapter Four. By loading A with H and then ORing L if HL is zero so will be the Accumulator. If HL is not zero then the instruction to increment the highest byte of FRAMES is skipped (INC 1Y + 40). 'INC(1Y + 40)' is used because IY always contains a base address of 5C3A. This is so that when the ROM has to make references to the system variables it can use 1Y and displacement rather than the HL register pair. For instance, if we wanted to use HL instead this would be necessary:

```
PUSH HL
LD HL, 5C3A
INC (HL)
POP HL
```


Instead of simply:

```
INC (IY+40)
```

The next thing to happen is that the other registers are stored on the stack, before a call to the subroutine which builds up a keyboard scan in the system variable K–SCAN and then determines the last key pressed and stores it in LAST K. The registers are then restored to their original values by POPping them off the stack.

Finally, an EI instruction is encountered before returning.

The EI instruction is the partner of DI. We have not yet met these instructions so let us have a closer look.

Their function is quite simple. I have already mentioned the fact that there are two types of interrupt: maskable and non-maskable. The Spectrum uses a maskable interrupt for its timer up-dating and keyboard scanning. When a maskable interrupt is made, the CPU automatically disables it, eg ignores any further maskable interrupts. This is so an interrupt is not interrupted! If you see what I mean. Almost as if it is hanging a 'Do Not Disturb' sign on the door. Once it has finished servicing the interrupt it effectively 'takes the sign down' by using the EI instruction. EI stands for enable interrupt and means just what it says. This is used because otherwise interrupts would be ignored for evermore. And without interrupts the ZX Spectrum will never know if a key has been pressed.

There are other times when the ROM will not want to be disturbed. When it is doing a tape operation, for example. If the logic chip 'interrupts' in the middle of loading a byte, for example, by the time the timer has been updated and the keyboard scanned that byte may have passed by on the tape recorder.

So, to prevent disturbances we can use the 'DI' instruction

meaning disable interrupt. As we have described this simply tells the CPU to ignore maskable interrupts until further notice, eg by way of EI. Incidentally whenever the CPU is reset the maskable interrupt is *always* enabled.

INTERRUPT MODES

What I haven't told you is that the CPU doesn't *have* to go to the subroutines at 0038 Hex when it is interrupted. It goes there because it is in 'interrupt mode one'. So what are these mysterious interrupt modes? The first thing to say is that to a Spectrum owner, they are quite irrelevant because the other modes are unusable unless you are a hardware freak. The reason that I am mentioning them is because one mode (namely mode two) uses the 'I' register whose use is probably a mystery to you.

I am afraid that at this point I am forced to step into the daunting world of hardware to explain the modes so please try to bear with me and I will keep everything as simple as possible.

INTERRUPT MODE ZERO

Interrupts are often used as the word 'Hello' when computers are talking to each other. Obviously they do not actually speak, they send messages either serially (in the same way a program is stored on tape) or in parallel, eg a whole byte goes in at once. Before they talk the message sender needs to draw the other one's attention. This is done by way of interrupt (unfriendly computers such as the BBC and Spectrum just ignore each other by using DI!) But what if we also have an interrupt every 50th of a second for something such as the timer? What we need to do is have the computer announce itself by saying 'Hello, I'm the other computer, can we talk' so the CPU doesn't mistake it for the timer interrupt. One way of doing this is by using interrupt mode zero.

What happens is that once an interrupt has been made, the interrupting device has enough time to 'speak' one byte to the CPU. When the CPU gets that byte it will quite simply execute

it like any other instruction. Bearing in mind RST instructions are one byte long, the computer can 'say' RST 28, for instance. So at 0028 Hex there could be a routine to receive the other computer's message. On the other hand, if the logic chip interrupts, meaning that it is time to increment the timer, it can say 'RST 38' so that the CPU goes to 0038 Hex to update the timer. By using these RST directives in mode zero a 'device' can tell the CPU what routine to do.

INTERRUPT MODE ONE

This is the interrupt mode that the computer is set to when it is turned on. When an interrupt is generated, a call is made to the subroutine at 0038 Hex. The Spectrum uses this interrupt mode as we have described.

INTERRUPT MODE TWO

This is where the I register becomes involved. Its full title is the 'Interrupt vector' or IV register. 'Vector' is a technical term for something that points somewhere, in this case, into a table. For each device that can interrupt there is an appropriate subroutine address stored as part of a table in memory.

INTERRUPT TABLE	8000	10	00
	8002	00	70
	:	:	:
	:	:	:

To find which subroutine to call, a pointer is formed when the CPU is interrupted. This pointer will point to two data bytes in the table. This pointer is formed by taking the I register as the most significant byte and the byte that the 'device' sends to the CPU as the least significant byte. So if we use the example table, for instance, if the I register contains 80 and the device sends a zero the subroutine at 0010 will be called. Let's take it in stages to make it simpler:

1. An interrupt is generated.
2. The CPU collects a byte from the device.

3. Using the I register as the high byte and the collected byte as the low byte we have a vector (pointer).
4. At the location pointed to by this vector we have the low byte followed by the high byte of the subroutine address.
5. These two bytes are used as the subroutine CALL.

To change the interrupt mode we can use the instructions IMØ, IM1 and IM2. When the machine is turned on, the ROM does a routine to initialise everything and before the interrupt is enabled, an IM1 instruction is done. Here are the codes for the 'interrupt mode set' instructions:

IMØ	ED 46
IM1	ED 56
IM2	ED 5E

NON-MASKABLE INTERRUPTS

As I have mentioned there are two types of interrupt. The NMI (non-maskable interrupt) is the one we shall now look at. By being non-maskable the only time the computer will ignore it is while another interrupt is being serviced. When NMI is generated the computer goes to the subroutine at ØØ66 Hex, no ifs and buts, no modes, just simply to ØØ66.

Unfortunately the ØØ66 NMI routine appears to have a bug. Here is the listing.

```
0066 F5          PUSH AF
0067 E5          PUSH HL
0068 2A B0 5C    LD   HL,(5CB0)
006B 7C          LD   A,H
006C B5          OR   L
006D 20 01       JR    NZ,0070
006F E9          JP     (HL)
0070 E1          POP  HL
0071 F1          POP  AF
0072 ED 45       RETN
```

The first thing to point out is that the routine is ended with 'RETN'. This instruction is used instead of 'RET' to end a NMI-service routine. The more important point is the apparent bug.

If we study the program, we notice that first the AF and HL pairs are stacked. Then the contents of address 5CBØ and 5CB1 are fetched into HL. If you bother to look up these addresses you will find they are the two free bytes in the system variables. The plot thickens...an 'OR' zero test is performed on HL using the method I described in Chapter Four. The next instruction should have been JR Z, ØØ7Ø but instead is JR NZ. The problem being that if it had been a 'JR Z' then if those bytes were not zero then a jump to HL would have been made. In this way one could have a key wired to the NMI so that providing those bytes were loaded with the address of a suitable routine, 'an escape from crash' key could be made.

But as it stands it will only make a jump if the bytes are zero which acts as an almost useless total reset function. If the 'mistake' had not been made the gateway would be open to programmable function keys and many other applications. Oh well, better luck next time....

Appendices

A: Hex, decimal and ASCII code look up table plus Z80 mnemonics.

B: Z80 mnemonics and flags.

C: System variables and explanation.

D: Z80 Mnemonics and brief definitions.

Appendix A
Hex, decimal & ASCII characters. Z8Ø mnemonics.

APPENDIX A

This appendix serves as an all-in-one look up table including the Hex numbers ØØ to FF, their decimal equivalents, the ZX Spectrum ASCII characters, and last, but not least, the Z8Ø machine language mnemonics. They are listed under three sections: no prefix; prefixed by CB; prefixed by ED. Those instructions involving the index registers IX and IY have Hex codes that are simply the equivalent involving HL prefixed by DD for IX and FD for IY.

HEX	DECIMAL	CHARACTER	Z8Ø-NO PREFIX	PREFIX CB	PREFIX ED
ØØ	Ø	not used	NOP	RLC B	
Ø1	1	not used	LD BC,nn	RLC C	
Ø2	2	not used	LD (BC),A	RLC D	
Ø3	3	not used	INC BC	RLC E	
Ø4	4	not used	INC B	RLC H	
Ø5	5	not used	DEC B	RLC L	
Ø6	6	**PRINT** comma	LD B,n	RLC (HL)	
Ø7	7	**EDIT**	RLCA	RLC A	
Ø8	8	cursor left	EX AF,AF'	RRC B	
Ø9	9	cursor right	ADD HL,BC	RRC C	
ØA	1Ø	cursor down	LD A,(BC)	RRC D	
ØB	11	cursor up	DEC BC	RRC E	
ØC	12	**DELETE**	INC C	RRC H	
ØD	13	**ENTER**	DEC C	RRC L	
ØE	14	number	LD C,n	RRC (HL)	
ØF	15	not used	RRCA	RRC A	
1Ø	16	**INK** control	DJNZ dis	RL B	

HEX	DECIMAL	CHARACTER	Z80-NO PREFIX	PREFIX CB	PREFIX ED
11	17	**PAPER** control	LD DE,nn	RL C	
12	18	**FLASH** control	LD (DE),A	RL D	
13	19	**BRIGHT** control	INC DE	RL E	
14	20	**INVERSE** control	INC D	RL H	
15	21	**OVER** control	DEC D	RL L	
16	22	**AT** control	LD D,n	RL (HL)	
17	23	**TAB** control	RLA	RL A	
18	24	not used	JR dis	RR B	
19	25	not used	ADD HL,DE	RR C	
1A	26	not used	LD A,(DE)	RR D	
1B	27	not used	DEC DE	RR E	
1C	28	not used	INC E	RR H	
1D	29	not used	DEC E	RR L	
1E	30	not used	LD E,n	RR (HL)	
1F	31	not used	RRA	RR A	
20	32	space	JR NZ,dis	SLA B	
21	33	!	LD HL,nn	SLA C	
22	34	"	LD (nn),HL	SLA D	
23	35	#	INC HL	SLA E	
24	36	$	INC H	SLA H	
25	37	%	DEC H	SLA L	
26	38	&	LD H,n	SLA (HL)	
27	39	'	DAA	SLA A	
28	40	(	JR Z,dis	SRA B	
29	41	)	ADD HL,HL	SRA C	
2A	42	*	LD HL,(nn)	SRA D	
2B	43	+	DEC HL	SRA E	
2C	44	,	INC L	SRA H	
2D	45	—	DEC L	SRA L	
2E	46	.	LD L,n	SRA (HL)	
2F	47	/	CPL	SRA A	
30	48	0	JR NC,dis		
31	49	1	LD SP,nn		
32	50	2	LD (nn),A		
33	51	3	INC SP		
34	52	4	INC (HL)		
35	53	5	DEC (HL)		
36	54	6	LD (HL),n		
37	55	7	SCF		
38	56	8	JR C,dis	SRL B	
39	57	9	ADD HL,SP	SRL C	
3A	58	:	LD A,(nn)	SRL D	
3B	59	;	DEC SP	SRL E	
3C	60	<	INC A	SRL H	
3D	61	=	DEC A	SRL L	
3E	62	>	LD A,n	SRL (HL)	
3F	63	?	CCF	SRL A	
40	64	@	LD B,B	BIT 0,B	IN B,(C)
41	65	A	LD B,C	BIT 0,C	OUT (C),B
42	66	B	LD B,D	BIT 0,D	SBC HL,BC

HEX	DECIMAL	CHARACTER	Z80-NO PREFIX	PREFIX CB	PREFIX ED
43	67	C	LD B,E	BIT 0,E	LD (nn),BC
44	68	D	LD B,H	BIT 0,H	NEG
45	69	E	LD, B,L	BIT 0,1	RETN
46	70	F	LD B,(HL)	BIT 0,(HL)	IM 0
47	71	G	LD B,A	BIT 0,A	LD I,A
48	72	H	LD C,B	BIT 1,B	IN C,(C)
49	73	I	LD C,C	BIT 1,C	OUT (C),C
4A	74	J	LD C,D	BIT 1,D	ADC HL,BC
4B	75	K	LD C,E	BIT 1,E	LD BC,(nn)
4C	76	L	LD C,H	BIT 1,H	
4D	77	M	LD C,L	BIT 1,L	RETI
4E	78	N	LD C,(HL)	BIT 1,(HL)	
4F	79	O	LD C,A	BIT 1,A	LD R,A
50	80	P	LD D,B	BIT 2,B	IN D,(C)
51	81	Q	LD D,C	BIT 2,C	OUT (C),D
52	82	R	LD D,D	BIT 2,D	SBC HL,DE
53	83	S	LD D,E	BIT 2,E	LD (nn),DE
54	84	T	LD D,H	BIT 2,H	
55	85	U	LD D,L	BIT 2,L	
56	86	V	LD D,(HL)	BIT 2,(HL)	IM 1
57	87	W	LD D,A	BIT 2,A	LD A,I
58	88	X	LD E,B	BIT 3,B	IN E,(C)
59	89	Y	LD E,C	BIT 3,C	OUT (C),E
5A	90	Z	LD E,D	BIT 3,D	ADC HL,DE
5B	91	[	LD E,E	BIT 3,E	LD DE,(nn)
5C	92	/	LD E,H	BIT 3,H	
5D	93	]	LD E,L	BIT 3,L	
5E	94	↑	LD E,(HL)	BIT 3,(HL)	IM 2
5F	95	—	LD E,A	BIT 3,A	LD A,R
60	96	£	LD H,B	BIT 4,B	IN H,(C)
61	97	a	LD H,C	BIT 4,C	OUT (C),H
62	98	b	LD H,D	BIT 4,D	SBC HL,HL
63	99	c	LD H,E	BIT 4,E	LD (nn),HL
64	100	d	LD H,H	BIT 4,H	
65	101	e	LD H,L	BIT 4,L	
66	102	f	LD H,(HL)	BIT 4,(HL)	
67	103	g	LD H,A	BIT 4,A	RRD
68	104	h	LD L,B	BIT 5,B	IN L,(C)
69	105	i	LD L,C	BIT 5,C	OUT (C),L
6A	106	j	LD L,D	BIT 5,D	ADC HL,HL
6B	107	k	LD L,E	BIT 5,E	LD HL,(nn)
6C	108	l	LD L,H	BIT 5,H	
6D	109	m	LD, L,L	BIT 5,L	
6E	110	n	LD L,(HL)	BIT 5,(HL)	
6F	111	o	LD L,A	BIT 5,A	RLD
70	112	p	LD (HL),B	BIT 6,B	IN F,(C)
71	113	q	LD (HL),C	BIT 6,C	
72	114	r	LD (HL),D	BIT 6,D	SBC HL,SP
73	115	s	LD (HL),E	BIT 6,E	LD (nn),SP
74	116	t	LD (HL),H	BIT 6,H	

HEX	DECIMAL	CHARACTER	Z80-NO PREFIX	PREFIX CB	PREFIX ED
75	117	u	LD (HL),L	BIT 6,L	
76	118	v	HALT	BIT 6,(HL)	
77	119	w	LD (HL),A	BIT 6,A	
78	120	x	LD A,B	BIT 7,B	IN A,(C)
79	121	y	LD A,C	BIT 7,C	OUT (C),A
7A	122	z	LD A,D	BIT 7,D	ADC HL,SP
7B	123	{	LD A,E	BIT 7,E	LD SP,(nn)
7C	124	|	LD A,H	BIT 7,H	
7D	125	}	LD A,L	BIT 7,L	
7E	126	~	LD A,(HL)	BIT 7,(HL)	
7F	127	©	LD A,A	BIT 7,A	
80	128		ADD A,B	RES 0,B	
81	129		ADD A,C	RES 0,C	
82	130		ADD A,D	RES 0,D	
83	131		ADD A,E	RES 0,E	
84	132		ADD, A,H	RES 0,H	
85	133		ADD A,L	RES 0,L	
86	134		ADD A,(HL)	RES 0,(HL)	
87	135		ADD A,A	RES 0,A	
88	136		ADC A,B	RES 1,B	BLOCK GRAPHICS
89	137		ADC A,C	RES 1,C	
8A	138		ADC A,D	RES 1,D	
8B	139		ADC A,E	RES 1,E	
8C	140		ADC A,H	RES 1,H	
8D	141		ADC A,L	RES 1,L	
8E	142		ADC A,(HL)	RES 1,(HL)	
8F	143		ADC A,A	RES 1,A	
90	144	(a)	SUB B	RES 2,B	
91	145	(b)	SUB C	RES 2,C	
92	146	(c)	SUB D	RES 2,D	
93	147	(d)	SUB E	RES 2,E	
94	148	(e)	SUB H	RES 2,H	
95	149	(f)	SUB L	RES 2,L	
96	150	(g)	SUB (HL)	RES 2,(HL)	
97	151	(h)	SUB A	RES 2,A	
98	152	(i)	SBC A,B	RES 3,B	
99	153	(j) user	SBC A,C	RES 3,C	
9A	154	(k) graphics	SBC A,D	RES 3,D	
9B	155	(l)	SBC A,E	RES 3,E	
9C	156	(m)	SBC A,H	RES 3,H	
9D	157	(n)	SBC A,L	RES 3,L	
9E	158	(o)	SBC A,(HL)	RES 3,(HL)	
9F	159	(p)	SBC A,A	RES 3,A	
A0	160	(q)	AND B	RES 4,B	LDI
A1	161	(r)	AND C	RES 4,C	CPI
A2	162	(s)	AND D	RES 4,D	INI
A3	163	(t)	AND E	RES 4,E	OUTI
A4	164	(u)	AND H	RES 4,H	
A5	165	**RND**	AND L	RES 4,L	
A6	166	**INKEY$**	AND (HL)	RES 4,(HL)	

HEX	DECIMAL	CHARACTER	Z80-NO PREFIX	PREFIX CB	PREFIX ED
A7	167	**PL**	AND A	RES 4,A	
A8	168	**FN**	XOR B	RES 5,B	LDD
A9	169	**POINT**	XOR C	RES 5,C	CPD
AA	170	**SCREEN$**	XOR D	RES 5,D	IND
AB	171	**ATTR**	XOR E	RES 5,E	OUTD
AC	172	**AT**	XOR H	RES 5,H	
AD	173	**TAB**	XOR L	RES 5,L	
AE	174	**VAL$**	XOR (HL)	RES 5,(HL)	
AF	175	**CODE**	XOR A	RES 5,A	
B0	176	**VAL**	OR B	RES 6,B	LDIR
B1	177	**LEN**	OR C	RES 6,C	CPIR
B2	178	**SIN**	OR D	RES 6,D	INIR
B3	179	**COS**	OR E	RES 6,E	OTIR
B4	180	**TAN**	OR H	RES 6,H	
B5	181	**ASN**	OR L	RES 6,L	
B6	182	**ACS**	OR (HL)	RES 6,(HL)	
B7	183	**ATN**	OR A	RES 6,A	
B8	184	**LN**	CP B	RES 7,B	LDDR
B9	185	**EXP**	CP C	RES 7,C	CPDR
BA	186	**INT**	CP D	RES 7,D	INDR
BB	187	**SQR**	CP E	RES 7,E	OTDR
BC	188	**SGN**	CP H	RES 7,H	
BD	189	**ABS**	CP L	RES 7,L	
BE	190	**PEEK**	CP (HL)	RES 7,(HL)	
BF	191	**IN**	CP A	RES 7,A	
C0	192	**USR**	RET NZ	SET 0,B	
C1	193	**STR$**	POP BC	SET 0,C	
C2	194	**CHR$**	JP NZ,nn	SET 0,D	
C3	195	**NOT**	JP nn	SET 0,E	
C4	196	**BIN**	CALL NZ,nn	SET 0,h	
C5	197	**OR**	PUSH BC	SET 0,L	
C6	198	**AND**	ADD A,n	SET 0,(HL)	
C7	199	< =	RST 0	SET 0,A	
C8	200	> =	RET Z	SET 1,B	
C9	201	<>	RET	SET 1,C	
CA	202	**LINE**	JP Z,nn	SET 1,D	
CB	203	**THEN**		SET 1,E	
CC	204	**TO**	CALL Z,nn	SET 1,H	
CD	205	**STEP**	CALL nn	SET 1,L	
CE	206	**DEF FN**	ADC A,n	SET 1,(HL)	
CF	207	**CAT**	RST 8	SET 1,A	
D0	208	**FORMAT**	RET NC	SET 2,B	
D1	209	**MOVE**	POP DE	SET 2,C	
D2	210	**ERASE**	JP NC,nn	SET 2,D	
D3	211	**OPEN #**	OUT (n),A	SET 2,E	
D4	212	**CLOSE #**	CALL NC,nn	SET 2,H	
D5	213	**MERGE**	PUSH DE	SET 2,L	
D6	214	**VERIFY**	SUB n	SET 2,(HL)	
D7	215	**BEEP**	RST 16	SET 2,A	
D8	216	**CIRCLE**	RET C	SET 3,B	

HEX	DECIMAL	CHARACTER	Z80-NO PREFIX	PREFIX CB	PREFIX ED
D9	217	**INK**	EXX	SET 3,C	
DA	218	**PAPER**	JP C,nn	SET 3,D	
DB	219	**FLASH**	IN A,(n)	SET 3,E	
DC	220	**BRIGHT**	CALL C,nn	SET 3,H	
DD	221	**INVERSE**	PREFIXES INSTRUCTIONS USING IX	SET 3,L	
DE	222	**OVER**	SBC A,n	SET 3,(HL)	
DF	223	**OUT**	RST 24	SET 3,A	
E0	224	**LPRINT**	RET PO	SET 4,B	
E1	225	**LLIST**	POP HL	SET 4,C	
E2	226	**STOP**	JP PO,nn	SET 4,D	
E3	227	**READ**	EX (SP),HL	SET 4,E	
E4	228	**DATA**	CALL PO,nn	SET 4,H	
E5	229	**RESTORE**	PUSH HL	SET 4,L	
E6	230	**NEW**	AND n	SET 4,(HL)	
E7	231	**BORDER**	RST 32	SET 4,A	
E8	232	**CONTINUE**	RET PE	SET 5,B	
E9	233	**DIM**	JP (HL)	SET 5,C	
EA	234	**REM**	JP PE,nn	SET 5,D	
EB	235	**FOR**	EX DE, HL	SET 5,E	
EC	236	**GO TO**	CALL PE,nn	SET 5,H	
ED	237	**GO SUB**		SET 5,L	
EE	238	**INPUT**	XOR n	SET 5,(HL)	
EF	239	**LOAD**	RST 40	SET 5,A	
F0	240	**LIST**	RET P	SET 6,B	
F1	241	**LET**	POP AF	SET 6,C	
F2	242	**PAUSE**	JP P,nn	SET 6,D	
F3	243	**NEXT**	DI	SET 6,E	
F4	244	**POKE**	CALL P,nn	SET 6,H	
F5	245	**PRINT**	PUSH AF	SET 6,L	
F6	246	**PLOT**	OR n	SET 6,(HL)	
F7	247	**RUN**	RST 48	SET 6,A	
F8	248	**SAVE**	RET M	SET 7,B	
F9	249	**RANDOMIZE**	LD SP,HL	SET 7,C	
FA	250	**IF**	JP M,nn	SET 7,D	
FB	251	**CLS**	EI	SET 7,E	
FC	252	**DRAW**	CALL M,nn	SET 7,H	
FD	253	**CLEAR**	PREFIXES INSTRUCTIONS USING IY	SET 7,L	
FE	254	**RETURN**	CP n	SET 7,(HL)	
FF	255	**COPY**	RST 56	SET 7,A	

Appendix B
FLAGS

How the instructions affect the flags.

APPENDIX B

This appendix lists each Z80 instruction and next to each its effect on the flags. Only the important flags are shown, the Carry flag Parity/Overflow flag, Zero flag, Sign flag, the other flags are of no direct use to the programmer since they do not play any part in decision making so far as JR, JP, CALL, and RET are concerned. Throughout the table a number of symbols are used:

In the mnemonics:	nn	Single byte data
	nnnn	Double byte data
	r	Single byte register (A,B,C,D,E,H,L)
	d	Double byte register
	c	Condition
	dis	Two's complement displacement data
In the flags:	0	The flag is reset to zero
	1	The flag is set to one
	.	The flag is unaffected
	R	The flag is changed to reflect result
	?	The flag is set or reset randomly

B This flag is set to one if the B or BC register pair (whichever appropriate for given instruction) is zero at the end of the operation

INSTRUCTIONS mnemonic	FLAGS S	Z	P	C
ADC A,r	R	R	R	R
ADC HL,d	R	R	R	R
ADD A,r	R	R	R	R
ADD HL,d	.	.	.	R
ADD IX,d	.	.	.	R
ADD IY,d	.	.	.	R
AND r	R	R	R	Ø
BIT b,r	?	R	?	.
CALL nnnn	.	.	.	.
CALL c,nnnn	.	.	.	.
CCF	.	.	.	R
CP r	R	R	R	R
CPI	R	B	R	.
CPD	R	B	R	.
CPIR	R	B	R	.
CPDR	R	B	R	.
CPL	.	.	.	.
DAA	R	R	R	R
DEC r	R	R	R	.
DEC d	.	.	.	.
DI	.	.	.	.
DJNZ dis	.	B	.	.
EI	.	.	.	.
EX AF,AF'	.	.	.	.
EX DE,HL	.	.	.	.
EX (SP),HL	.	.	.	.
EX (SP),IX	.	.	.	.
EX (SP),IY	.	.	.	.
EXX	.	.	.	.
HALT	.	.	.	.

INSTRUCTIONS mnemonic	FLAGS S	Z	P	C
IM Ø	.	.	.	.
IM 1	.	.	.	.
IM 2	.	.	.	.
INC r	R	R	R	.
INC d	.	.	.	.
IN A,(nn)	.	.	.	.
IN r,(C)	R	R	R	.
INI	?	B	?	.
IND	?	B	?	.
INIR	?	1	?	.
INDR	?	1	?	.
JP nnnn	.	.	.	.
JP c,nnnn	.	.	.	.
JP (HL)	.	.	.	.
JP (IX)	.	.	.	.
JP (IY)	.	.	.	.
JR dis	.	.	.	.
JR c,dis	.	.	.	.
LD (d),A	.	.	.	.
LD A,(d)	.	.	.	.
LD A,R	R	R	R	.
LD A,I	R	R	R	.
LD I,A	.	.	.	.
LD R,A	.	.	.	.
LD SP,HL	.	.	.	.
LD SP,IX	.	.	.	.
LD SP,IY	.	.	.	.
LD r,r	.	.	.	.
LD r,nn	.	.	.	.
LD d,nnnn	.	.	.	.
LD A,(nnnn)	.	.	.	.
LD (nnnn),A	.	.	.	.
LD d,(nnnn)	.	.	.	.
LD (nnnn),d	.	.	.	.
LDI	.	.	B	.
LDD	.	.	B	.

INSTRUCTIONS mnemonic	FLAGS S	Z	P	C
LDIR	.	.	Ø	.
LDDR	.	.	Ø	.
NEG	R	R	R	R
NOP	.	.	.	.
OR r	R	R	R	Ø
OUT (nn),A	.	.	.	.
OUT (C),r	.	.	.	.
OUTI	?	B	?	.
OUTD	?	B	?	.
OTIR	?	1	?	.
OTDR	?	1	?	.
POP AF	R	R	R	R
POP d	.	.	.	.
PUSH AF	.	.	.	.
PUSH d	.	.	.	.
RES b,r	.	.	.	.
RET	.	.	.	.
RET c	.	.	.	.
RETN	.	.	.	.
RETI	.	.	.	.
RLA	.	.	.	R
RLCA	.	.	.	R
RRA	.	.	.	R
RRCA	.	.	.	R
RL r	R	R	R	R
RLC r	R	R	R	R
RR r	R	R	R	R
RRC r	R	R	R	R
RRD	R	R	R	.
RST ØØ	.	.	.	.
RST Ø8	.	.	.	.
RST 1Ø	.	.	.	.
RST 18	.	.	.	.
RST 2Ø	.	.	.	.
RST 28	.	.	.	.
RST 3Ø	.	.	.	.

INSTRUCTIONS mnemonic	FLAGS S	Z	P	C
RST 38	.	.	.	.
SBC A,r	R	R	R	R
SBC HL,d	R	R	R	R
SCF	.	.	.	1
SET b,r	.	.	.	.
SLA r	R	R	R	R
SRA r	R	R	R	R
SRL r	R	R	R	R
SUB r	R	R	R	R
XOR r	R	R	R	Ø

n.b. in cases where 'r' is shown in the mnemonics it not only represents single byte registers but also (HL) & (IX+dis), (IY+dis) and direct data 'nn' where applicable.

Appendix C
SYSTEM VARIABLES

APPENDIX C

This appendix lists all the system variables and explains them, many more fully than the manual. It is best understood by those who have a knowledge of the ROM's mechanism.

SYSTEM VARIABLES

X — Only adjust this variable if you understand the effect.

Bytes	*Hex Address*	*Label*	*Function*
8	5C00	KSTATE	This variable consists of eight bytes. Each byte holds information about the key pressed, such as when it is due to repeat, and its code in extended mode.
1	5C08	LAST K	This is set to the last key pressed depending on the mode. It is only changed when another key is pressed. Automatic repeat operates on this. By resetting it to zero then testing it one can wait for a keypress.
1	5C09	REPDEL	The time (50th of a second or 60th of a second in N. America) which a key

Bytes	Hex Address	Label	Function
			must be held down before it repeats. Initialised with 23 Hex.
1	5C0A	REPPER	The delay (50th of a second or 60th of a second in N. America) between successive repeats of a key. Initially 05 Hex. Decrease to a minimum of 01 Hex to speed up repeating.
2	5C0B	DEFADD	Address of arguments of user-defined function if one is being evaluated.
1	5C0D	K DATA	When a colour control code is entered direct from the keyboard, eg extended shift–1 (INK blue) the second byte, eg the colour, FLASH etc, is stored here while the INK, PAPER, BRIGHT, FLASH or INVERSE code is printed. Once that is done the ROM recalls the second byte from here so that can be printed following the colour control code.
2	5C0E	TVDATA	This is used by the print routine to store AT, TAB and the colour controls going to television.
X38	5C10	STRMS	This is used as a store for offsets in CHANS. For each of 16 user files and three system files, there is an offset. When this is added to CHANS, it

Bytes	Hex Address	Label	Function
			points to an address which is the start of the file handling a routine for that file.
2	5CB6	CHARS	100 Hex less than the address of character set (space to copyright). Normally set to 4C00 Hex (character set at 4D00 Hex) but can be altered to point to a user-defined character set.
1	5C38	RASP	The computer 'rasps' at you if you type in colour control coders in such a way as to make an illegal colour. It also rasps when the edit line grows above 23 lines. To change the length of this, alter this variable.
1	5C39	PIP	The length of the keyboard bleep/click. A larger PIP makes keypresses more audible.
1	5C3A	ERR NR	One less than the report code. Starts off at FF Hex. If POKEd in BASIC program then program ends at last line then the error code POKEd is displayed.
X1	5C3B	FLAGS	Flags to control the basic system.
X1	5C3C	TVFLAG	Flags associated with the display and printing.

Bytes	Hex Address	Label	Function
X2	5C3D	ERR SP	This points to an item on the machine stack. When an error occurs this item is the address that is jumped to after the stacks are reset by RST Ø8. By altering this item, new error handling routines can be written.
2	5CBF	LIST SP	This points to the return address on the machine stack which is jumped to after an automatic listing.
1	5C41	MODE	Specified a K,L,C,E or G cursor.
2	5C42	NEWPPC	Line to be jumped to. Used with GOTO and GOSUB.
1	5C44	NSPPC	Statement number in line to be jumped to. POKEing first NEWPPC and then PPC forces a jump to a specified statement in a line.
2	5C45	PPC	Line number of the statement currently being executed.
1	5C47	SUBPPC	Number of statement currently being executed.
1	5C48	BORDCR	This is the attribute byte for the lower half of the screen. Bits zero to two are the INK colour and bits three to five are the PAPER/BORDER colour. The FLASH and BRIGHT bits are not used.

Bytes	Hex Address	Label	Function
2	5C49	E PPC	Number of line with program cursor.
X2	5C4B	VARS	Points to the start of where the variables are stored.
2	5C4D	DEST	Address of variable in assignment.
X2	5C4F	CHANS	Points to table of file handling addresses — used by STRMS.
X2	5C51	CURCHL	Points to the address (in table of file handling addresses) that is being used for the file handling routine.
X2	5C53	PROG	Address of BASIC program.
X2	5C55	NXTLIN	Address of next line in BASIC program.
X2	5C57	DATADD	Points to terminator of last DATA item. If no DATA in the program then it points to the 8Ø Hex at the end of the channel data.
X2	5C59	E LINE	Address of command being typed in.
2	5C5B	K CUR	Address of cursor within the command line.
X2	5C5D	CHADD	Address of next character to be interpreted.
2	5C5F	X PTR	Address of the character after the '?' marker.

Bytes	Hex Address	Label	Function
X2	5C61	WORKSP	Address of temporary workspace.
X2	5C63	STKBOT	Address of bottom of calculator stack.
X2	5C65	STKEND	Address of start of spare space.
1	5C67	BREG	Calculator's register used for a variety of counting purposes.
2	5C68	MEM	Address of area used for the calculator's six memories (usually MEMBOT but not always).
1	5C6A	FLAGS2	More flags.
X1	5C6B	DF SZ	The number of lines (including one blank line) in the lower part of the screen.
2	5C6C	STOP	The number of the top line in automatic listing.
2	5C6E	OLDPPC	Line number to which CONTINUE jumps.
1	5C72	STRLEN	Length of string type destination in assignment.
2	5C74	T ADDR	Address of next item in the syntax table. In the ROM there is a large table which defines where the routine for each command is and how to collect the information needed.

Bytes	Hex Address	Label	Function
2	5C76	SEED	The seed for RND. This is the variable that is set by RANDOMIZE.
3	5C78	FRAMES	Three byte frame counter incremented every 5Øth of a second or 6Øth of a second in N. America. See Chapter 18 of Sinclair manual.
2	5C7B	UDG	Address of first user-defined graphic. Remember that when RAMTOP is moved down for machine code, UDG is *not*. So if you put machine code in the UDG area then it could corrupt any graphics that are defined.
1	5C7D	CO ORDS	Used as a temporary store for the X co-ordinate while plotting calculations take place.
1	5C7E		As above but for the y co-ordinate.
1	5C7E	P POSN	33 column number of printer position.
1	5C8Ø	PR CC	Less significant byte of address of next position for LPRINT AT (in printer buffer).
1	5C81		Not used.
2	5C82	ECHO E	33 column number and 24 line number (in lower half) of end of input buffer.

Bytes	Hex Ad- dress	Label	Function
2	5C84	DFCC	Address of PRINT position for top slice of character in display file. Can be redirected.
2	5C86	DFCCL	Like DF CC for lower part of screen.
X1	5C88	S POSN	33 column number for PRINT position.
X1	5C89		24 line number for PRINT position.
X2	5C8A	SPOSNL	Like SPOSN for lower part.
1	5C8C	SCR CT	Count scrolls; it is always one more than the number of scrolls that will be done before stopping with 'scroll?'. If this is regularly POKEd with 255 then it will scroll on and on without stopping.
1	5C8D	ATTR P	Permanent attribute (as set up by global INK, PAPER statements, etc).
1	5C8E	MASK P	Used for transparent attributes. Any bit that is one shows that the one corresponding attribute bit is not taken from ATTR P but from what is already on the screen.
1	5C8F	ATTR T	Temporary current attributes (eg set up in PRINT, PLOT, DRAW statements, etc).
1	5C90	MASK T	Like MASK P, but temporary.

Bytes	Hex Ad- dress	Label	Function
1	5C91	P FLAG	More flags.
30	5C92	MEMBOT	These are where the calculator can store six different five byte floating point numbers in special 'memories'.
2	5CB0	INTERR*	Ex-interrupt vector unused due to a feature of the ROM's programming.
2	5CBZ	RAMTOP	Address of last byte of BASIC area.
2	5CB4	PRAMT	Address of last byte of physical RAM.

* See 'May I Interrupt'.

APPENDIX D
Z8Ø MNEMONICS AND EXPLANATION

APPENDIX D

This appendix provides a brief description of each of the Z8Ø commands. It details the operation and flag adjustment. Hex codes are not given but can be found in Appendix A where all the Z8Ø mnemonics are listed with their op-codes and associated Spectrum ASCII and standard ASCII codes.

SOME ABBREVIATIONS ARE USED:

r = single byte register:

A,B,C,D,E,H or L

nn = single byte of direct data.

nnnn = double byte of direct data.

d1 = double byte of register:

BC,DE,HL, and SP

d2 = double byte register:

BC,DE,HL,IX,IY and SP

x = bit number Ø,1,2,3,4,5,6 or 7.

dis = displacement byte, according to two's complement convention.

res = single byte Hex value:

00,08,10,18,20,28,30 or 38.

ADC A,r : Adds register r to the Accumulator and also adds the Carry Flag to the least specification bit position at the start of the operation. With the exception of ADCA,A the contents of the operand register are left unchanged. The N flag is reset to zero by ADC, and the remaining flags will reflect the final status of the Accumulator.

ADC A,(HL)
ADC A,(IX + dis)
ADC A,(IY + dis)
ADC A,nn : Identical to ADC A,r except that data pointed to by (HL), (IX + dis), (IY + dis) and direct data nn respectively is added to the Accumulator.

ADC HL,d1 : Performs double byte addition firstly adding the Carry Flag to the least significant bit of the L register, then adding double byte register d1 (BC, DE,HL,SP) to HL. The N flag will be set to zero and the other flags will reflect the status of HL.

ADD A,r : Performs a simple single byte addition, adding register r to the Accumulator. ADD sets the N flag to zero and the other flags will reflect the status of the Accumulator.

ADD A,(HL)
ADD A,(IX + dis)
ADD A,(IY + dis)
ADD A,nn : Exactly the same as ADD A,r except that the data pointed to by (HL), (IX + dis),(IY + dis) and direct data nn is added instead of register r.

ADD HL,d1
ADD IX,d1
ADD IY,d1 : Perform double byte addition to the HL, IX and IY registers respectively. A double byte register d1 (HL,BC,DE,SP) is added and remains unchanged at the end of the operation.

AND r : Performs a logical AND operation on the Accumulator. The bits of register r are compared with more of the Accumulator, any corresponding bit which is both one in the Accumulator and register r results in the bit being kept at one in the Accumulator, else it is set to zero. The operand register r is unaffected by this instruction. The flags are set accordingly.

AND (HL)
AND (IX + dis)
AND (IY + dis)
AND nn : Logical AND operation is performed on the Accumulator. The data pointed to by HL, IX + dis, IY + dis, and direct data nn is ANDed with the Accumulator. The results are left in the Accumulator, but the operand (either (HL), (IX + dis), (IY + dis) or nn) is left unchanged. The flags are set to the status of the Accumulator.

BIT x,r
BIT x,(HL)
BIT x,(IX + dis)
BIT x,(IY + dis) : This instruction text bit x (where x is between zero and seven) of register r, (HL), (IX + dis) or (IY + dis). If the bit is zero then the Zero Flag is set. If the bit is one then the Zero Flag is reset. The C flag is left unchanged, the H flag is set to one and the N flag is cleared. The status of the S and P/V Flags cannot be predicted.

CALL nnnn : Causes a jump to a subroutine at address nnnn. The address of the next

instruction is stacked and remains there until a RET is reached. The flags are unaffected by CALL.

CALL C,nnnn
CALL M,nnnn
CALL NC,nnnn
CALL NZ,nnnn
CALL P,nnnn
CALL PE,nnnn
CALL PO,nnnn
CALL Z,nnnn
: These directives operate in exactly the same way as an unconditional CALL except that they will be ignored unless the condition is satisfied. They leave the flags unaffected.

CCF : The condition of the Carry Flag is reversed from its current state. The N flag is set to zero, but the other flags remain unaltered.

CP r : This instruction subtracts register r from the Accumulator but does not actually change the Accumulator or register r, but the flags are set in reflection with the result.

CP(HL)
CP(IX + dis)
CP(IY + dis)
CP nn
: These instructions behave exactly like CP r except that the data pointed to by HL?, IX + dis, IY + dis and direct data nn are compared as opposed to register r. The flags reflect the result of the subtraction.

CPD : The contents of the memory location pointed to by HL are compared with the contents of the Accumulator. The N flag is set to 1, the other flags reflect the result of the comparison, except the Carry Flag, which remain unaltered. The BC register, acting as a 'byte count' is then decremented and

the HL register pair are also decremented to poiont the next, lower memory location. If the BC register pair becomes zero then the P/V flag is set to zero otherwise it is set to one.

CPDR : This performs exactly the same as CPD except that after HL has been decremented and BC decremented, if BC is not zero then the operation is repeated. It will not be repeated if BC is zero or the Accumulator value matches that of the memory location pointed to by HL, in other words until a match has been found.

CPI : This operates just like CPD except that the HL register is incremented instead of decremented.

CPIR : This operates just like CPDR except that the HL register is incremented on each completion of the operation, instead of decremented.

CPL : The contents of the Accumulator are complemented. That is, the bits equal to one are set to zero and those equal to zero are set to one. All CPU flags are unaffected except H and N which are set to one.

DAA : This instruction alters the value of the Accumulator from its binary value into two Binary Coded Decimal digits. The most significant four bits reflect the 'tens' digit and the last significant four bits reflect the 'units' digit.

EI : The enable interrupt command instructs the CPU to accept maskable interrupts. After an interrupt is received no further interrupts will be accepted until the next EI command is encountered.

EX AF,AF' : This single byte command causes the values of the AF and AF' register to exchange places. The flags will reflect the contents of the new bank zero flags after the exchange.

EXX : The double byte registers, BC,DE and HL are exchanged with their duplicates in bank one.

EX DE,HL : This instruction causes the values of double byte registers DE and HL to be exchanged.

EX (SP),HL : The top value on the stack is exchanged with HL by this instruction. SP and the flags are unaltered by this instruction.

EX (SP),PX : Similar to the EX(SP), HL instruction these two exchange the index registers IX and IY respectively. The flags remain unaltered.

HALT : This causes the CPU to halt all operations, and it will remain in this state until an interrupt or reset signal is received. Refresh of dynamic memory (such as that used in the Spectrum) is refreshed because the processor actually repeatedly processes the NOP instruction to maintain memory

refreshing. This instruction does not alter any CPU registers or flags.

IM Ø : This instruction sets the interrupt mode to zero. When an interrupt is received, the CPU allows a properly designed and activated external device to force an instruction onto the data bus. The CPU will execute that instruction. The flags are unaltered by any IM directive.

IM 1 : This instruction sets the interrupt mode to one. When an interrupt is received the CPU will execute a RESTART to location 38 Hex. The CPU flags are not affected by this command.

IM 2 : This instruction sets the CPU to interrupt mode two. The CPU, when 'interrupted' will jump to a location formed by taking the interrupt vector register (IV or just I) as the most significant eight bits and the information places on the Data Bus and the least significant eight bits.

IN r,(C) : This directive causes the CPU to read the value at port C and place it into register r. The CPU flags are unaltered by this instruction.

IN A,nn : Operates in exactly the same manner as the above but direct data nn is used, and register A instead of register C and r respectively.

DEC r : This instruction decrements register r by one. This directive does not affect the C flag. The N flag is reset to zero. The other flags reflect the resulting value of register r.

DEC (HL)
DEC (IX + dis)
DEC (IY + dis) : These directive decrement the contents of the location point to by HL, IX + dis and IY + dis. The flags are affected as previously described for DEC r.

DEC d2 : This instruction decrements double byte register d2 (BC,DE,HL,SP,IX and IY) by one. This command does *not* affect the flags.

DI : This command instructs the CPU not to accept a maskable interrupt.

DJNZ dis : This directive causes register B to be decremented by one. If B does not equal zero then a Jump Relative is made, using dis and two's complement, dis is added to the Program Counter.

INC r : This instruction increments register r by one. The Carry Flag is unaffected, the N flag is reset to zero and the other flags reflect the resulting value of register r.

INC HL
INC(IX + dis)
INC(IY + dis) : This directive increments the contents of the location pointed to by HL,IX + dis and IY + dis. The flags are affected as previously described for INC r.

INC d2 : This instruction increments double byte register d2 (BC,DE,HL,SP,IX or IY) by one. This command does *not* affect the flags.

IND : This directive results in data being accepted from the input port specified by register C. The data is transferred to the location pointed to by HL; HL is then decremented by one. The B register, serving as a counter is decremented and if this results in the B register becoming zero then the Z flag is set. The status of the S,M and P/V cannot be predicted. The N flag is always set to one by this directive and the Carry Flag is unaltered.

INDR : This instruction behaves exactly like the previously described IND but if B is not zero at one end of the operation then it is repeated. This means that at the end of the operation, the flags will be as before but the Z flag will be set to one.

INI : This directive has exactly the same effect as IND but the HL register pair is incremented instead of decremented.

INIR : This behaves exactly like INDR except that again the HL register pair is incremented.

JP(HL) : This causes a jump to the address specified by double byte register HL.

JP(IX)
JP(IY) : This directive causes a jump to the address specified by index register IX

or IY respectively. The flags are unaltered by these instructions.

JP nnnn : This causes a direct jump to address nnnn. The flags are unaltered by this directive.

JP C,nnnn
JP M,nnnn
JP NC,nnnn
JP NZ,nnnn
JP P,nnnn
JP PE,nnnn
JP PO,nnnn
JP Z,nnnn : These directives are ignored by the CPU unless the condition is satisfied. If it is satisfied, then a direct jump to address nnnn is executed. The flags are unaltered by these directives.

JR dis : This instruction causes a Jump Relative to be executed. The destination address is formed by using the displacement byte dis and two's complement convention. The displacement acts from the address of the next instruction. JR dis leaves the flags unaltered.

JR C, dis
JR NC, dis
JR NZ, dis
JR Z, dis : These directives act in the same way as JR dis except that they are ignored by the CPU unless the condition is satisfied.

LD (nnnn),A
LD (nnnn),d2
LD (BC),A
LD (DE),A
LD (HL),r
LD (HL),nn
LD (IX + dis),r
LD (IX + dis),nn
LD (IY + dis),r
LD (IY + dis),nn
LD A,(nnnn)
LD A,(BC)
LD A,(DE)
LD r,(HL)
LD r,(IX + dis)
LD r,(IY + dis)
LD r,(r)
LD r,nn
LD d2,(nnn)
LD d2,nnnn
LD A,I
LD I,A
LD A,R
LD R,A
LD SP,HL
LD SP,IX
LD SP,IY : The LD directive has an incredible number of combinations. All the possible syntax are listed. It simply copies the value from the right-hand data, which can be the contents of a location, a single or double byte register or direct data, into the left-hand destination, which can be a single or double byte register, or a memory location. Not all combinations are possible so check, when programming, that the syntax you use is on the list or look in Appendix A to ensure that you do not write an impossible program . . .

LDD : The contents of the memory location pointed to by the HL register pair are transferred to the location pointed by the DE register pair. DE and HL are then decremented. BC is decremented and if that results in BC becoming zero then the P/V flag will be set to zero; otherwise it will be set to one. The H and N flags are reset to zero, but the other flags are left unaltered.

LDDR : This directive operates in exactly the same way as LDD, except that if BC is not zero at the end of the operation the whole thing is repeated. This means that when execution is completed the P/V flag will be zero and the other flags will be the same as described above.

LDI : This behaves just like LDD, except that both HL and DE are incremented instead of decremented. The flags behave in exactly the same way.

LDIR : This directive behaves just like LDIR except that, again, both the HL and DE register pairs are incremented instead of decremented.

NEG : This instruction negates the Accumulator in accordance with two's complement convention. What happens is that first, all the bits are inverted, that is those that are zero are changed to one and *vice versa*. Then one is added to the result. The S and Z flags reflect the result of the operation and the P/V flag is set to one if the Accumulator started, and so finished with 80 Hex. Otherwise it is cleared to zero. The C flag will be cleared to zero if the Accumulator started, and so finished with 00 Hex. Otherwise it is set to one. The N flag is always set to one.

NOP : This directive merely causes the CPU to waste time by doing little but advance the Program Counter to the next location! None of the flags are affected.

OR r : Single byte register r is logical ORed with the Accumulator. What happens is that bit by bit the Accumulator bits are compared with the corresponding ones of register r. If either bit is one or both bits are one, then the

Accumulator bit is set to one; otherwise it is set to zero. Register r is unaffected. The Carry Flag is reset to zero, so is the N flag. The H flag is always set to one and the other flags reflect the result of the operation.

OR(HL)
OR (IX + dis)
OR (IY + dis)
OR nn : In the same way as described for OR r, the contents of the memory location pointed to by HL,IX + dis, IY + dis and direct data nn is ORed with the Accumulator.

OTDR : In a similar way as described for INDR data is transferred *from* the memory location pointed to by HL to the port specified by register C, with register B acting as a 'byte count', and HL being *decremented* after each single run of the operation.

OTIR : This directive operates in exactly the same manner as OTDR except that HL is *incremented*.

OUT (C),r : This outputs the data held in register r to the I/O port specified by register C. The flags are unaltered.

OUT nn,A : This directive is similar to OUT(C),r except that direct data nn specifies the port and the data is taken from A.

OUTD : This is very similar to the previously described IND directive except that with OUTD the data is being sent out to the port from the location specified by HL.

OUTI : This directive is similar to OUTD, but after each operation the HL register pair is incremented instead of decremented.

POP AF : The top two byte value from the stack is removed and put into the AF register pair. The Stack Pointer, SP, is then incremented twice so that it points to the data theoretically below. The flags are unchanged.

POP BC
POP DE
POP HL
POP IX
POP IY : These directives act in exactly the same way as POP AF except that double byte registers BC,DE,HL,IX and IY respectively receive the value.

PUSH AF : This directive has the opposite effect of POP AF. The value held in the AF register pair is put on the stack and the Stack Pointer AP is decremented by two. The flags are unaffected.

PUSH BC
PUSH DE
PUSH HL
PUSH IX
PUSH IY : These instructions act in exactly the same way as PUSH AF, except that BC,DE,HL,IX and IY respectively have their values PUSHed.

RES x,r : This directive resets bit x of single byte register r to zero. The CPU flags are unaffected.

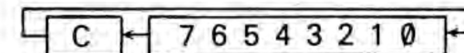
RES x,HL
RES x,(IX + dis)
RES x,(IY + dis) : These instructions reset bit x of the contents of the memory location pointed to by HL, IX+dis and IY+dis respectively.

RET : The top value of the stack is removed and transferred to the Program Counter, PC. The Stack Pointer is incremented twice so that it points to the data theoretically below. Program execution continues using the new value held in PC. The CPU flags are unaffected.

RET C
RET M
RET NC
RET NZ
RET P
RET PE
RET PO
RET Z : These directives act in exactly the same way as RET, but they are ignored unless the condition is satisfied. The flags are unaffected.

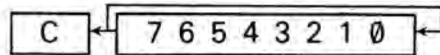
RETI
RETN : These two instructions you will not need, but they are to do with the Z80 interrupt system. In the Spectrum the interrupts are used for scanning the keyboard. This is turned off at certain times, e.g. when the printer operates and when the tape interface is operating. See Chapter Eight.

RL r
RL (HL)
RL (IX+dis)
RL (IY+dis) : The bits of register r or memory location to by HL, IX+dis or IY+dis, are rotated through the carry bit as shown in the diagram below.

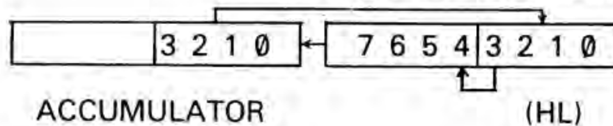


The H and flags are cleared to zero. The S, Z and P/V flags reflect the result of the operation. The P/V flag reflects the parity status of the rotated register or memory location.

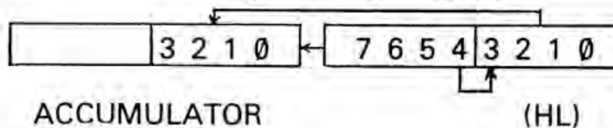
RLC r
RLC (HL)
RLC (IX + dis)
RLC (IY + dis)
: Register r or memory location pointed to by HL, IX + dis or IY + dis respectively, is rotated as shown in the diagram below. The flags behave in the same way as for RL r.



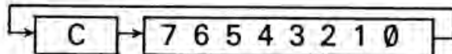
RLD
: This instruction allows a Binary Coded Decimal (BCD) four bit digit in the least significant half of the Accumulator to be rotated to the left with two digits in memory pointed to by HL. This is shown diagramatically below:



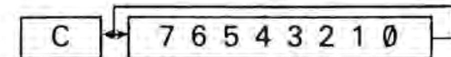
RRD
: This directive is quite similar to the above in that it rotates BCD digits. But in this case, the digits are rotated to the right as shown below:



RR r
RR(HL)
: These directives operate in a similar way to RL except that the rotation is done in the opposite direction as shown below:



RRC r
RRC(HL)
RRC (IX + dis)
RRC (IY + dis)
: These instructions run in a similar way to RLC except that the rotation is done in the opposite direction as shown below:



RST res
: This directive is an effectively shortened version of CALL, which is restricted to address (res) which is restricted to ØØ, Ø8,1Ø,18,2Ø,28,3Ø and 38. RST is short for RESTART. These directives do not affect the flags.

SBCA,r
: This directive subtracts register r from the Accumulator. The content of the Carry Flag is subtracted from the least significant bit of the Accumulator. The N flag is set to one, but the remaining flags are unaffected.

SBC A,nn
SBC A,(HL)
SBC A,(IX + dis)
SBC A,(IY + dis)
: This instruction operates in the same way as SBC A,r except that direct data nn, memory location HL, IX + dis and IY + dis respectively are subtracted instead of register r.

SBC HL,d1
: This directive performs double byte subtraction with Carry Flag. Double byte register d1 is subtracted from HL and also the Carry Flag is subtracted from the least significant bit of HL. The N flag is set to one. The remaining flags reflect the status of HL.

SCF
: The Carry Flag is simply set to one by this directive. The H and N flags are cleared to zero, and the remaining flags are left unaltered.

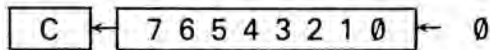
SET x,r
: This directive sets bit x of register r to one. None of the CPU flags are affected.

SET x,(HL)
SET x,(IX + dis)
SET x,(IY + dis)

: These instructions set bit x to one, in the memory location pointed to by HL, IX + dis or IY + dis respectively. None of the flags are affected.

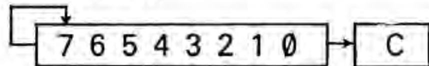
SLA r
SLA (HL)
SLA (IX + dis)
SLA (IY + dis)

: These directive shift register r, or the contents of the memory locations pointed to by HL, IX + dis or IY + dis respectively, as shown in the diagram below. The flags behave as described for RL r.



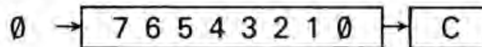
SRA r
SRA (HL)
SRA (IX + dis)
SRA (IY + dis)

: These directives are similar to those described for SLA except that the shifting is done in the other direction as shown below:



SRL r
SRL (HL)
SRL (IX + dis)
SRL (IY + dis)

: These instructions shift register r or memory location pointed to by HL, IX + dis or IY + dis respectively to the left as shown below:



The flags are affected as described for RL.

SUB r

: This directive does a simple single byte subtraction. Register (r) is simply subtracted from the Accumulator. Register r is left unaffected, the N flag is set to one and the remaining flags represent the final contents of the Accumulator.

SUB (HL)
SUB (IX + dis)
SUB (IY + dis)
SUB nn

: These instructions operate in exactly the same way as SUB (r) except that instead of register r being subtracted, the contents of the location pointed to by HL, IX + dis, IY + dis or direct data nn are used.

XOR r

: Register (r) is exclusively ORed with the Accumulator. Each bit in turn is compared. If the Accumulator bit and the register bit are both zero or they are both one then the Accumulator bit is set to zero. Otherwise it is set to one. The Carry Flag is set to zero, as is the N flag, the H flag is set to one and the remaining flags reflect the resulting value of the Accumulator.

XOR nn
COR (HL)
XOR (IX + dis)
XOR (IY + dis)

: These directives act in the same way as above except that direct data nn, memory location pointed to by HL, IX + dis or IY + dis respectively are XORed with the Accumulator.