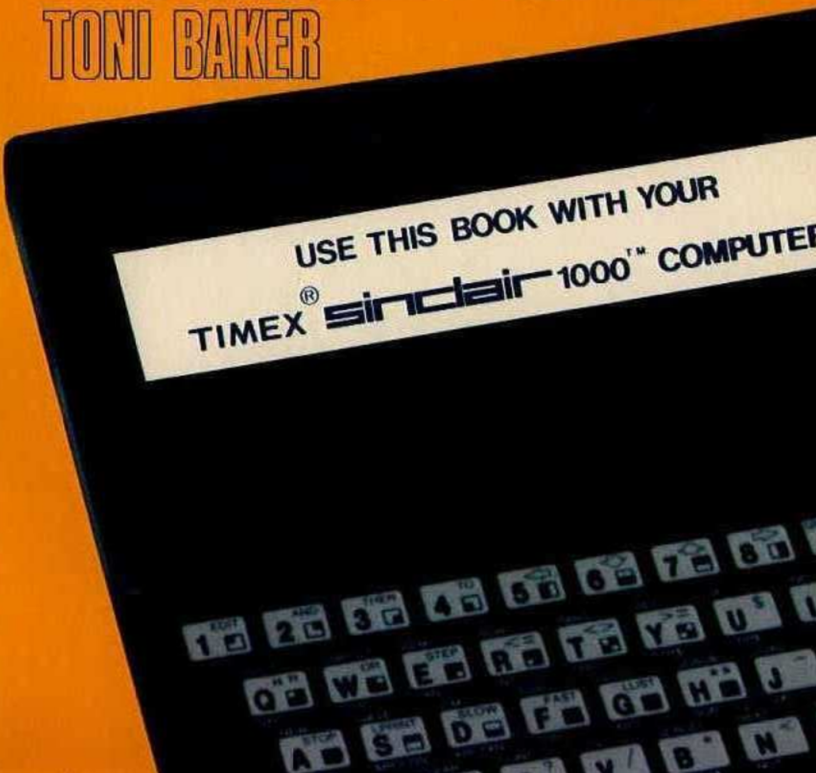
A REWARD BOOK

$12.95

MASTERING MACHINE CODE ON YOUR ZX 81

TONI BAKER



Mastering Machine Code on Your ZX81

Toni Baker

with illustrations by Cathy Lowe

Reston Publishing Company, Inc.
A Prentice-Hall Company
Reston, Virginia

Contents

Foreword

I was staggered when Toni first brought the manuscript for this book to us at the National ZX80 and ZX81 Users' Club. We'd talked about it, and Toni had given me a broad idea of the contents of the book, but until I had the chance to read it, I did not realize just what a comprehensive and easy-to-understand work it would be.

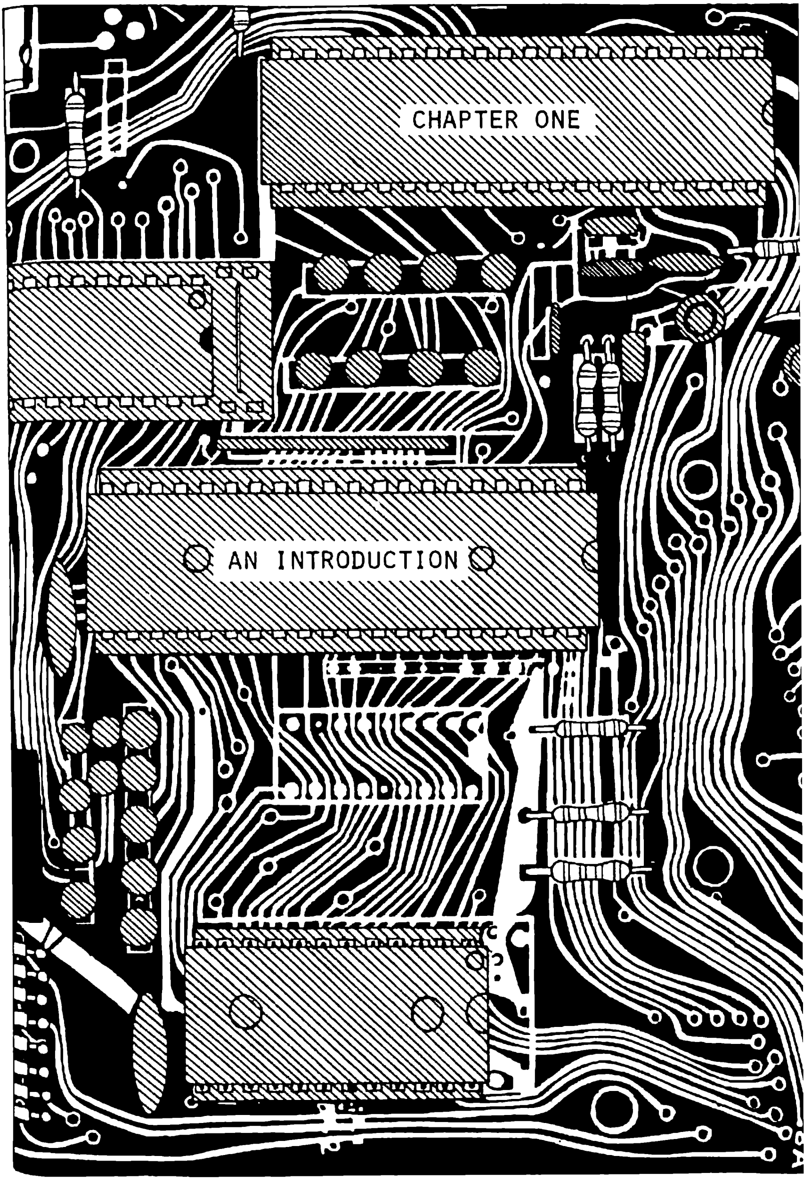
The book has been written for those who know BASIC, but haven't much idea about machine code, and want to get down and master this most useful addition to one's programming skills. We've waited for over a year for a book like this, and now it is here.

If you've decided that GUESS MY NUMBER and SIMON are OK for a while, but now it's time to start exploring the full potential of your computer, and time to begin developing all your potential programming skills, then this book may well prove just what you've been waiting for.

When Toni first came to us with the idea for the book, I stressed that it must be designed to lead someone who knew absolutely nothing about machine code through from the true basics to the point where they would have a real knowledge of how to use it. I'm pleased to say that she has done just that, and if you work through the book with your ZX81 or ZX80 turned on, entering the programs and routines as instructed, you'll certainly end up *Mastering Machine Code on Your ZX81 or ZX80.*

Tim Hartnell
National ZX80 and
ZX81 Users' Club,
London,
August 1981

Mastering Machine Code on Your ZX81


CHAPTER ONE
AN INTRODUCTION

AN INTRODUCTION

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

Machine language is no more than a second computer language - very much like BASIC is in fact. We start by learning the simplest of instructions, and become familiar with them by using them in BASIC programs. An example would be a SCROLL program given in chapter four, which moves the screen downward instead of upward. This effect is rather interesting, and certainly surprizing.

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

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

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

The computer is made to generate many intricate and fascinating displays in the program LIFE. It challenges the skill of an unwary human operator in graphics games such as SPIRALS. A draughts program is included, with several interesting features. This is actually a teaching game because you are encouraged to add your own features to it as you progress.

Careful study of the listings of these programs will teach you a great deal about machine code, but of course the biggest steps in learning will come from experiment. By writing your own programs, or by adapting mine - by all means do - they are intended for this purpose, and some in fact are deliberately improvable for this reason.

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

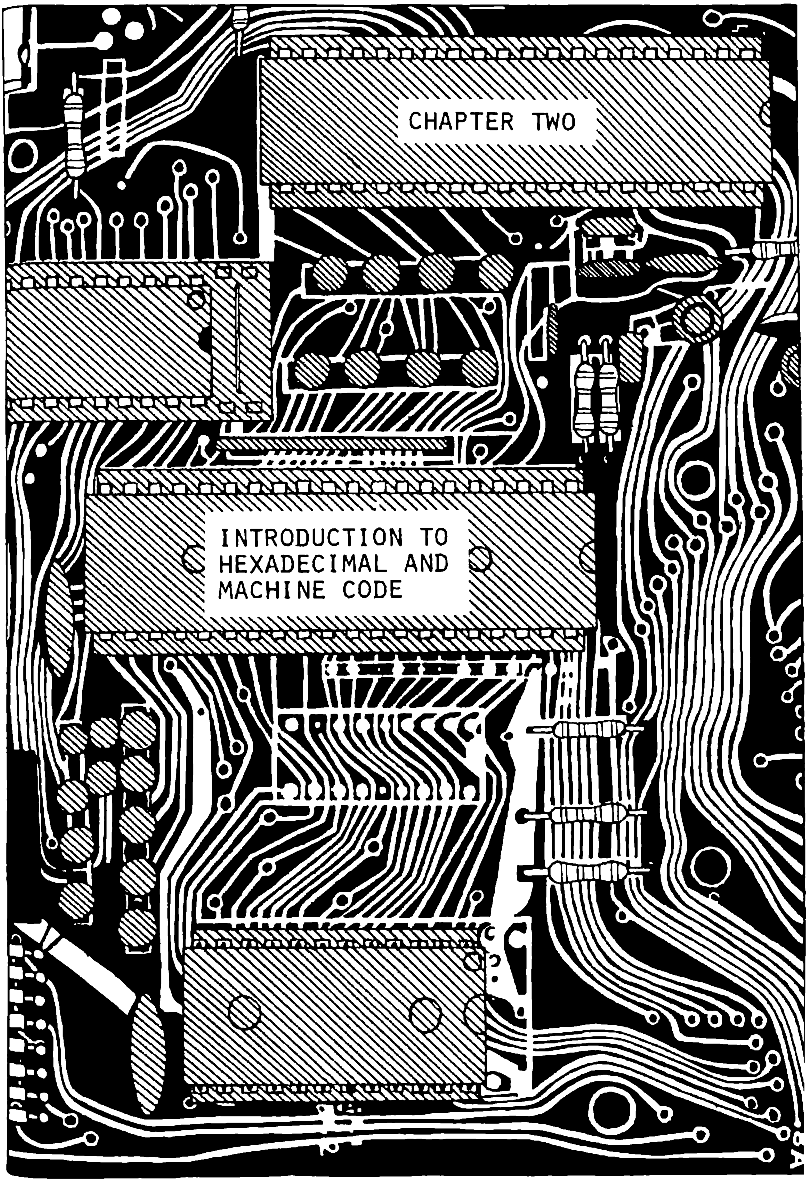
The last two chapters in the book are rather ambitious. An algorithm by which the ROM may be disassembled is given, but only guidelines are given as to how you may write a program for it. All of the arithmetic subroutines are explained in detail, even NEW ROM floating point functions like SIN and COS, and how the numbers are stored.

The heavily tabulated appendices at the back are designed to be used as a source of reference throughout the book. Any piece of information you need to know can generally be found in these appendices, or in chapter eight, which is a kind of "catalogue" of machine code.

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

CHAPTER TWO

INTRODUCTION TO HEXADECIMAL AND MACHINE CODE



OK, so your ZX80/81 is all fired up and ready, and that ominous inverse-K is sitting there glaring at you from its little corner and waiting for you to type something in. What do you do?

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

If you have a NEW ROM machine type one of the following sequences, depending on how much memory you have:

1K	4K	16K
POKE 16388,173	POKE 16388,32	POKE 16388,48
POKE 16389,67	POKE 16389,78	POKE 16389,117
NEW	NEW	NEW

The effect of this is quite straightforward. The addresses 16388 and 16389 together hold a system variable called RAMTOP. It contains the address of the first byte which the computer cannot use - at least not for BASIC. Under ordinary circumstances this address is the one immediately after the last byte in memory, so that the whole of the memory is available for BASIC programming. What we have done is to alter that address, so that some of the memory is unavailable for BASIC, and becomes a safe place in which to store machine code.

If you have an OLD ROM machine, don't worry - you can still store machine code in spare areas of the memory, but you MUST NOT type NEW, or you will lose it all.

The best addresses in which to store machine code are best found by trial and error. We shall adopt the following standard addresses, which should work perfectly for all of the routines in this book:

OLD ROM 1K:	17225
NEW ROM 1K:	17325
4K:	20000
16K:	30000

Throughout the remainder of this book I shall use the address 30000. Please read this as one of the alternatives above if you have less than 16K.

OK:- Now we're ready to start. Type in the following BASIC program:

When you have typed this program in name it "HEXLD" and don't forget to SAVE it.

```
10 LET X=30000
20 LET A$=""
30 IF A$="" THEN INPUT A$
40 IF A$="S" THEN STOP
50 POKE X,16*CODE A$+CODE A$(2) -476
60 LET X=X+1
70 LET A$=A$(3 TO )
80 GO TO 30
```

(For the OLD ROM you must replace lines 50 and 70 as follows:)

```
50 POKE X,16*CODE(A$)+CODE(TL$(A$))+36
70 LET A$=TL$(TL$(A$))
```

Can you see how the program works? Or at least what it does? In brief - it will accept a machine code program, & will store it at addresses 30000 onwards. (Or 20000, or whatever.) The program will stop when you input an "S". Note that although it will enter machine code, it will NOT attempt to run it.

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

Unlike BASIC, however, the computer isn't programmed to understand all of the various instructions as we do. If you were to RUN the above program and enter "RET" then this simply would not make sense to the poor old ZX81 (or '80). To make life easier for it, every instruction has a numerical code, which it DOES understand directly. For example the code for RET is 201. Every code lies somewhere in the range 0-255, and it is usually more convenient to write these codes in a system called HEXADECIMAL.

COUNTING IN HEXADECIMAL

Our friend Mr. Sinclair briefly covers this obscure system of counting in the ZX81 instruction manual by describing an imaginary race of sixteen fingered "Martians" who would regard counting in tens as being equally absurd. In these modern days of science we know enough about Mars to realise that it is extremely unlikely to host sixteen fingered people, but the principle of counting in sixteens is still very very sound.

Briefly, for those who have not read the ZX81 manual, hexadecimal, or hex for short, is a means of counting which uses sixteen symbols instead of ten. The first ten symbols are the same as the ones we're used to. These are:

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

There are six new symbols which represent the numbers 10 to 15. These are:

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

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

A complete table of all of the numbers from 0 to 255 is shown here. This is intended to help you to understand the hexadecimal system of counting. You should try to refer to it as little as possible, but don't worry if you find yourself using it all the time at first, you'll find you get used to it much quicker than you expect.

The symbols down the left hand side are the first hex digit, the symbols along the top are the second digit. The leading zeros may of course be ommitted if there are any, but it is sometimes more convenient to leave hex codes as two digits rather than one.

If there is ever any confusion about whether a number is written in hex or not, you should make it clear by writing a small letter h (standing for hex) or a small letter d (for decimal) after the number, so that 19h means twenty-five, and 19d means nineteen. Usually you wont need to do this because numbers like CD can only possibly be hexadecimal, and numbers like 118, which are three digits long, can only be in decimal. (Computing does not use hex numbers which are three digits long, though it does use ones which are FOUR digits long).

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

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
2	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
3	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
4	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
5	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
6	96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111
7	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127
8	128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143
9	144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159
A	160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175
B	176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191
C	192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207
D	208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223
E	224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239
F	240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255

There are fundamental differences between machine code programming and BASIC programming. One of the most fundamental differences is that of LINE NUMBERS.

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

In machine code, there are no line numbers. Also, the ZX80/81 will not allow you to use machine code instructions as commands, they MUST form part of a program. The instructions are executed in the order that they are stored. For example, if the computer had just finished executing the instruction which was stored in location 30000, it would then go on to execute the instruction held in location 30001. It will continue in this way until it recieved an instruction telling it to do otherwise.

Unlike BASIC, it will NOT automaticly stop when it reaches the end of the program. It will plough right on through the addresses, and every time it finds a number which is not zero it will simply treat that number as a code for some instruction and try to execute it. Usually this will result in what is called a CRASH.

ABOUT CRASHING

Crashing is the name we give to what happens when your (up until now at least moderately well-behaved) Sinclair machine unwittingly tries to execute something it shouldn't, or if there is a drastic mistake in your machine-coding which will confuse the poor machine and give it a rather nasty headache. The effect of a crash is very unmistakable- The screen will either go blank or will go into its "LET'S-PRODUCE-SOME-MODERN-ART" mode. If this happens you will get pretty (or otherwise) patterns on your TV not unlike those produced during SAVE.

When this happens you will undoubtedly try to break out, by using the BREAK key, and will discover to your horror that the BREAK key doesn't work! In fact NONE of the keys will work after a crash, except possibly to produce slight variations in the TV picture. This is the prime reason why we dislike crashes, for THE ONLY WAY TO THEN GET BACK TC NORMAL IS TO DISCONNECT THE POWER SUPPLY! When you reconnect you will of course have lost all of your program and will have to reLOAD it.

If a BASIC program contains a mistake it will usually NOT WORK.

If a machine-code program contains a mistake it will usually CRASH!

HOW TO PREVENT CRASHES

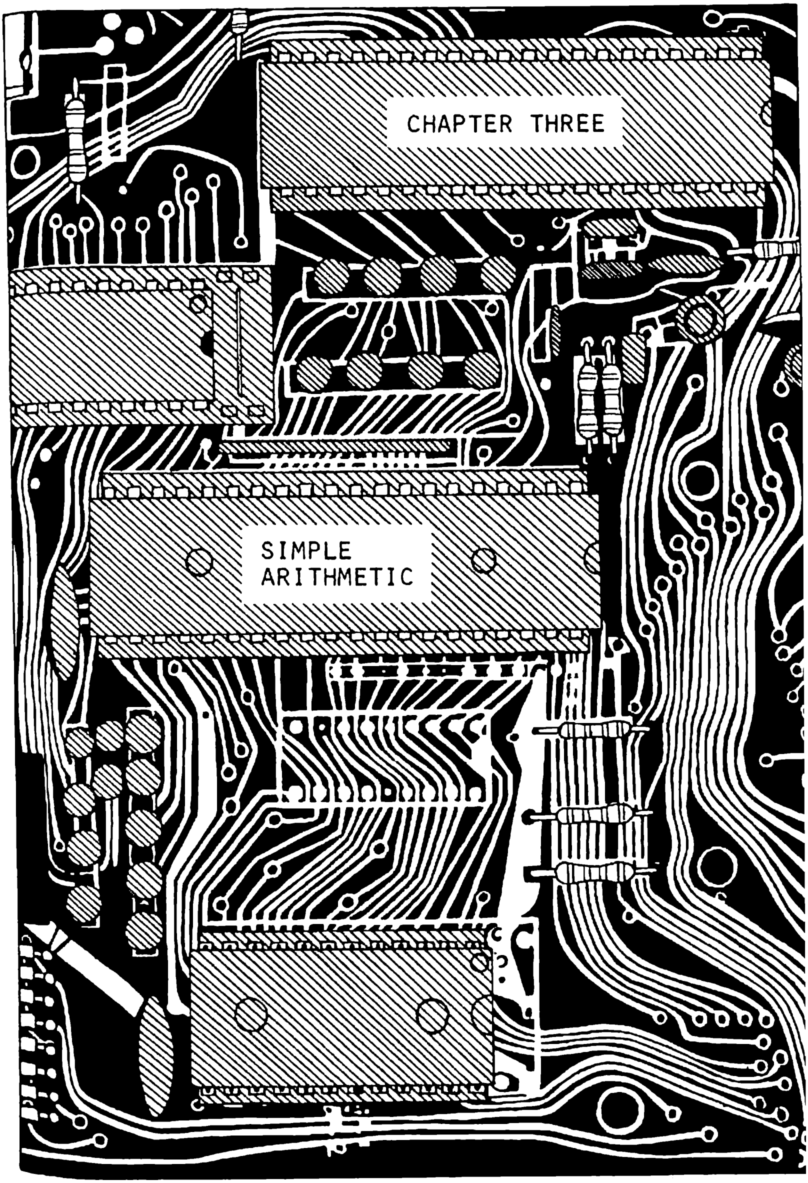
We have already stated that a machine code program will not automaticly stop at the end of a program - it must be told to do so by a specific instruction. For The ZX80/81, that instruction is RET. (Return - ie return to BASIC).

There is an instruction similar to STOP in BASIC, that instruction is HALT. DO NOT USE THIS INSTRUCTION! On other computers you can use HALT to end a program, but not on the ZX's. HALT produces a condition similar to a crash, for it means "Do nothing whatsoever until somebody breaks out." The problem is of course that you CAN'T break out because you'll find that the keyboard no longer works. To summarise: To end a machine code program ALWAYS use RET. NEVER use HALT.

A program must have at least one return instruction in it somewhere, otherwise it will never return to BASIC, unless you actually disconnect the power supply, and this is not usually a desirable thing to do.

This chapter has dealt with how to reserve space for machine code programs, and has given you a program with which to load it. It has not told you how to make use of this program, nor has it explained how to run machine code programs once they have been loaded. The fundamentals of counting in hex have been introduced, and the notion of a crash has been mentioned.

Once you have understood this chapter, you may turn to chapter three for your first lesson in machine language programming.


CHAPTER THREE
SIMPLE ARITHMETIC

"HEXLD" REVISITED

You remember the program I asked you to save in chapter two? Well now it' time to break it out, wipe the dust from it, and after you've reserved yourself some machine code space as described at the start of the previous chapter, you can LOAD it.

Now press RUN, and newline.

The program is waiting for a string input. What it in fact wants is some kind of HEXADECIMAL input. This means that every time you want to input a machine language instruction you have to know its numerical code, and you have to know it in hex.

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

Input C9. You have now told the computer that the first instruction of you machine language program is RET.

The computer is now waiting for another input. Break out of the program by inputting "S".

Your program is now complete. It consists of the single instruction RET. This is usually written

```
C9          RET
```

to remind you that the hex-code for RET is C9. The machine language instuctions are sometimes called OPCODES to distinguish them from their corresponding HEX-CODES. C9 is a hex-code, RET is an Opcode. Hex-codes are used by the machine - it will not understand opcodes. Conversely, opcodes are used by humans, because we would find it extremely difficult to work in hex-codes.

If you now look at the screen you'll see that the computer has gone back to command mode. It is waiting for an instruction. Suppose we now wish to run the machine code program that we've just typed in. We can do this either as part of a BASIC program, or, as we are going to do, as a direct command. If your routine was loaded to address 30000 then the command is

```
PRINT USR 30000
```

If your routine began at some other address simply use this figure instead of the 30000 in the above command. Note that OLD ROM users will need brackets around the number following the word USR.

You will have found that the computer has printed 30000 in the top left hand corner of the screen. Can you see why this is so? It started off with the number 30000 - this is the address you gave it when you typed PRINT USR 30000. The program told it to RET, or return to BASIC, having done nothing at all to this number, so that's exactly what it did - it returned to BASIC and it returned the number 30000 with it.

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

There are seven registers which are most commonly used for machine code routines. Their names are A, B, C, D, E, H, L. To give a larger degree of flexibility it is also possible to use the registers in pairs. When this is done you can alternatively store numbers either in the range -32768 to 32767 or in the range 0 to 65535, using the register-pairs, as they are known, BC, DE, and HL.

To make this clear, if register H contains the value 2, and register L contains the value 23, the register-pair HL is said to contain the value 2*256+23, which is 535. If H were to contain a value of 128 or more, then HL could instead be thought of as containing a negative value, equal to (H-256)*256 L.

THE INSTRUCTION LD

Consider the BASIC instruction LET A=42. In machine language we assign varibles (registers) using the instruction LD. We could, for example write LD A, 42. Note there is no equals symbol as there is in BASIC, instead a comma (,) is used to separate the A from the number. The effect of this instruction is exactly what you'd expect it to be - the previous value of A is overwritten, and a new value, in this case 42, is assigned in its place.

Each different LD instruction has a different code. For example the code for LD A, is 3E. The number 42 is 2A in hex, so the full instruction in hex is 3E2A. Note that this is TWO BYTES in length (every two hex digits is one byte). Compare this with the number of bytes in the BASIC instruction LET A=42.

The remaining codes are as follows:

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

Using the program "HEXLD" enter the following program, by inputting the symbols in the left hand column. Once the whole program has been entered, break out by inputting "S".

```
2600        LD H,00h
2E2A        LD L,2Ah
C9          RET
```

Now that the program is loaded you can run it by typing as a direct command PRINT USR 30000. What happens?

Now try entering this program:

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

If you possess an OLD ROM then the first program should return a value of forty-two, and the second program should return a value of 30000. However the NEW ROM will work the other way round, and return 30000 for the first program, and forty-two for the second. The reason is the fact that USR works differently for the two ROMs. For the OLD ROM, USR something means load HL with that something and then run the machine code. On the NEW ROM it means load BC with that something before running the machine code. When

BASIC returns the number you are left with is the value of HL (OLD ROM) or BC (NEW ROM). The first program leaves BC unchanged (on the NEW ROM it will have been assigned 30000) but will load HL with 42. The OLD ROM will return HL (42) and the NEW ROM will return BC (30000). The second program is the reverse. It will leave HL unchanged. (On the OLD ROM HL will have been assigned 30000) BC will then be loaded with 42. Which ROM will return which number? Which ROM do you have? Try it and see.

HL, by the way, stands for High/Low. Because any number in HL is stored in two parts the part that is stored in H is called the HIGH part, and the part that is stored in L is called the LOW part. BC and DE also have high and low parts, with the first letter for the high part, and the second letter for the low part.

What is 42 in hexadecimal to FOUR digits? Answer:- 002A. What do you think the following program will do? Try it and find out.

OLD ROM	NEW ROM
21002A	01002A
C9	C9

You may be surprized to discover that when you type PRINT USR 30000 to run it you get the answer 10752 - NOT 42! Now run this program:

OLD ROM	NEW ROM
212A00	012A00
C9	C9

NOW you will get 42. Notice the way the 2A and the 00 have been swapped around. Although this is rather strange it is in fact USUAL for the ZX80/81 to think of its numbers as having the low part FIRST, and the high part SECOND. In fact with the exception of line numbers, and in FOR/NEXT loops the ZX80/81 will always store its numbers "the wrong way around." In the instruction LD HL, the first byte is always 21h. The second byte is the new value of L, and the last byte is the new value of H. Note that this is always three bytes long.

To summarise: The LD instructions which operate on register pairs, rather than on single registers, use values stored "the wrong way round."

LDing From One Variable To Another

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

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

In the above table you read the left-hand-column registers first, and the top-row registers second, so that the code for LD D,A is 57, and the code for LD A,D is 7A. Notice how each of these is a mere ONE BYTE in length. Compare this with the equivalent BASIC instruction LET A=D, which takes a total of ten bytes in all (eight on the old ROM) if you include the line number, the line length, and the end of line character.

And now for some simple arithmetic. Those of you who have been thinking ahead may have been wondering how we can add and subtract registers like we can in BASIC. After all, the single-byte representation of LD A,B, for example, doesn't leave a lot of room for manoeuvre.

In fact, we use a different instruction altogether to add registers together. The instruction is ADD. You can think of an ADD instruction as being a LET statement with an expression involving "plus" on the right hand side of the equals. A useful example would be

```
                                ADD HL,DE
which has the effect            LET HL=HL+DE
```

The instruction ADD HL,DE will take the contents of the register-pair DE, and will add this number to the contents of register-pair HL. The result of this calculatio will then be stored in register-pair HL. As you can see, if we were working in BASIC and we were dealing in variables instead of register-pairs then we would have performed the operation LET HL=HL+DE.

Well almost, but not quite. There is in fact one small difference - the difference is what happens when you get what is called an overflow. You see register pairs can store all of the (hexadecimal) numbers between 0000 and FFFF. Those from 0000 to 7FFF are the integers 0 to 32767 in decimal, those from 8000 to FFFF can either represent numbers from 32768 to 65535, or numbers in the range -32768 to -1. You can use either form, but when the USR function returns a decimal number to BASIC the OLD ROM will use -32768 to 32767 and the NEW ROM will return a number between 0 and 65535. An OVERFLOW is what happens when you go beyond these ranges. In BASIC any overflow will simply stop the program and give you an error message. What do you suppose will happen in machine code?

OLD ROM first then: the BASIC for the OLD ROM deals with numbers from -32768 to 32767. What is the number 32767 in hexadecimal? Dividing by 256 to split it into two bytes gives 127 remainder 255, so the first byte is 127 (7F) and the second byte is 255 (FF). Now enter this program:

<u>OLD ROM ONLY:</u>

```
110100      LD DE,1
21FF7F      LD HL,32767
19          ADD HL,DE
C9          RET
```

The program will simply attempt to add one to the number 32767. Run it (using the direct command PRINT USR(30000)) and the result may astonish you. By the way, did you notice how the 00 and 01, and also the 7F and FF, had been swopped around in the above listing? You must always remember to do this in machine code. Did you notice also that the code for adding the registers (ADD HL,DE) was only one byte long? In fact the byte 19h. All of the ADD codes are one byte in length.

If you want to add one to BC for instance then you must do something like this

```
210100      LD HL,1
09          ADD HL,BC
44          LD B,H
4D          LD C,L
```

Notice how B and C have to be loaded sepatately since there is no such instruction as LD BC,HL. If you have a NEW ROM and you want to see what happens on an overflow load and run this program:

NEW ROM ONLY:		
	210100	LD HL,1
	01FFFF	LD BC,65535
	09	ADD HL,BC
	44	LD B,H
	4D	LD C,L
	C9	RET

Another thing you should notice is that only register-pairs may be added to register pairs, and that only single-registers may be added to single-registers. You may NOT add a single-register to a register pair, or vice versa. ADD A,HL is WRONG.

ADD HL,BC	09	ADD A,A	87
ADD HL,DE	19	ADD A,B	80
ADD HL,HL	29	ADD A,C	81
		ADD A,D	82
		ADD A,E	83
		ADD A,H	84
		ADD A,L	85

If overflowing register-PAIRS had you thinking, then think about overflowing SINGLE registers, for they can only hold numbers from 0 to 255. What happens when they overflow? Well yes, they simply start again at zero, but the question is can we do anything about this? In fact we can. Whenever we add two numbers, sometimes there is an overflow, or CARRY, and sometimes there isnt. The computer sets aside a NEW register, called F (which we cannot use directly) to store various bits of information. One of these bits of information is called the CARRY BIT.

An ADD instruction will always reassign the CARRY BIT. If there is no carry, it will be set to zero. If there is a carry, it will be set to one. We can use the value of the CARRY BIT by using the machine code instruction ADC, which means "ADD with CARRY".

It works like this. Suppose the machine comes across the instruction ADC A,B. It will take the contents of register B, and it will add the contents of register A, as in the previous instruction ADD A,B, and then it will add the CARRY BIT to this new number. Having done this it will store the result in register A, overflowing if necessary. The carry bit will always be reassigned to either zero or one, depending on whether or not there is an overflow.

So	ADD A,B	effectively means	LET A=A+B
		followed by	LET CARRY=INT((A+B)/256)
wheras	ADC A,B	effectively means	LET A=A+B+CARRY
		followed by	LET CARRY=INT((A+B+CARRY)/256)

Study the programs that follow. If the value of the A register is irrelevant, then are these programs equivalent (ie do they both do the same thing?) or not? Can you understand why?

The first program is

```
118533      LD DE,13189
21C77B      LD HL,31687
19          ADD HL,DE
(44         LD B,H )
(4D         LD C,L ) NEW ROM only
C9          RET
```

and the second program is

```
1633        LD D,51
1E85        LD E,133
267B        LD H,123
2EC7        LD L,199
7D          LD A,L
83          ADD A,E
6F          LD L,A
7C          LD A,H
8A          ADC A,D
67          LD H,A
(44         LD B,H)  NEW ROM only
(4D         LD C,L)
C9          RET
```

In actual fact they are exactly the same. You can learn two things from this; firstly that the instruction LD does not in any way affect or alter the value of CARRY, for if it did the two LD instructions between ADD A,E and ADC A,D would really mess things up; secondly that the instruction ADD HL,DE is much shorter, and much neater, than going all round the houses by adding each byte separately. And never forget to swop the order of the bytes round in LD instructi on pairs - compare the first two lines of program one with the first four lines of program two.

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

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

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

Notice how the codes for ADC HL, are all TWO bytes long, rather than one. The first byte is ED, and the second byte depends on what you are adding. Do not think of ED as meaning ADC HL, though, since it may have many other possible meanings as well, depending on what follows it.

ADDING CONSTANTS

We can also use the ADD and ADC instructions to add numerical constants directly to the A register. An example would be ADD A,3 which would, as you'd expect, add three to the current value of A. It would also assign CARRY to one or zero, depending on whether or not this addition caused A to overflow beyond 255.

The code for ADD A, is C6, and the code for ADC A, is CE. Note that we cannot add constants to any register other than A.

Suppose we wished to add 57 to HL. One way would be as follows:

```
113900      LD DE,57d
19          ADD HL,DE
```

but this method has the disadvantage that it requires the use of DE, which may be needed for other things. Another way of achieving the same thing, but this time only bringing the A register into use, is thus:

```
7D          LD A,L
C639        ADD A,57d
6F          LD L,A
7C          LD A,H
CE00        ADC A,0
67          LD H,A
```

Notice how the instruction ADC A,0 was used to add any carry digit there may have been from adding 57 to L.

AND FINALLY....

There is one more way that we can add constants to a register, and that is by using the instruction INC.

INC A means add one to the value of A. Unlike ADD, INC may be used on ANY register, so statements like INC D (add one to the value of D) or INC DE (add one to the value of register-pair DE) are allowed.

If A contained the value 255, then INC A will set A to zero, but WITHOUT setting CARRY equal to one. In fact INC will not alter the value of CARRY at all. If it was one before an INC instruction, it will be one after such an instruction. It it was zero before an INC, it will be zero after an INC.

In short:

INC B is equivalent to LET B=B+1

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

Remember, the difference between ADD A,1 and INC A is that ADD A,1 will assign a new value to CARRY, whereas INC A will leave it unaltered. INC, by the way, is short for INCREMENT.

The value of CARRY can be altered directly without any of the other registers being affected. There is an instruction SCF, which stands for SET CARRY FLAG, and its job is to assign to CARRY a value of one. The code for this instruction is 37h. Alternatively, it is possible to reset CARRY to zero, although there is no specific instruction to do this. One way would be to say ADD A,0 for example. Adding zero will of course leave the value of A unchanged, but an ADD instruction will always reassign CARRY.

CARRY is called a FLAG rather than a register, because it can only store the numbers one and zero. It is not possible to assign a value of two to CARRY, nor any other number in fact, only one and zero.

There is one other way to directly change the value of the carry flag, that is by using the instruction CCF, which stands for COMPLEMENT CARRY FLAG. It will change the value of CARRY from one to zero, or from zero to one. In BASIC terms these three instructions may be listed thus.

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

SUBTRACTION

In machine language, there are codes for subtraction, which are used in exactly the same way as the addition codes. The instruction is SUB, for SUBTRACT, and in exactly the same way as ADD, there is also an instruction SBC, for SUBTRACT WITH CARRY.

It works like this. SUB A,B will take the value of register B, and will subtract it from the value of register A. The result of this calculation is stored in register A. The carry flag is reassigned to zero if there is no overflow, or to one if the result overflows to below zero (in which case the value of A will have 256 added to it.)

SUB A,B may also be written as simply SUB B, because it is only the A register which may have things subtracted from it. Do not get confused by this convention - the two terms mean exactly the same thing.

The codes for SUB are:

SUB A,A	97
SUB A,B	90
SUB A,C	91
SUB A,D	92
SUB A,E	93
SUB A,H	94
SUB A,L	95

It is also possible to subtract numerical constants from the A register. For example the instruction SUB A,100 will subtract 100 from the number stored in register A. The result is stored in register A, and the carry flag is re-assigned to zero if there is no overflow, or to one if there is an overflow. The code for subtracting constants is D6, so that SUB A,100 is D664 (since 100 is written as 64 in hexadecimal)

SUB A,	D6

You should note the fact that although there are instructions such as ADD HL,BC, there are NO instructions to subtract register-pairs.

SUBTRACT WITH CARRY (SBC) on the other hand, WILL work for register pairs, but as with ADD and ADC, only the value of HL may be altered. For single registers it is only the value of A that may be changed.

SBC A,C will subtract the value of C from the value of A, and will then subtract the value of CARRY from this result. The final answer will be stored in register A. CARRY will be reassigned as before.

The codes for SBC are:

SBC HL,BC	ED42	SBC A,A	9F
SBC HL,DE	ED52	SBC A,B	98
SBC HL,HL	ED62	SBC A,C	99
		SBC A,D	9A
		SBC A,E	9B
		SBC A,H	9C
		SBC A,L	9D

To SUBTRACT WITH CARRY a numerical constant from the A register the code is DE followed by the number itself in hex. What is the code for SBC A,200? What precisely does this instruction do?

DEC is short for DECREMENT. It is, as you may have gathered from its wierd sounding name, the opposite of INC (Increment). Its pupose is to decrease the value of any register by one without changing the value of the carry flag. So DEC DE has the effect of LET DE=DE-1, remembering of course that if you decrement zero you get 255.

Compare these two routines:

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

and

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

Are they the same? And if not, why not? One of these two routines will subtract two from A, and will subtract DE from HL - The other routine is wrong. Which is which?

In fact it is the first example which is wrong. The instruction SBC HL,DE will subtract both DE and the carry flag, so the carry flag must first be reset to zero. This is what ADD A,0 is for. But having done that, the first example will alter the carry flag AGAIN with the instruction SUB A,2. The chances are that it will be reset to zero, but if A happens to equal one or zero then the SUB will not only change A to 255 or 254, it will also set the carry flag to ONE. So that the effect of SBC HL,DE would then be to assign HL a value of HL-DE-1, NOT HL-DE. In the second example, the instruction DEC A is used twice. DEC will not change the carry-flag, so it will still be zero when the instruction SBC HL,DE is reached, and the subtraction will then go ahead correctly.

Got it? INC and DEC do not alter the value of the carry flag - the other arithmetic instructions do. The other instructions we've covered are RET and LD. Neither of these will alter CARRY at all.

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

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

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

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

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

```
LD BC,0
LD HL,0
ADD HL,BC
(LD B,H)
(LD C,L)-NEW ROM only
RET
```

Now use the direct command PRINT USR 30000 to run it. What did you get? If you got zero well done. If, on the other hand, you got -31004 or 34532 then you did something fundamentally wrong. The instructions LD BC, and LD HL, both need THREE bytes altogether to make them work, not two. What

instructions did you really give the computer to make it come up with -31004 or 34532? And how exactly did it arrive at that answer? Now try again until you get zero.

Delete HEXLD by typing NEW (or on the old ROM by deleting each line individually) The machine code program will STILL BE THERE. Type in the following BASIC program:

```
 10 INPUT A
 20 INPUT B
 30 POKE 30001,A-INT(A/256)*256
 40 POKE 30002,INT(A/256)
 50 POKE 30004,B-INT(B/256)*256
 60 POKE 30005,INT(B/256)
 70 PRINT A,B
 80 PRINT USR 30000
 90 PRINT
100 GO TO 10
```

This BASIC program will replace the second, third, fifth, and sixth bytes of the machine code routine by the values you input in lines 10 and 20. Run the program and input some values to see what happens. Try going outside the range -32768 to 32767.

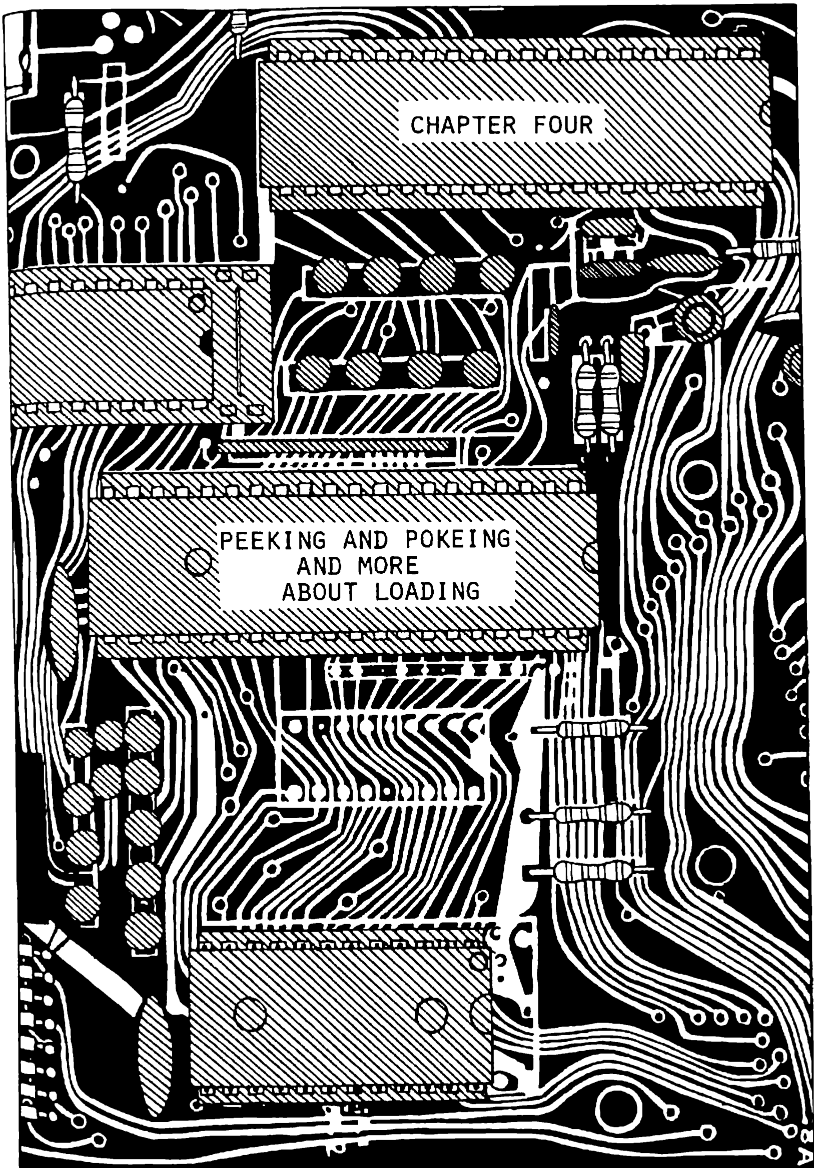
Now see if you can write a similar program, including a COMPLETELY NEW machine code routine, which will print a TABLE of values of A and B on the screen, and the result of <u>subtracting</u> A from B in each case. Let A and B both take on all of the values from 1 to 10 inclusive.

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

Write a short machine code routine which will set the carry flag equal to one, but without altering any of the registers. Do it WITHOUT using the instructions SCF, CCF, or ADD A,0.

CHAPTER FOUR

PEEKING AND POKEING AND MORE ABOUT LOADING



PEEKING AND POKING AND MORE ABOUT LD-ING

For those of you who thought maybe seven registers might not be enough, it's just as well we can PEEK and POKE, and thus make use of all the addresses in the RAM. (The RAM, which stands for Random Access Memory by the way, is the portion of memory which we are allowed to alter - the addresses numbered from 16384 upwards. The add-on 16K pack is RAM for instance.) If there's any number we have to store somewhere, either permanantly or temporarily, then it makes sense to just POKE that number somewhere - (almost anywhere will do) then when we need it again all we have to do is to PEEK at that address and voila - there it is!

A LESSON IN PEEKING

If you've ever seen any machine language printed anywhere, you may have wondered why obscure brackets kept turning up here and there. What, for example, is the difference between LD HL,16396, and LD HL,(16396)?

It's not just for variety, or to make it look pretty, they do actually mean something: brackets around a number or register-pair will refer to the contents of the ADDRESS in the brackets. So

	LD HL,16396	means	LET HL=16396
and	LD HL,(16396)	means	LET HL=PEEK 16396+256*PEEK 16397

The second example may have confused you. The only address in brackets is 16396, so how does 16397 come into it? What happened is a kind of side-effect. H and L can each hold ONE BYTE, so the pair HL stores TWO BYTES altogether. The address 16396 only holds ONE byte, so another one has to come in from somewhere. In practice this other byte comes from the next possible address, in the above case, 16397. The real effect of the instruction LD HL,(16396) is LET L=PEEK 16396, followed by LET H=PEEK 16397.

There is also a reverse instruction, which is

```
LD (16396),HL
```

This is effectively POKEing. The result of the instruction is

```
POKE 16396,HL-INT(HL/256)*256
POKE 16397,INT(HL/256)
```

or if you think of H and L separately:

```
POKE 16396,L
POKE 16397,H
```

In BASIC, this particular pair of instructions is used quite frequently. I'll give you an example. Suppose you've just written a BASIC program, and you want to know how long it is. You can find out the number of bytes your program occupies by using the expression PEEK 16404+256*PEEK 16405 to find the address of the END of your program (including the screen and all of your variables) and then subtract 16509 (the START of your program) from this number. There is a similar expression for the OLD ROM, which is PEEK(16394)+256*PEEK(16395)-16424. A very simple machine code program to calculate this value would be:

OLD ROM		NEW ROM	
112840	LD DE,16424	117D40	LD DE,16509
2A0A40	LD HL,(16394)	2A1440	LD HL,(16404)
C600	ADD A,0	C600	ADD A,0
ED52	SBC HL,DE	ED52	SBC HL,DE
C9	RET	44	LD B,H
		4D	LD C,L
		C9	RET

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

Notice how the hex-code for LD HL,(16404) is built up. The first byte is 2A. Now, although you're not expected to remember this, the last time we used a LD HL, instruction the code was 21 (hex). The difference is the BRACKETS! LD INSTRUCTIONS WHICH USE BRACKETS HAVE A COMPLETELY DIFFERENT HEX-CODE. The next two bytes are 14h and 40h:- this is the number 16404 in hexadecimal - if you divide 16404 by 256 you get sixty-four (40h) remainder twenty (14h). In the HEX-CODE these two bytes have been switched around to give 1440 rather than 4014. You must always remember to do this in machine code.

If you store this machine code program above RAMTOP (This is something that only NEW ROM users can do easily) as I've described then you can type in or LOAD any BASIC program and find its length in bytes simply by the by now familiar direct command PRINT USR 30000

16404 will ALWAYS contain the address of the end of all the variables in your program - this is its job. It is one of the SYSTEM VARIABLES which are used to help the ROM know what it is doing. If you alter this value by POKEing or LDing then the poor machine will get very confused, although, as we shall see later, this is sometimes an advantage.

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

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

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

AND POKEING:

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

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

Suppose you needed to know how far down the screen the PRINT position was. If you look in your instruction manual you'll find that PEEKing 16442 will tell you exactly that. (On the OLD ROM you'll need 16421 instead) The problem is to LD this into HL, because the number we're after is ONE BYTE long - it ISN'T stored in either 16441 or 16443 - and one way of doing it is this:

OLD ROM		NEW ROM	
2A2540	LD HL,(16421)	ED4B3A40	LD BC,(16442)
2600	LD H,0	0600	LD B,0
C9	RET	C9	RET

As you can see, the first instruction will successfully load the contents of 16421/16442 into the L or C register as required, but it will also load H or B with 16422/16443, so H or B must be reset to zero before we return to BASIC, otherwise the figure printed by the routine will be virtually meaningless.

The other way of getting PEEK 16442 into BC is to go via the A register, since this register can be LDed directly all by itself. But as you will see this offers no advantages, since we still have to reset B to zero anyway.

OLD ROM		NEW ROM	
3A2540	LD A,(16421)	3A3A40	LD A,(16442)
2600	LD H,0	0600	LD B,0
6F	LD L,A	4F	LD C,A
C9	RET	C9	RET

If you still aren't convinced that the second instruction is necessary try omitting it to see what happens. You'll find you get the number 29952 added to the real answer. Can you see why? You started off with the number 30000 and only altered the LOW part. The HIGH part was unchanged. (The HIGH part is INT(30000/256).) It happens to be 117. The factor of 29952 comes in because 117*256 is 29952.

Both of the above programs, as they are written, will have the same effect - they will tell you the line number of the PRINT position, that is, they will tell you how far down the screen the next character to be printed will be.

Try feeding in ONE of the above two programs, and then type in this BASIC program:

```
10 FOR I=0 TO 20
30 PRINT USR 30000
50 NEXT I
```

Remember, only NEW ROM users may type NEW without wiping out the machine code. Run it and see what happens. Now insert more lines.

```
20 FOR J=0 TO 3
30 PRINT TAB(8*J);USR 30000;
40 NEXT J
```

and again, RUN it and see what happens. OLD ROM users should replace the new line 30 by PRINT USR 30000, (ie with a comma at the end of the statement).

POKEING IN MACHINE CODE

POKEing is just as easy. To put line 50 of your BASIC program at the top of the screen at the next automatic listing you can POKE 16419,50. (On the OLD ROM it is POKE 16403,50.) You must make sure the ■ cursor is 50 or more first though. In machine code:

OLD ROM		NEW ROM	
3E32	LD A,50	3E32	LD A,50
321340	LD (16403),A	322340	LD (16419),A
C9	RET	C9	RET

Note that it doesn't actually matter what number returns to BASIC - (in actual fact it will be 30000) - the important thing is that the system variable called S-TOP (Screen Top) is POKEd with 50. That is what this program does.

Now look at the HEX-CODE of LD (16419),A. The first byte is 32h. This is the code for LD (pq),A, where pq represents some arbitrary address. The remainder of the code is 2340, which is the number 16419 in hexadecimal (with of course the first and last bytes switched around) So even though we humans would write our OPCODE with the (16419) first, and the ,A second, the machine language code always puts the instruction itself FIRST - despite the fact that the instruction itself actually incorporates the A at the end of the OPCODE. You must not put the 32h last, for the instruction 234032 would mean something totally different. In fact it would probably end up crashing, because it would take it to mean

23	INC HL
40	LD B,B
32	LD (????),A

With the (????) address made up of your next two bytes of machine code.

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

LD A,(BC)	0A	LET A=PEEK BC
LD A,(DE)	1A	LET A=PEEK DE
LD A,(HL)	7E	LET A=PEEK HL
LD B,(HL)	46	LET B=PEEK HL
LD C,(HL)	4E	LET C=PEEK HL
LD D,(HL)	56	LET D=PEEK HL
LD E,(HL)	5E	LET E=PEEK HL
LD H,(HL)	66	LET H=PEEK HL
LD L,(HL)	6E	LET L=PEEK HL
LD (BC),A	02	POKE BC,A
LD (DE),A	12	POKE DE,A
LD (HL),A	77	POKE HL,A
LD (HL),B	70	POKE HL,B
LD (HL),C	71	POKE HL,C
LD (HL),D	72	POKE HL,D
LD (HL),E	73	POKE HL,E
LD (HL),H	74	POKE HL,H
LD (HL),L	75	POKE HL,L

If you study the codes of the instructions that have (HL) in them you'll see that they form a regular pattern. In fact it looks very much like there ought to be an instruction LD (HL),(HL) with code 76 just to fill up a small hole in the regular pattern. In actual fact there is no such instruction, and code 76 corresponds to an instruction called HALT,

To demonstrate what I mean, here is a small table of all of the LD codes, which use registers A to L, and address (HL):

LD	B	C	D	E	H	L	(HL)	A
B	40	41	42	43	44	45	46	47
C	48	49	4A	4B	4C	4D	4E	4F
D	50	51	52	53	54	55	56	57
E	58	59	5A	5B	5C	5D	5E	5F
H	60	61	62	63	64	65	66	67
L	68	69	6A	6B	6C	6D	6E	6F
(HL)	70	71	72	73	74	75	—	77
A	78	79	7A	7B	7C	7D	7E	7F

Do you see what I mean about a regular pattern with LD (HL),(HL) missing? Of course, it's not an instruction you'll ever want to use, since it does absolutely nothing, but it's worth pointing out that you must never even ATTEMPT to use it becauuse, as I've said, 76 is the code for HALT.

Why is any variable in brackets a register pair rather than a single register? Why is any variable NOT in brackets a single register rather than a register pair? If HL contained a value of 16434, what is the difference between LD B,(HL) and LD BC,(16434)? What is the precise effect of each? See if you can write a program in machine language which will assign to HL a value of PEEK 16442 <u>ONLY</u>, using one of the LD ,(HL) instructions.

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

<u>HOW TO LOAD BLOCKS</u>

Loading BLOCKS means loading huge chunks of memory all in one go. For example, if you had a machine code routine stored beginning at location 30000 and you wanted to move it completely to location 20000, then if you were really really patient you could write a new machine code routine along the lines of

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

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

LDI	EDA0	POKE DE,PEEK HL LET HL=HL+1 LET DE=DE+1 LET BC=BC-1

The above program could therefore have been completely rewritten as

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

There is no list of variables after the opcode LDI, because the instruction will ALWAYS load from (HL) to (DE). You must not write LDI (DE),(HL) because this does not make sense. Further, it is impossible to load in this manner in any other combination. Loading from (HL) to (BC) for example simply cannot be done in a single instruction.

There is also an instruction LDD, or LOAD WITH DECREMENT, which has the same effect as LDI except that DE and HL are decremented and not incremented. Neither of these instructions, as with all LD instructions, will in any way alter the value of CARRY. The code for LDD is EDA8.

REPEATING THINGS

Even with LDI and LDD at our disposal, it would still be a very tedious affair to move something from, say, 30000 to 20000 if that something were around fifty bytes long. If it were a hundred we'd probably give up in dispair. Fortunately for us both LDI and LDD have a REPEAT facility. If, instead of writing LDI we wrote LDIR, with the extra R standing for REPEAT, then the instruction LDI would be carried out over and over again, and would not stop until the value of BC was zero. So if the routine we wanted to move was in fact 100 bytes long then we could move it using the routine

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

When the machine reaches the instruction LDIR, BC will contain a value of 100. After LDI had been carried out once, the first byte would have been transfered, DE would be increased to 20001, HL would be increased to 30001, and BC would be decreased to 99. After a second attempt, the second byte would have been transfered, and BC would contain a value 98. After LDI had been carried out one hundred times, the whole routine would have been successfully transferred, and BC would contain a value zero and so the program would continue with the next instruction. If this routine were the entire program then the next instruction should of course be RET.

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

LDI	EDA0
LDD	EDA8
LDIR	EDB0
LDDR	EDB8

I shall now give you a program which will enable you to SCROLL the screen BACKWARDS, so that the screen moves downwards, not upwards, and the print position is moved to the top of the screen. It will work on the OLD ROM provided 1)all twenty-two lines of the screen are full, ie contain thirty-two characters plus a newline character, 2)you do not attempt to PRINT anytning again (however you can alter the screen by POKEing the display file). It will work on the NEW ROM provided 1)RAMTOP is at least 19712 (effectively this means if you have 4K or more plugged in) 2)every time you use the statement SCROLL you fill the bottom line (for example by using the statement PRINT "thirty-two spaces", your next PRINT should be a PRINT AT.

A complete explanation of the program will also be given.

```
01D602      LD BC,726
2A0C40      LD HL,(16396)
09          ADD HL,BC
54          LD D,H
5D          LD E,L
01B502      LD BC,693
2A0C40      LD HL,(16396)
09          ADD HL,BC
EDB8        LDDR
C9          RET
```

The screen may now be scrolled BACKWARDS by using the NEW ROM statement PRINT AT USR 30000,0; On tne OLD ROM the corresponding statement is LET L USR(30000) but remember that on the OLD ROM once the screen is full you can only "PRINT" by POKEing into the display file. The machine code routine will leave a value of zero in BC (See the description of the last instruction, LDDR) so having executed the machine code it will then PRINT AT 0,0; ie it will move the NEW ROM print position to the top of the screen. This is precisely the opposite of SCROLL.

The first instruction is LD BC,726. This is the number of characters in the screen. There are twenty-two lines and each line contains tnirty-three characters (thirty-two plus one new-line character) hence the total number is 22*33=726. The address 16396 (together with 16397) contains the address of the START of the diplay file. (The first character in the display file is a new-line, so the screen itself actually starts one character further on.) This address is LDed into HL. Remember that LD HL,(16396) will load TWO bytes into HL, not one. The ADD instruction will then calculate the address of the LAST byte of the screen.

In order for LDDR to work, we need this address in DE, not in HL, and so since LD DE,HL is not a valid instruction it needs TWO instructions, LD D,H and LD E,L to accomplish this. We can now use HL for something else.

We need the address of what WILL BE the last character of the screen after we've finished scrolling (or antiscrolling if you want to call it that). Since it is the bottom line that will be lost, then this will be the last character of what is currently the TWENTY-FIRST line. So we need tne start address plus 21*33, or 693.

The next three instructions in the program: LD BC,693; LD HL,(16396); and ADD HL,BC will achieve this, and the result will be left in HL. This is precisely what we need for LDDR to work. LDDR will transfer from the address contained in HL to the address contained in DE, ie it will move the last character of the twenty-first line to the last character of the twenty-second line, before HL and DE are both decremented, or decreased by one.

How many times do we need to make such a transfer? We have to move twenty-one lines altogether, so we have to make sure that we do not use LDDR until BC contains a value of 21x33, or 693. As it happens, it already does, since we assigned it to 693 earlier on in the program. We may now quite happily use the instruction LDDR to BLOCK LOAD the first twenty-one lines of screen down to their new position occupying the LAST twenty-one lines of screen. Note that the old screen will be completely overwritten by the new screen with the exception of the first (top) line, which will be left unchanged. This is why the BASIC statement PRINT AT 0,0;"thirty-two spaces" is needed after every antiscroll.

The following NEW ROM BASIC program is designed to demonstrate the ANTISCROLL feature at work. It isn't a terrificly exciting game, or a pattern making artistic genius, or anything, but it will show you exactly what the machine code we've just been working on will do. You can of course insert the routine into any program - there are some graphics games which would be immensely enhanced by the ability to SCROLL in either direction. This program sets up a striped pattern accross the screen, with each stripe composed of a random character chosen from the whole ZX81 set. The pattern on the screen will then wait for you to tell it what to do. Pressing the "up" key will move the pattern upwards, and pressing the "down" key will move the pattern downwards. These are of course the standard cursor control keys I'm refering to, except that you don't need to use SHIFT.

The listing is written for both FAST and SLOW modes. In FAST, line 110 should read PAUSE 40000, but in SLOW it should be changed to IF INKEY$="" THEN GOTO 110. Otherwise enter the program as listed.

UP AND DOWN

```
 10 DIM A$(22,32)
 20 FOR I=0 TO 22
 30 LET B$=CHR$(63*RND+128*(RND<.5))
 40 FOR J=1 TO 5
 50 LET B$=B$+B$
 60 NEXT J
 70 LET A$(I)=B$
 80 PRINT A$(I)
 90 NEXT I
100 LET A=1
110 PAUSE 40000
120 LET B=A+1
130 IF B=23 THEN LET B=1
140 LET C=A-1
150 IF C=0 THEN LET C=22
160 LET B$=INKEY$
170 IF B$="6" THEN PRINT AT USR 30000,0;A$(C)
180 IF B$="7" THEN SCROLL
190 IF B$="7" THEN PRINT A$(B)
200 IF B$="6" THEN LET A=C
210 IF B$="7" THEN LET A=B
220 GOTO 110
```

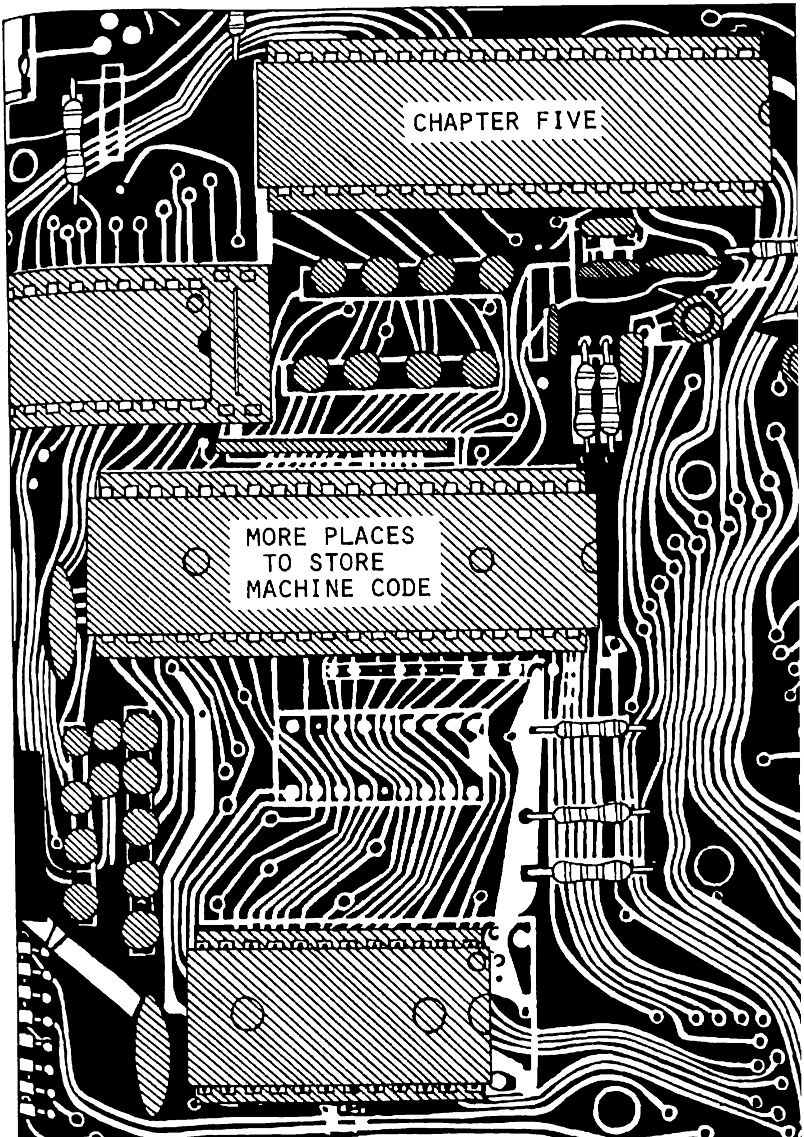
This chapter has tried to develop a deeper understanding of the LD instruction, and has explained how LD can be used to access the memory addresses of the computer. The specialised load instructions LDI (Load with Increment), LDD (Load with Decrement), LDIR (Load with Increment and Repeat, or BLOCK LOAD with Increment), LDDR (Load with Decrement and Repeat, or BLOCK LOAD with Decrement) have been covered.

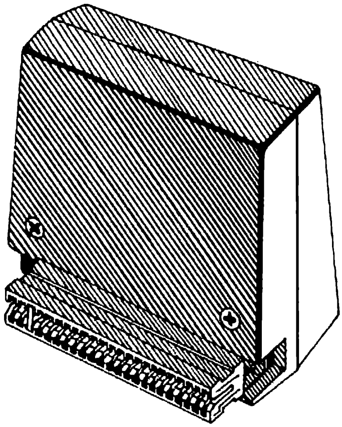
EXERCISES

Based on the Antiscroll program in this chapter, write a machine language program to SCROLL forwards, as the keyboard SCROLL does. (This exercise is especially useful if you do not have SCROLL on your keyboard.) Then see if you can write a machine language program which scrolls forward, but which will ONLY SCHOLL THE BOTTOM HALF OF THE SCREEN, so that the top ten lines are unaltered, the eleventh line is lost, and the twelfth to twenty first lines are all moved up one line.

Write a BASIC program making use of the routine. You will need the BASIC statement PRINT AT 21,0;"thirty-two spaces" every time the machine code routine is used. Try leaving this out just to see what happens.

If you can't cope with the challenge of writing such a SCROLL program, then I'll give you a hint or two. You will need to use LDIR instead of LDDR, otherwise all you'll get is a pretty pattern, and you'll need to start block loading at the BEGINNING of the screen, NOT the end. The instruction LD HL,(16396) will always give you the address at which the screen begins. Don't forget that a full line contains thirty-three characters, not thirty-two, since there is always a new-line character there as well.


CHAPTER FIVE
MORE PLACES
TO STORE
MACHINE CODE



SOME NEW PLACES TO STORE MACHINE CODE

Storing machine code above RAMTOP will protect it from being erased by NEW, or overwritten by a program, but it has the disadvantage that you can never save it. There are several alternative locations in which we can store machine language programs, and we shall explore a few of the possibilites in this chapter.

Using REM.
To store a machine language routine that is fifty bytes long, make the first line of your program

```
1 REM 12345678901234567890123456789012345678901234567890
```

is a REM statement with fifty characters after it. If your routine was sixty bytes long then you'd need sixty characters after the word REM. If it were only three bytes long you would only need three characters after the word REM. It doesn't actually matter what these characters actually are, but counting upwards in ones, as I have done, will ensure that you don't lose count halfway through. You will need to LOAD "HEXLD" before you add this new line one, and then change line 10 to

```
10 LET X=16514      (or 16427 on the OLD ROM)
```

OLD ROM users should ensure that line one does not appear on the automatic LISTing. You can use the command POKE 16403,10 to remove it. If this has no effect try moving the cursor to line 10 and try again.

NOW you can enter a machine code program exactly as before, except that to execute it you must say USR 16514 instead of USR 30000. On the OLD ROM you must say USR(16427). BUT you MUST NOT type NEW. Delete HEXLD by entering the line numbers one at a time, and do not delete line one! On the OLD ROM you must not even attempt to list line one or you may cause a crash.

Now there are two very important differences between using 16514 and using 30000. Firstly, SAVE will store the machine code as well as the BASIC program - this is something you cannot do in upper memory. Secondly, the command NEW will erase it. It is thus an integral part of the program, and can only be used with that one BASIC program and no other (unless you delete it line by line and then type in a new program line by line). If you have written a machine code routine specifically to accompany some BASIC program then this method is an obvious choice, but it does have one big disadvantage - on the OLD ROM the command LIST will usually cause a system crash.

There is another very very good place to store machine code, that is immediately after the program area. This has several advantages: 1) The BASIC surrounding program can be safely listed - even on the OLD ROM. 2) The MACHINE CODE can be SAVED. 3) Using RUN, as opposed to GOTO 1, will not wipe it out. To load a machine code routine that is, say, 20 bytes long, type the following BEFORE you type in any BASIC:

OLD ROM:	1 REM 45678901234567890
NEW ROM:	1 REM 678901234567890

Then as a direct command type:

OLD ROM:	POKE 16424,-1
NEW ROM:	POKE 16509,-1

You have now reserved a space of twenty bytes in which to store whatever machine code you like. The starting address is a little more complicated though - it is on the OLD ROM PEEK(16392)+256*PEEK(16393)-20, and on the NEW ROM PEEK 16396+256*PEEK 16397-20 . The PEEK expression is the end of the machine code, and the minus twenty is there to find the start. This is an excellent way of storing machine language routines. You begin loading it from address PEEK 16396+256*PEEK 16397-length-of-routine, and you can execute it with the expression USR (PEEK 16396+256*PEEK 16397-length-of-routine). First though, there is one disadvantage to get round. As I've explained things so far there is no way you can actually load an editing program like HEXLD: If you LOAD <u>before</u> you apply the above technique then HEXLD will disappear along with the REM statement as soon as you POKE 16509. If you try to LOAD <u>after</u> you've reserved a space then the very act of LOADing will overwrite this space.

Here then is a step by step method of reserving a space for machine code in a place that is 1)editable, 2)SAVEable, and 3)unLISTable.

<u>STEP ONE.</u> LOAD an editing program such as HEXLD.

<u>STEP TWO.</u> Add a new line at the END of the program: 9999 REM followed by a number of arbitrary characters. On the OLD ROM you'll need three characters less than the number of bytes in the machine code routine, on the new ROM you'll need five bytes less than the machine code. The best way of doing this is to fill the REM statement with digits, and simply start counting from 4 (OLD ROM) or 6 (NEW ROM). Like this - for a fifteen byte routine:

OLD ROM:	9999 REM 456789012345
NEW ROM:	9999 REM 6789012345

Of course it doesn't actually matter if you have too many characters, but it is a waste of space if you reserve area and then don't use it.

STEP THREE. Add the following lines anywhere in the program. I've put them at 9000, but it doesn't matter. If you use 8000 then just remember to read 8000 every time you see 9000 written on this page.

OLD ROM	9000 LET X=PEEK(16392)+256*PEEK(16393)
NEW ROM	9000 LET X=PEEK 16396+256*PEEK 16397
OLD ROM	9010 POKE X-(four more then the number of characters in the REM statement),-1
NEW ROM	9010 POKE X-(six more than the number of characters in the REM statement),-1
BOTH	9020 STOP

If you counted up to fifteen in line 9999 (as above) then 9010 should be POKE X-16,-1. If you counted up to twenty then line 9010 should instead be POKE X-21,-1, and so on. Remember though to start counting at four or six though, as above.

STEP FOUR. Run the program from line 9000, and then delete lines 9000, 9010, and 9020.

STEP FIVE. Replace all references to the machine-code-starting-address on your editing program by the expression PEEK 16396+256*PEEK 16397 minus the number you counted up to in the REM statement. OLD ROM users should instead use PEEK(16392)+256*PEEK(16393) minus the number you counted up to in the REM statement.

You are now complete. The only thing you must not do is type NEW, since this will erase the machine code. Other than that you are in complete command.

REM STATEMENTS

For the purposes of storing machine code, OLD and NEW ROM REM statements are completely different. Let's examine them one at a time. First of all for the Old ROM:

There are several important points about OLD ROM REM statements. Most people already know that a "blank" REM statement - that is a statement consisting of the work REM and nothing else - has the effect of ensuring that the next line is not executed. It is therefore the same as GOTO the-line-after-next, and can be used in BASIC programs deliberately with this meaning.

The biggest limitation of an OLD ROM REM statement is the fact that you may not store the byte 76 (hex) in the line, except in extremely limited cases, which I shall explain. The reason is that a character 76 is interpreted by the ROM as an end of line marker. The two bytes immediately after such a character will be interpreted as representing the line number of the next BASIC program line, and the following byte will be the first character in that line. Thus if the following data were POKEd into a REM statement in line one the following would happen:

DATA: 39 76 01 01 F8 E4 D5

```
RESULT:  1 REM T
       257 LET < THEN
         2 next line of program...
```

If you tried to RUN this program you would get a syntax error in "line 257". Typing RUN 2 would be useless, because the program searches for line numbers from top to bottom, and as soon as it hit the "line number" 257 it would think to itself "ah - there obviously isn't a line 2 in the program - I'll have to RUN it from here instead." The same applies to all GO TO's in the program which have destinations between 2 and 257. You must only allow 76's in your data IF the next two bytes form a "line number" less than the next line number in the program, and IF you never try to execute this "new line".

On the other hand - this treatment does offer one or two advantages. For instance, if you made your REM statement too long and you want to shorten it, if your machine code data ends at address A just type

```
POKE A+2,2
POKE A+1,0
POKE A,118
```

then simply delete "line 2" by typing in it's line number. It doesn't matter if there is already a line numbered 2 in the program - typing the line number alone will only delete the first "line 2" in the program - all your excess REM characters in other words.

Conversely, if you find you don't have enough characters after the word REM just type in a line 2 consisting of a second REM statement full of arbitrary characters. In this way as soon as the "real" end of line marker is over-writen line 2 will become part of line 1, with enough characters for whatever you need.

Alas, the NEW ROM does not fit any of these descriptions. NEW ROM REMs are quite, quite different.

The first, and most important differance, is that you can put character 76's into the REM data and the machine won't notice. BUT if you do so be prepared to be confused by the LISTing - even the ROM gets confused over it - but you don't need to worry because even with supposed new-line markers in mid-line the program will RUN quite smoothly, and will not interpret the remainder of the line as a different line.

On the other hand, it's a little more difficult to extend the length of a REM statement. If you want to overrun into line two you'll have to do some very clever POKEing first, but I'll explain how to get round that in a minute. The obvious way of making a line longer is simply to use EDIT and add more characters. Unfortunately for us this is usually not a very wise thing to do.

If the data in the line does not contain a byte 7E then by all means go ahead and use EDIT - you are quite safe, and nothing will go wrong.

If the data in the line does contain a byte 7E then DO NOT use EDIT. In the listing, a byte 7E is invisible, and the five bytes of data that follow immediately after it will also be invisible, but they are still there! If on the other hand you use EDIT, all six of these invisible bytes will simply vanish without a trace.

7E is used by Sinclair to mean "This is a (floating point) number". Whenever you use a decimal number in a program listing the ROM will automatically follow this number with a byte 7E, followed by five more bytes which contain the number itself in floating-point-binary-form. Both the byte 7E and the five bytes that follow will be invisible from the listing. This is what causes all the problems in editing REM statements. Now although I agree that this is a very very efficient means of storing floating point numbers in a program, it is also true that Sinclair Research could have used ANY byte for this purpose - they didn't specifically have to use 7E. It is of course the purest of coincidences that 7E happens to be one of the most commonly used machine language instructions of all.

The only practical means of adding more characters to a REM statement containing machine code on the NEW ROM is to let the data overrun into line two, but there are problems even there, thanks to our kind friends at Sinclair Research. You see the start of every line of program is preceeded by two invisible bytes which store the length of the line, so that even if you overwrite the end-of-line-marker, the ROM will still try to interpret the second line from the same point. To get round this you have to actually POKE these invisible bytes with different values. The following is a small routine which will enable you to increase the length of a REM statement at line one.

Step one is to insert a new line 2 to your BASIC program consisting of the word REM followed by a number of arbitrary characters. Then, at ANY point in the program insert the following five lines - (They will shortly be deleted anyway):

```
LET A=16515+PEEK 16511+256*PEEK 16512
LET A=A+PEEK A+256*PEEK (A+1)-16511
POKE 16511,A-256*INT (A/256)
POKE 16512,INT (A/256)
STOP
```

Simply run this routine and line 2 will automatically be a part of line 1. You can delete this routine now - its job has been done. LIST line one - you'll see that line two still looks quite separate, but try moving the cursor down - you'll find it skips over line two altogether. Try deleting line 2 by typing in its line number - it won't work because now the computer doesn't know that line 2 is there! Whatever the listing may <u>look</u> like, the ROM will now ignore line 2 altogether, taking it to be part of line one. You may now quite happily overwrite the end-of-line-marker at the end of line one with no ill effects.

Conversely, the following routine will shorten a REM statement by a <u>minimum</u> of six bytes.

```
LET A=the address of the last byte which you wish to preserve
        in the REM statement of line 1.
LET B=A-16511
LET C=PEEK 16511+256*PEEK 16512-B-4
POKE 16511,B-256*INT (B/256)
POKE 16512,INT (B/256)
POKE A+1,118
POKE A+2,0
POKE A+3,2
POKE A+4,C-256*INT (C/256)
POKE A+5,INT (C/256)
STOP
```

Again you simply RUN the routine once, and then delete it. Now LIST the program and you'll find a new line 2 has appeared. Delete this by typing its line number and your REM statement will now be as short as you need it.

USING THE VARIABLES AREA

Another place where machine code may be stored is in the variables area. To do this you must first of all reserve the space. To store a machine code routine of n bytes (n is the length) OLD ROM users should type DIM O(n/2), and NEW ROM users should type DIM O$(n). You may now write your machine code.

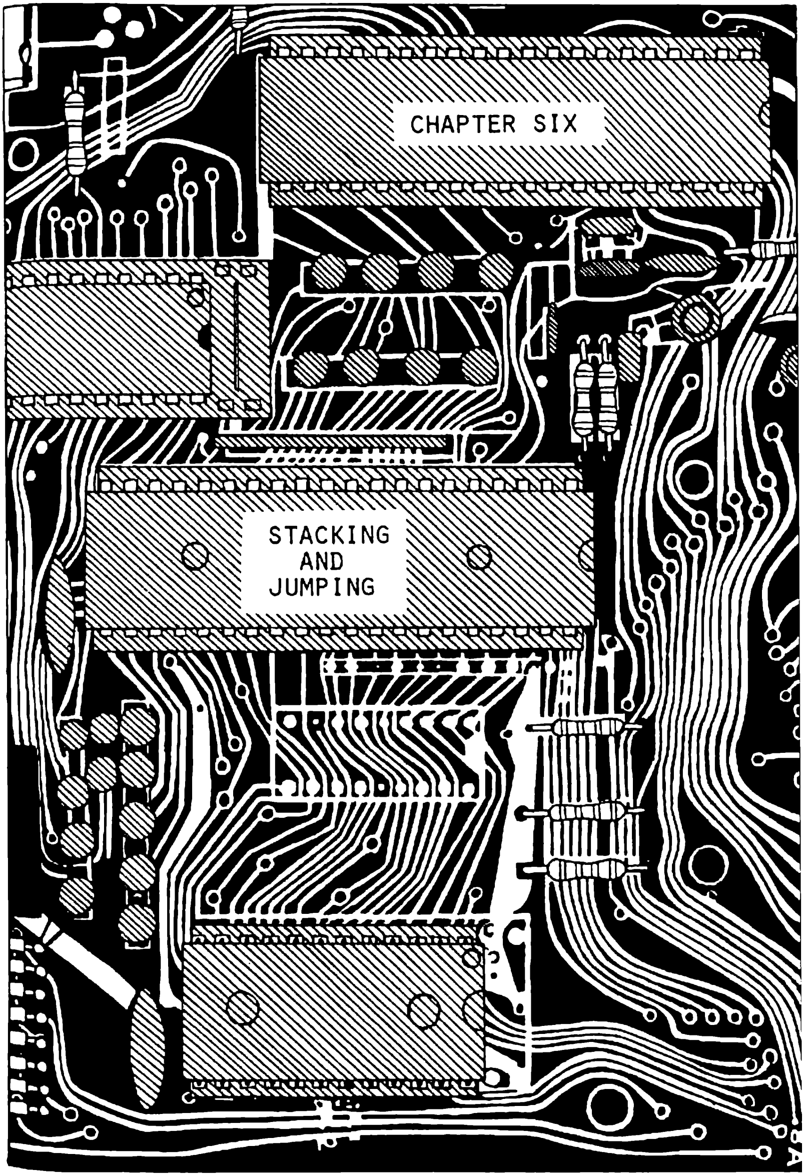
On the OLD ROM the starting address will be PEEK(16392)+256*PEEK(16393)+2, provided the array O is the first item in the variables area. This will be the case if the DIM was the first DIM, FOR, INPUT, or LET statement executed since the last time you used RUN or CLEAR. If you DIMensioned O as a direct command you should remember to type CLEAR first. You can say in your program something along the lines of LET A=PEEK(16392)+256*PEEK(16393)+2 right at the very start, and this value will not change throughout the program.

On the NEW ROM the starting address is PEEK 16400+256*PEEK 16401+6, provided the character array O$ is the first item in the variables area. This will be true if the DIM was the first DIM, FOR, INPUT, or LET statement executed since the last time you used RUN or CLEAR. You can dimension O$ as a direct command, but you must remember to type CLEAR first. There is however one big difference between the OLD and NEW ROMs here. On the NEW ROM the value PEEK 16400+256*PEEK 16401+6 will <u>change</u> during the running of your program if you have less than 3¼K plugged in. If you have more than 3¼K then you don't need to worry, but otherwise you must recalculate the expression every time you wish to access the machine code.

One last important point is that having stored machine-code in the variables area, any future use of either RUN or CLEAR will completely wipe it all out, never to be seen again. For this reason I do not advise using it for machine code storage. It WILL SAVE and RE-LOAD, again provided you never type RUN or CLEAR.

CHAPTER SIX

STACKING AND JUMPING



THE STACK

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

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

The memory locations in the stack are just like that. You can put things on top of it, but ONLY at the top, and you can take things FROM THE TOP. There are two special words that go with the stack - one word which means "stacking a new number onto the top", and a second word that means "removing a number from the top". The first word is PUSH, and the second word is POP, so if you PUSH the number five onto the stack, and then you PUSH the number one-thousand, and then you PUSH say 16426, the first number you can POP is 16426, because this number is at the top since it was put there last. The next number to be POPped will be 1000, and then five.

The stack is stored very very high in the address, so that there is less chance of programs "colliding" with the stack as either one or the other is built up. In the old ROM the bottom of the stack is at the very top of memory - 17407 for 1K, 20479 for 4K, and 32767 for 16K. In the new ROM the whole stack moves around - the bottom of the stack is at an address stored in one of the system variables - ERR-SP - to be found at 16386 and 16387. The stack is actually very peculiar, because it's UPSIDE DOWN. The BOTTOM of the stack is at the TOP of available memory, and the TOP of the stack is BELOW it! It turns out to be more efficient this way. It's not actually a deliberate plot to confuse the whole human race so that the world may be taken over by ZX computers, even if it does at times seem like it. So remember - the stack, or the MACHINE STACK as it's sometimes called, is like a stack of cardboard boxes piled up on a shop floor, except that in a daring feat of defiance of Newton's laws this stack instead decides to reside on the ceiling and build up downwards. The top - the only part you can easily get at - is lower down than the bottom!

The stack is so important to the computer that a special REGISTER is set aside just to store the position of the TOP of the stack. (The part with the lowest address - the part we can get to.) That register is called SP, which stands for STACK POINTER. It is actually a register-PAIR, because it can store two separate bytes, but unlike the other register-pairs BC, DE, and HL, we CANNOT treat the two halves independently - they just won't separate.

Here's how the instructions PUSH and POP work. Suppose HL contained a value 12345. This means that H contains a value of INT(12345/256), or 48, and L contains a value of 12345-256*INT(12345/256), or 57. Now the instruction PUSH HL would store the number 12345 at the top of the stack. It would do it by first of all stacking the HIGH part, and then stacking the LOW part. It would also alter the value of SP accordingly since two more bytes have been added to the stack, and the position of the top will therefore have moved (down) by two addresses.

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

PUSH can be thought of in BASIC as a sequence of three statements:

```
PUSH HL          POKE SP-1,H
                 POKE SP-2,L
                 LET SP=SP-2
```

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

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

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

PUSH

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

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

The register-pair AF, which cannot normally be used in this way, is made up of smaller single registers A and F, in the same way that BC is composed of B and C. A is the register which we've been using throughout the book so far, but F is something completely different. The F stands for FLAGS, and is so called because it stores the value of all the FLAGS used. (A FLAG is a memory that can only store zero or one). One of these FLAGS we've already seen - the CARRY flag. The F register has the capability to store eight flags altogether, but in fact only six of them are used. We shall see what these are, and how to use them, later on.

POP

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

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

One of the major uses of PUSH AF and POP AF is simply to put the value of A onto the stack. The fact that F has been stacked with it is irrelevant. PUSH AF will conveniently store the value of A until it's needed again, at which point its value may be recovered by the use of POP AF. This can be useful if you have to use the A register to perform calculations of some kind that couldn't be performed by any other register, but when the value of A will still be needed later on in the program.

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

```
F5               PUSH AF
78               LD A,B
C619             ADD A,25d
47               LD B,A
F1               POP AF
```

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

ALTERING SP

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

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

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

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

The final instruction INC SP was necessary in order to restore the Stack Pointer to its original value. If this is not done you may cause a pretty nasty crash.

SP is not the only very specialised register in use. There is another two byte register called PC, or PROGRAM COUNTER. Its job is to remember whereabouts we are in the program. Every time it has to execute an instruction it will take a look at what PC says. If it says 30004 then it will execute the instruction at location 30004, and then it will increment the value of PC by the number of bytes in that instruction, so that NEXT time round it will be looking at the next instruction in sequence. For example, if 30004 contained the instruction LD A,B then this would be carried out and PC would be increased to 30005. If the instruction at 30005 was LD A,2 then once this was carried out PC would be increased by TWO, since LD A,2 is a TWO-BYTE instruction. PC would then be reading 30007 where the next instruction begins.

If you alter the value of PC then the effect is like a BASIC GO TO. The only difference is that machine code does not use line numbers, so you have to GO TO the right ADDRESS rather that the right line number. The machine language instruction that does this job is JP, which of course is short for JUMP. JP 30000 means GO TO address 30000 and continue executing this machine code program from there. Of course all this instruction REALLY does is to load the number 30000 into register PC (but without incrementing it at the end of the instruction), so that it thinks 30000 is the next address in the program. It is far more useful for us human beings to think of it as kind of GO TO though, because that's what we're used to.

Be careful with JP though. If you create an infinite loop in machine code then TOUGH! You're stuck with it, and what's more you can never break out unless you actually switch the machine off at the mains. Some other computers will let you break out of machine code, but the ZX81 will not, neither will the ZX80. An example of an infinite loop would be

```
77          30000     LD (HL),A
23          30001     INC HL
C33075      30002     JP 30000
```

I've written the actual addresses in the middle column. Usually this isn't done, and important lines are marked with LABELS, or words which tell us which lines do what. These LABELS do not appear in the hex, and we only in fact write them for our own convenience. If for instance we decided to call the first line START then our pretty bad program could be written

```
77          START     LD (HL),A
23                    INC HL
C33075                JP START
```

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

It is also possible to jump backwards using JR, since there is a convention that any hex number greater than 7F will be treated as a negative number, obtained by subtracting 256 from the number it would normally represent. To make life easier I have included a second table of hexadecimal numbers, only this time using the negative sign convention.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
8	-128	-127	-126	-125	-124	-123	-122	-121	-120	-119	-118	-117	-116	-115	-114	-113
9	-112	-111	-110	-109	-108	-107	-106	-105	-104	-103	-102	-101	-100	-99	-98	-97
A	-96	-95	-94	-93	-92	-91	-90	-89	-88	-87	-86	-85	-84	-83	-82	-81
B	-80	-79	-78	-77	-76	-75	-74	-73	-72	-71	-70	-69	-68	-67	-66	-65
C	-64	-63	-62	-61	-60	-59	-58	-57	-56	-55	-54	-53	-52	-51	-50	-49
D	-48	-47	-46	-45	-44	-43	-42	-41	-40	-39	-38	-37	-36	-35	-34	-33
E	-32	-31	-30	-29	-28	-27	-26	-25	-24	-23	-22	-21	-20	-19	-18	-17
F	-16	-15	-14	-13	-12	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

Here the number -5 is represented in hex by FB, and so it is therefore possible to use the instruction JR -5, but note that because of this convention we are unable to say JR 129 for instance, because 129 in hex is 81, which would here be taken to mean -127, and would be a jump backwards. The range we are limited to is therefore from -128 to 127.

JR 0, as we have said, does absolutely nothing. It will continue with the next instruction. It is important to remember that all relative jumps are counted from the NEXT instruction. JR 0 means execute the NEXT PLUS ZERO instruction, JR 1 means execute the NEXT PLUS ONE instruction. Consequently if we were to say JR -2 then you must count backwards for two bytes, starting at zero with the NEXT instruction. You will find that two bytes leads you to exactly the instruction we have just executed - the instruction JR -2. JR -2 is therefore an infinate loop, and is not a recommended instruction to use in a program.

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

```
77          START    LD (HL),A
23                   INC HL
18FC                 JR -4      or JR START
```

You have probably by now realised that JP and JR are more or less useless on their own, in the same way that the BASIC statement GOTO would be useless if it weren't for IF/THEN statements and GOTO N. We need some kind of a CONDITIONAL jump, so that we can say IF some condition is true THEN jump to a new address pq, otherwise we are virtually certain to produce an infinite loop. Although machine language doesn't have quite the same kind of flexibility as an IF/THEN statement, there are four conditions we can check for using JR, and eight conditions we can check for using JP. These are:

JR e	18	JUMP RELATIVE by e bytes.
JR Z e	28	IF the last result calculated was zero then JUMP RELATIVE by e bytes.
JR NZ e	20	IF the last result calculated was non-zero then JUMP RELATIVE by e bytes.
JR C e	38	IF CARRY=1 THEN JUMP RELATIVE by e bytes.
JR NC e	30	IF CARRY=0 THEN JUMP RELATIVE by e bytes.

and for JP:

JP pq	C3	JUMP to address pq.
JP Z pq	CA	IF the last result calculated was zero THEN JUMP to pq.
JP NZ pq	C2	IF the last result calculated was non-zero THEN JUMP to pq.
JP C pq	DA	IF CARRY=1 THEN JUMP to pq.
JP NC pq	D2	IF CARRY=0 THEN JUMP to pq.
JP PE pq	EA	see below.
JP PO pq	E2	see below.
JP M pq	FA	IF the last result calculated was negative (Minus) THEN JUMP to pq.
JP P pq	F2	IF the last result calculated was positive (Plus) THEN JUMP to pq.

Now although this is a far cry from IF A$ "HELLO" THEN PRINT "GOODBYE" as you're used to, you'll soon see that even this horrendous task may be evaluated in machine code. First though I think I ought to explain about the instructions JP PE and JP PO. The P actually stands for PARITY, and the E and O mean Even and Odd. What we are doing is testing one of the flags - a flag called P/V. It's not all that difficult to understand - it works like this.

P/V stands for Parity/Overflow. V stands for Overflow because O is too confusing - it could mean zero or it could mean Odd (as in JP PO), so in their wisdom, and expertise at spelling, the computing bods decided to call it V. The P/V flag is a rather overworked little beast because it does two jobs at once. The first job is to check the PARITY of the last result calculated. This means you simply count the number of 1's (or 0's) in the binary form of the last result. (The binary form is always written to eight digits even if this means adding several leading zeroes.) If the number of 1's is ODD then the Parity is ODD. If the number of 1's is EVEN then the parity is EVEN.

The second job this flag has to do is to check for an overflow. If we regard numbers from 00 to 7F as positive, and from 80 to FF as negative (as described in the section on JR) Then an overflow happens if the "sign" is changed accidently. For example 41 (positive) plus 41 (positive) equals 82 (which is negative). This is an <u>overflow</u>, but note this is NOT a CARRY. JP PE in this case means JUMP if there has been an overflow, and JP PO means JUMP if there has not been an overflow.

The various tests, if combined with other instructions properly, can really check for any situation conceivable. In fact there's only one other kind of instruction you need in order to make JP and JR as powerful as IF/THEN/GOTO - that instruction is CP, or COMPARE.

CP will compare the register A with any other register, or with any constant number. It will do this by working out what would happen if that register or number were to be subtracted from A. It will not alter the value of any of the registers, but it will alter all of the FLAGS. The conditional JP and JR instructions work by checking the value of the flags. Apart from the carry flag, some of the other flags are the sign flag, which stores a one if the last calculation was negative, and a zero if the last calculation was positive; the zero flag, which stores a one if the result of the last calculation was zero, and a zero otherwise; and the parity flag, which stores a one for parity-even, and a zero for parity-odd. Although this may sound complicated you don't actually remember any of it, as long as you know how to use CP.

If A=B THEN GOTO pq is quite easy to represent in machine code. It is CP B followed by JR Z,e. CP B will compare B with A (CP always compares with A, so that CP A is meaningless) which is does by working out A-B. The result isn't stored in any of the registers, so A and B both remain unchanged. The next instruction JR Z,e, will only jump if the result A-B is zero - in other words if A equals B.

IF A<B THEN GOTO pq may be achieved in machine code in two ways. The first instruction is CP B which will compare B with A by performing A-B. Now if A is less than B then A-B will be negative, and so you could well use JP M pq, but you could also do it in another way which will allow you to use JR instead of JP, since if A is positive, and A-B is negative, then there will be a carry, and so you may use the instruction JR C e. Of course this will not work if A was "negative" (ie in the range 80-FF) to start with unless subtracting B caused another overflow by going through 00. This could not happen unless B was in the range 80-FF as well.

<u>CALLING...</u>

Even in machine code we can have subroutines. GOSUB the routine starting at address pq is CALL pq. RETURN is RET. This particular instruction should look very familiar, since it is the very same RET that we've been using to get back to BASIC at the end of a routine. This is because every USR routine is really a SUBROUTINE, even though we consider it as a program in its own right. Unfortunately there's no such thing as a CALL RELATIVE instruction, as there is with JUMP, so CALL must always be a three byte instruction. In exactly the same way as with JP we can impose IF/THEN conditions, which work in precisely the same way and are written with the same letters to define the conditions. These are:

CALL pq	CD	RET	C9
CALL Z pq	CC	RET Z	C8
CALL NZ pq	C4	RET NZ	C0
CALL C pq	DC	RET C	D8

CALL NC	D4	RET NC	D0
CALL PE	EC	RET PE	E8
CALL PO	E4	RET PO	E0
CALL M	FC	RET M	F8
CALL P	F4	RET P	F0

As you may or may not have guessed, instructions like RET Z (return if zero) can also be used to end a machine code routine, ie IF RESULT 0 THEN RETurn to BASIC.

It is very important however that the value of SP is not altered during a subroutine, since the instructions CALL and RET both use the stack. CALL works by PUSHing what would have been the next address to be executed onto the stack, and RET works by POPping the first item on the stack. Thereafter both of these instructions act exactly like JP. Therefore it is possible to alter the RET address, should you need to, by POPping the first item on the stack (the previous RET address) and then PUSHing a new address. For example, to change the RET address to 20000 you could use the sequence

```
E1        POP HL
21204E    LD HL,2000
E5        PUSH HL
```

Another useful trick is to store the value of the stack pointer somewhere at the start of a subroutine, and then retrieve it at the end. On the new ROM a good place to store this value is the address 16507 because neither this nor 16508 are used at all by the ROM - it is the two "spare" bytes between the system variables and the program. On the old ROM you don't have this spare space, but you can overwrite some of the other systems variables, for example the frame counter at address 16414. The advantage of doing this is that you can PUSH and POP to your heart's content and still be sure of a safe RETurn.

At the start of a subroutine:

```
ED737B40      LD (16507),SP
```

and at the end of a subroutine:

```
ED7B7B40      LD SP,(16507)
C9            RET
```

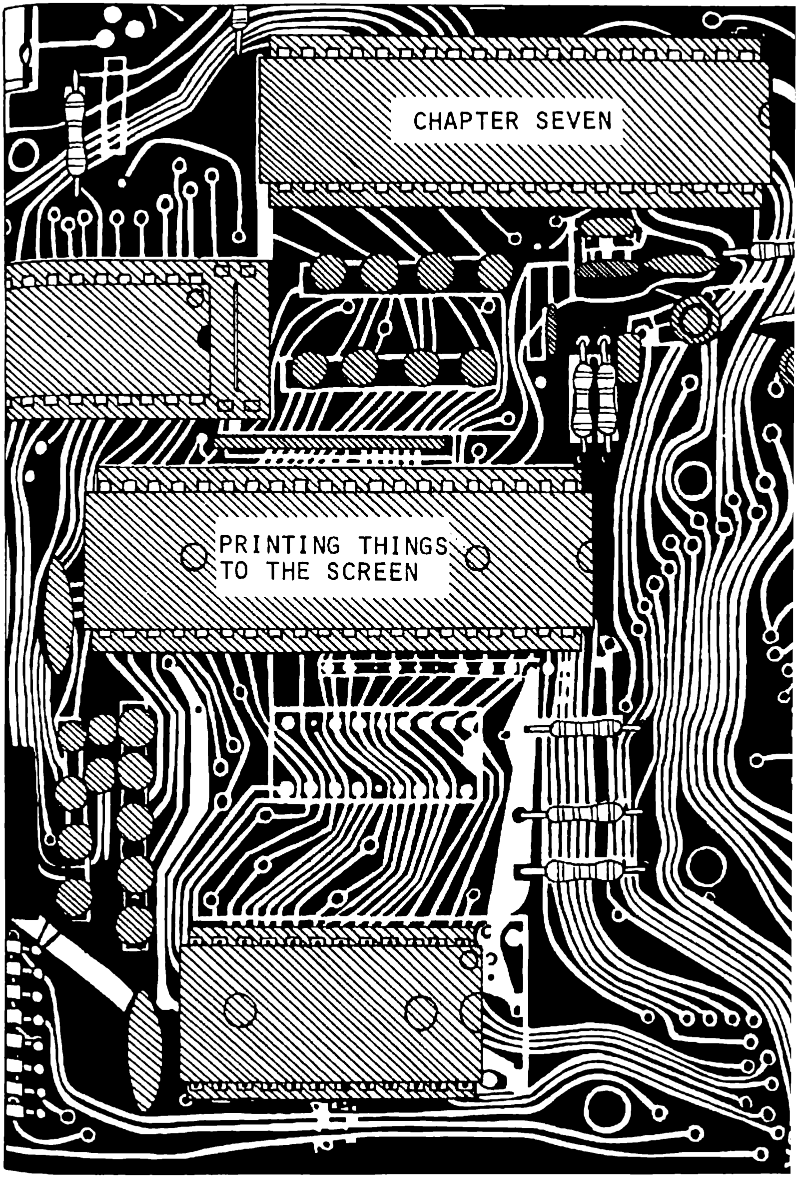
__EXERCISES__

To make sure you have understood using the stack, and conditional jumps, write a program which will PUSH every number between one and fifty onto the stack (using PUSH AF) and then somehow manage to successfully return to BASIC. (HINT: CP 32 (Compare with 32 (hex) (50 decimal)) is quite a useful instruction here.

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

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

In the next chapter we'll begin loading a program which is designed to play a game of draughts. Now don't worry if this sounds rather complicated- I did say we'd __begin__ loading it. I'm afraid you won't get the whole program until you've nearly completed the whole book, so keep a cassette handy reserved just for this program, and you can reSAVE it at each new stage. You'll need at least 4K for this.


CHAPTER SEVEN
PRINTING THINGS
TO THE SCREEN

DRAUGHTS

In order to write a program as extensive as draughts, we'll need a fairly powerful BASIC program in order to help us load it. The following is a second version of HEXLD - called HEXLD2 - which has a couple of improvements over its predecessor. One such improvement is the ability to input strings of characters such as "To be or not to be" which will then be incorporated in the machine code one character at a time. To achieve this you must input ";TO BE OR NOT TO BE;" - that is, the text must be surrounded by semicolons - this is very important.

HEXLD2 lists as follows. OLD ROM users should use the version on the left, and NEW ROM users the version on the right.

OLD ROM

```
10 PRINT "WRITE TO ";
20 INPUT A$
30 PRINT A$
40 GOSUB 200
50 PRINT
60 LET A$=""
70 IF A$="" THEN INPUT A$
80 IF A$="S" THEN STOP
90 IF CODE(A$)=215 THEN GOTO 300
100 PRINT CHR$(CODE(A$));
         CHR$(CODE(TL$(A$)));
         "two spaces";
110 POKE X,16*CODE(A$)+CODE(TL$(A$))+36
120 LET X=X+1
130 LET A$=TL$(TL$(A$))
140 GOTO 70
200 LET X=0
210 FOR I=1 TO 4
220 LET X=16*X+CODE(A$)-28
230 LET A$=TL$(A$)
240 NEXT I
250 RETURN
300 LET A$=TL$(A$)
310 PRINT ".";CHR$(CODE(A$));
         "two spaces";
320 POKE X,CODE(A$)
330 IF CODE(A$)=226 THEN POKE X,118
340 LET A$=TL$(A$)
350 LET X=X+1
360 IF NOT CODE(A$)=215 THEN GOTO 310
370 LET A$=TL$(A$)
380 GOTO 70
```

NEW ROM

```
10 PRINT "WRITE TO ";
20 INPUT A$
30 PRINT A$
40 GOSUB 200
50 PRINT
60 LET A$=""
70 IF A$="" THEN INPUT A$
80 IF A$="S" THEN STOP
90 IF CODE A$=25 THEN GOTO 300
100 PRINT A$( TO 2);"two spaces";
110 POKE X,16*CODE A$+CODE A$(2)-476
120 LET X=X+1
130 LET A$=A$(3 TO )
140 GOTO 70
200 LET X=4096*CODE A$+256*CODE A$(2)
         +16*CODE A$(3)+CODE A$(4)
         -122332
210 RETURN
300 LET A$=A$(2 TO )
310 PRINT ".";A$(1);"two spaces";
320 POKE X,CODE A$
330 IF CODE A$=216 THEN POKE X,118
340 LET A$=A$(2 TO )
350 LET X=X+1
360 IF CODE A$<>25 THEN GOTO 310
370 LET A$=A$(2 TO )
380 GOTO 70
```

This program is basicly the same as HEXLD except for two features. Firstly you are required to input the starting address (in hexadecimal) at which the machine code is to be loaded, and secondly it will allow you to input strings of data using their character codes, rather than hex - this is what the routine starting at line 300 is for. If you input "CD0808C9" it will be interpreted as CALL 0808 followed by RET - this is exactly the same as before - however if you instead input ";LN graphic A graphic A TAN;" it will mean exactly the same thing. If you compare character codes with hexadecimal by looking it up in the manual you'll find the hex for LN is CD, hex for graphic A is 08, and hex for TAN is C9. The semicolan is used to tell the program where the data starts and ends.

SUBROUTINES WITH DATA

Let's look at some uses for this. Perhaps the most useful subroutine we could imagine would be one which prints a string of characters to the screen. There is already a subroutine in the ROM which will print a single character. Try this program. Load it to address 4E00. (If you only have 1K you'll have to find some other suitable address).

OLD ROM	NEW ROM		
CDE006		START	CALL PRPOS (OLD ROM only)
3E94	3E97		LD A,inverse asterisk
CD2007	CD0808		CALL PRINT
3A2540			LD A,(S.POSN))
3D			DEC A)- (OLD ROM ONLY)
C8			RET Z)
18F1	18F9		JR START

You'll discover upon running it that the screen fills up with inverse asterisks, and that it fills up very, very fast. (Much faster than PRINT "inverse asterisk"/RUN). The ROM subroutine PRINT will place the character whose code is stored in the A register at the current PRINT position on the screen. In the new ROM, locating the print position is automatic, but in the old ROM you have to call up a completely different subroutine - PRPOS (Print Position) - first, in order that the second subroutine, PRINT, knows where to place the image on the screen. PRPOS wipes out the value of the A register, but PRINT does not. Note that OLD-ROM-PRINT, and NEW-ROM-PRINT, work by two completely different methods, even though we are using them in precisely the same way, except that for the OLD ROM we have to check for end-of-screen.

It is in fact possible to put this entire program into a REM statement. NEW ROM users with only 1K might like to try clearing the machine with NEW and then typing line 1 REM Y inverse asterisk LN graphic A graphic A / RAND (You'll need to type THEN RAND and delete the word THEN to get the word RAND in position) This is precisely the above program, but entered direct from the keyboard instead of loaded via a separate program. Now the command RAND USR 16514 will almost instantly fill the screen! Shock - Horror - A full screen in 1K!!?

What we want though is a subroutine which can print any message, from "YES" to "OH WHAT A BEAUTIFUL MORNING". Suppose such a subroutine exists and it's called SPRINT (String Print) We want to be able to use an instruction something along the lines of CALL SPRINT WITH "OH WHAT A BEAUTIFUL MORNING". Here's how it will work:

```
CD????          CALL SPRINT
2D2A313134      DEFM HELLO
FF              DEFB FF
```

Here DEFM means Define Message. It's not actually a machine language instruction, but is used to specify data within a program. If you look at the hex equivalent you'll see that 2D is hexadecimal for the character code of H, 2A for E, 31 for L and 34 for O. DEFB is also data - it means define byte. We could have put DEFB C9 and it would have meant the byte C9. Here we are using it to specify the end of the data to be used by SPRINT. We must ensure, however, that the machine does not try to execute these bytes, since in machine language terms they don't make a great deal of sense. Let's take a look at how we could go about writing such a subroutine as SPRINT which at the same time ensures that we don't try to execute the data (ie the word "HELLO" and the byte FF)

You may remember from the last chapter that CALL works by PUSHing the return address onto the stack and then jumping to the required address. RET works similarly - it POPs an address from the stack and then jumps to it. Therefore if the word "HELLO" immediately follows a CALL instruction then the address at the top of the stack will be the address of the first character of data - the "H" - we can obtain this with the single instruction POP HL. If we then increment HL by one and PUSH it back onto the stack then the effect of the next RETURN will be to jump back to the NEXT address in line - the "E". We can test for the end of the data by looking for the byte FF (which is not a printable character). Follow this subroutine through.

OLD ROM	NEW ROM			
E1	E1	SPRINT	POP HL	
7E	7E		LD A,(HL)	
23	23		INC HL	
E5	E5		PUSH HL	
FEFF	FEFF		CP FF	
C8	C8		RET Z	
F5			PUSH AF	OLD ROM ONLY
CDE006			CALL PRPOS	OLD ROM ONLY
F1			POP AF	OLD ROM ONLY
CD2007	CD0808		CALL PRINT	
18EF	18F4		JR SPRINT	

The first four lines are designed to look at the character stored at the current return address and then increment the return address. The next two lines will only return from the subroutine if the byte FF has been found. Note that CP FF will compare A with FF, not HL which was the last thing referred to. CP will always compare A with something - in this case the hex value FF. The RET instruction (actually a RET Z or return if zero, but it works in precisely the same way) will, if you examine the listing closely enough, return you to the byte AFTER the FF, not to the FF itself. Finally, if FF has not yet been found, the subroutine PRINT will be called and the single character now in the A register will be printed to the screen. The whole routine will then be repeated over and over again until the end of the message is found.

Enter the program HEXLD2 to enable you to load machine code. Add an additional line to it - line one - which should be a REM statement with fifty arbitrary characters after the word REM. OLD ROM users must ensure that this line is never listed. LIST 9999 followed by LIST 2 will ensure this. Now RUN the program. The message WRITE TO will greet you. Input "402B" for the OLD ROM, or "4082" for the NEW ROM. This is the address, in HEX, of the first character after the word REM. When promted type in the following - (here / means newline) - E1/7E/23/E5/FEFF/C8/ (old ROM users only should type F5/CDE006/F1) CD2007 or CD0808/18EF or 18F4/ CD2B40 or CD8240/;OH WHAT A BEAUTIFUL MORNING;/FF/C9. The last four lines were CALL SPRINT (Notice how the two bytes of the address have been switched around), DEFM OH WHAT A BEAUTIFUL MORNING, DEFB FF, and RET.

Now do you see the purpose of the BASIC routine in HEXLD2 which begins at line 300. Imagine how tedious it would have been to have had to type in 342D003C2D263900... and so on instead of ;OH WHAT A BEAUTIFUL MORNING; It has exactly the same effect. Now type in as a direct command RANDOMISE USR (16444) (OLD ROM) or RAND USR 16526 (NEW ROM) and what happens?

We shall use this routine to print a draughts board for us. You'll need at least 4K to load this program, but once loaded it will quite happily fit and run in 1K. If you only have 1K altogether it might be an idea to try and borrow some memory from somewhere, and then give it back only once you've got the whole of draughts in - but be warned - the listing is spread very thinly throughout the whole of the book.

If you take a look at line 330 of HEXLD2 you'll see that every time you input a double-asterisk (**) it will automatically be changed into a newline. This is a point of convenience. We can input a newline if we want, by just deleting the quote marks at the input prompt and instead typing CHR$(118), but it is far simpler, and far more convenient, to only have to type shift-H. If of course you ever need two asterisks in a row you can always type a single asterisk twice.

The next machine code program forms the very first part of DRAUGHTS. It is the routine which enables us to print the playing board. For the OLD ROM we shall begin loading this program such that the first address used is 4C04. For the NEW ROM the first address will be 4C09. NEW ROM users should remember (or write down) the sequence of BASIC commands

```
POKE 16389,76
NEW
```

which should be typed in BEFORE HEXLD2 is entered. Now enter the following machine code. WRITE TO 4C04 (OLD) or 4C09 (NEW).

```
E1              SPRINT   POP HL              Increment the return
7E                       LD A,(HL)           address.
23                       INC HL
E5                       PUSH HL
FEFF                     CP FF               Return if no more text.
C8                       RET Z
F5                       PUSH AF    )
CDE006                   CALL PRPOS } OLD ROM ONLY
F1                       POP AF     )
CD2007/CD0808            CALL PRINT          Print one character of
18EF/18F4                JR SPRINT           the text at a time.

CD044C/CD094C            CALL SPRINT         Print the draughts
                                             board.
001D1E1F2021222324 76      DEFM  12345678          Data for
1D00BC00BC00BC00BC1D76           1 W W W W1        the SPRINT
1EBC00BC00BC00BC001E76           2W W W W 2        subroutine.
1F00BC00BC00BC00BC1F76           3 W W W W3
2080008000800080002076           4_ _ _ _ 4
2100800080008000802176           5 _ _ _ _5
22A700A700A700A7002276           6B B B B 6
2300A700A700A700A72376           7 B B B B7
24A700A700A700A7002476           8B B B B 8
001D1E1F2021222324 76             12345678
76
76
76
0000000000000000000000000000
FF                                           End of data.
C9                                           Return to Basic.
```

The command RAND USR 19477 (The address of the CALL SPRINT instruction) will produce a complete draughts board picture on your screen almost instantly. Try it.

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

OLD ROM

```
500 PRINT "4C00 TO ";
510 INPUT A$
520 PRINT A$
530 GOSUB 200
540 LET Y=(X-19454)/2
550 DIM O(Y)
560 FOR X=1 TO Y
570 LET A=PEEK(19455+2*X)
573 IF A>127 THEN LET A=A-256
576 LET O(X)=PEEK (19454+2*X)+256*A
580 NEXT X
590 SAVE
600 FOR X=1 TO Y
610 POKE 19454+2*X,O(X)
615 POKE 19455+2*X,O(X)/256
620 NEXT X
630 CLEAR
640 STOP
```

NEW ROM

```
500 PRINT "4C00 TO ";
510 INPUT A$
520 PRINT A$
530 GOSUB 200
540 LET Y=X-19456
550 DIM O$(Y)
560 FOR X=1 TO Y
570 LET O$(X)=CHR$ PEEK (19456+X)
580 NEXT X
590 SAVE "inverse space"
600 FOR X=1 TO Y
610 POKE 19456+X,CODE O$(X)
620 NEXT X
630 CLEAR
640 STOP
```

Now, to SAVE your machine code type RUN 500. At this stage enter 4CA0. It doesn't actually matter which address you give it, so long as this address is larger than the last address of machine code. (So far the last address happens to be 4C96).

The program will then SAVE automatically (line 590). Incidently if you're wondering why I've put SAVE "inverse space" in the NEW ROM version try instead using SAVE "space" and see what happens to the line. When you LOAD the program back, OLD ROM users will need to type GOTO 600 before doing anything else. NEW ROM users won't because the program will continue automatically. Here's how the program works: An array of sufficient size to hold all the bytes to be saved is dimensioned in line 550, after which the machine code is copied into this array and SAVED. The routine at 600 does the reverse - it copies the machine code from the array up to the required address.

AND....

We leave draughts for the moment in order to introduce a few more machine language instructions which we'll need in order to continue with the program. The first of these is AND. Unfortunately for sanity the word AND doesn't mean quite the same thing as it does in BASIC. We're all used to seeing expressions like IF X=1 AND Y=1 THEN... In machine code however we use the word in a completely different context. For example AND B is a complete machine code instruction. So is AND (HL) or AND F0. In order to see how it works it is necessary to take a brief look at numbers in BINARY.

BINARY is yet another form of counting - like decimal or hex. Decimal makes use of the digits 0, 1, 2, 3, 4, 5, 6, 7, 8 and 9. Hex uses A, B, C, D, E and F as well. Binary on the hand uses only the digits 0 and 1. Converting from hex to binary is very simple - much simpler than changing from decimal to hex - simply convert each digit one at a time from this table:

HEXADECIMAL	BINARY	HEXADECIMAL	BINARY
0	0000	8	1000
1	0001	9	1001
2	0010	A	1010
3	0011	B	1011
4	0100	C	1100
5	0101	D	1101
6	0110	E	1110
	0111	F	1111

Therefore C9 (hex) is the same as 11001001 (binary). Can you see how the binary splits up into two halves, 1100 (C) and 1001 (9)? The same is true of all numbers. What is 1E (hex) in binary? What is 01100111 in hex? Now see if you can work out what 123 (decimal) is in binary. (Hint: convert to hex first)

AND assigns a new value to the A register. This new value is determined by a) the previous value of the A register, and b) the value written after the word AND in the instruction. Suppose A contains 5A, and B contains 1F, and the computer then comes across the instruction AND B. Here's how the new value is calculated:

```
          A 01011010
          B 00011111
new value 00011010
```

as you can see, the digits of the new value are zero if there is a zero in the corresponding position of either or both of the old values, and a one if both the old values contained a one in that position. To make this clear just look at the columns - you'll see that in all cases two zeroes lead to a zero, two ones lead to a one, and a mixture of zeroes and ones lead to a zero. The function is called AND since a one is only obtained if A <u>AND</u> B have a corresponding one. It may appear to you to be rather a useless function, but it is in fact one of the most widely used machine language instructions there is. Some examples of its use are:

AND A — leaves A unchanged, but resets the carry flag.
AND 7F — if A contains a printable character, prevents it from being inverse - both of these examples we shall make use of.

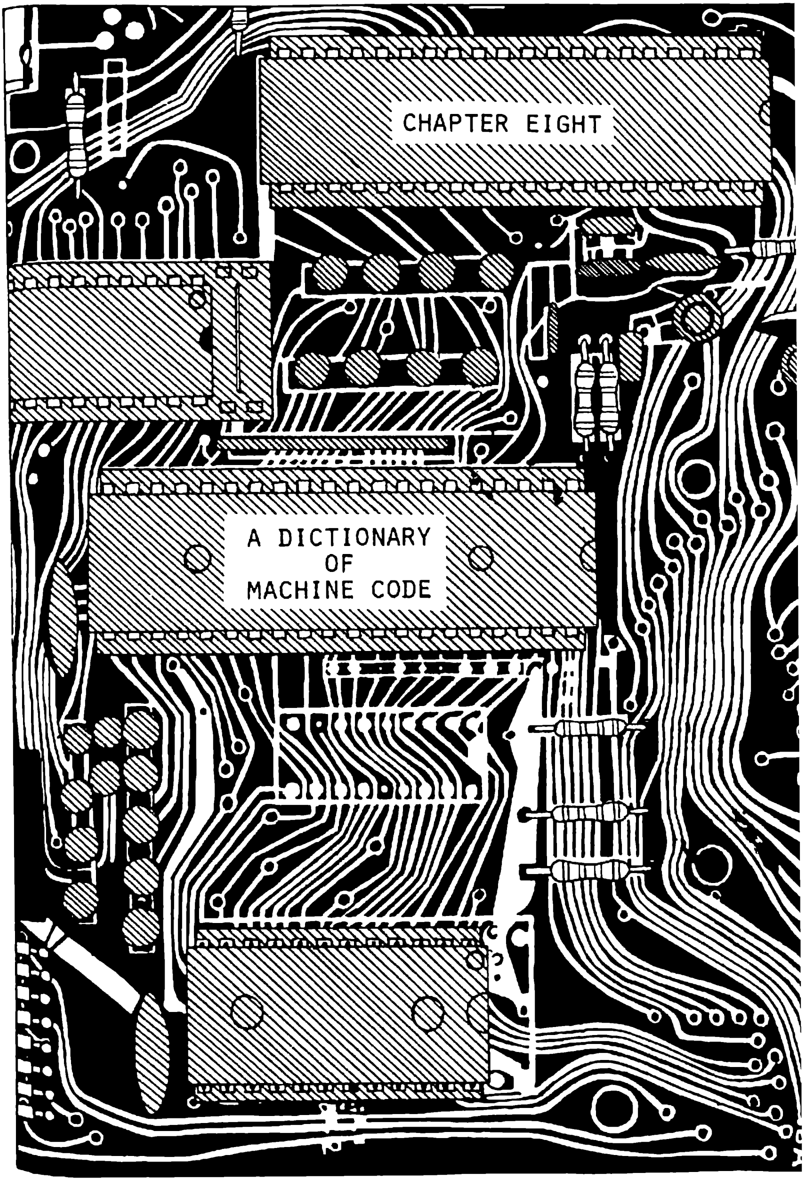
OR....

OR is pretty similar. The rules are that two zeroes lead to a zero, two ones lead to a one. The difference here is that a mixture of zeroes and ones lead to a one rather than a zero. Instead of AND A then we could have used OR A to reset the carry flag. The function is called OR since a one is obtained if A <u>OR</u> B have a corresponding one. One use for the OR function could be OR 80 which, if A is a printable character, will ensure that it is an inverse character. This also we shall use. There is one other function we need to know - it is called XOR.

XOR....

XOR is not a character out of Flash Gordon, despite its sound, it is in fact short for Exclusive-OR, which is a variation on ordinary OR. Its difference is that two ones will lead to a zero. Everything else is the same as in ordinary OR, ie two zeroes equals zero, a mixture equals one. It follows then that XOR FF will change every single binary digit of A (this is called "complementing") from a zero to a one or vice versa. Also note that XOR A will combine A with itself and hence come up with eight zeroes. It in effect resets both A and the carry flag to zero, having the same effect as SUB A,A. This too is useful.

The reason we are interested in these functions is the manner in which we shall represent kings in our draughts game. As you have seen from the initial playing board ordinary pieces are inverse B or inverse W (for Black or White). Kings however shall be ORDINARY B or ORDINARY W. Thus a human's piece can either be 27 hex (the character code of B) or A7 (the character code of inverse B), so to check whether or not we've found one we just put it into the A register, use OR 80, and compare it with A7. This saves us from making two separate comparisons.


CHAPTER EIGHT
A DICTIONARY
OF
MACHINE CODE

SPECIAL REGISTERS

The Z80 has two special registers which can be made use of. The first is called IX.

It is special because as well as just assigning it, as it can be used just like any other register pair with LD IX,0000 for instance, we can use it to find the contents of an address - using (IX) - just like we can with (HL). IX is different because we can add a constant to the address. Thus LD B,(IX+7B) works! If IX were 0000 then LD B,(IX+7B) will load B with the contents of memory location 007B. In no other way can we assign a single register from an address in one instruction.

There is a warning that goes with using IX though. If you are using SLOW then you must not alter the value of IX at all, otherwise you might cause a crash.

The other special register is called IY. It is used in exactly the same way as IX, except that the ROM itself gives us an added advantage. When you jump into a machine language routine, IY starts off as 4000 (hex), so that all of the system variables may be accessed directly. (The system variables start off at 4000.) For example, LD L,(IY+0C) will load L with the low part of the address at which the display file begins.

Changing the value of IY will not cause a crash. It will be reset to 4000 as soon as you return to BASIC. This is done automatically by the ROM.

To find the hex code of any instruction involving IX or IY pretend you are using HL instead and look up the code for that. Then preceed it by DD for IX, or FD for IY. If the IX or IY is in brackets then it must have a displacement, even if that displacement is 00. (For instance, in LD B,(IY+04) the displacement is 04.) This byte should be added to the hex code, and should be the third byte of the code, even if this means splitting the original code in two.

Thus if the code of LD (HL),44 is 3644, then the code of LD (IX+20),44 is DD362044. Note how the displacement 20 has been inserted into the middle of the original code in order to make it the third byte. We have now reached the stage of using four byte instruction codes. This is the longest a Z80 instruction can possibly be.

THE FLAGS REGISTER

Another special register is the FLAGS register, sometimes called the STATUS register. Usually it abbreviates itself to just F, and cohabitates with A in the hope that no-one will notice it. Its purpose is to store various bits of information about the results of calculations. Some instructions will alter all of the flags, some will alter only some of them, and some won't actually alter any flags at all. A complete list of what instruction does what is included in an appendix at the back of this book.

As for the register itself: it is, like any other register, eight bits in length, but each bit has a different purpose (although two of them aren't used). These bits are each used as an individual flag which can store a value of either zero or one. The flags are, from left to right; Sign, Zero, not-used, Half-carry, not-used, Parity/Overflow, Subtract, and Carry. The two unused flags are both more or less random, but the rest are quite specific. They work like this:

The SIGN flag stores the sign (positive or negative) of the last result. A positive number resets this flag to zero, and a negative number sets it to one. For the purposes of this flag, zero is counted as positive. The value of the S flag is therefore always equal to the leftmost bit of the result. It may be tested using instructions like JP P (Jump if positive) of JP M (Jump if negative (Minus)).

The ZERO flag checks whether or not the last result was actually zero. If so the flag is set to one, but otherwise it is reset. Watch out for this flag though - it can be very deceiving - many of the register-pair instructions simply do not change it as you'd expect: instructions like DEC or ADD for instance will only change the zero flag if applied to single registers. You are advised to check with the appendix if you are unsure.

The HALF-CARRY flag is set if there is a carry from bit 3 into bit 4, or, in the case of register-pairs, from bit 11 into bit 12. It is used internally by the Z80 for such instructions as DAA, but cannot easily be tested by the programmer. It is possible to examine it using the sequence PUSH AF/POP BC/ BIT 4,C and then testing the zero flag, but this is rarely done.

The PARITY/OVERFLOW flag does two jobs at once. The PARITY of a result is either odd or even, depending on the number of ones in the result (when written in binary). The Parity flag is assigned in exactly the opposite manner to that which you'd expect. If the parity is even, the flag is one (an odd number), and if the parity is odd, the flag is zero (an even number). The following instructions assign this flag according to the parity of the result: AND r, OR r, XOR r, RL r, RLC r, RR r, RRC r, SLA r, SRA r, SRL r, RLD, RRD, DAA, and IN r,(C). An OVERLOW represents an "accidental" change of sign of the result - a carry from bit 6 into bit 7 effectively. The following instructions assign this flag according to whether or not we have an overflow: ADD A,r, ADC A,r, ADC HL,s, SUB A,r, SBC A,r, SBC HL,s, CP r, NEG, INC r, and DEC r.

The subtract flag, also called the N flag, simply lets the machine know whether or not the last instruction was an addition, or a subtraction. You can't get hold of this flag unless you make use of PUSH and POP as I've described under HALF-CARRY, but in general you'll know what the last instruction was anyway. This flag is primarily used internally by the Z80 for instructions such as DAA.

The CARRY flag you know about. It detects a carry from bit 7 into (the non-existant) bit 8, or in the case of register-pairs, from bit 15 into what would have been bit 16. It is also assigned by shift and rotate instructions, in which one bit is "lost" from a register and moves into the carry. This is probably the most frequently accessed flag of all.

ALL THE INSTRUCTIONS

By now we've seen a fair number of Z80 instructions, so you'll be wanting to expand your vocabulary of these. Here now is a detailed list of all of the instructions that are available to you. I shall cover them in alphabetical order so that you may use this chapter as a kind of dictionary of instructions. For precisely the same reason I shall re-cover the ones you've already seen. You should reread them anyway since it will prove a useful memory aid.

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

ADD Very similar to ADC except that the carry flag is not used in the initial calculation. It is however still altered by the final result. There are one or two important differences between ADD and ADC however. Firstly the set of instructions ADD HL,s (where s means the same as it did in ADC) are one byte instructions rather than two, and secondly it is permissable to use two further sets of instructions ADD IX,s and ADD IY,s. Altering the value of IX however is not advisable if you are using SLOW IY may be safely altered but will always be reset to 4000 (hex) on return to BASIC.

AND Only one form here - AND r. The value of the A register is altered one bit at a time. If such a bit is zero it will be unaltered. If a bit is one it will take on the value of the corresponding bit of r. Thus AND 00 is always zero, AND FF will leave A unchanged. AND alters all of the flags - specifically the carry flag will always be reset to zero.

BIT Now this is a new one. What happens is that from time to time you'll want to know whether an individual bit of some register is one or not, but for some reason or other it becomes impractical to try and rotate or shift it into the carry. BIT is specially designed to help you out here. Suppose you wanted to know the value of BIT 5 of B. The instruction is simply BIT 5,B - the result is then either zero or non-zero, which you can exploit using JR Z for instance, or RET NZ. BIT does not alter the value of ANY of the registers, nor does it change the value of the carry flag. Its hex codes are listed in a table at the end of this book - it is a two-byte instruction. I tend to find it's not used very often, but when it is used it comes in very handy indeed.

CALL You've seen this one before - it's rather like GOSUB. Its exact function is as follows: PUSH the return address onto the stack, and JUMP to the call address. The return address is used by the RET instruction so it is vitally important that a subroutine should not alter the stack. You may only push things onto the stack in a subroutine if you pop them off again before you attempt to return. CALL may also be used with conditions - for example CALL Z,pq (pq is an absolute address) which means CALL pq if the last calculation was zero, otherwise continue with the next instruction.

CCF Complement carry flag. If the carry flag was zero then change it to one. If it was one then change it to zero.

CP In the form CP r it will calculate the result of subtracting r from A, however the answer NOT stored anywhere, nor is the previous value of either A or r altered. It will on the other hand alter all of the flags, so conditions like jump if zero, or jump if carry, will still work. CP r followed by JR Z will jump if A equals r.

CPD Imagine this as CP (HL), followed by DEC HL, followed by DEC BC. The zero flag is altered as if a single CP (HL) instruction had been executed. Another flag altered is the P/V flag, which works as follows: if BC decrements to zero then the P/V flag is also zero. If BC does not decrement to zero then the P/V flag is set to one. Thus JP PO will jump only if BC now equals zero. JP PE will jump only if BC is not now equal to zero. The carry flag is not altered at all by this instruction.

CPDR Basicly this is the same as CPD except that the instruction is executed over and over again - a kind of automatic loop. CPDR stands for Compare with Decrement and Repeat. The loop will end in one of two cases. a) if A equals (HL) - in which case the zero flag will be set, or b) if BC reaches zero - this will affect the P/V flag as in CPD. If neither of these conditions is true the instruction is re-executed.

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

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

CPL An abbreviation for complement. The register A is altered bit by bit. If any particular bit starts off as zero it is changed to one and vice versa. In other words if A starts off as 11010101 (binary) the instruction CPL will change it to 00101010 (binary). The flags are not altered, nor are any of the other registers.

DAA Suppose you wanted to add 16 (decimal) to 26 (decimal) without converting them to hex. The following seems plausible: LD A,16 then ADD A,26. Unfortunately, because the machine works in hex the final value of A will be 3C, not 42. The instruction DAA (Decimal Adjust accumulator) will change A from 3C to 42. How it works is rather complicated - it makes a note of what's been carried where and whether you've added or subtracted and so on - but it does always work. For instance the sequence LD A,42 then SUB A,06 will again leave A with 3C, but this time round DAA will change A to 36, since 42 (decimal) minus 6 (decimal) is 36 (decimal). The instruction changes every flag appropriately.

DEC This is another one of those instructions that comes in two forms. It can be dec r (a single register) or dec s (a register pair). dec r is very simple to understand - the value of the register r is decreased by one, the carry flag is unaltered, and the zero flag is changed appropriately. Dec s is the one you want to watch for, because the zero flag is NOT ALTERED! Nor are any of the other flags! Thus DEC BC followed by JR NZ,-3 is either an infinite loop or has no effect! You'll have to be very careful to remember this - a lot of my earlier programs crashed because I didn't.

DI Not a Welsh name, nor is it short for Diane or Diana. It is in fact an abbreviation (surprize! surprize!) It stands for Disable Interrupts, and although this sounds pretty confusing its use is immensely simple. An interrupt is what you get if you send little bleeps into the pins of the Z80 chip. DISABLING the interrupts means that if such a thing happens in future it is to be ignored. That's about all I can tell you I'm afraid - you'll have to consult the hardware boffs for a more detailed explanation.

DJNZ Yet another abbreviation - this time for Decrement B and Jump-relative if Not Zero. So if B is 7, DJNZ will reduce it to 6. If B is zero, DJNZ will change it to FF. If B is one however, DJNZ will change it to zero, and will then Jump to a new destination. The form of the instruction is DJNZ e, where e is a single byte. If B is not decremented to zero the e is ignored, if it is then e specifies how far to jump. If e is between 00 and 7F then the jump is FORWARDS, if e is between 80 and FF then the jump is BACKWARDS (with FF -1, FE -2, and so on). Start counting from the next instruction, so that DJNZ 00 is just the same as DEC B, except that DJNZ does not alter any of the flags.

EI Guess what? Another abbreviation. EI stands for Enable Interupts, and is the opposite of DI. From now on, if the Z80 recieves an interrupt, then execution of the current instruction is completed, and control then jumps to an interrupt routine. For a slightly better explanation look under IM.

EX At last - an instruction with a sensible name. Ex means exchange. There are five different EX instructions - these are EX AF,AF', EX DE,HL, EX (SP),HL, EX (SP),IX, and EX (SP),IY. They don't alter any of the flags. What they do is, as you'd expect, swop the values over - thus EX DE,HL replaces DE by the value HL used to contain, and HL by the value DE used to contain. The last three are rather interesting - the old value of HL (or IX or IY) is pushed onto the stack, but simultaneously the old value at the top of the stack is popped and loaded into HL. The position of the stack pointer is therefore unchanged. AF' (Pronounced AF dash) is a register pair distinct from the real AF, and this is the only instruction which uses it. It is used by the SLOW hardware, so don't use EX AF,AF' while you're in SLOW.

EXX As well as AF' there are also BC', DE' and HL', which are just a set of six new registers (or three new register pairs) which can only be accessed by this one single instruction. EXX is an exchange instruction. It means exchange BC with BC' (ie B with B' and C with C'), DE with DE', and HL with HL' - all in the same go. This is quite safe, and does not affect SLOW in the way that AF' does. It is useful for preserving the values of the registers when calling a ROM subroutine which relies upon A but wipes out the other registers, eg EXX/CALL ROM-SUBROUTINE/EXX. The previous values of BC, DE, and HL are now unchanged. Some of the programs later on in this book will make use of this technique.

HALT Don't be fooled by your own intuition - this isn't the same as STOP. It means do nothing, or wait forever. Once you hit a HALT instruction it will just sit there, effectively executing NOP instructions, over and over again. In fact the only way you can get out of it, once you're stuck there, is by sending the little chip an interrupt signal, so EI followed by HALT is safe since the hardware ensures that interrupts turn up pretty frequently, wheras DI followed by HALT is rather disasterous.

IM There are three forms of this instruction. These are IM 0, IM 1, and IM 2. They are there to change the Interrupt Mode (yes, another abbreviation) to either zero, one, or two. What this means is that the next time an interrupt is detected the following will happen. IF THE INTERRUPT MODE IS ZERO: The interrupt device itself must supply an instruction to be executed. IF THE INTERRUPT MODE IS ONE: The instruction RST 38 is executed. IF THE INTERRUPT MODE IS TWO: The interrupt device must supply one byte of data. This is used as the low part of an address. There is a register called I (which we so far haven't used) and the value of this register is used as the high part of an address. The machine then looks up this address and should find a second address stored there. Confusing isn't it? This second address is used as a subroutine call.

IN Short for input, but nothing like the INPUT we are used to in BASIC. It is this instruction from which Sinclair builds the LOAD routine and a keyboard scan. It has two forms - the first is IN A,(n) where n is a numerical constant. n refers to an external device - a different n for each different device. One byte of data is read from device n, and loaded into A. IN A,(n) has no effect on the flags. The second form DOES alter the flags - it is IN r,(C). The number held by the C register is used to specify the device. The number input is loaded into register r.

IND Input with decrement. This is a deliberate digression from alphabetical order so that all of the input instructions can go together. IND can be thought of as IN (HL)(C) followed by DEC B followed by DEC HL. The carry flag is not altered, but the zero flag is altered to show whether or not B has decremented to zero.

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

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

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

INC Don't Panic! At long last we're back to sensible instructions we can all understand. INC r increases the value of register r by one. Every flag except the carry flag is altered. INC s on the other hand (where s is a register-pair rather than a single register) will not change ANY of the flags. It still does the same job of course, increasing the value of register-pair s by one and zooming back round to 0000 if s starts off at FFFF, but don't use a check for zero after an INC s instruction because it simply won't work. INC HL/JR Z means jump if the instruction before INC HL came to zero, NOT if HL has reached zero. INC H/JR Z does work.

JP If you can understand GOTO 10 you can understand JP 4300. The destination is an address, not a line number, but the principle is exactly the same. JP is the machine language GOTO. We can also have conditional jumps, for example JP NZ,4300 means jump to 4300 IF NOT ZERO. (In other words if the zero flag is not set.) There is another form of JP which also has an analogy in BASIC - variable destinations. If you understand GOTO N you'll understand JP (HL). In this form you can't have conditions. JP NC,(HL) for instance is not allowed. Also only three registers may be used as variables - these are HL, IX, and IY. Even so these are very powerful instructions - HL can be the result of a calculation, possibly even generated at random.

JR The same as JP but slightly less powerful, and one byte shorter. Only four of the eight conditions may be used - JR Z, JR NZ, JR C, and JR NC. It is impossible to say JR PO. It is also impossible to say JR (HL). JR does not use an absolute address - the R stands for relative. You write the instruction as JR e (or JR Z,e or whatever) where the e is a single byte which specifies how far we must jump. JR 0 has no effect, and JR FE is an infinite loop, since FE represents minus two. The jump is forward if e is between 0 and 7F, and backward if e is between 80 and FF.

LD The most used instruction in the whole of machine language. All it does is to transfer data from one place to another. It has many, many forms, the simplest being LD r1,r2, that is to transfer data from one register to another. LD A,(BC) is also legal and is a one byte code, so is LD A,(DE). These are reversable, ie LD (BC),A and LD (DE),A are also legal. Remember that the brackets mean the contents of the address BC (or DE). Two special registers R (the memory refresh register as it's called which is used in outputting to the screen) and I (see IM) may be loaded to and from A (but only A) as in LD A,I, LD A,R, LD I,A, and LD R,A. The register pairs may all be loaded with either numerical constants or the contents of absolute addresses - LD s,mn or LD s,(pq). Conversely any address may be loaded with the contents of one of the register pairs - LD (pq),s. Note that register-pairs hold two bytes not one, and these are transferred to and from pq and pq+1. You can do the same with A on its own - LD A,(pq) and LD (pq),A are both allowed, but no other register can do this on its own. Finally the register pair SP - the stack pointer - may be loaded directly with either HL, IX, or IY.

In other words there's a lot you can do and a lot you can't do. You can't say LD HL,SP for instance, even though LD SP,HL is allowable. Fortunately, since LD is used so very, very often it is extremely easy to become familiar with.

LDD Load with decrement. Effectively LD (DE),(HL) followed by DEC HL, DEC DE, and DEC BC all in one go. The carry flag and zero flag are unaltered, as is the sign flag, but the P/V flag becomes zero if BC becomes zero, one otherwise, thus JP PO will jump only if BC is zero after the instruction.

LDDR As LDD, but the instruction is repeated continually until BC reaches zero.

LDI As LDD, except that DE and HL are both incremented instead of decremented.

LDIR As LDDR except that DE and HL are both incremented instead of decremented.

NEG Neg alters the accumulator and all of the flags. As you may have gathered from the name it negates A. If A contains 1 then NEG will change it to minus one (FF). If A contains minus six (FA) then NEG will alter it to plus six (06). The same effect may be achieved using CPL followed by INC A - this alternative means of negating a number does not affect the carry flag as NEG does, but NEG is faster.

NOP This wonderous little instruction (which incidently is short for No Operation) has a very simple purpose - its purpose is to waste time, for it does nothing at all! It's almost like a REM statement in fact, except that you can't put messages after it. It has two major uses: 1) as a delay, and 2) to overwrite previous machine coding when debugging. I'd say it was virtually indispensable.

OR In the form OR r this instruction is practically the opposite of AND r. Bit by bit, the value of the A register is changed. If a bit is one then it will be unaltered, but if it is zero it will take on the value of the corresponding bit in r. If A contains 00 then OR r is the same as LD A,r (except for the flags). If A contains FF then OR r will not change it. All of the flags are changed as you'd expect them, and the carry flag is reset to zero.

OUT As with IN, OUT is nothing like the BASIC understanding of output. The instruction OUT (n),A, where n is a one-byte numerical constant, will transfer the contents of A to external device n. Similarly OUT (C),r will transfer the contents of register r to the device pointed to by register C. OUT is used in the ROM to SAVE things. OUT has no effect whatsoever on the flags.

OUTD Output with Decrement. The carry flag is unchanged, but the zero flag depends on the final result of B. OUTD is equivalent to OUT (C),(HL) followed by DEC HL followed by DEC B.

OTDR A slightly different spelling in no way alters the fact that this is still an Output with Decrement and Repeat instruction - all it does is leads us to digress from alphabetical order in order to maintain consistancy. Equivalent to OUTD repeated until B is zero.

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

OTIR As CTDR except that HL is incremented instead of decremented.

POP Remove two bytes of data from the top of the stack and load them into a register pair. Any register pair may be used except for SP. In addition the flags register may be combined with A, allowing the instruction POP AF. Specifically, the low part of the register pair is popped first, and then the high part. The machine remembers that the stack is now two bytes shorter by altering the value of SP automatically.

PUSH PUSH s is the opposite of POP s. It stores the contents of any register pair (except SP, but including AF) at the top of the stack. It "remembers" that it has done this by altering the value of SP. The high part of s is pushed first, then the low part, so that the low part is at the top. After a PUSH instruction SP will point to the address of this low part.

RES With this instruction you can actually alter individual bits of any register. In computing circles "set" means change to one, and "reset" means change to zero, so RES is the instruction that changes the required bit to zero. For instance, to reset bit 3 of D the required instruction is RES 3,D. RES has no effect on any of the flags.

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

RETI Used to end an interrupt subroutine (see IM). Its function is the same as RET, but RETI must be used instead of RET because the chip does clever things if you get a second interrupt in the middle of an interrupt subroutine! As soon as an interrupt subroutine is called a DI instruction is automatically executed, but there are such things as non-maskable interrupts, that it almighty superhigh-powered interrupts that override even DI, these can cause confusion if you don't use RETI.

RETN Used to end a non-maskable interrupt subroutine. Its function is the same as RETI except that the Interrupt Mode (which was altered by the non-maskable interrupt in the first place) is also restored to its previous value.

RLA An abbreviation for Rotate Left Accumulator. Each bit of A is moved one position to the left. The leftmost bit is moved into the carry, and the rightmost bit takes on the previous value of the carry. For example, if A contained 10010101 (binary) and the carry contained 0 then after a RLA instruction A will contain 00101010 and the carry will contain one. Only the carry flag is altered by this instruction.

RL On the other hand, there is another instruction which may be applied to any register. It is RL r. In fact every now and again the instruction RL A tends to disguise itself as RLA - due possibly to printing errors or bad handwriting. On the face of it they seem to do the same thing - RL means Rotate Left and its function is exactly as described in RLA. The difference however, is in what happens to the flags, for RL will alter ALL of them, RLA will only alter the carry. RL may of course be applied to any register, not just A.

Incidently - RL A does precisely the same thing as ADC A,A, down to the last flag - except one - one you can't get at - called the H flag. The only way you can possibly tell the difference is by following it with a DAA instruction. ADC A,A, by the way, is twice as fast.

RLCA Almost the same as RLA, but not quite. Each bit of A is moved one position to the left. The leftmost bit is moved BOTH into the carry, AND into the rightmost position of A. If, as before, A started off with 10010101 and the carry was zero, then after RLCA it will be 00101011. The carry will also be one. Only the carry flag is changed - the previous value of which is lost forever.

RLC RLC r will Rotate Left with Carry the register r in the same way that RLCA does with A. RLC A is a valid instruction, which is not the same as RLCA. RLC B is a valid instruction, but please note there is no such instruction as RLCB. The spacing is very important here. RLC r will alter all of the flags.

RLD Not to be confused with RL D, this is a COMPLETELY DIFFERENT instruction which works as follows: Write the value of A and the value of address (HL) in hex. The second hex-digit of (HL) is shifted left so that it becomes the first digit. The first digit overwrites the second digit of A. The second digit of A moves to the second digit of (HL). Thus if A contains 25 (hex) and (HL) contains EB then after an RLD has been carried out A will contain 2E and (HL) will contain B5. RLD, incidently, stands for Rotate Left Decimal.

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

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

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

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

RRD The contents of (HL) are moved one hex-digit to the right, the rightmost digit moving into the rightmost digit of A, which in turn becomes the left digit of (HL). If A equals 25 and (HL) equals EB then after RRD A will equal 2B and (HL) will equal 5E. Note that RRD twice is the same as RLD once, and vice versa. All of the flags except carry are altered.

RST The same as CALL, except that it is only one byte long ALTOGETHER! It is much less powerful though for two reasons: 1) you may not use conditions. RST 0 is legal but RST NZ,0 is not. 2) only one of eight specific addresses may be called. There are 0, 8, 10, 18, 20, 28, 30, or 38. On the OLD ROM, RST 0 is the same as NEW. On the NEW ROM however RST 0 will move RAMTOP to its highest possible location, which the BASIC instruction NEW will not do. RST 0 is the same thing as pulling out the mains lead and then reconnecting it.

SBC SBC, like ADC, comes in two forms. The first is SBC A,r, which will first of all subtract r from A, and will then subtract the carry digit. Similarly SBC HL,s will subtract both s and the carry flag from HL. SBC A,A is quite useful - if the carry is zero both A and the carry will end up zero - if the carry is one then A will be reassigned FF and the carry will still be one.

SET The opposite of RES. SET 4,H will change the value of bit 4 of H to one. Any bit of any register may be set.

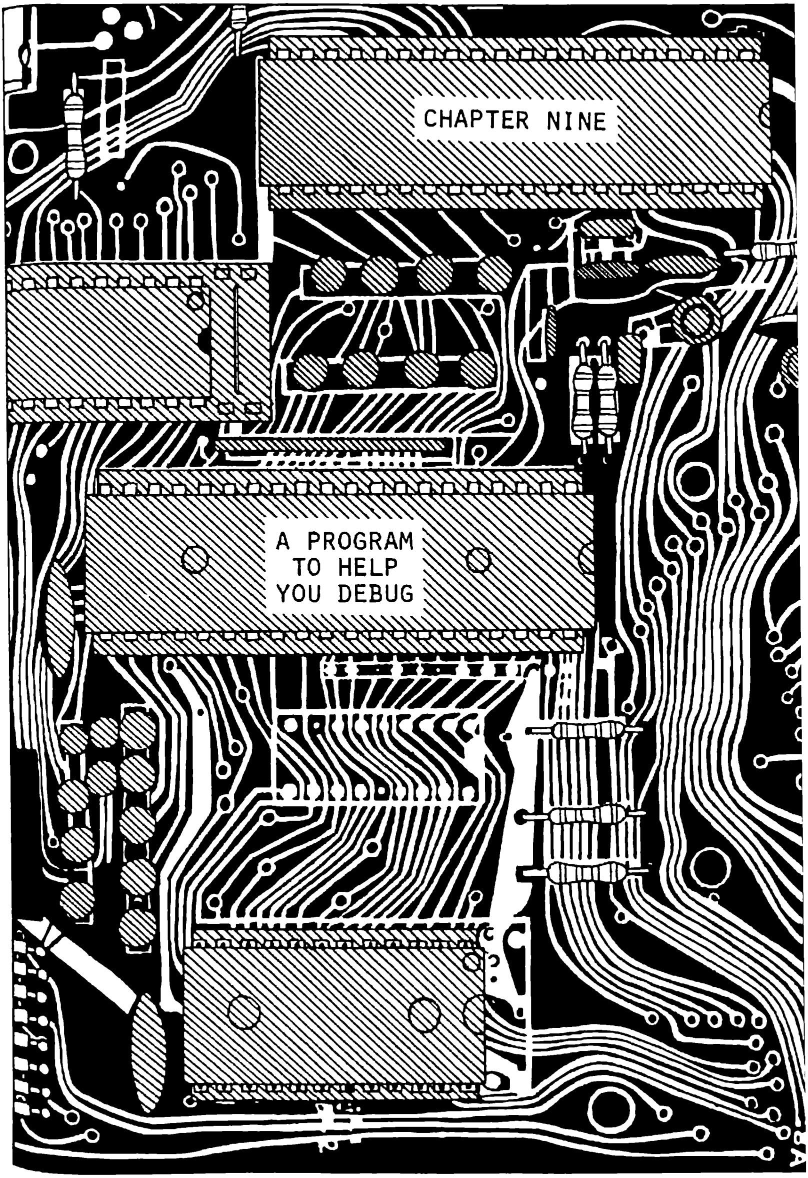
SLA Shift Left Arithmetic. The form is SLA r. It is similar to RL r except that the rightmost bit is automatically replaced by zero. It alters all of the flags. Note that SLA A does the same thing as ADD A,A, except that ADD A,A is faster.

SRA   Shift Right Arithmetic. Any register may be shifted right using the format SRA r. The rightmost bit falls into the carry, but the leftmost bit remains unaltered. Thus after a SRA instruction bit 6 will always be the same as bit 7. The effect of SRA is to divide both positive and negative numbers by two: FC (minus four) becomes FE (minus two). What happens if the number is odd?

SRL   Shift Right Logical. As SLA except that the bits are shifted right instead of left, and the leftmost bit becomes zero.

SUB   Sometime written as SUB r, sometimes as SUB A,r, both mean the same thing. The value of r is subtracted from the A register. Note that unlike ADD, there is no corresponding instruction SUB HL,s. If you wish to do this you must first of all reset the carry flag (usually by use of AND A) and then use SBC HL,s.

XOR   XOR r alters all of the flags, resetting the carry to zero, and the A register alone. r is not altered. What happens is that A is altered bit by bit, in the same manner as AND and OR. If a bit is zero it takes on the value of the corresponding bit of r. If on the other hand a bit is one then its new value is the complement of the appropriate bit of r. XOR A is very useful since in one byte it zeroes both the accumulator and the carry flag. Incidently so does SUB A.


CHAPTER NINE
A PROGRAM
TO HELP
YOU DEBUG

Now we more or less know what machine language is, it's about time we learned a bit more about how to handle it. What we shall do now is to write a new program - HEXLD3 - which will allow us to do five things. 1) Input machine code. 2) Insert machine code in between previous routines, but without overwriting anything. 3) Delete machine code, closing up the gap that it occupied. 4) List machine code. 5) SAVE machine code. The important point about this program is that the principle parts of it will themselves be in machine code, although all of the surrounding fabric will be BASIC. To work it all you will need to do is enter one of the following.

RUN	To List your stored machine code.
RUN 100	To Write new machine code.
RUN 200	To Insert new machine code.
RUN 300	To Delete previous machine code.
RUN 400	To Save machine code.
GOTO 500	To Reload saved machine code (OLD ROM only)

More to the point - you'll need HEXLD2 in order to help you load it. The addresses used in this chapter assume that the machine code is being loaded into a REM statement in line one of a NEW ROM machine. If this is so you'll actually need 255 characters after the word REM. However, you don't have to use the same addresses as me if you don't want to. OLD ROM folk are specifically advised NOT to use a REM statement, since the machine code contains newline characters. To store machine code at different addresses to those I've used simply change the listed addresses to yours.

Let's create it one part at a time. First of all a special subroutine for OLD ROM people - designed to AUTOMATICALLY print a character to the screen, in much the same way that the NEW ROM PRINT routine does. The routine will also protect all of the registers. Study this listing:

<u>FOR OLD ROM PEOPLE ONLY</u>

E5	APRINT	PUSH HL	Store the value of HL.
D9		EXX	Store the remaining registers.
F5		PUSH AF	And the A register.
CDE006		CALL PRPOS	Find print-position.
F1		POP AF	Retrieve A.
CD2007		CALL PRINT	Print character A.
D9		EXX	Retrieve BC and DE.
E1		POP HL	Retrieve HL.
C9		RET	End of subroutine.

Note that HL needs to be stacked, since CALL PRINT changes the value of HL'. The next subroutine we'll need is a mechanism for printing to the screen the <u>value</u> of the A register in hexadecimal. This subroutine will INCLUDE a subroutine-call to APRINT, at least for OLD ROM people. New ROM people in fact already have an automatic print routine which protects all of the registers, since there is one in the ROM itself. It is not quite the same as the PRINT routine, since it also preserves the values of all the registers - this is something that CALL PRINT will not do. CALL PRINT will erase the values of B, C, D, E, H, and L. The address at which APRINT begins in the NEW ROM is 0010, and so CALL 0010 would print a character without changing any register. This is very useful indeed.

One of the Z80 instructions designed to speed things up a bit is RST. It is in effect the same as CALL except that only one of eight addresses may be called. It just so happens that 0010 is one of these possible addresses. RST is better than CALL for two reasons: 1) it is <u>faster</u> to execute, and 2) it is only one byte in length. The code for RST 10 is D7. D7 then has precisely the same effect as CD1000, that is, to print a character. OLD ROM users should note that although D7 still produces a call to 0010, it will <u>not</u> print a character, since in the OLD ROM there is no PRINT subroutine located at this point. RST is short for <u>RESTART</u>.

```
F5      HPRINT  PUSH AF      Store A for later use.
E6F0            AND F0       This isolates the first digit.
1F              RRA          Move this first digit to
1F              RRA          its proper position within
1F              RRA          the A register.
1F              RRA
C61C            ADD A,1C     Add twenty-eight to the character
                             code, making it a hex-digit.
D7              RST 10       Print this hex-digit. OLD ROM
                             users should of course replace
                             RST 10 by CALL APRINT.
F1              POP AF       Retrieve the original value of A.
E60F            AND 0F       Isolate the second digit.
C61C            ADD 1C       Add twenty-eight.
D7              RST 10       Print it. OLD ROM users should
                             instead use CALL APRINT.
C9              RET.
```

By the way, did you understand all those ANDs and RRAs? If you didn't I'll explain exactly what's going on.

In binary, F0 is 1111 0000. This means that when you apply AND to F0 and another number, then the first four binary digits of A will be unchanged, and the second four binary digits will all become zero. Do you remember how to change from binary to hex? You have to look at it four bits at a time. The first four representing the first digit, and the second four the second digit. Thus all we have done is to change the second digit to zero.

If A were 36 then it would become 30. If it were 99 it would become 90. If it were D5 it would become D0. And so on. This is not what we want. We must shift A four bits to the right.

RRA moves A <u>one</u> bit to the right, replacing bit 7 (the leftmost bit) by the value of the carry. In this case the carry is zero, since we have just done an AND instruction. The new value of the carry will be the previous value of bit 0 (the rightmost bit). This will also be zero since there are now <u>four</u> zeroes at the right of A.

RRA then, repeated four times, will change A from 30 to 03, from 90 to 09, and from D0 to 0D. All that remains now is to add 28 (decimal) to this number and print it. We print it using the instruction RST 10.

Back to our new program. The BASIC part of the List routine will look like this:

```
10 PRINT "keyword LIST"
20 GOSUB 600
30 RAND USR 16539
```

to obtain the keyword LIST in line 10, either type THEN LIST (NEW ROM only) and delete the word THEN, or type the whole line as 10 LIST quote backspace backspace PRINT quote newline.

```
600 LET A = 16533
610 PRINT "ADDRESS space";
620 INPUT A$
630 PRINT A$
640 POKE A+1,16*CODE A$+CODE A$(2)
    -476
```

```
650 POKE A,16*CODE A$(3)+CODE A$
    (4)-476
660 CLEAR
670 RETURN
```

What about this USR routine at 16539 then? What will that do? And what about this business of POKEing 16533 and 16534? What's that all about? Well using my addresses, 16539 is the start of a routine called HLIST, which we haven't yet written. It is designed to actually LIST a machine code program in hexadecimal (hence H-List). The address 16533 is the number I've used to hold a "variable" called ADDRESS. That is to say, it is a place at which we can store a two-byte number. Any address may be used for this purpose provided that BASIC will not change that two-byte number.

This program demands four such "variables", or two-byte memory locations. They will be called BEGIN, ADDRESS, ADD2, and LIMIT. They will be used by the program as follows:

BEGIN	The address at which the subject-program begins.
ADDRESS	The address we are currently looking at.
ADD2	The address beyond which we must not progress.
LIMIT	The address at which the subject-program ends.

I ought to explain here what is meant by "subject-program". The program we are writing is a replacement for HEXLD2. As such it is to be called HEXLD3. This is the "object-program" - the one we are writing now. But the purpose of HEXLD3 is to enable us to be able to create and examine machine code programs. The program that HEXLD3 will be used to examine is called the "subject-program". These distinctions are clearly necessary in order to avoid confusion between the two different concepts. It is of course possible to use HEXLD3 to examine itself, in which case it becomes both the object and the subject, but for the time being keep these two ideas separate in your mind.

The addresses which I've used to store the "variables" BEGIN, ADDRESS, ADD2, and LIMIT are as follows:

Decimal	Hex	Explanation
16514	4082	The start of the subroutine HPRINT
16531	4093	The variable BEGIN
16533	4095	The variable ADDRESS
16535	4097	The variable ADD2
16537	4099	The variable LIMIT
16539	409B	The start of the USR routine HLIST.

Lines 640 and 650 POKE into the variable ADDRESS - Giving the address at which our listing (input in hex as A$) is to begin. This idea of using part of the RAM in machine-code-area to store numbers is a very useful one. You can use it in many different programs. The numbers will be safe there even after the program ends and you are in command mode. You can type RUN or CLEAR and they won't be wiped out. They will even SAVE and reLOAD.

Now for the subroutine HLIST (Short for Hexadecimal List). It is a very very simple routine indeed, and should be no trouble for you to follow.

```
2A9940   HLIST    LD HL,(LIMIT)        Ensure that we don't progress beyond
229740            LD (ADD2),HL         the end of the subject-program.
54                LD D,H
5D                LD E,L
2A9540            LD HL,(ADDRESS)      Compare the current address with
0616              LD B,16 (OLD ROM ONLY)  final address.
A7       NXTAD    AND A
ED52              SBC HL,DE
19                ADD HL,DE
301F              JR NC,DONE
7C                LD A,H               Print the high-part of the current
CD8240            CALL HPRINT          address in hex.
7D                LD A,L               Print the low-part of the current
CD8240            CALL HPRINT          address in hex.
AF                XOR A                Reset A to zero.
D7                RST 10               Print a space.
7E                LD A,(HL)            Print the contents of the current
CD8240            CALL HPRINT          address in hex.
CB76              BIT 6,(HL)           If this character is unprintable
2004              JR NZ,NOPRINT        then do not print it.
AF                XOR A                Reset A to zero.
D7                RST 10               Print a space.
7E                LD A,(HL)            Load A with this character
D7                RST 10               and PRINT it.
3E76     NOPRINT  LD A,76              Load A with a newline character.
D7                RST 10               Print newline.
23                INC HL               Look at the next address.
229540            LD (ADDRESS),HL      Store the current address.
18DB              JR NXTAD (NEW ROM ONLY)
10DB              DJNZ NXTAD (OLD ROM ONLY)
CF       DONE     RST 08               See below.
00                DEFB 00
```

The above program will run as listed on a NEW ROM machine. OLD ROM users should replace every RST 10 instruction by CALL APRINT as before, and are reminded that the JR byte-count must be changed accordingly at two points in the program.

There are several things we can note about this program. Firstly, two new instructions have been used - BIT 6,(HL) and RST 08. Here's what they do.

BIT 6,(HL) tests the value of bit 6 of the address (HL). The result will either be 1 (if bit 6 is 1) or 0 (if bit 6 is 0). This result is not stored in any of the registers, but we can still check it with the next line JR NZ,NOPRINT, which says jump to NOPRINT if the last result (that is bit 6 of (HL)) is not zero.

Why do we need to do this? Take a look at the character set. In particular look at their character codes in hex. Notice that all of the <u>expandable</u> characters lie between C0 and FF (except for RND, INKEY$, and PI on the NEW ROM - these are treated slightly differently by the ROM) and that all of the characters between 40 and 7F are not printable at all (again, except for RND, INKEY$, and PI on the NEW ROM. The machine has to make a special check for these.) (You could argue that the NEW ROM cursor (7F) was printable, but of course it looks different depending on what mode the machine is in.) In fact all of the printable characters are either between 00 and 3F, or between 80 and BF, and conversely every character between 00 and 3F, or 80 and BF, is printable. What have all these in common? The fact that <u>BIT 6</u> of the character code is zero. In binary these codes run between 0000 0000 and 0011 1111, and then from 1000 0000 to 1011 1111. So all we have to do to find out whether or not a character in the set is printable, all we have to do is to look at BIT 6. The above program won't attempt to print them unless BIT 6 is zero. This is because the RST 10 routine won't expand the expandable characters, nor will it replace the others by question marks. It <u>will</u> crash though!

The other new instruction is RST 08. This will cause an immediate return to BASIC, stopping the program with an error code. The byte immediately after the RST 08 instruction tells it which error code to use. An error code 1 needs the data 00, since this byte has to be one less than the report code. If we wanted to be really flash we could have used 1C and got an error code of T!

Now follow the program through carefully and see what it does. Note the way we check whether or not the address ADD2 has been reached (it is stored in DE) - especially the use of AND A to reset the carry flag.

You can check that this program works by POKEing the address at which HPRINT starts into both BEGIN and ADDRESS, and by POKEing the address at which HPRINT ends into LIMIT. Then, if you type RAND USR HLIST (this is the location 16539 using my addresses) you should end up with a more or less instant listing of the subroutine HPRINT.

Now if you simply type RUN and enter 4082 the program will instantly list out the start of this program. In other words we are using it to examine itself. Typing CONT or CONTINUE repeatedly will continue the listing until the end of the program is reached, when you will get a report code of 9.

Now for the second part of our program, HEXLD3. The BASIC part is to look like this:

```
100 PRINT "WRITE"
110 GOSUB 600
120 INPUT A$
130 PRINT A$;"two spaces";
140 RAND USR 16589
150 GOTO 120
```

This part calculates the length of the string A$, which because of the CLEAR Statement in subroutine 600 is the first (and only) item in the variable store.

OLD ROM ONLY:

```
        2A0840    WRITE     LD HL,(VARS)
        E5                  PUSH HL
        06FF                LD B,FF
        23        ANOTHER   INC HL
        7E                  LD A,(HL)
        04                  INC B
        3D                  DEC A
        28FA                JR Z,ANOTHER
        E1                  POP HL
        CB28                SRA B
```

This routine leaves the length of the string divided by two (since it needs two characters to specify one byte of machine code) in the B register and leaves HL pointing to the byte immediately before the start of the contents of the string. Notice how LD A,(HL)/INC B/DEC A/JR Z is used to check for a character 1 (a quote mark, or end of string character) as well as counting the number of characters so far (in B). Can you also see how SRA B will divide B By two?

Strings are stored differently in the NEW ROM. This actually makes things easier, not harder! Look at the corresponding NEW ROM routine which does the same job.

NEW ROM ONLY:

```
16589   2A1040    WRITE     LD HL,(VARS)
        23                  INC HL
        46                  LD B,(HL)
        23                  INC HL
        CB28                SRA B
```

This works because the NEW ROM works by storing the length of a string immediately before the string itself. It takes two bytes for this, but notice that in both of our versions we are only using one byte for the length, so don't input more than 255 characters in one go.

Here's the rest of the routine.

```
28F4                      JR Z,DONE
ED5B9540                  LD DE,(ADDRESS)
23          NEXTBYTE      INC HL
7E                        LD A,(HL)
87                        ADD A,A
87                        ADD A,A
87                        ADD A,A
87                        ADD A,A
23                        INC HL
86                        ADD A,(HL)
C624                      ADD A,24
12                        LD (DE),A
13                        INC DE
ED539540                  LD (ADDRESS),DE
E5                        PUSH HL
2A9940                    LD HL,(LIMIT)
ED52                      SBC HL,DE
E1                        POP HL
3004                      JR NC,CHECK
ED539940                  LD (LIMIT),DE
10E1        CHECK         DJNZ NEXTBYTE
C9                        RET
```

You can learn several things from this routine. Firstly, notice that if you input the empty string the program will jump back to the RST 08 instruction in the previous section. This is so that you can end the program without actually having to break out.

Now look at the first few lines from CHECK onwards. What they do is this - if the end of the program (the program that WRITE is editing) is greater than the current address, do nothing, otherwise make a note of the fact that the program has got longer by altering our variable LIMIT.

You now have two segments of machine code which, if you've typed them in properly, will work first go. Now delete the WHOLE of HEXLD2 (except of course for line 1) but be very careful not to attempt to list line one. The first line now contains more characters, when the keywords in the REM are expanded, than will fit on the screen. In this circumstance the ROM will go into an infinite loop if it tries to list it - this is a design fault - the ROM should not be capable of making infinite loops. You won't be able to break out if it happens. To avoid it, type POKE 16403,10 (OLD) or POKE 16419,10 (NEW). Then type in lines 10 to 30, then delete the rest of the program one line at a time, lowest line number first. Now type in the rest of the program and SAVE it before you do anything else.

For NEW ROM users, it should be made clear that the REM statement will, when keywords are expanded, be longer than will fit on the screen, thus although the command LIST is acceptable (the result of which is that part of line one is listed and an error 4 message displayed), if you LIST 10, to ensure that line 10 is always at the top of the screen (sometimes this doesn't work - if not type POKE 16419,10 which always works) be warned never to delete line 10. If you do the ROM will go into an infinite loop trying to reshuffle the lines so that it can list them. In SLOW this can be quite amuzing to watch, but it is always irritating because the only way you can get out of it is by pulling the plug.

Now to complete the transition from HEXLD2 to HEXLD3 let's rewrite the section that will SAVE things in upper memory. The BASIC:

OLD ROM	NEW ROM
400 DIM O(USR(ARRAY))	400 DIM O$(USR ARRAY)
410 RANDOMISE USR(STORE)	410 RAND USR STORE
420 SAVE	420 SAVE "HEXLD3"
500 RANDOMISE USR(RETRIEVE)	500 RAND USR RETRIEVE
510 CLEAR	510 CLEAR
520 STOP	520 STOP

as you can see there are three different parts of machine code. The first, in line 400, alters nothing, but returns a numerical value to BASIC, which is then used by BASIC to reserve the correct amount of space using a DIM statement. Let s look at that part first:

Using my addresses, ARRAY is 16635, STORE is 16651, and RETRIEVE is 16669.

```
        2A9940    ARRAY     LD HL,(LIMIT)
        ED5B9340            LD DE,(BEGIN)
        A7                  AND A
        ED52                SBC HL,DE
        229740              LD (ADD2),HL
for the OLD ROM only:
        CB2C                SRA H
        CB1D                RR L
for the NEW ROM only:
        44                  LD B,H
        4D                  LD C,L
for both:
        C9                  RET
```

The first part is obvious. The beginning address is subtracted from the end address.Again we see AND A being used to zero the carry flag so that SBC gives the right answer. Now, for OLD ROM users, this number is divided by two, because arrays use two bytes per element. For NEW ROM users we move the answer into the BC register because this is what will return to BASIC. Now for the machine code that accompanies line 410. Use RUN 100 to load it in the first place.

You may be wondering why ADD2 was loaded with the number of bytes in the code to be SAVEd. Well ADD2 is just a convenient place to store it, since it will be needed in line 410.

```
2A1040      STORE       LD HL,(VARS)
110600                  LD DE,0006
19                      ADD HL,DE
EB                      EX DE,HL
2A9340                  LD HL,(BEGIN)
ED4B9740                LD BC,(ADD2)
EDB0                    LDIR
C9                      RET
2A1040      RETRIEVE    LD HL,(VARS)
110600                  LD DE,0006
19                      ADD HL,DE
ED5B9340                LD DE,(BEGIN)
ED4B9740                LD BC,(ADD2)
EDB0                    LDIR
C9                      RET.
```

In case you're beginning to lose track, here's a quick round up of all the addresses we've used so far:

Decimal	Hex	Routine/Variable
16514	4082	HPRINT
16531	4093	BEGIN
16533	4095	ADDRESS
16535	4097	ADD2
16537	4099	LIMIT
16539	409B	HLIST
16589	40CD	WRITE
16635	40FB	ARRAY
16651	410B	STORE
16669	411D	RETRIEVE
16687	412F	next spare byte.

Briefly, STORE moves machine-code from upper memory and stores in an array. RETRIEVE moves it back from the array to its previous position. Both of the routines start off by working out the address of the first free byte in the array. The array is the first item in the variable store, but because the OLD and NEW ROMs think differently, we have to add two to this location in the OLD ROM, and six on the NEW ROM. Can you spot the different ways in which this is done?

This is also the first time we've used the instruction LDIR. What is does is to automatically move a block of elements from address (HL) to address (DE), assuming that the number of elements contained in this block is BC. This is of course precisely what we want to do. LDIR does alter the value of each of the register pairs BC, DE, and HL, but that doesn't concern us since the next thing we do is RET.

LDIR is very, very useful indeed, but you must remember which way round it goes. It loads from (HL) into (DE). Have you ever pressed 'record' instead of 'play' when trying to load programs from tape? Well that's exactly what will happen to your machine code if you get DE and HL the wrong way round for LDIR - it will just be wiped out - and there's no going back.

As long as you can see exactly what's happening you're OK. If you can't then get a piece of paper and write down the values of each register at each stage. Work through until you're convinced you know exactly what's happening all the way through.

We now have a BASIC program called HEXLD3 which contains a fair number of machine code subroutines. As it stands it will both LIST and WRITE machine code, and can also be used to SAVE any machine code or data which is stored in spare RAM space high in memory. This is all that HEXLD2 did. You now have the ability to enter your own machine code programs very easily, but what you can't yet do is edit them if you make a mistake. That is what the next section is for - it is called INSERT, and will insert whatever you input between the surrounding code, without over-writing it. The BASIC part of the routine is this

```
200 PRINT "INSERT"
210 GOSUB 600
220 INPUT A$
230 PRINT A$;"two spaces"
240 RAND USR 16687
250 GOTO 220
```

And the machine code which goes with it (which NEW ROM users should write to address 16687) is as follows:

OLD ROM

```
2A0840    INSERT    LD HL,(VARS)
E5                  PUSH HL
01FFFF              LD BC,FFFF
23        MORE      INC HL
7E                  LD A,(HL)
03                  INC BC
3D                  DEC A
28FA                JR Z,MORE
E1                  POP HL
```

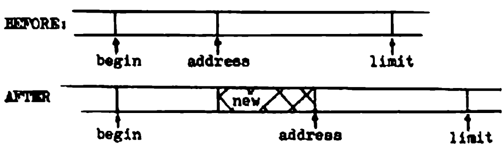
NEW ROM

```
2A1040    INSERT    LD HL,(VARS)
23                  INC HL
4E                  LD C,(HL)
23                  INC HL
46                  LD B,(HL)
```

```
CB28        COPYUP      SRA B
CB19                    RR C
2002                    JR NZ,NOTEMPTY
CF                      RST 08
08                      DEFB 08
C5          NOTEMPTY    PUSH BC
2A9940                  LD HL,(LIMIT)
ED5B9540                LD DE,(ADDRESS)
A7                      AND A
ED52                    SBC HL,DE
23                      INC HL
44                      LD B,H
4D                      LD C,L
E1                      POP HL
ED5B9940                LD DE,(LIMIT)
19                      ADD HL,DE
229940                  LD (LIMIT),HL
EB                      EX DE,HL
EDB8                    LDDR
CDCD40                  CALL WRITE
C9                      RET
```

Now exactly how this works is quite complicated, so think carefully. The part between INSERT and COPYUP finds the length of the string A$. As you can see it required a completely different method for each ROM. See WRITE on this, since it is very similar here.

Between COPYUP and NOTEMPTY the length of the string is divided by two, and if it is zero returns to BASIC with error code 9. This is the job of the RST 08/DEFB 08 sequence. From then on we are concerned with moving part of the program being edited. Look at the diagram below.



As you can see, we need to load a complete block of elements from one point to another, but unlike before the new and old positions overlap. This is a slight problem, and we have to be very careful how we load it. If we were to simply assign HL to ADDRESS(before) and DE with ADDRESS(after), and then use LDIR as before (having assigned BC to the number of elements in the block first) then since LDIR moves things one byte at a time the first few elements would end up in the middle of the block, only to be copied up for a second time. The program would be completely corrupted.

We can get round this flaw by sneaking up on the problem sideways while it's not looking. What we do is we block load it from the other end! This means loading HL with LIMIT(before) and DE with LIMIT(after) and use LDDR instead of LDIR.

Having found the length of the new section, this length is pushed onto the stack. BC is then loaded with the length of the block to be moved. See how this is worked out. Then HL and DE are correctly assigned, making use of the fact that the length of the new section is at the top of the stack, and the new limit is stored in our "variable" LIMIT.

After the block load is successfully carried out we call the WRITE subroutine to fill the shaded area in the diagram with the contents of the input string. This will work because the above program does not change the value of the variable ADDRESS. WRITE will simply overwrite the shaded region, moving the current address pointer to its new position. We then return to BASIC for the next input.

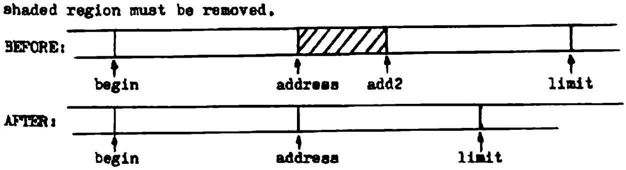
To test the program, use WRITE to write "9D9E9FA0A1A2A3A4A5" to the point just beyond where our program currently ends. This will list as [redacted] Now use INSERT. Give it the address of the inverse five, and input "00"/ "201E"/"00". Here / means newline. When you list it you'll find four new characters have been inserted. Notice that the routine allows you to input as many characters as you like in one go, and that it allows you to press newline as many times as you like. Newline on its own (ie inputting the empty string)will break out of the program.

The final section to add to our program is DELETE. This will look (in BASIC) like this

```
300 PRINT "DELETE"
310 GOSUB 600
320 LET A 16535
330 GOSUB 610
340 RUN USR 16732
```

The first four lines load the initial and final addresses into the variables ADDRESS and ADD2. Line five calls the machine language routine that will do the task for us.

Here's what the machine code has to do. Look at the diagram below. Here the shaded region must be removed.

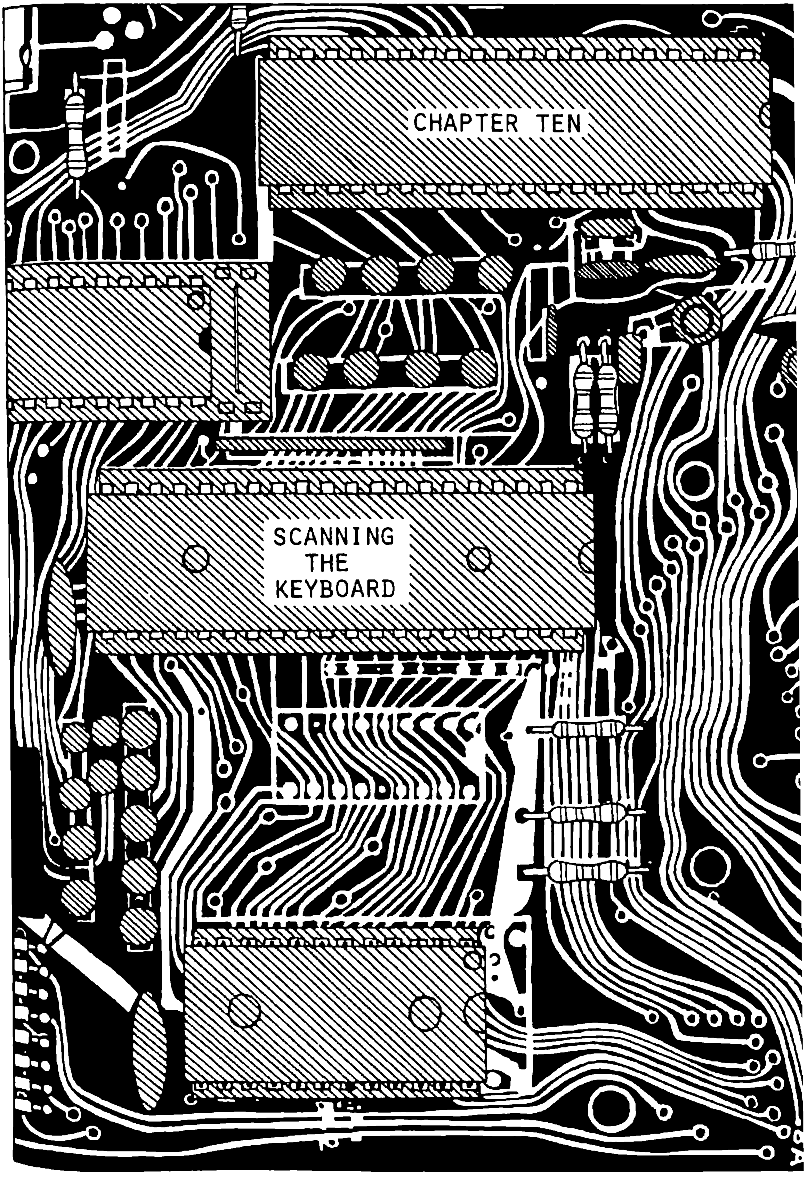


This is quite simple - we just use LDIR quite straightforwardly. You might think there would be some effort involved in calculating the new limit, but not so. LDIR alters the value of HL and DE for us in quite an advantageous way - as we shall see.

```
2A9940      DELETE    LD HL,(LIMIT)
ED5B9740              LD DE,(ADD2)
D5                    PUSH DE
A7                    AND A
ED52                  SBC HL,DE
44                    LD B,H
4D                    LD C,L
E1                    POP HL
23                    INC HL
ED5B9540              LD DE,(ADDRESS)
EDB0                  LDIR
1B                    DEC DE
ED539940              LD (LIMIT),DE
CF                    RST 08
08                    DEFB 08
```

As LDIR moves from one end of the blocks being shifted to the other, HL and DE move with it, so HL ends up to the right of the original block, and DE ends up to the right of the copy. Thus a simple DEC DE after the LDIR will set it to exactly the right place for our new limit. Load this routine to address 410D (OLD)/415A (NEW), using INSERT. You should now have one or two spare characters after the end of the program. Use DELETE to wipe them out - this will of course test whether or not you have typed in DELETE correctly.

Now SAVE this program permanently. This is the final version. All you have to do in order to use it in future is to type RUN 100 and enter the address of the variable BEGIN. (403C or 4093). Then input the address to which the program you are about to write will begin, then simply newline on its own. RUN 100 a second time to actually begin inputting a program.


CHAPTER TEN
SCANNING
THE
KEYBOARD

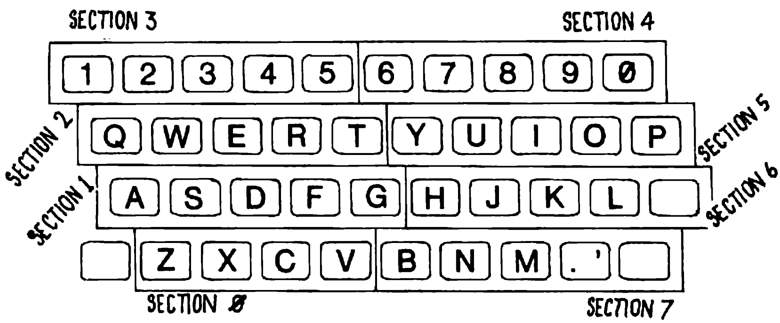
Now it's time to explore how we can make use of some of the other subroutines that are remarkably well-hidden within the ROM. Specifically we'll cover two of these subroutines, which between them will enable us to scan the keyboard and locate which, if any, of the keys on the keyboard are being depressed. On the NEW ROM we can make use of these subroutines just by calling them, but we can't on the OLD ROM because they're simply not there. For the benefit of the people with OLD ROMs I shall include a section at the end of this chapter explaining how these programs may be made to work by actually inputting these subroutines yourselves. This section will also be of interest to those of you with NEW ROMs, since it will give you an insight into how the subroutines actually work.

The first such subroutine is an amazing little keyboard scan, which begins at address 02BB. It may be accessed simply by calling that address, ie CALL KSCAN, or CDBB02 in hex. It doesn't actually produce a very useable answer though. Let's see exactly what it does do.

It returns a value to the HL register pair. Actually it returns separate and independent values - one to H and one to L. Here's how the value of L is interpreted:

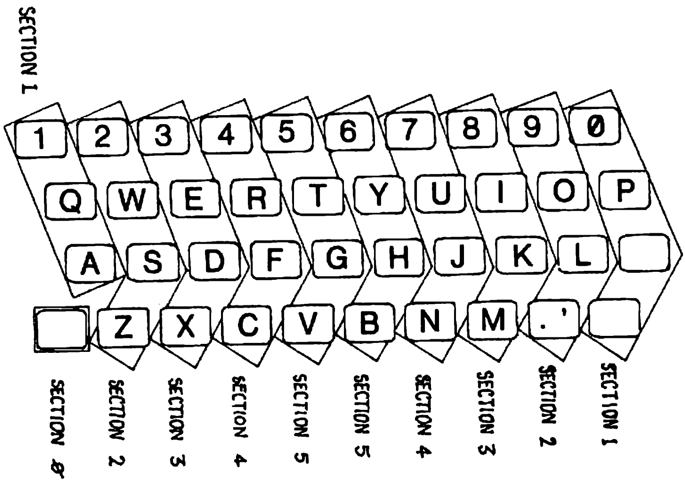
Imagine the keyboard (excluding SHIFT) divided up into eight horizontal sections, each containing five keys (except for section zero which only contains four, because SHIFT is ommitted). Notice how each section has a corresponding number between zero and seven. Now, if there is no key depressed then L will return a value FF. However, if one or more keys are depressed, then the appropriate BIT (of L) will be reset to zero. In other words, if you are pressing Q, W, E, R, or T then bit 2 will be reset - if you are pressing B, N, M, full-stop, or space, then bit 7 will be reset. This means that L can return the following:

	BINARY	HEX
If no key is depressed	11111111	FF
If a section 0 key is depressed	11111110	FE
If a section 1 key is depressed	11111101	FD
If a section 2 key is depressed	11111011	FB
If a section 3 key is depressed	11110111	F7
If a section 4 key is depressed	11101111	EF
If a section 5 key is depressed	11011111	DF
If a section 6 key is depressed	10111111	BF
If a section 7 key is depressed	01111111	7F



As an exercise see if you could work out what L would return if both S and P were depressed at the same time.

The value returned by H is determined by a similar principle, but notice how the keyboard is divided up here - not horizontally but vertically. Notice also that the SHIFT key has a section all to itself - section 0. Now if you press key S for instance then H will return FB (in binary 11111011). We have already seen that L would give FB as well, so that HL returns FBFB. Can you see why it is impossible for this value to be obtained from any other key?



Now let's see what would happen if you pressed SHIFT S. Both bits zero and two would be reset giving, in binary, 11111010. In hex this is FA, so HL would return as FAFB - which is different to the value produced without shift. We can see the precise effect of SHIFT from this table:

	WITHOUT SHIFT		WITH SHIFT	
	BINARY	HEX	BINARY	HEX
If no key is depressed	11111111	FF	11111110	FE
If a section 1 key is depressed	11111101	FD	11111100	FC
If a section 2 key is depressed	11111011	FB	11111010	FA
If a section 3 key is depressed	11110111	F7	11110110	F6
If a section 4 key is depressed	11101111	EF	11101110	EE
If a section 5 key is depressed	11011111	DF	11011110	DE

It should by now be reasonably clear how each individual key, with or without shift, produces its own unique code in the HL register pair. If two keys are both in the same horizontal section they cannot possibly both be in the same vertical section. Note that SHIFT on its own returns a value of FEFF which is not the same as no key depression at all.

The subroutine which I've called KSCAN does have one big disadvantage though - it will completely wipe out the previous values of all the registers! If you want to preserve them you'll have to make use of the stack as follows:

```
F5        PUSH AF
C5        PUSH BC
D5        PUSH DE
CDBB02    CALL KSCAN
D1        POP DE
C1        POP BC
F1        POP AF
```

Now we want to turn these rather obscure numbers into real character codes. It just so happens that all of these codes are rather cleverly stored in the ROM beginning at address 007E. By "rather cleverly" I mean in a most convenient order, as follows: First the straightforward characters:

007E   ZXCV ASDFG QWERT 12345 09876 POIUY newline LKJH space .MNB

(There are no spaces between the characters - they are printed here to make the ordering more obvious.) Then the shift characters:

00A5   :;?/ STOP LPRINT SLOW FAST LLIST "" OR STEP <= <> EDIT AND THEN TO cursor-left RUBOUT GRAPHICS cursor-right cursor-up cursor-down ")($ >= FUNCTION =+- ** £,><*

Can you see how the ordering relates to which sections the key lies in? We could quite easily write a subroutine now to convert from the strange number we already have in HL to an address between 007E and 00CB (the last item in the table - the *) but it turns out that we don't need to because that nice man Uncle Clive has already done it for us with a subroutine which I shall call FINDCHR beginning at address 07BD. The ROM people probably have their own name for it but they keep it shrouded in mystery. The subroutine performs the following task - given a value as defined above, in the BC register, it will work out the address at which the appropriate character is stored - the final result ending up in HL.

It does have a problem though. If you're not pressing a key then surely you shouldn't end up with a character to print! You'll have to prevent this yourself. One way would be as follows. Notice though how we move the result from the first subroutine into the BC register before calling the second.

```
CDBB02    START     CALL KSCAN
44                  LD B,H
4D                  LD C,L
51                  LD D,C
14                  INC D
3E00                LD A,00
2804                JR Z,NOCHR
CDBD07              CALL FINDCHR
7E                  LD A,(HL)
...       NOCHR     ...
                    rest of program
```

There are several things to note about this example. Firstly that two separate instructions, LD B,H and LD C,L, are needed to transfer HL to BC since there is no single instruction LD BC,HL. Secondly that the condition JR Z means if D is zero, not A - LD does not alter any of the flags. If D is zero after being incremented then it must have been FF beforehand, which means that L must have been FF after it came out of the first subroutine. This is the check that a key is being pressed. A is loaded with zero and if no key is pressed it remains zero, otherwise it takes on the code of whichever character you're touching on the keyboard.

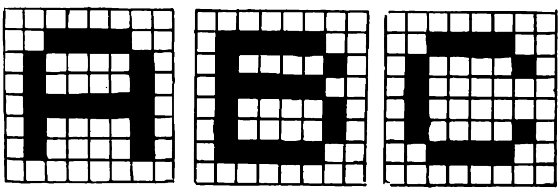
There is here a slight ambiguity in that zero is also produced if you press space. You could use LD A,01 instead of LD A,00 since the character whose code is one (▘) is not available from the keyboard. Now there is no ambiguity since zero means space and one means no character is being pressed. If you have SLOW at your disposal you could omit LD A,00 altogether and use JR Z,START instead of JR Z,NOCHR. Now the program will WAIT until a key is pressed before continuing. Without SLOW it will still wait but you'll have to suffer a blank screen in the meantime.

The A register now contains a value corresponding precisely to the function INKEY$. In this way real time games are just as feasable in machine code as they are in BASIC.

Another interesting part of the ROM is the very last bit - the half of a K that runs from 1E00 to 1FFF. It's not a subroutine, it's a table - a very long table - actually the longest table in the ROM. It stores the dot pattern of every symbol used by the computer - that is all of the printable characters. It takes eight bytes to store a single character symbol, so for example, the characters A, B and C are represented, in binary, by:

```
0 0 0 0 0 0 0 0    0 0 0 0 0 0 0 0    0 0 0 0 0 0 0 0
0 0 1 1 1 1 0 0    0 1 1 1 1 1 0 0    0 0 1 1 1 1 0 0
0 1 0 0 0 0 1 0    0 1 0 0 0 0 1 0    0 1 0 0 0 0 1 0
0 1 0 0 0 0 1 0    0 1 1 1 1 1 0 0    0 1 0 0 0 0 0 0
0 1 1 1 1 1 1 0    0 1 0 0 0 0 1 0    0 1 0 0 0 0 0 0
0 1 0 0 0 0 1 0    0 1 0 0 0 0 1 0    0 1 0 0 0 0 1 0
0 1 0 0 0 0 1 0    0 1 1 1 1 1 0 0    0 0 1 1 1 1 0 0
0 0 0 0 0 0 0 0    0 0 0 0 0 0 0 0    0 0 0 0 0 0 0 0
```

Can you pick the letters A, B and C from the digits above? The pattern is held by the positions of the "ones" amongst the "zeroes". When they finally appear on your TV screen they look like this:



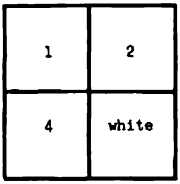
Suppose we now wished to reconstruct these letters in an enlarged form - using a pixel (quarter-square) for each dot. This means that each character we print should be a graphics character (space and inverse-space both count as graphics characters) chosen so that the correct quarters are black.

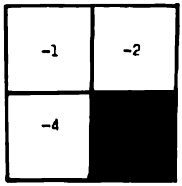
There are two ways of doing this. One is to make use of the NEW ROM character codes, in which the graphics are arranged in a very clever order - unfortunately we would not be able to adapt this sytem to the old ROM. The second is to include sixteen bytes of data within our program representing the graphics symbols in any order we care to choose. Let's take a look at the first method first.

Suppose the bottom right corner is WHITE. If we give the other pixels numbers 1, 2 and 4 then simply adding them up gives the required character code. You can check this by comparing the diagram below with the character set in the Sinclair manual.

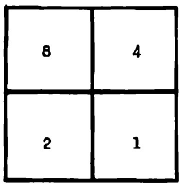
If the bottom right hand corner is BLACK then we need to give the other pixels the numbers -1, -2, and -4. To work out the code of any graphics symbol here we start off with the number 135 (decimal) and subtract appropriately the required number for each black pixel. Again you can check this by comparing the diagram below with the Sinclair Manual.

Incidently it is worth pointing out here that many copies of the Sinclair Manual incorrectly give the character of 135 as . This is a misprint - it should of course be . Try typing PRINT CHR$ 135 to check. Character seven is - the manual gives this correctly.





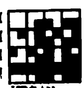
The character code of the OLD ROM graphics symbols are unfortunately rather random, so there is no simple system for working out the code, given which pixels should be black and which should be white. In order that the program to follow should work on both ROMs we shall adopt a slightly different method. Instead of distinguishing two different cases (that is the colour of the bottom right-hand pixel) we shall treat every quarter-square the same, and code them as follows:



We would then have to include in our program a DATA section which lists the graphics symbols in the order space 

Move RAMTOP to address 4380 (this is a hex address) by typing POKE 16388,128 POKE 16389,67 then NEW. Now load the following program to address 4380. (In decimal this is 17280, meaning 1K users will still be able to run it.) As it stands this program is best run in SLOW. We shall see how to alter it so that it will run in FAST later.

```
00870483   DATA    DEFM      This is the table of
02850681           DEFM      graphics symbols in
01860582           DEFM      the required order.
03840780           DEFM
CDBB02     START   CALL KSCAN   Wait for human to take
2C                 INC L        finger off of key.
20FA               JR NZ,START
```



```
CDBB02   WAIT        CALL KSCAN          Wait for new key to be
44                   LD B,H              pressed.
4D                   LD C,L
51                   LD D,C
14                   INC D
28F7                 JR Z,WAIT
CDBD07               CALL FINDCHR        Locate appropriate
7E                   LD A,(HL)           character code.
A7                   AND A
17                   RLA                 Multiply by eight, but
17                   RLA                 return to BASIC if a non-
D8                   RET C               printable character is
17                   RLA                 pressed .
1600                 LD D,00
CB12                 RL D
5F                   LD E,A
21001E               LD HL,CTABLE        Find start of dot pattern.
19                   ADD HL,DE
0E04                 LD C,04
0604     OUTERLOOP   LD B,04
56                   LD D,(HL)           Transfer two lines of dots
23                   INC HL              into D and E
5E                   LD E,(HL)
23                   INC HL
E5                   PUSH HL
AF       INNERLOOP   XOR A               Compute which graphics
CB12                 RL D                character is to be
17                   RLA                 printed.
CB12                 RL D
17                   RLA
CB13                 RL E
17                   RLA
CB13                 RL E
17                   RLA
218043               LD HL,DATA          Get this character from
85                   ADD A,L             the table of graphics
6F                   LD L,A              symbols.
7E                   LD A,(HL)
D9                   EXX                 Print this symbol
CD0808               CALL PRINT
D9                   EXX
10E6                 DJNZ INNERLOOP      Next print position.
E1                   POP HL
3E76                 LD A,76             End of current line.
D9                   EXX
CD0808               CALL PRINT
D9                   EXX
0D                   DEC C
20D4                 JR NZ,OUTERLOOP     Next line begins.
18AF                 JR START            Start again.
```

The program is now complete. Make sure you are in SLOW mode and start the program off by typing RAND USR 17296. <u>DO NOT</u> type RAND USR 17280 since this is purely data and will not run. You should see a completely blank screen. Press "C", and watch what happens. Now press "A". Interesting isn't it? Try experimenting with different keys to see what happens - and what happens when you run out of screen?

You may have been confused by the use of the instruction EXX which was used four times in the above program. Its function is very easy to explain. As you know, the registers B, C, D, E, H, L, and A can be very easily manipulated, but there are also seven other registers, called B', C', D', E', H', L', and A'. (Pronounced A-dash or A-prime.) These are not so easy to manipulate and can in practice only be used for storage purposes. The instruction EXX means exchange B and B', C and C', D and D', E and E', H and H', L and L'. Thus all the registers except A lose their previous values but take on the values of their alternative registers. Likewise the alternative registers take on the original values of the usual registers.

The reason we need to do this is because the ROM subroutine PRINT destroys the previous values of BC, DE, and HL. We could have preserved them by pushing them onto the stack, but EXX works just as well here and is only one instruction.

Lets take a closer look at the above program and sort out exactly what each bit does. First of all we find the right character code, which gets stored in the A register. The instruction AND A resets the carry flag to zero. RLA will then multiply A by two. Now we know that this character is on the keyboard and can be obtained in one touch, so it is not an inverse character. Rotating left then will move the leftmost bit, which must be a zero, into the carry flag. RLA a second time will again multiply by two (since we know the carry is zero), however, if the character is NOT PRINTABLE (such as newline or STOP) then bit 6 of the original value will be a one. This will now be moved into carry. The instruction RET C ensures that if this circumstance ever occurs the program will terminate.

Knowing then that the carry is still zero we can use RLA once more to multiply by two. Here however, bit 5, which a necessary part of the character code, will be moved into the carry flag. To move the carry digit into D we use two instructions LD D,00 and RL D. D will then contain either zero or one. LD E,A ensures that register-pair DE now contains eight times the original value of A.

The other interesting part is the first nine lines of the INNER-LOOP. A is loaded with the first two bits of D and the first two bits of E. This gives a number between zero and fifteen which corresponds to the required graphics symbol. It is NOT the character code, it is the specially designed code we worked out earlier on. Notice how the NEXT bits of D and E are now automatically in place at the extreme left.

For those of you who do not have SLOW I suggest replacing the last instruction, JR START by RET. You could then have a surrounding BASIC program as follows:

```
10 RAND USR 17296
20 PAUSE 40000
30 RUN
```

THE SUBROUTINES

Old ROM users will by now be feeling quite envious at NEW ROM people for having these subroutines at their disposal. Of course there is a keyboard scan in the OLD ROM, but it isn't a subroutine - ie it doesn't end in RET. One call to it and you're stuck there forever! What we'll have to do is rewrite them ourselves. We can do this by taking all of the important bits from the subroutines in the NEW ROM.

First of all KSCAN. This is the required subroutine. Don't worry if there are parts of it you don't understand - all will become clear in due course.

```
21FFFF     KSCAN     LD HL,FFFF
01FEFE               LD BC,FEFE
ED78                 IN A,(C)
F601                 OR 01
F6E0       LOOP      OR E0
57                   LD D,A
2F                   CPL
FE01                 CP 01
9F                   SBC A,A
B0                   OR B
A5                   AND L
6F                   LD L,A
7C                   LD A,H
A2                   AND D
67                   LD H,A
CB00                 RLC B
ED78                 IN A,(C)
38ED                 JR C,LOOP
1F                   RRA
CB14                 RL H
C9                   RET
```

Now - if you enter this subroutine into RAM you can then replace every CDBB02 in the chapter by a call to the appropriate address in RAM. The other subroutine you'll need to be able to emulate is FINDCHR. This may be done as follows.

```
1600       FINDCHR   LD D,00
CB28                 SRA B
9F                   SBC A,A
F626                 OR 26
2E05                 LD L,05
95                   SUB L
85         LOOP      ADD A,L
37                   SCF
CB19                 RR C
38FA                 JR C,LOOP
0C                   INC C
C0                   RET NZ
48                   LD C,B
2D                   DEC L
2E01                 LD L,01
20F2                 JR NZ,LOOP
217D00               LD HL,KTABLE-1
5F                   LD E,A
19                   ADD HL,DE
C9                   RET
```

The address 007D, referred to in my listing as KTABLE-1, is for the NEW ROM only. THE ADDRESS OF KTABLE IN THE OLD ROM IS 006C, and so this line should be changed to LD HL,006B. This is far easier to understand than the first subroutine. The second and third lines are rather interesting.

If you remember BC should contain a code corresponding to one of the keys at the start of the subroutine. Now bit zero of B is a one if SHIFT is not pressed, and zero if shift is pressed. SRA B will shift B to the right, will set bit 7 to one (Do you remember the difference between SRA and SRL?), and will set the carry flag equal to the previous value of bit zero.

SBC A,A will first of all subtract A from A - effectively reseting it to zero - and will then subtract the carry flag. In other words, if SHIFT is pressed A will end up as 00, if SHIFT is not pressed A will end up as FF.

The fourth line, OR 26, will ensure that A is 26 for a shifted character, FF for a non-shifted character.

You should recall here that B contains information about which VERTICAL section the key is in, and C about which HORIZONTAL section. If you take a closer look at the order the characters are stored in the keyboard table (KTABLE) which was shown a few pages back you'll see that the horizontal-section-number needs to be multiplied by five, and the vertical-section-number added to it, in order to find a specific key in the table. This is what the next part does:

L is loaded with 5 - the multiplying factor. Notice how the next two lines cancel each other out the first time round the loop. This is one way of adding L nought times should we need to. The next two lines are SCF and RR C. This is not the same thing as SRA C, since bit 7 could be zero. (ie if a horizontal-section-7 key is pressed.) Apart from shifting C to the right it also moves one bit into the carry. If this bit is a one we haven't found the right section yet and the loop is re-executed. Note that five is added each time round the loop. Note also that if A starts off as FF it is just as easy, if not easier, to think of it as minus-one.

Now that we're out of the loop, C should be all ones, that is, it should be FF, so that INC C should ensure that it becomes zero, so what's this RET NZ instruction for? Of course this condition is simply to check that you're not holding down two keys at once. What would C contain if you were?

LD C,B moves the vertical-section-data into the B register, so that the same loop may be used over again.

DEC L followed by LD L,1 looks confusing. Actually it's not. At the moment L is five, and so DEC L makes it four, which is NOT ZERO. LD L,1 doesn't alter the zero flag, so JR NZ,LOOP sends it back through the same loop, but this time checking the vertical sections, and only incrementing by one instead of five.

When it comes out of the loop DEC L will reduce to zero, so after reloading L with one JR NZ will not be satisfied and the program will continue.

LD HL,KTABLE-1. Why minus one? Well if there was a "real key" in the position where SHIFT is and you were pressing it then A would end up as zero. Since there isn't the smallest value A can end up as is one, which happens if you hold down "Z", hence LD HL,KTABLE-1 takes this into account.

LD E,A is effectively loading A into DE. This works because D is already zero - see the first line of the program. Then ADD HL,DE will find the correct address. Notice that we could have replaced these two instructions by ADD A,L followed by LD L,A. This has the advantage that the first instruction (LD D,00) becomes unnecessary, and that DE is not at all altered by the subroutine. The ROM however uses the version as listed.

KTABLE in the OLD ROM looks like this:

```
006C    ZXCV ASDFG QWERT 12345 09876 POIUY newline LKJH space .MNB
0093    ::?/[illegible] NOT AND THEN TO cursor-left RUBOUT HOME
        cursor-right cursor-up cursor-down *)($" edit =+- ** £,>< OR
```

For the actual printing process itself, the instruction CALL PRINT for the NEW ROM should be replaced by PUSH AF/CALL PRPOS/POP AF/CALL PRINT. In HEX this is F5/CDE006/F1/CD2007.

The Character Table (CTABLE) which gives the dot patterns for the characters is located in the OLD ROM at address 0E00, rather than 1E00. Again it is stored at the very end of the ROM. All of the characters are slightly different.

The data for the table of graphics symbols in the character printing program should run 00 07 06 03 05 82 08 84 04 88 02 85 83 86 87 80 if the program is to be used with an OLD ROM. Replace the PAUSE 40000 BASIC statement given in the following text by INPUT A$

GRAFFITTI

It only requires a slight modification to the original version in order to make a really excellent program, demonstrating the immense speed which machine code offers over BASIC. In this program, GRAFFITTI, you touch a key and an enlarged version of the required symbol appears on the screen. In this program the whole screen is used (even the bottom two lines) thus allowing a total of forty-eight symbol on the screen. To load it move RAMTOP to any address not less than 4D00 and NEW (ie this can't be done in 1K - at least not in this version). The program is as follows.

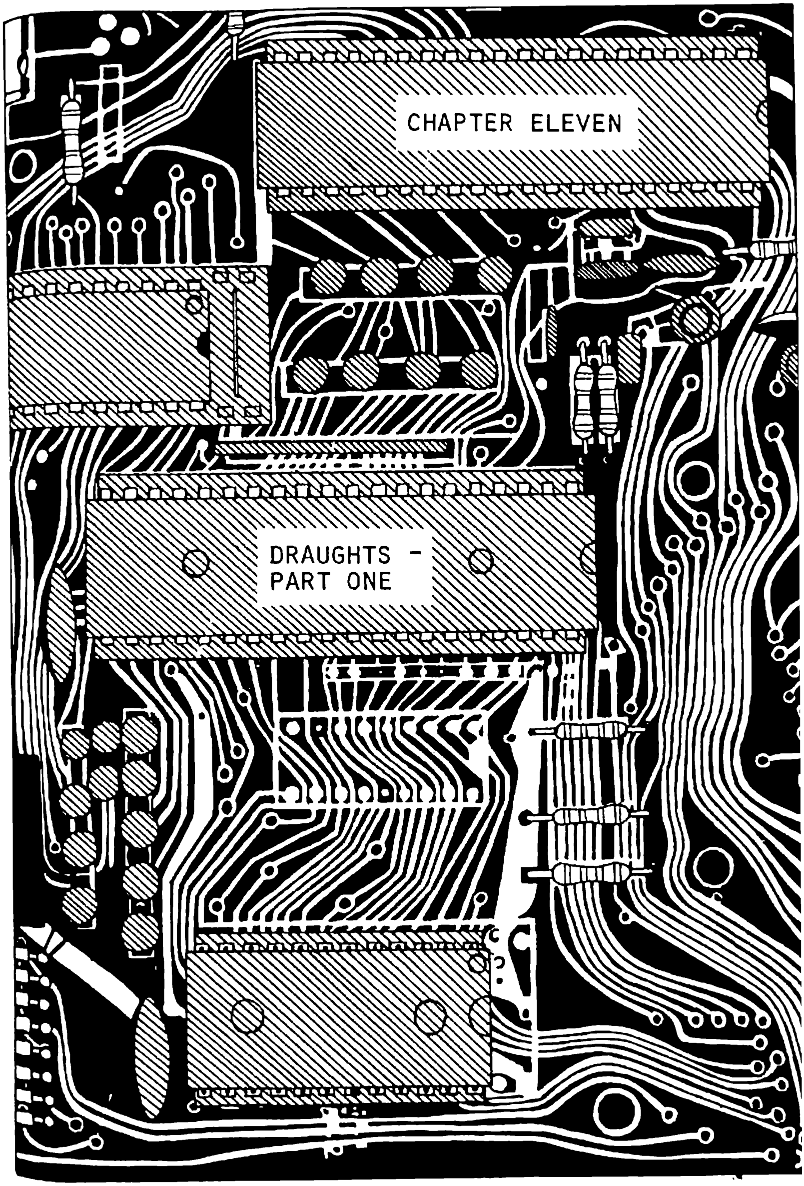
```
2A0C40   GRAFFITTI LD HL,(D-FILE)    Set the print position
23                 INC HL            to the start of the screen.
220E40             LD (DF-CC),HL
36B0     START     LD (HL),B0        Print a cursor
CDBB02   PAUSE     CALL KSCAN        Wait for human to take
2C                 INC L             finger off of key.
20FA               JR NZ,PAUSE
CDBB02   WAIT      CALL KSCAN        Wait for new key to
44                 LD B,H            be pressed.
4D                 LD C,L
51                 LD D,C
14                 INC D
28F7               JR Z,WAIT
CDBD07             CALL FINDCHR      Locate the correct
7E                 LD A,(HL)         character code.
A7                 AND A
17                 RLA               Multiply by eight, but
17                 RLA               return to BASIC if a
D8                 RET C             non-printable character
17                 RLA               is pressed.
1600               LD D,00
CB12               RL D
5F                 LD E,A
21001E             LD HL,CTABLE      Find start of dot
19                 ADD HL,DE         pattern.
0E04               LD C,04
0604     OUTERLOOP LD B,04
56                 LD D,(HL)         Transfer two lines of
23                 INC HL            dots into D and E
5E                 LD E,(HL)
23                 INC HL
E5                 PUSH HL
AF       INNERLOOP XOR A             Compute which graphics
CB12               RL D              character is to be
17                 RLA               printed.
CB12               RL D
17                 RLA
CB13               RL E
```

```
17                  RLA
CB13                RL E
17                  RLA
21-data-address     LD HL,DATA          Get this character from
85                  ADD A,L             the table of graphics
6F                  LD L,A              symbols.
7E                  LD A,(HL)
2A0E40              LD HL,(DF-CC)       Print this character
77                  LD (HL),A           in required position.
23                  INC HL              Store new print
220E40              LD (DF-CC),HL       position.
10E3                DJNZ INNERLOOP
D5                  PUSH DE             Move print position
111D00              LD DE,001D          ready for next row
19                  ADD HL,DE           of symbols.
220E40              LD (DF-CC),HL
D1                  POP DE
E1                  POP HL
0D                  DEC C
20CF                JR NZ,OUTERLOOP
1180FF              LD DE,FF80          Move print position
2A0E40              LD HL,(DF-CC)       ready for next enlarged
19                  ADD HL,DE           character.
220E40              LD (DF-CC),HL
7E                  LD A,(HL)           Check for end of line.
FE76                CP 76
209B                JR NZ,START
116400              LD DE,0064          Move print position to
19                  ADD HL,DE           left of screen ready for
220E40              LD (DF-CC),HL       next enlarged character.
23                  INC HL
ED5B1040            LD DE,(VARS)        Check for end of screen.
ED52                SBC HL,DE
19                  ADD HL,DE
383A                JR C,START
C9                  RET
```

This program is intended to be run on a ZX81 in the SLOW mode. See if you can work out how to adapt it so that it will print inverse characters instead of ordinary ones. It may even be possible to offer a choice!

CHAPTER ELEVEN

DRAUGHTS - PART ONE



DRAUGHTS

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

```
1 INPUT A$
2 RAND/RANDOMISE USR(something)
```

As you can see, the vast, vast majority of it will be entirely in machine code. The machine code will begin immediately after the BASIC program ends. However, in order that we may edit it we shall temporarily store it a little higher up in memory than that - in the fourth K.

Also, in order that we may have the BASIC part of the program at the right location it will be necessary to MOVE the machine code part of HEXLD3. New ROM users should start typing POKE 16389,74, and then NEW, and then load the program HEXLD3.

Now, to move it, write the following program to the few spare characters at the end of the REM statement:

```
010002   MOVE   LD BC,0200
11004A          LD DE,4A00
210040          LD HL,4000
EDB0            LDIR
C9              RET              Run this.
```

Now for the tedious bit. Every address used in the machine code part either begins with 40 or 41. You'll have to go through the listing and change each 40 to a 4A, and each 41 to a 4B. (The changes are to be made in the copied version, not the original version.) That done, change every address in the BASIC parts that calls a USR routine. To make a change you must add 2560 to each number. Now delete line one by typing its line number. The program should still work, but now you'll need to type RUN 400 in order to SAVE it. Make sure that the variable BEGIN (Now at 4A3C or 4A93) contains a value of 4A00. New ROM users change line 500 to:

500 RAND USR (PEEK 16400+256*PEEK 16401+161) -In this way RETRIEVE is called from within the variables area, ie address (VARS)+A1.

Now type RUN 100 to start the WRITE routine and re-enter the board printing routine. Again you'll need to load it to address 4C09. The listing is the same as it was before. Turn to chapter seven and simply retype the whole thing.

The instruction RAND USR 19477 should now print a picture of a draughts board in the top left hand corner of the screen. Try it and see. Now each part of this program will be explained in great detail, so dont worry if a program this size seems a daunting prospect. Right now we are only going to input the first part. It starts off with some data.

```
4C97     FAFB0605   TABLE     DEFB FA FB 06 05
```

This represents the directions in which we are about to allow moves. The numbers in the data are -6, -5, 6 and 5, which, in the board numbering system the computer will use, are simply the numbers we add to one square to reach another.

The first, and simplest thing to do, is to make a copy of the board as it appears on the screen. The copy is called WKBOARD, for it is the part of RAM on which the computer will do its working out. The address of WKBOARD is to be 403C. That's not a misprint, it really does say four zero three C. For OLD ROM users this is just beyond the end of the BASIC part of the program, but for NEW ROM users it is slap bang in the middle of the system variables. Is this wise?

We will in fact be overwriting the 33 byte area PRBUFF and part of the calculating store MEMBOT. This doesn't matter since we will not be using LPRINT, not be attempting to use floating point calculations, and in fact not using this area at all. This will not cause a crash.

During the construction of this program, OLD ROM users should use the address 4B3C instead, since 403C is in mid-program. You can always change it when the program is complete.

Here's the copying routine. You should load this to address 4CE4.

```
2A0C40     BOARDCOPY LD HL,(D-FILE)     Make a copy of the board
110D00               LD DE,000D         from the screen to the
19                   ADD HL,DE          working area.
113C40/4B            LD DE,WKBOARD
062A                 LD B,2A
EDA0       NSCOPY    LDI
23                   INC HL
10FB                 DJNZ,NSCOPY
C9                   RET
```

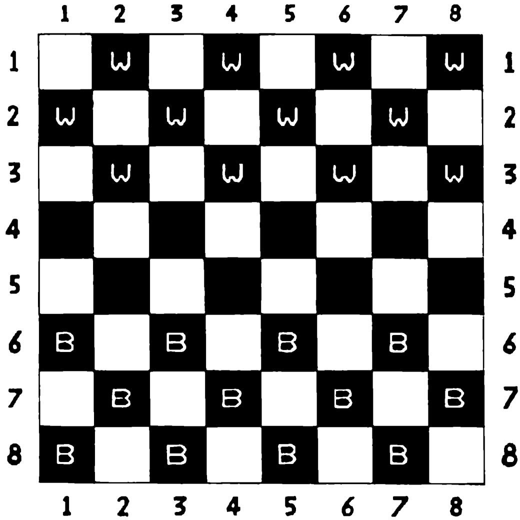
Notice the way LDI was used instead of LDIR. This is a very useful way of saving space. What we are doing is incrementing HL each time round, so that only the black squares are copied, not the white ones. This loop is repeated 2A (forty-two) times, since in addition to the squares on the board, one or two characters from the border are also copied. Notice that although LDI decrements BC, it is C that is decremented, not B, so that the DJNZ instruction will still count correctly.

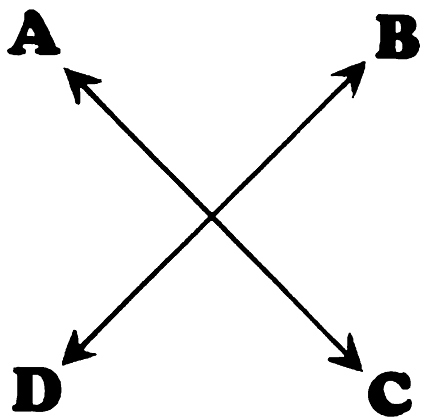
OLD ROM users can easily check that the routine is working by listing from 4B3C using HEXLD3, after the program is run. NEW ROM users can check by replacing the RET instruction to LD HL,WKBOARD/LD (ADDRESS),HL/JP HLIST. You must not return to basic (NEW ROM users that is) since PRBUFF will be wiped out by doing so. You can quite safely return after you've listed.

The next part of the program is just as simple. If you take a look at the board printing program, you'll see that the last thing printed is a row of fourteen spaces. What this is is a "window" in which our machine code program can display messages to the user, so the next thing to do is to fill this window with spaces in order to wipe out any error message that may have been there.

```
4CF5   000000   NEXTLINE  six NOP's
4CF8   000000
4CFB   2A0C40   CLWIND    LD HL,(D-FILE)     Find start of window.
4CFE   117000             LD DE,0070
4D01   19                 ADD HL,DE
4D02   060E               LD B,0E            Fill it with fourteen
4D04   3600     WIPEOUT   LD (HL),00         spaces.
4D06   23                 INC HL
4D07   10FB               DJNZ WIPEOUT
4D09   C9                 RET                Return to BASIC.
```

Notice that we have actually overwritten the previous routine's RET instruction, so that it will automatically continue into this one. The next part is for NEW ROM users only. OLD ROM users please ignore it.


1 2 3 4 5 6 7 8
1 W W W W 1
2 W W W W 2
3 W W W W 3
4 4
5 5
6 B B B B 6
7 B B B B 7
8 B B B B 8
1 2 3 4 5 6 7 8


A
B
D
C

The following will cause line one (that is BASIC line one) to be re-executed as soon as the next RET instruction is recieved. Note that this overwrites the six NOPs in the previous section.

```
4CF5    217D40    NEXTLINE  LD HL,FIRSTLINE
4CF8    222940              LD (NXTLIN),HL
```

This fools the ROM into thinking that the next line to be executed begins at address 407D, which is the first byte of the program. It doesn't return to BASIC immediately however, it will continue with draughts until a RET instruction is reached.

Now the program seriously starts. We assume that a move has been input as A$, which is the first item in the variable store.

Here's how to input a move. Look at the diagram of the board. There are sixty-four squares, but only thirty-two of them are playable. Each square has a coordinate from 11 to 88. Notice that these are printed without separation. The first digit refers to the number down the left (and right) hand side of the board, and the second digit refers to the number along the top or bottom of the board.

There are four different directions you may move in. These are called A (up-left), B (up-right), C (down-right) and D(down-left). This is indicated on the diagram. To input a move simply type in the coordinates and a letter (A, B C or D). There should be no spaces in this input. For instance, to move from square 61 to square 52 you should input "61B".

Now for a program to interpret this input. Follow this carefully:

```
4D09    2A1040 (NEW ROM)      MOVE
        2A0840 (OLD ROM)      LD HL,(VARS)
        23                    INC HL
        7E                    LD A,(HL)
        3D                    DEC A
        3D                    DEC A
        3D                    DEC A
        2001                  JR NZ,NOTZERO
        2F                    CPL
4D14    5F        NOTZERO     LD E,A
```

A small amount of additional explanation concerning the input here,which applies to OLD ROM users only. To input a simple move, such as from 61 to 52, you in fact need to input "shift W space 61B". A simple move must always be preceeded by shift W space, and this also applies to single jumps. Double jumps, triple jumps etc are a little different, and we shall cover them later. As I have said, this is for OLD ROM users only.

The above routine initially loads A with the length of the input string, and then subtracts three, so that for an ordinary move A ends up as zero, for a double jump A ends up as one, for a triple jump A ends up as two, and so on. Then IF A is 00 it is changed to FF. This is so that we can check up on whether or not a player has made a move, or a jump, later on in the game.

This quantity, which is ordinarily FF, is stored in the register E. We then continue.

```
4D15    23                  INC HL         The first character of the
        23                  INC HL         coordinates is found.
        7E                  LD A,(HL)
        47                  LD B,A         This number is multiplied
        87                  ADD A,A        by eleven, since the board
        4F                  LD C,A         on screen is eleven char-
        87                  ADD A,A        acters across.
        87                  ADD A,A
        23                  INC HL         The position within the
        E5                  PUSH HL        string is stored.
        80                  ADD A,B
        81                  ADD A,C
        86                  ADD A,(HL)     The next coordinate is added.
        1F                  RRA            Divide by two, since the
                                           copy contains only the black
                                           squares.
        3805                JR C,NOERROR1  If the coordinate points to
                                           a black square there is no
                                           cheating.
```

In the above routine the first coordinate is multiplied by eleven, by making use of registers B and C, and then the second coordinate is added. Note that if you input "12" as your square then because of the Sinclair character codes the program thinks that the first coordinate is actually 1D, and that the second coordinate is 1E. This actually leads to a result of 5D. Rotating right gives 2E, together with a carry indicating that the player has not cheated by giving a white square instead of a black one. The next five bytes deal with what happens if the player has cheated. These are

```
4D25    E1        ERROR1    POP HL         Restore the stack pointer.
        CDA74C              CALL ERROR     Call to an error message
        1D                  DEFM 1         subroutine.
```

The subroutine ERROR, which requires one byte of data (here the byte 1D, the character code of "1") looks like this:

```
4C9B    2E31312A  IMOVE     DEFM ILLE      These are the words "ILLEGAL
        2C263100            DEFM GAL       MOVE" - data to be printed.
        32343B2A            DEFM MOVE
4CA7    E1        ERROR     POP HL         Fetch the byte of data
        7E                  LD A,(HL)
        2A0C40              LD HL,(D-FILE) Find the start of the window.
        117000              LD DE,0700
        19                  ADD HL,DE
        EB                  EX DE,HL
        219B4C              LD HL,IMOVE    Copy the words onto the
        010C00              LD BC,000C     screen.
        EDB0                LDIR
        13                  INC DE         Print the byte of data onto
        12                  LD (DE),A      the screen.
        C9                  RET            Return to BASIC.
```

Notice what happens. The message "ILLEGAL MOVE 1" appears on the screen, and no piece is moved. The player is then required to re-input her move which will then be checked in exactly the same way.

If no error is found (yet) the program continues.

```
4D2A    C60E      NOERROR1  ADD A,0E       Find the position of the
                                           square in WKBOARD.
```

0E is simply the required factor to exactly match A to the low part of the address of the working-board square. For instance, adding 0E to 2E gives 3C, and 403C is the start of WKBOARD.

```
4D2C    6F                  LD L,A
        2640/4B             LD H,WKBOARD-high
        4E                  LD C,(HL)         Find which piece is on
                                              that square.
        0680                LD B,80           Replace that square by a
        70                  LD (HL),B         black empty square.
4D33    E3        LOOP      EX (SP),HL        Store the square position,
                                              and retrieve the pointer
                                              to the input string.
        23                  INC HL
        227B40              LD (POINTER),HL   Store this value.
        7E                  LD A,(HL)
        C671                ADD A,71          Find the direction being
        6F                  LD L,A            moved from the TABLE.
        264C                LD H,TABLE-high
        56                  LD D,(HL)
        E1                  POP HL            Retrieve square position.
        78                  LD A,B            Check whether the player is
        A2                  AND D             moving one of her own pieces,
        2F                  CPL               and in a legal direction.
        A1                  AND C
        FE27                CP 27
        20DE                JR NZ,ERROR1
```

A brief explanation of the last six lines here. A is loaded by 80. D is the direction to be moved, which will be FA, FB, 05, or 06. AND D will therefore produce 00 for a backward direction, and 80 for a forward direction. CPL will change this to FF or 7F. C is the piece to be moved. If it is a black king it will be 27, if it is a black piece it will be A7, so AND D will produce a value of 27 if EITHER the piece being moved is a king, OR if the piece is moving forwards. If you try to move a piece backwards, or give a square which does not contain one of your own pieces, then a value of 27 will NOT be produced. In this case the program will send an "ILLEGAL MOVE 1" error message.

```
4D48    7D                  LD A,L            Find destination square.
        82                  ADD A,D
        6F                  LD L,A
        7E                  LD A,(HL)         Check the contents of that sq.
        B8                  CP B              Is it an empty square?
        2008                JR NZ,NEXT
        7B                  LD A,E            If so, is this a single
        3C                  INC A             move?
        2815                JR Z,CONTINUE
        CDA74C              CALL ERROR        If not a single move, give
        1E                  DEFM 2            "ILLEGAL MOVE 2" message.
4D57    B0        NEXT      OR B
        FEBC                CP BC             Does square contain a comp-
        2804                JR Z,NOERROR3     uter's piece?
        CDA74C    ERROR3    CALL ERROR        Give message "ILLEGAL MOVE
        1F                  DEFM 3            3" if not.
4D60    70        NOERROR3  LD (HL),B         Overwrite computer's piece
                                              with a black empty square.
        7D                  LD A,L            Find next destination
        82                  ADD A,D           square.
        6F                  LD L,A
```

```
4D64    7E      CONTENT   LD A,(HL)       Find the contents of the
                                          new square.
        B8                CP B            Is this square empty?
        20F4              JR NZ,ERROR3    Give "ILLEGAL MOVE 3" if not.
4D68            CONTINUE
```

At this stage the program will jump or move as the case may be (in other words it will decide for itself - you don't need a special input) and will so far check for three types of error. These are:

1) Attempting to move a piece that is not your own, or moving one of your own non-king pieces backwards.
2) Attempting to make a non-jump move in the middle of a multiple move sequence.
3) Attempting to move to a square which is non empty.

You may like to check all of these things. This isn't too difficult to do. Simply write JP 4DDE to the end of the program and add the following routine.

```
4DDE    2A0C40     BDPRINT   LD HL,(D-FILE)
        110D00               LD DE,000D
        19                   ADD HL,DE
        EB                   EX DE,HL
        213C40/4B            LD HL,WKBOARD
        062A                 LD B,2A
        EDA0       LDI       LDI
        13                   INC DE
        10FB                 DJNZ LDI
        C9                   RET
```

This will copy the computer's working-board back onto the screen so that you can see what has happened. You can also alter the data for the board-print routine, and so set up a board in mid-game in order to test some of the error checks if you want.

To make the program run, add the following BASIC program lines.

OLD ROM	NEW ROM
1 INPUT A$	1 INPUT A$
2 RANDOMISE USR(19683)	2 RAND USR 19683
3 RUN	3 STOP
4 RANDOMISE USR(19477)	4 RAND USR 19477
5 RUN	5 RUN

The program is activated by the command RUN 4. Don't forget you can still use HEXLD3 to list, but you must now use RUN 10 to bring this into operation. If you type RUN on its own accidently you will get an input prompt. Break out immediatly! If you don't the results will be unpredictable. I don't think it will crash, but just to be on the safe side....

And now a check to determine whether or not the human player has reached the other end of the board. If so, this next routine will automatically change her piece into a king.

```
4D68    7D      CONTINUE  LD A,L          If the low part of the
        FE40              CP 40           current address is less than
        300C              JR NC,NOKING    40hex then the other side
                                          has been reached.
        7B                LD A,E          If this is not the last
        3C                INC A           move then give an "ILLEGAL
        FE02              CP 02           MOVE 4" message.
        3804              JR C,NOERROR4
        CDA74C            CALL ERROR
        20                DEFM 4
```

```
        0E27      NOERROR4  LD C,27        Make piece a king.
        71        NOKING    LD (HL),C      Put back on board.
        E5                  PUSH HL        Store current position on
                                           board.
```

Notice the new error check. If a player attempts to make a king in <u>mid-move</u>, that is, if she jumps to the back row and intends to jump out again in the same go, then an error will be detected and "ILLEGAL MOVE 4" printed to the screen. This is because according to official rules a piece does not become a king until after you remove your fingers from it. Of course in this game your fingers are never on the piece in the first place, but we presume that this is what the rules are intended to mean.

Remember that E contains FF for a single jump, and 01 for a double jump. LD A,E/INC A/CP 02 will only give an error if E is one or more. If E is 00, (which if you've input a multiple jump it will eventually be) the move will go ahead successfully.

```
4D7B    2A7B40          LD HL,(POINTER)     Retrieve the position pointed
                                            to in the input string.
        1D              DEC E               Decrease the number of moves
                                            left in a multi-jump seq.
        7B              LD A,E              Check whether last move
        E3              EX (SP),HL          has been made.
        17              RLA
        30AE            JR NC,LOOP          The input pointer is replaced
                                            at the top of the stack
                                            ready for the next time
                                            round the loop.
        E1              POP HL
4D85    C3DE4D          JP BDPRINT          Exit.
```

Well, all of the possible error checks have been made, and the program contains a loop which will allow for the inputting of multiple jumps. Here's how a multiple jump should be input. To jump from square 63 first in direction A, then in direction B, then finally in direction C, just input "63ABC" - it's that simple. OLD ROM users need to note the following convention though:

<u>OLD</u>	single moves or single jumps should be preceeded by shift W space.
<u>ROM</u>	double jumps should be preceeded by shift E space.
<u>ONLY</u>	triple jumps should be preceeded by shift R space.
	4-ply jumps should be preceeded by shift D space.

And so on... The sequence is W, E, R, D, F, S, A, T, G. I doubt very much whether you'll ever need a 4-ply jump though. Even using a triple jump seems rather unlikely.

The next thing that should happen is that the computer should make a move in response, but we'll leave that to another chapter, since it has a bit of decision making to do. But there is one question to be answered first. What if it now has no pieces left to move? What if the player's last move removed its last piece? This has to be checked for. If this is the case then the player has won, and we must somehow indicate this.

Here is the final check:

```
4D85    0EBC            LD C,BC
4D87    CDBC4C          CALL GAMEOVER
4D8A    C3DE4D          JP BDPRINT
```

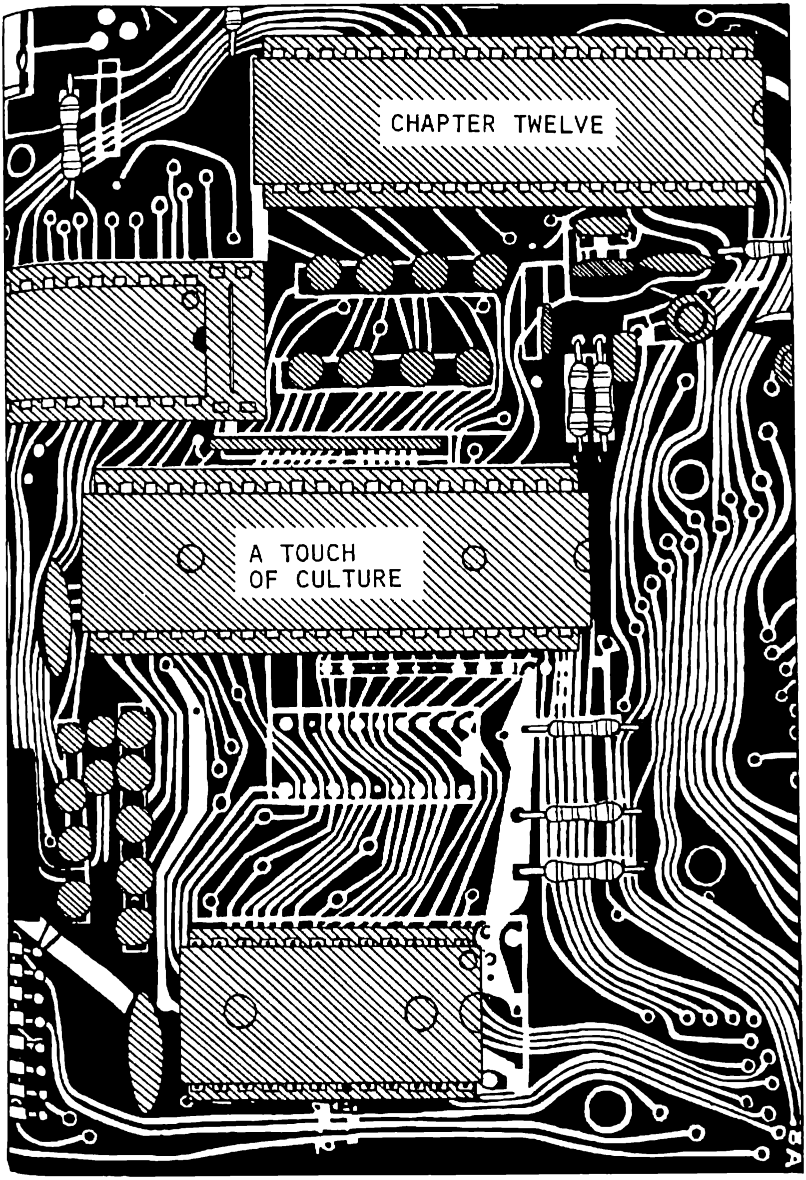
And the subroutine....

```
4CBC    213C40/4B GAMEOVER  LD HL,WKBOARD       Look at first square.
        062A                LD B,2A
        7E        POSSIBLY  LD A,(HL)           Find contents of square.
        F680                OR 80               Make it an inverse graphic.
        B9                  CP C                Is it what we're looking
        C8                  RET Z               for? If so we're OK and
                                                can return to draughts.
        23                  INC HL              Look at next square.
        10F8                DJNZ POSSIBLY       Try again.
4CC9    the next six bytes are for the new rom only. Old Rom users should
        replace them with six NOPs (00).
        219740    STOPPROG  LD HL,STOPLINE      Fool the ROM into thinking
        222940              LD (NXTLIN),HL      that line 3 is to be carried
                                                out next.
```

Now we reach the exciting bit. What happens if the player HAS won? I'm not actually going to tell you - just input it and find out. To test it you'll have to alter the data that sets up the initial board, and arrange it so that you can take all of the computer's pieces.

```
4CCF    2A0C40    INVERT    LD HL,(D-FILE)
        066C                LD B,6C
        23        COVER     INC HL
        7E                  LD A,(HL)
        FE25                CP 25
        3006                JR NC,NOINV
        A7                  AND A
        2803                JR Z,NOINV
        F680                OR 80
        77                  LD (HL),A
        10F2      NOINV     DJNZ COVER
        E1                  POP HL
        C9                  RET
```

Notice how, in the last two lines the return address is removed from the stack, so that the next item on the stack is the return to BASIC address. The next RET will of course do just that.


CHAPTER TWELVE
A TOUCH
OF CULTURE

MUSIC

Music from your TV speaker? Is it possible? More to the point - is it possible on a ZX? The answer is yes!

As you know, your machine is designed to work without sound. It does make a kind of horrible buzzing noise, but hardly anything you'd want to make music out of. The manual itself tells us to turn the volume right down so as to cut the noise out completely.

The little computer, on the other hand, has a mind of its own. Completely ignoring its own design specifications it thinks to itself "Anything a bigger computer can do, I can do better", and as a result of this rebellion you'll find that REAL MUSICAL NOTES can be produced with just a tiny speck of machine code.

Those of you who have tried the music routines in Interface are undoubtedly thinking to yourself "Huh! I've heard this so called 'music' - it's rubbish!" Well I assure you this is not the same thing. The reason? Well one big advantage machine code does have over BASIC is precision - and this program is in machine code, not BASIC. The music is musical. You can even tune it if you have a tuning fork handy.

This is called CATHY'S PROGRAM, dedicated to someone who believes computers should be artful, not just attack you with space invaders. The machine code is best stored in a REM statement. The addresses given in the listing assume you have a NEW ROM machine. If you have an OLD ROM machine all you have to change is the addresses (although you will have to supply two of the subroutines yourself - see chapter ten)

Cathy's Program

```
           9B897369    NOTES   C D E F          This data represents
           00937E005E          - C# D# - F#     the various notes that
           003B312824          - C D E F        are available from the
           0000362C00          - - C# D# -      keyboard.
           00000F161E          - - A# G# F#
           000A0C121A          - C B A G
           000000414C          - - - A# G#
           00383C4653          - C B A G
           78          PAUSE   LD A,B           Subroutine causing a
           3D          HOLD    DEC A            delay of a precise
           20FD                JR NZ,HOLD       length.
call       C9                  RET
here -->   CDBB02      START   CALL KSCAN       Wait until a key is
           44                  LD B,H           pressed.
           4D                  LD C,L
           51                  LD D,C
           14                  INC D
           28F7                JR NZ,START
           CDBD07              CALL FINDCHR     Find which key is being
           110440              LD DE,NOTES-7E   pressed.
           19                  ADD HL,DE
           46                  LD B,(HL)        Select note.
           AF                  XOR A
           B8                  CP B             Check that this note is
           28EB                JR Z,START       not a "pause".
           DBFF                IN A,(FF)        Play this note.
           CDA940              CALL PAUSE
           D3FF                OUT (FF),A
           CDA940              CALL PAUSE
           18E0                JR START         Go round loop again.
```

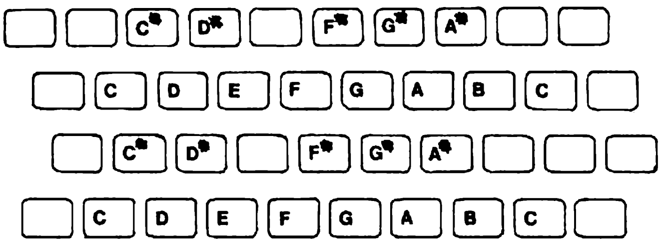
If you store the whole machine code routine in a single REM statement in line one, then you only need one more line of BASIC to make the program complete. This is line 2 RUN USR 16558, which calls the machine code from the address labelled START. Delete any extra lines you may have, and SAVE the program a couple of times <u>before</u> you RUN it.

You now have two octaves at your disposal - the keyboard below shows where the notes are. A fair number of tunes may be played quite successfully.

<u>Always</u> run the program in the FAST mode - it's not that the speed makes the notes sound different - it's simply that the program doesn't work AT ALL when in SLOW.

The notes as listed in the program are roughly right, but exactly how they sound will depend mainly on your television set, (incidently you may have to alter the tuning slightly to get the best sound quality,) so in case you need to "re-tune" the notes, here's how you do it:

The data at the start of the program (labelled NOTES) contains one byte for each note. A zero indicates there is no note on that key. The data is in the following order:



data	key	note	
9B 89 73 69	Z X C V	C D E F	lower octave
00 93 7E 00 5E	A S D F G	- C# D# - F#	lower octave
00 3B 31 28 24	Q W E R T	- C D E F	upper octave
00 00 36 2C 00	1 2 3 4 5	- - C# D# -	upper octave
00 00 0F 16 1E	0 9 8 7 6	- - A# G# F#	upper octave
00 0A 0C 12 1A	P O I U Y	- C B A G	upper octave
00 00 00 41 4C	nl L K J H	- - - A# G#	lower octave
00 38 3C 46 53	sp . M N B	- C B A G	lower octave

to alter the frequency of any note just change the byte of data that represents it. To make a note higher you must decrease the number, and to make it lower you must increase the number.

THE PROGRAM'S DISADVANTAGES

(And how you can cure them)

The biggest disadvantage is the lack of a RET instruction anywhere in the program, which means that once you enter the program you can never leave. You can cure this by adding a few lines somewhere near the START label. As an exercise, see if you can adjust the program so that it returns to BASIC whenever the key SHIFT-ZERO (rubout) is pressed. (HINT: HL equals FCEF when it returns from KSCAN)

The second disadvantage is that if you press SHIFT while playing notes some very random things seem to happen. See if you can make the shift key inactive (except for breaking out as described above) by adding a SET 0,H instruction somewhere in the program.

ZX Music is a fascinating subject, and it is possible to store in data a list of notes to be played, and how long each note is to be played - a tune in other words. I'll leave that one to you though, because the only real way to learn is by experiment. We'll leave the subject of music altogether now and turn to something slightly different: pictures....

PICTURES

This is yet another program which relies on the artistic ability of the human operator. It is strictly for NEW ROM users ONLY, but it is intended to be run in the FAST mode. You will require at least four-K for this.

The program stores in memory three or more different pictures, and cycles through them one at a time, displaying each on the screen for as long as you want. A "picture" can be anything whatsoever - you can compose it out of graphics symbols, letters, spaces, inverse asterisks - whatever.

The first thing you do is to reserve some memory in which to store these pictures. If you have 4K type POKE 16388,182/POKE 16389,70/NEW for three pictures, or POKE 16388,206/POKE 16389,73/NEW for two pictures. If you have 16K you can find enough room for about twenty pictures. To work out how far down you have to move RAMTOP with 16K just start off with 32768 and subtract 793 for each picture.

Now you're ready: Write the following machine code to a REM statement in line one:

```
2A0C40   STORE   LD HL,(D-FILE)
11B646           LD DE,PICTURE1
011903           LD BC,0319
EDB0             LDIR
C9               RET
```

The address labelled PICTURE1 refers to those people using 4K. For those same people PICTURE2 would be 49CE and PICTURE3 would be 4CE7. If only two pictures will be used you should omit PICTURE1, not PICTURE3. If you have 16K you have more or less limitless freedom. In the interests of simplicity you could use addresses 5000, 5400, 5800, 5C00, and so on.

Now type POKE 16389,77 followed by CLS if you are using 4K, or if you are using 16K but earlier POKEd 16389 with a number less than 77.

Now write a BASIC program (without deleting line one) which prints a picture. The last line of this program should be RAND USR 16514. 4K users may find themselves running out of space. If this is so you'll just have to give up and make do with two pictures instead of three.

A useful fact to know is that if you make the first line of your program (first apart from the REM that is) POKE 16418,0 then you can print to all twenty-four lines of the screen. Even PRINT AT 23,0; works!

Now delete all the PRINT lines. DO NOT TYPE NEW. Change the address in the machine code to that of a different picture, and write a new BASIC program printing a different picture, again ending in RAND USR 16514. Do this until every picture you wish to cycle through has been stored.

Now move RAMTOP back to the address described in paragraph three
Type NEW. Now you are ready....

For the first time in the book we are going to make use of the PAUSE facility. The instruction CALL PAUSE will display the TV picture indefinately, or until a key is pressed. To PAUSE for a specific number of TV frames it is necessary to LD (FRAMES) with the required number first. Enter this machine language program:

```
0602        PICTURES  LD B,number of pictures
21B646                LD HL,address of first picture
C5          NEXTPIC   PUSH BC
ED5B0C40              LD DE,(D-FILE)
011903                LD BC,0319
EDB0                  LDIR
E5                    PUSH HL
210001                LD HL,length of pause
223440                LD (FRAMES),HL
CD2902                CALL PAUSE
E1                    POP HL
C1                    POP BC
10E8                  DJNZ NEXTPIC
C9                    RET
```

This is the complete program. See how it works - the first picture is copied into the display file using LDIR, and the PAUSE subroutine is called from the ROM. Then when the PAUSE is over the next picture is copied onto the screen, and so on. The value of HL is not changed between each picture, since they are stored in memory immediately after each other. If they are not (for instance if you are using easy to remember addresses) you'll need to alter the program slightly . HL should point to the start of a new picture each time round the loop.

The BASIC program to go with this is

```
10 RAND USR pictures
20 RUN
```

In this way you can break out of the program at the end of the sequence. Alternatively you could replace the last RET instruction by JR PICTURES, which would eliminate the need for a second BASIC instruction. You can of course always break out during a PAUSE.

LIFE

In the last program in this chapter we turn the tables slightly. We humans have been artistic for long enough - now it's time to let the computers take their turn....

This program is called LIFE - it is supposed to represent the birth/growth/ death cycle of a colony of cells living on a square grid. It produces rather fascinating results. Before your very eyes you see a constantly evolving pattern - starting off totally random - which finishes sometimes with the ultimate death of the cell colony, sometimes with a fixed and unmoving cell structure which has reached equilibrium, and sometimes with a continuous cycle of patterns, called dynamic equilibrium. It really is amazing to watch.

LIFE was invented in 1970 by a man called John Conway of Cambridge University, and it's rather surprizing that the Tate Gallery hasn't yet got a copy of it. Although it is in fact about the growth of cells which follow hard and fast mathematical rules it in reality becomes a rather effective pattern generating algorithm.

The principle of LIFE is very simple. A grid - usually square - is dotted with approximately one quarter of its available squares filled with cells. These positions are usually chosen entirely at random. This configuration of the grid is called GENERATION ZERO.

Successive generations are then worked out by a fairly simple to understand principle. Each square on the grid has eight neighbouring squares. These squares either contain another cell or they are empty. Every cell with two neighbouring cells; or with three neighbouring cells, will survive to the next generation, but no other cells will survive. A new cell is born in every empty space which has precisely three neighbouring cells, but no other cells are born. With these fairly simple rules it is rather surprizing that the game should produce the rather impressive results that it does.

In this version of LIFE our grid is sixteen by sixteen, because of course sixteen is a fairly easy number to work with in hexadecimal. Further, our grid is rather strangely constructed in a curved space continuum, meaning that every square on the left hand edge is connected to the corresponding square on the right hand edge, and vice versa, also every square on the top edge is connected to the corresponding square on the bottom edge and vice versa.

The program is best run in SLOW, although of course it will run in FAST if you add a PAUSE or INPUT statement.

NEW ROM people are advised to store the machine code in a REM statement. OLD ROM people are advised to store the machine code <u>anywhere but</u> a REM statement, since it contains characters 76h. The machine code contains exactly one hundred and thirty nine bytes.

The surrounding BASIC program is

```
 2 RAND USR START
 3 RAND USR NEXTGEN
(4 PAUSE 25 or INPUT A$ - optional extra for FAST users)
 5 GOTO 3
```

where START and NEXTGEN are addresses in the machine code program. In the following listing we assume that the first address is 4082. You can quite easily change it if you wish.

```
EF010110  TABLE      DEFB EF 01 01 10   Data representing the displacements
10FFFFF0             DEFB 10 FF FF F0   of the neighbouring squares.

call here

0E10      START      LD C,10            C counts the number of rows printed.
0610      NEWROW     LD B,10            B counts the number of columns.
2A3240    NEXT       LD HL,(SEED)       This next section generates a
54                   LD D,H             random number.
5D                   LD E,L
29                   ADD HL,HL
29                   ADD HL,HL
19                   ADD HL,DE
29                   ADD HL,HL
29                   ADD HL,HL
29                   ADD HL,HL
19                   ADD HL,DE
223240               LD (SEED),HL       The new random-number-seed is stored.
7C                   LD A,H
FEC4                 CP C4              Decide which character to print, based
3804                 JR C,BLACK         on choice of random number.
3EB4                 LD A,B4
1802                 JR CHAR
3E80      BLACK      LD A,80
D7        CHAR       RST 10             Print this character.
10E3                 DJNZ NEXT          Same for the next character in the row.
3E76                 LD A,76
D7                   RST 10             Print a newline symbol at the end
0D                   DEC C              of the row.
20DB                 JR NZ,NEWROW       Same for next row.

C9                   RET                Generation zero printed completely.
0600      NEXTGEN    LD B,00            B counts the number of cell positions.
110043               LD DE,DUMP         DE stores the start of the working-
                                        area used to compute the next gen.
2A0C40               LD HL,(D-FILE)
E5                   PUSH HL            Stack the start of the display-file.
7E        COPY       LD A,(HL)          Copy the current generation (but not
23                   INC HL             newlines) to the working space.
FE76                 CP 76
28FA                 JR Z,COPY
12                   LD (DE),A
13                   INC DE
10F6                 DJNZ COPY

110043               LD DE,DUMP         Stack the start of the dump.
D5                   PUSH DE
0E00      NEXTCELL   LD C,00            C Counts the number of neighbours a
                                        particular cell has.
D1                   POP DE
E1                   POP HL             Skip over the next character in the
7E                   LD A,(HL)          display file if it is a newline.
FE76                 CP 76
2001                 JR NZ,VALID
23                   INC HL
E5        VALID      PUSH HL
EB                   EX DE,HL           Store the position within the dump of
E5                   PUSH HL            the cell being examined in HL, and
                                        also stack it.
```

```
118240            LD DE,TABLE        Point DE to table of displacements.
1A       NEXDIS   LD A,(DE)          Find displacement.
FE0E              CP 0E              If this "displacement" is 0E we have
280B              JR Z,COUNTED       reached the end of the table.
13                INC DE             Point DE to next item in table.
85                ADD A,L            Find neighbouring cell-position.
6F                LD L,A
7E                LD A,(HL)          Is there a cell there?
FEB4              CP B4
20F3              JR NZ,NEXDIS
0C                INC C              Increase count if so.
18F0              JR NEXDIS
E1       COUNTED  POP HL             Retrieve cell position.
79                LD A,C
FE02              CP 02              Are there less than two neighbours?
380F              JR C,NOCELL        If so no cell appears.
FE04              CP 04              Are there four or more?
300B              JR NC,NOCELL       If so no cell appears.
FE03              CP 03              Are there precisely three?
2803              JR Z,CELL          If so, a cell does appear.
7E                LD A,(HL)
1806              JR PUT
3EB4     CELL     LD A,B4
1802              JR PUT
3E80     NOCELL   LD A,80            A now contains the right character.
E3       PUT      EX (SP),HL         Retrieve print position.
77                LD (HL),A          Print character.
23                INC HL             Move print position along one.
E3                EX (SP),HL         Retrieve cell-position.
23                INC HL             Look at next cell-position.
E5                PUSH HL            Stack this position.
7D                LD A,L             Check the value of L to find out
A7                AND A              whether or not we have printed the
20BF              JR NZ,NEXTCELL     last cell-position.
E1                POP HL             Restore the stack to its original
E1                POP HL             state and return to BASIC
C9                RET
```

If you used the same addresses as in the listing then START is 16522 and NEXTGEN is 16562. SAVE the program. Do not RUN it yet because if you do it will crash! NEW ROM users MUST first of all type POKE 16389,67 followed by NEW, and OLD ROM users should ensure that they have at least 2K of memory. You will then have to reLOAD the program from tape.

The first thing you should type is RAND/RANDOMISE. You may now type RUN.

An interesting point about this program is that it is capable of producing its own random numbers. The part labelled NEXT does this - you should study how this is achieved, and by all means use the same principle in your own programs.

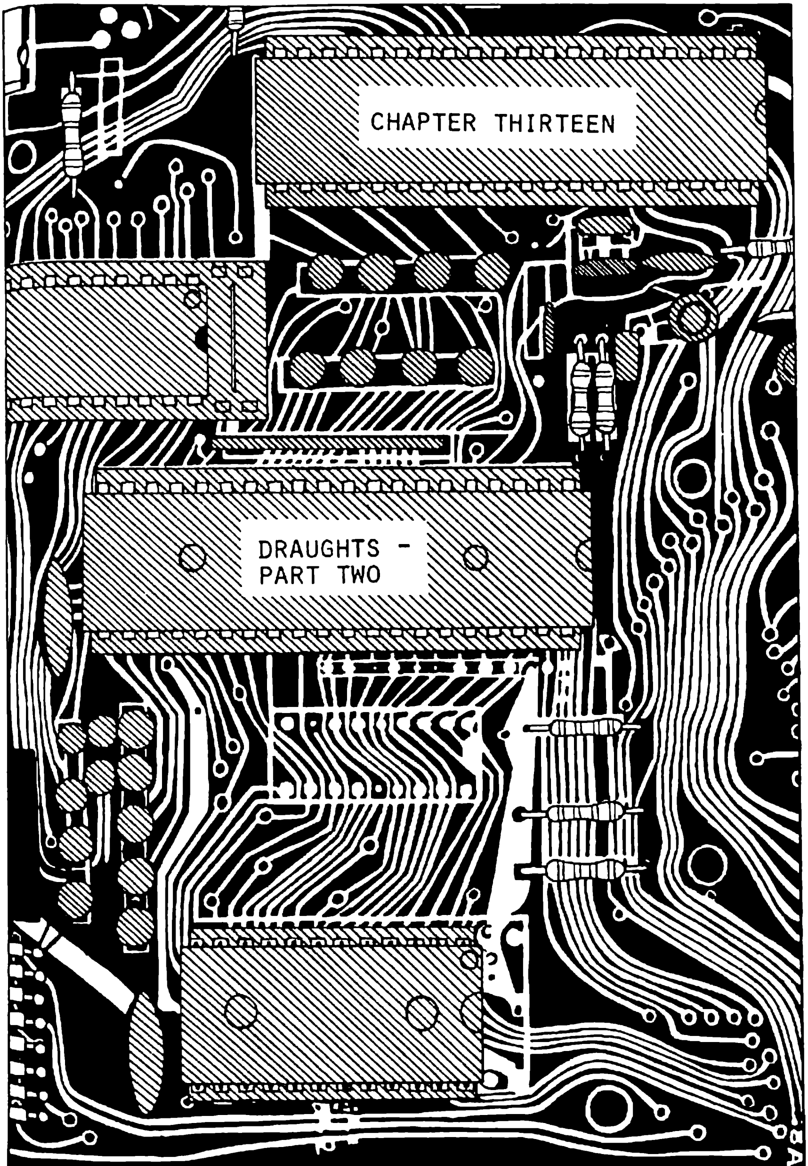
LIFE will print out a randomly constructed generation zero in just ONE SECOND when in the SLOW mode. The successive generations will then be produced at the staggering rate of three and a half generations per second! If you find this is much too rapid you can slow it down by adding a few more lines of BASIC - I suggest LET X=0/LET X=X+1/PRINT AT 17,0;X with the last two being inside the loop - this has the added advantage of telling you how many generations have been shown.

Finally you should follow the manner in which this program , unlike some other LIFE programs, calculates each new generation entirely on the basis of the previous one. It does not work out the new first row and then calculate the second row by counting the neighbours in the now-changed new first row, the second row is determined by the previous status of the first row, (this is what the area of memory labelled DUMP in the machine code listing is for), thus each new generation is correctly set up.

There are many other pattern generating programs, some much simpler, but none with the elegance of LIFE. If you own 16K you might like to try writing a 24 by 24 LIFE, or even a 24 by 32 version - remember, in machine code there is nothing to stop you printing on the very bottom two lines.

The biggest LIFE you could possibly hope to achieve is 48 by 64 using white quarter-squares for cells, but that would be quite a complicated program. If you feel really enthusiastic you might like to have a bash at this monumental task. I will let that decision rest with your sanity.

The next chapter completes the discussion on DRAUGHTS and leaves you with the horrifying prospect of completing the program....


CHAPTER THIRTEEN
DRAUGHTS -
PART TWO

DRAUGHTS

This is the section which decides upon which is the "best" move the computer can make, after the human's move.

You may have to follow this thinking we are about to embark upon very carefully. Here in brief is a systematic breakdown of the way in which the move is chosen.

We scan the board, one (black) square at a time, and whenever we find a computer's piece we sit and think about it for a bit. To each move we find possible we assign a numerical value, such that the bigger the number, the better we think the move is. It then follows that to select a move we merely have to choose the one with the highest possible value.

Of course this idea won't let the computer plan ahead - it can only think one move at a time. In order to construct this list of moves, and accompanying numerical values we don't actually have to store every single move we find. Having located a possible move, and worked out its score, what happens is this:

If the score is LOWER than those on the list, the move is ignored.

If the score is EQUAL to those on the list, it is added to the end.

If the score is HIGHER than those on the list, then the list is abolished and a new one started.

In this way the list is always as short as it can possibly be. When the final decision time actually arrives the computer now merely has to select one of these moves at random. Next question - where will the list be stored? Answer The Stack. This simplifies things, but it does mean that we must keep a record of where the start of the list is. We shall store this at address 407B (OLD ROM 4022) and call this quantity LBASE. You will notice that in an earlier part of the program we used 407B/4022 to store a quantity called POINTER. Don't worry - this is quite alright. POINTER is not used in the previous section, and it's value does not need to be preserved. LBASE was not used in the last section, and again its value does not need to be preserved. Using the same space twice for two different things is a space-saving trick you should get to know.

The decision making of the computer begins at address 4D8A. The first instruction is LD (LBASE),SP. The start of the list is now preserved. We can play around with the stack now as much as we like, as long as we remember to restore its value before we return to BASIC. The second and third instructions are LD BC,0000 and PUSH BC, which will indicate that there is nothing at all in the current list.

The checking loop thus looks like this. Notice that a new variable SQCHK is used. It is listed as residing at 4077, but OLD ROM owners should replace this address by 401C:

```
4D8A    ED737B40  BOARDSCAN LD (LBASE),SP      Initialise the list.
        010000              LD BC,0000
        C5                  PUSH BC
        213C40              LD HL,WKBOARD      Scan the board, one square
        7E        NXTCHK    LD A,(HL)          at a time.
```

```
        F680                  OR 80
        FEBC                  CP BC              Have we found a computer's
                                                 piece?
        227740                LD (SQCHK),HL
        CA434E                JP Z,EVALUATE
        2A7740      KPCHKNG   LD HL,(SQCHK)
        2C                    INC L              Have we reached the end
        7D                    LD A,L             of the board yet?
        FE66                  CP 66
        20EC                  JR NZ,NXTCHK       Loop back if not.
```

As you can see, this particular bit is quite straightforward. You only need to (temporarily) add a few extra instructions to avoid crashing. These are:

```
4E43    C3D54D      EVALUATE  JP 4DD5            These additional lines
4DA9    C3D54D      CHOOSE    JP 4DD5            are temporary only. They
4DD5    ED7B7B40              LD SP,(LBASE)      will stop the program
4DD9    0EA7                  LD C,A7            crashing, but will not
4DDB    CDBC4C                CALL GAMEOVER      run it.
```

Can you see that loading SP with (LBASE) eliminates the need to POP everything from the stack before returning. LDing SP will fool the machine into thinking that the stack hasn't changed since we went into the loop.

Now we need to think about what form we want the list to take. Let's examine the problem in reverse. What form would we <u>like</u> the list to take, in order to make removing items from the stack easier.

The first item on the stack should be the number of steps involved in the move - that is one for a single move/jump, two for a double jump, three for a triple jump, and so on. The second item should be the numerical value which the items in the list have been assigned - the <u>priority</u> as we shall call it. Following these items of information we should have the list itself, starting with the square to be moved from, followed by a sequence of one or more directions in which to be moved. Immediately after this the second item in the list in the same form, then the third, and so on...

You'll notice that each thing we need on the stack will only need to be one byte in length. The number of steps cannot possibly be more than 255. The priority can be chosen however we like - we can always make it one byte if we wish. The initial square can be stored by only stacking the <u>low</u> part of its address in WKBOARD. The directions to be moved can be stored in the same manner as before - 05, 06, FA, or FB for plus or minus five or six. In order to make this program as space efficient as we can it makes sense to do just that.

To make a random decision let's assume there are B possible choices. We want therefore to choose a random number between 1 and B - or as we shall do between 0 and B-1. We shall do this by the following means:

```
4DA9    3A3440      CHOOSE    LD A,(FRAMES)low   Select a random number
        90          REPEAT    SUB B              between 0 and B-1. This
        30FD                  JR NC,REPEAT       number to be stored in
        80                    ADD A,B            the A register.
        C3D54D                JP 4DD5
```

OLD ROM users should replace the address 4034 by 401E. The final JP 4DD5 is merely a means of exiting the program.

Imagine the list is complete and we are about to remove one item from it. The stack now looks like this:

no. of steps	priority	initial square	direction one	initial square	direction one	....	direction one

↑sp ... lbase↑

If we now use the instruction POP BC, B will contain the priority, and C the no. of steps. The priority is now a redundant piece of information, since it was only needed to construct the list in the first place. C however is very important. In the diagram above C would be one, but it doesn't have to be.

The stack now looks like this - but let's generalise a bit more by assuming there are two steps per move, not one:

initial square	direction one	direction two	initial square	direction one	direction two	....	direction two

↑sp ... lbase↑

If A is an indication of which of these moves we are to choose then it seems logical that we must remove A of them from the stack. Then the required move would be at the top of the stack. Thus if A is zero we do nothing, otherwise we must use some kind of loop. Can you see that POP HL followed by DEC SP will remove one byte from the stack rather than two, and that INC SP can be used to skip over one of the bytes.

The required loop is this:

```
4DB0    C1                POP BC             Find the number of steps
        41                LD B,C             per move.
        2808              JR Z,FIRSTOFF      Do nothing if A is zero.
        33      NSQOFF    INC SP             Remove a total of A
        33      NEXTOFF   INC SP             complete moves from the
        10FD              DJNZ NEXTOFF       stack.
        41                LD B,C
        3D                DEC A
        20F8              JR NZ,NSQOFF
```

The selected move is now at the top of the stack. To carry it out let's first take a look at what the stack is now like:

initial square	direction one	direction two	.....

↑sp

To find the initial square the sequence is POP HL followed by LD H,WKBOARD-high. You see "initial square" is the low part of the address. By assigning H with the high part we ensure that the register pair HL contains the absolute address of the square from which we must move. H must be assigned after the POP HL instruction though, since there is no real way we can manage to remove L on its own. Finally the instruction LD B,C once more will assign B with the number of steps we have to make. The proceedure for carrying out these steps is much simpler than before since we don't have to check for cheating - we shall write the program such that the computer cannot cheat.

```
4DBA    E1      FIRSTOFF  POP HL             Find the absolute address
        2640              LD H,WKBOARD-high  from which we must move.
```

To remove one direction at a time from the stack we shall use the sequence DEC SP/POP DE. In this way E will be assigned with the required direction. D will contain useless information.

```
4DEF    41                  LD B,C
        3B      NEXTSTEP    DEC SP          Find which direction the
        D1                  POP DE          computer is to move.
        4E                  LD C,(HL)       Get computer's piece.
        3680                LD (HL),80      Overwrite with black sq.
        7D                  LD A,L          Find destination square.
        83                  ADD A,E
        6F                  LD L,A
        7E                  LD A,(HL)       Is this square empty?
        FE80                CP 80
        2805                JR Z,SQUARE     If so, move.
        3680                LD (HL),80      If not, jump.
        7D                  LD A,L
        83                  ADD A,E
        6F                  LD L,A
        71      SQUARE      LD (HL),C       Put piece in position.
        10EB                DJNZ NEXTSTEP   Same for next direction.
```

You should now be at address 4DD5, at which is stored the sequence

```
4DD5    ED7B7B40            LD SP,(LBASE)
        0EA7                LD C,A7
        CDBC4C              CALL GAMEOVER
4DDE    2A0C40    ENDPRINT  LD HL,(D-FILE)
                  ...
                  and so on down to
4DF0    C9                  RET.
```

This means that provided the stack is correctly set up we can actually see this whole mechanism working. What I want you to do now is to write a short routine to set up the stack so that all of the possible opening moves are stored. You should be able to do this all by yourself. I will tell you though that the routine should be placed at address 4E43 (what will eventually be the EVALUATE routine) and should be terminated by the instruction JP CHOOSE (C3A94D). One way of doing this bit would be LD HL,something/PUSH HL/LD HL, something/PUSH HL/and so on, but if you can think of a better way by all means use it.

You may now RUN the draughts program by typing RUN 4. You will be asked for an input - make your move as you have been doing in the past. Now watch what happens to the computer's side - one of the pieces should move! Break out of the program, since as yet it can only decide upon the first move of the game.

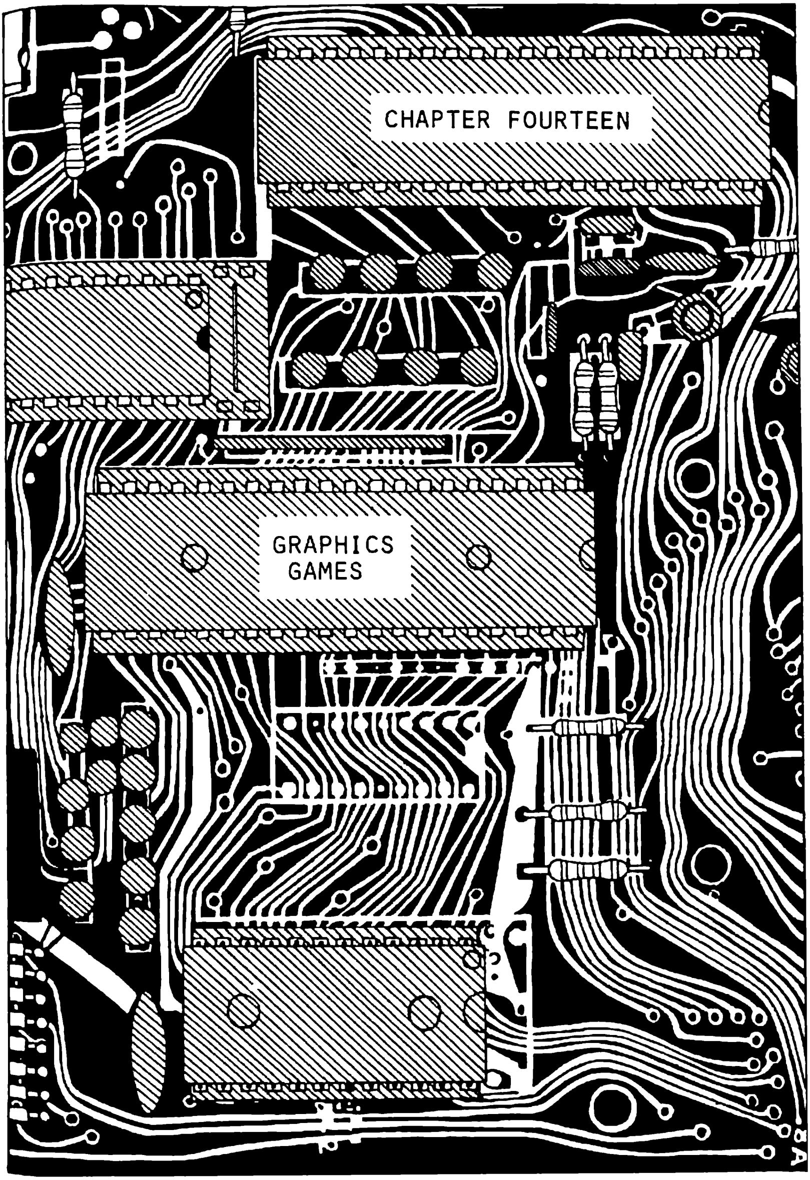
Now RUN it again - again by typing RUN 4. Does the computer make the same move? If it does it's purely coincidence, since choosing from the list is done at random. Try again, and again, remembering to break out of the program each time and re-run. You should get a different result each time.

We'll leave the program at this stage and continue later on with the mechanism of setting up the stack correctly in the first place, and actually deciding which moves are better than others.

In the next chapter we'll look at some complete (and short) games designed to demonstrate what machine code can achieve in terms of speed, and in very few bytes compared with BASIC.

CHAPTER FOURTEEN

GRAPHICS GAMES



SPIRALS

In this fast moving real-time graphics game (intended for use with SLOW) you are placed at the start of a square spiral and must reach the end of it in the minumum possible time. Your score is constantly displayed - it starts of at 99900 and decrements continuously, but you can't cheat by breaking out early with a high score - the program won't allow that.Now and again the score will reach zero before you reach the end of the spiral. If that happens you obviously need more practice!

This fascinating and highly amusing game is unfortunately for NEW ROM users with SLOW only. It will not work in FAST because although the program will still consider itself to be running perfectly smoothly, the average human operator won't know what's going on because of the fact that the screen in front of them is completely black.

This is a fascinating game to watch - witnessing the score decrease before your very eyes is surprisingly effective. You can make the game as difficult as you like by altering the initial value of the "timing" - held in BC. I've given it 0400, but you could use 0800 for a slower game, 0200 for a faster game, and so on.

There is one difficulty built in though - if you hit a wall you don't just bounce off, you actually become embedded in it, and the only way you can get out is to exactly reverse your direction.It can be quite tricky.

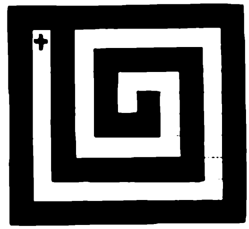
Well good luck on your race - keep a record of the high scores (no cheating) and see if you can master it.

The keys will move you as follows: Any key on the bottom row will move you downwards (except for shift, which has no effect), any key on the top row moves you up. The middle two rows move you left and right, with the left-hand ten keys (QWERTASDFG) moving you to the left, and the ten righthand keys (YUIOPHJKLn/l) moving you to the right. This system was adopted instead of using the cursor controls 5, 6, 7, and 8 for two reasons.
1) It is easier for people to understand and become familiar with.
2) It is easier to program, since we only need to test one register after the keyboard scan instead of two.

The program lists as follows, and can be relocated to any desired location by changing just one address. The program should be called from the point labelled START.

```
E1          SPRINT    POP HL         This subroutine prints
7E                    LD A(HL)       out a picture of the board,
23                    INC HL         along with your initial
E5                    PUSH HL        score. It must however be
FEFF                  CP FF          provided with a list of
C8                    RET Z          data terminated by FF.
D7                    RST 10
18F6                  JR SPRINT
CD-sprint START       CALL SPRINT    Calls the subroutine. The
                                     following is data for the
                                     subroutine.
DEFB 80 80 80 80 80 80 80 80 80 80 80 76
     80 15 80 00 00 00 00 00 00 00 00 76
     80 00 80 00 80 80 80 80 80 00 80 76
     80 00 80 00 80 00 00 00 80 00 80 76
     80 00 80 00 80 00 80 00 80 00 80 76
     80 00 80 00 80 80 80 00 80 00 80 76
     80 00 80 00 00 00 00 00 80 00 80 76
     80 00 80 80 80 80 80 80 80 00 80 76
     80 00 00 00 00 00 00 00 00 00 00 76
     80 80 80 80 80 80 80 80 80 80 80 76
     76
     3E 34 3A 37 00 38 28 34 37 2A 00 33 34 3C 00 25 25 25 1C 1C FF
```

```
2A0C40    SETUP      LD HL,(D-FILE)       This section initialises the
110E00               LD DE,000E           two "variables" used in our
19                   ADD HL,DE            program.
227B40               LD (POSITION),HL
210000               LD HL,0000
227940               LD (LASTMOVE),HL
2A0C40    LOOP       LD HL,(D-FILE)       Decrement the score.
118B00               LD DE,008B
19                   ADD HL,DE
```



```
7E        DECIMAL    LD A,(HL)
A7                   AND A
2008                 JR NZ,POSITIVE
0605                 LD B,05
23        RESET      INC HL
361C                 LD (HL),1C
10FB                 DJNZ RESET
C9                   RET
3D        POSITIVE   DEC A
FE1B                 CP 1B
2005                 JR NZ,OK
3625                 LD (HL),25
2B                   DEC HL
18EA                 JR DECIMAL
77        OK         LD (HL),A

010004               LD BC,SPEED          A timed delay. Altering the
0B        DELAY      DEC BC               initial value of BC changes
78                   LD A,B               the speed of the game.
B1                   OR C
20FB                 JR NZ,DELAY
CDBB02               CALL KSCAN           Scan keyboard. L now contains
7D                   LD A,L               a value corresponding to the
2F                   CPL                  direction required.
6F                   LD L,A
E681                 AND 81
2805                 JR Z,NOTDOWN         Find direction.
110C00               LD DE,000C
181C                 JR CHKMOVE
7D        NOTDOWN    LD A,L
E618                 AND 18
2805                 JR Z,NOTUP
11F4FF               LD DE,FFF4
1812                 JR CHKMOVE
7D        NOTUP      LD A,L
E660                 AND 60
2805                 JR Z,NOTRIGHT
110100               LD DE,0001
1808                 JR CHKMOVE
7D        NOTRIGHT   LD A,L
E606                 AND 06
28B2                 JR Z,LOOP
11FFFF               LD DE,FFFF
2A7940    CHKMOVE    LD HL,(LASTMOVE)     Is player embedded in wall?
7D                   LD A,L
B4                   OR H
2807                 JR Z,MOVE
19                   ADD HL,DE            If so, is player reversing?
7D                   LD A,L
B4                   OR H
2802                 JR Z,MOVE
18A1                 JR LOOP
```

```
2A7B40     MOVE     LD HL,(POSITION)    Reassign square with black
7E                  LD A,(HL)           or white space as required.
E680                AND 80
77                  LD (HL),A
19                  ADD HL.DE           Find new position.
7E                  LD A,(HL)           Draw black or white cross
F615                OR 15               as appropriate.
77                  LD (HL),A
227B40              LD (POSITION),HL
210000              LD HL,0000          Store direction moved if
17                  RLA                 a wall has been hit.
3002                JR NC,NOTHIT
62                  LD H,D
6B                  LD L,E
227940     NOTHIT   LD (LASTMOVE),HL
2A0C40              LD HL,(D-FILE)      Check to see whether the
113600              LD DE,0036          finishing square has been
19                  ADD HL,DE           reached.
ED5B7B40            LD DE,(POSITION)
ED52                SBC HL,DE
C8                  RET Z
C3-loop             JP loop
```

BREAKOUT

In this version of BREAKOUT, which incidently may only be run on a NEW ROM in SLOW, you move the bat with any of the keys on the keyboard - those on the left will move you to the left, and those on the right will move you to the right. The game is intended to be played only by those people with 3¼K or more, but it can be persuaded to run in less if the following few lines of machine code are added to the program - these should preceed the main program:

```
FD362200  EXTRA    LD (IY+22),00
210003             LD HL,0300
AF        SPACES   XOR A
D7                 RST 10
2B                 DEC HL
7C                 LD A,H
B5                 OR L
20F9               JR NZ,SPACES
```

The reason for this is that the main BREAKOUT program assumes that the screen is initially completely full - that is, that it contains twenty-four rows, each consisting of thirty-two spaces followed by a newline. If your machine has less than 3½K on board then this will not be so, because of the way that the ROM sets up the screen. To rectify this we first LD (IY+22) with 00. IY is always 4000 at the start of any USR routine, so IY+22 is 4022, which is the systems variable DF-SZ. This represents the number of rows in the bottom half of the screen (the part we cannot print to) - by telling the machine that this number is zero it follows that the number of rows that we cannot print to is also zero, thus the whole screen is at our disposal. HL counts the number of spaces to be printed to ensure that we do not try to run off the end of the screen.

BREAKOUT is a program in four parts. These parts are 1). Initialise everything. 2). Restart the game for each new ball. 3). Move the ball. 4). Move the bat. We will go over each of these steps in scrutinous detail.

Firstly to initialise everything. This involves a) printing the playing board, b) defining the initial ball position, and c) setting the initial speed of the game. To print the board:

```
20002200  TABLESTART  DEFW 0020 0022
E0FFDEFF              DEFW FFE0 FFDE
2A0C40    BREAKOUT    LD HL,(D-FILE)      Load all of the bricks into position.
118500                LD DE,0085
19                    ADD HL,DE
018080                LD BC,8080          B is the number of bricks, C is a
23        NXBRK       INC HL              constant used quite frequently in
7E                    LD A,(HL)           this section.
FE76                  CP 76
28FA                  JR Z,NXBRK
3608                  LD (HL),08
10F6                  DJNZ NXBRK
2A0C40                LD HL,(D-FILE)      Put top wall in position.
061E                  LD B,1E             This part puts in the first thirty
23        NXBL        INC HL              blocks.
71                    LD (HL),C
10FC                  DJNZ NXBL
23                    INC HL
369C                  LD (HL),9C          The current score - zero - is entered.
23                    INC HL
71                    LD (HL),C           The last block is set in place.
23                    INC HL
23                    INC HL
111F00                LD DE,001F          DE is one more than the number of
0617                  LD B,17             spaces between the walls.
71        SIDES       LD (HL),C           Both side walls are loaded into
19                    ADD HL,DE           position.
71                    LD (HL),C
23                    INC HL
23                    INC HL
10F9                  DJNZ SIDES
0620                  LD B,20             Now the base-line is drawn in.
361B      BASE        LD (HL),1B
23                    INC HL
10FB                  DJNZ BASE
```

You'll notice that in this version of the game I've ensured that a row of full stops is printed below the very bottom of the screen. This provides a convenient test for whether or not the ball has hit the base. Finally, to set the ball position and speed, the proceedure is:

```
11FCFE                LD DE,FEFC          This is the displacement from the
                                          current print position to the ball's
                                          starting point.
19                    ADD HL,DE           Locate this starting point.
223C40                LD (BALLINIT),HL    Store it.
210009                LD HL,0900          This is the initial speed.
224640                LD (SPEED),HL       Store it.
```

This is actually all the initialisation we need. You'll notice several things missing - for example although the ball is located it is not actually printed. The bat is not mentioned at all! The reason is that the bat is redrawn every time the game is restarted, and so is the ball. Why bother to find the initial position then? Well in this version, the ball starts off in a slightly different position each time. This ensures that it is possible to wipe out all of the bricks.

The variable SPEED has a dual purpose. Firstly it determines the speed of the game - that is, the speed at which the bat and ball will move (the bat moves at precisely twice the ball speed), but secondly it determines when the game is over. When SPEED decrements to zero (the lower the number, the faster the game) we know that the game is over.

Section two of the game does the following tasks. a) change the initial ball position, whilst also noting the current ball position and printing the ball. b) Set the initial direction of movement of the ball to up/right. c) change the speed of the game and check for end of game. d) print the bat, and at the same time delete any previous bat symbol that may have been there. e) give the human player a chance to recover from the last session, since presumably she won't want one ball to leap into the game immediately the last one vanishes. The section is this. Look at the manner in which the bat is printed and the previous bat overwritten.

```
2A3C40   RESTART  LD HL,(BALLINIT)    Change the starting position of
23                INC HL              the ball.
223C40            LD (BALLINIT),HL
224040            LD (BALLPOS),HL     Start the ball here.
3634              LD (HL),34          Print the ball.
21E0FF            LD HL,FFE0          Set the initial direction.
224440            LD (DIRECTION),HL
3A4740            LD A,(SPEED)high    Increase the speed
3D                DEC A
C8                RET Z               Return to BASIC if lives have run
324740            LD (SPEED)high,A    out.
2A0C40            LD HL,(D-FILE)      Reprint the bat in its starting
11B702            LD DE, 02B7         position
19                ADD HL,DE
3600              LD (HL),00
3E03              LD A,03             A contains the bat symbol.
23                INC HL
77                LD (HL),A
23                INC HL
77                LD (HL),A
23                INC HL
77                LD (HL),A
224240            LD (BATPOS),HL      Store the initial bat position. (This
23                INC HL              is the position of the centre of the
77                LD (HL),A           bat.
23                INC HL
77                LD (HL),A
0618              LD B,18             Now erase the rest of the row, in
23       ERASE    INC HL              case a previous bat symbol remains
3600              LD (HL),00          there.
10FB              DJNZ ERASE
210000            LD HL,0000          Set a very long delay, for the player
1803              JR DELAY            to recover for the next ball.
```

The last two lines, which cause a short pause between sessions, will become clear when the start of the next section is given.

To move the ball we first of all go through a timed delay loop (controlled by SPEED - the speed of the game) and then unprint the previous position of the ball. The contents of the next square in the direction the ball is travelling are examined, and one of the following will happen:

If a full stop has been reached then the ball has gone off the bottom of the screen - the game is restarted.

If either a space (ie nothing hit) or a brick is located, the ball is reprinted, at this new position.

If anything other than a space is reached, the direction of movement of the ball is changed at random.

If the ball was not reprinted then find the contents of the next square in this new direction and re-examine the situation.

If a brick was hit, the score is increased by 1.

Now, in order that we may choose a new random direction <u>validly</u> we require a table of directions to choose from. These valid directions are 0020, 0022, FFE0, and FFDE. You should store these numbers, low part first, at any address in RAM, and call the start of this table TABLESTART. The program which will then achieve all of this is as follows:

```
2A4640          LOOP        LD HL,(SPEED)         This is a short delay loop which
2B              DELAY       DEC HL                controls the speed of the game.
7C                          LD A,H
B5                          OR L
20FB                        JR NZ,DELAY
04                          INC B                 The ball is only moved every other
CB40                        BIT 0,B               time round the loop, so that the
205A                        JR NZ,MOVEBAT         bat moves twice as fast as the ball.
2A4040          MOVEBALL    LD HL,(BALLPOS)       The current ball position is found.
3600                        LD (HL),00            Erase the ball.
ED5B4440                    LD DE,(DIRECTION)     Find the next position of the ball.
19                          ADD HL,DE
7E                          LD A,(HL)             Find the contents of this new
FE1B                        CP 1B                 position.
28A6                        JR Z,RESTART          Has the ball hit the base?
4F                          LD C,A                Start next ball if so.
E6F7                        AND F7
2005                        JR NZ,DONTMOVE        Only reprint the ball if the new
3634                        LD (HL),34            position is either empty or contains
224040                      LD (BALLPOS),HL       a brick.
B1              DONTMOVE    OR C                  Retrieve previous contents
283E                        JR Z,MOVEBAT          Change direction if not a space.
E5                          PUSH HL
2A3240                      LD HL,(SEED)          Generate new direction at random.
54                          LD D,H
5D                          LD E,L
29                          ADD HL,HL
29                          ADD HL,HL
19                          ADD HL,DE
29                          ADD HL,HL
29                          ADD HL,HL
29                          ADD HL,HL
19                          ADD HL,DE
223240                      LD (SEED),HL
7C                          LD A,H                Choose this direction from a
E606                        AND 06                table.
C6tablestartlow             ADD A,TABLESTARTlow
6F                          LD L,A
26tablestarthigh            LD H,TABLESTARThigh
5E                          LD E,(HL)
23                          INC HL
56                          LD D,(HL)
ED534440                    LD (DIRECTION),DE
E1                          POP HL
79                          LD A,C                If the contents of the square is
FE08                        CP 08                 not a brick, then move again.
20BF                        JR NZ,MOVEBALL
2A0C40                      LD HL,(D-FILE)        Having established that a brick has
111F00                      LD DE,001F            been hit, the score is increased by
19                          ADD HL,DE             one.
7E              CARRY       LD A,(HL)
FE80                        CP 80
2002                        JR NZ,DIGIT
3E9C                        LD A,9C
3C              DIGIT       INC A
FEA6                        CP A6
2005                        JR NZ,INCREASED
369C                        LD (HL),9C
2B                          DEC HL
18EF                        JR CARRY
77              INCREASED   LD (HL),A
```

An interesting point to watch for is the way in which the score is increased. Compare the mechanism to that used in SPIRALS to decrease the score. There are one or two differences between this and the last. Firstly of course we are here using INVERSE digits instead of ordinary digits, though this difference is rather trivial. Secondly the BREAKOUT score <u>increases</u> instead of decreases. Thirdly, the SPIRALS score would terminate at zero, wheras the BREAKOUT score can increase indefinately.

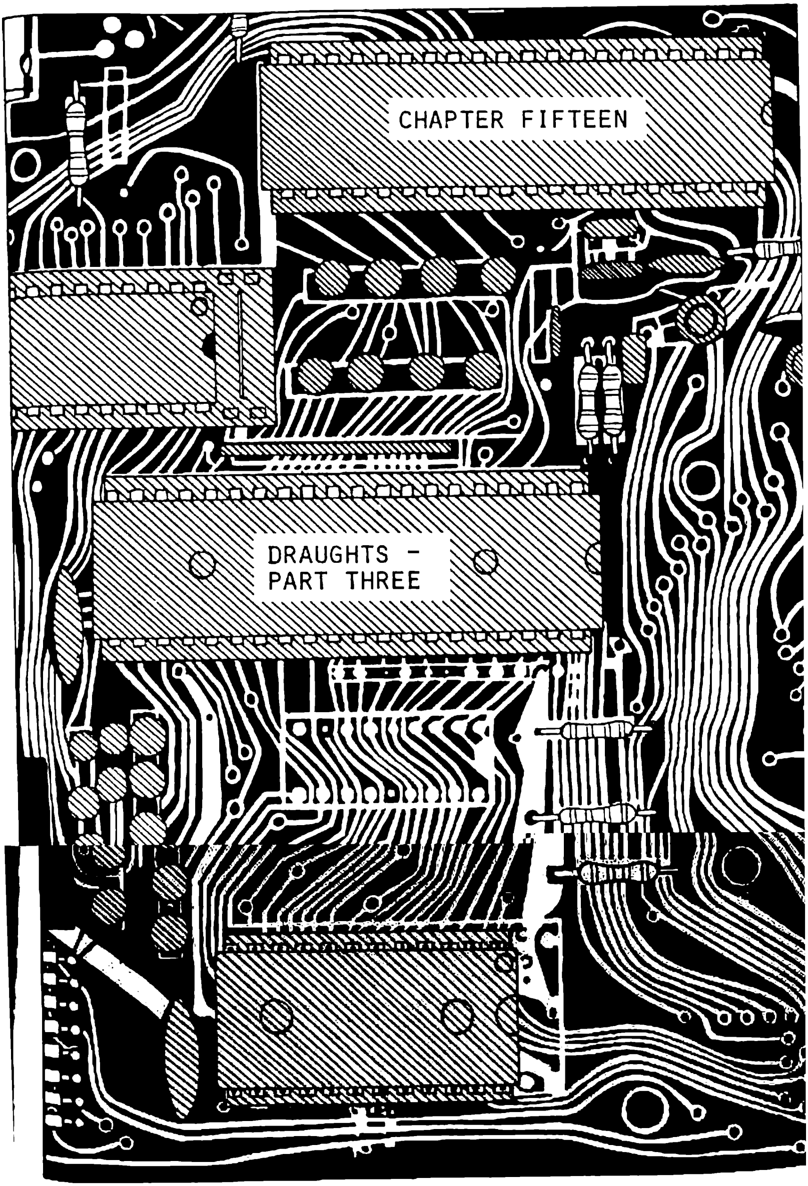
To move the bat, first of all the keyboard is scanned, and if a left-hand key is pressed the bat is moved to the left, provided of course there is not a wall in the way, and if a right-hand key is pressed then the bat is moved to the right, if possible. Note that if a left <u>and</u> right key are pressed simultaneously the bat should not move at all. In our program such a circumstance would cause the bat to move first to the left, and then to the right.

Study this, the final part of the program, and watch the way the bat is actually moved. Remember that the variable BATPOS stores the position of the <u>middle</u> of the bat.

```
C5         MOVEBAT   PUSH BC            Preserve the value of B.
CDBB02               CALL KSCAN         Scan the keyboard.
C1                   POP BC
7D                   LD A,L
2F                   CPL
F5                   PUSH AF            Stack contains a value corresponding
E60F                 AND 0F             the key pressed.
2817                 JR Z,NOTLEFT       If the player moves left....
2A4240               LD HL,(BATPOS)     Locate the bat.
2B                   DEC HL
2B                   DEC HL
2B                   DEC HL
7E                   LD A,(HL)          Is there a wall to our left?
FE80                 CP 80              If so, don't move.
2829                 JR Z,CYCLE1
3603                 LD (HL),03         Extend the bat to the left.
23                   INC HL
23                   INC HL
224240               LD (BATPOS),HL     Store new bat position.
23                   INC HL
23                   INC HL
23                   INC HL
3600                 LD (HL),00         Decrease bat to the right.
F1         NOTLEFT   POP AF
E6F0                 AND F0
2819                 JR Z,CYCLE2        If the player moves right....
[illegible]
3603                 LD (HL),03         Extend bat to the right.
2B                   DEC HL
2B                   DEC HL
224240               LD (BATPOS),HL     Store new bat position.
2B                   DEC HL
2B                   DEC HL
2B                   DEC HL
3600                 LD (HL),00         Decrease bat to the left.
E5                   PUSH HL            Same for next time round.
E1         CYCLE1    POP HL
C3loop     CYCLE2    JP LOOP
```

CHAPTER FIFTEEN

DRAUGHTS – PART THREE



The story so far... Once upon a time a human being input a move to a ZX computer. The computer checked this move to make sure that no cheating was going on, and cast a wicked spell on the poor human if it was which meant that the whole move had to be typed in all over again. The move was made. The computer started to search through the board for pieces that it could move. Having found a piece, but not knowing whether or not it could move, it then miraculously found itself at an address called EVALUATE. Where do we go from here?

Let's start off by saying that a neutral move - that is a move which achieves nothing, but also loses nothing - has a "priority" of 80, (hex).

The first point worth noting is that if a piece is in imminent danger of being captured then it stands to reason that we ought to move it out of the way - unless something more important crops up. Secondly, if a piece is preventing another piece from being captured, then we should be less likely to move it. Both of these conditions apply regardless of which direction we consider moving the piece. It stands to reason then that we should work out this part of the priority first, before we start analysing each of the different directions. We must therefore work out a numerical value that corresponds to the square that we are looking at. This value will then be added to 80, after which each direction in turn will be analysed.

EVALUATE will therefore start off

```
4E43    CDF14D    EVALUATE  CALL SQUAREVAL
        C680                ADD A,80
        322140              LD (INITIAL),A
```

The last instruction stores the value we've found for use later on in the game. On the OLD ROM the address of INITIAL should be changed to 4019. Now let's take a closer look at the subroutine SQUAREVAL. It will assign a value as follows - starting with zero, if a piece is in danger it will add five, or seven for a king. If it is protecting a piece it will subtract five, or seven for a king. Further, the subroutine, as with all subroutines from now on, must not be allowed to alter the values of any register except A. One way of doing this is to begin the subroutine

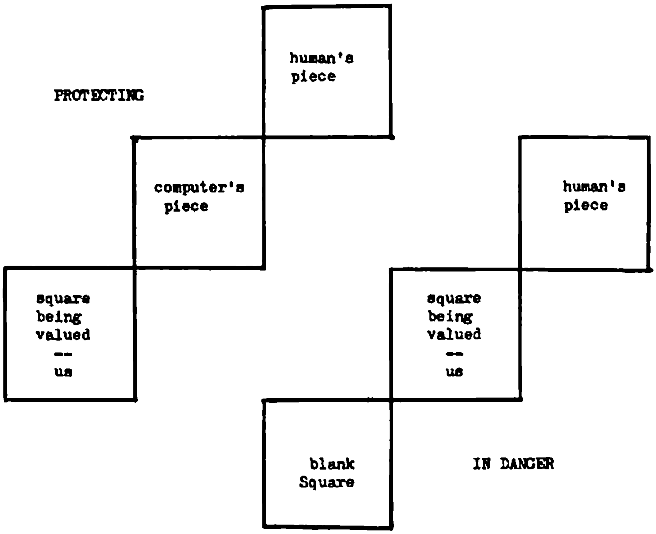
```
4DF1    C5        SQUAREVAL PUSH BC
        D5                  PUSH DE
        E5                  PUSH HL
```

Here is the complete subroutine. Follow it through carefully. It should be sufficiently annotated for you to make sense of exactly what it's doing.

```
C5      SQUAREVAL PUSH BC       Store the current value of the
D5                PUSH DE       registers on the stack, to be
E5                PUSH HL       retrieved at the end of the subroutine.
0600              LD B,00       B is being used as a flag here. The
                                first time round the loop it will be
        zero, the second time round it will be one. Watch the checks on B
        carefully. The loop will check for protection the first time round, but
        for danger the second time round.
11974C  STARTOFF  LD DE,TABLE   DE is a pointer, which points to the
                                table of directions of movement.
1A      NOWT      LD A,(DE)
4F                LD C,A        C now contains such a direction.
D62E              SUB 2E
282A              JR Z,EXIT     If this "direction" is 2E we have passed
                                the end of the table. We should exit
                                with value zero.
1C                INC E         Move pointer to next direction in table.
```

```
E1              POP HL
E5              PUSH HL         L contains the low part of the current
2640            LD H,40         square. We retrieve it without altering
                                the stack, and reassign H to tne high
                                part of this address.
7D              LD A,L
81              ADD A,C         Find square to be looked at in this
CB40            BIT 0,B         direction. Watch how B affects what
2001            JR NZ,LA        happens.
81              ADD A,C
6F      LA      LD L,A
3E7F            LD A,7F         Watch how A is constructed here. If
B1              OR C            a human's piece is present A will end
A6              AND (HL)        up as 27 UNLESS that piece is a non-
FE27            CP 27           king which can't move toward us. Then
20E5            JR NZ,NOWT      it will produce A7. No other piece can
                                generate the result 27.
7D              LD A,L
91              SUB C           Look at next square toward us. If B is
6F              LD L,A          zero we are looking at a possible piece
                                being protected. If B is one we are
                                looking at ourselves.
7E              LD A,(HL)
37              SCF             This is another way of checking for
17              RLA             a computer's piece rgardless of whether
CB40            BIT 0,B         or not it is a king, but watch the
2006            JR NZ,LB        carry flag.
FE79            CP 79
20D7            JR NZ,NOWT
7E              LD A,(HL)
17              RLA
3F      LB      CCF             Now notice the clever way we decide
3E81            LD A,81         on 5 for a piece, or 7 for a king.
17              RLA
17              RLA             A now contains 5 or 7 as needed.
CB40    EXIT    BIT 0,B         The loop is now ended.
2006            JR NZ,LC
04              INC B           This is what happens if B was zero.
67              LD H,A          The value 5, 7, or 0 is stored on the
E3              EX (SP),HL      stack behind HL.
E5              PUSH HL
18C3            JR STARTOFF
57      LC      LD D,A          This is what happens if B was one.
                                D now contains the current value
                                0, 5, or 7.
7D              LD A,L          The square behind us is located.
91              SUB C
6F              LD L,A
7E              LD A,(HL)       The contents of this square are
                                examined.
FE80            CP 80           If it is not a blank square we are
28BD            JR NZ,NOWT      not in danger.
7A              LD A,D          The current value is retrieved.
E1              POP HL
D1              POP DE          D now contains the previous value
                                0, 5, or 7.
92              SUB D           The final square-value is calculated.
D1              POP DE          The remaining registers are removed
C1              POP BC          from the stack.
C9              RET             End of subroutine.
```

This works because if you take a look at the diagram below you'll see very clearly the conditions under which we define a piece as being "in danger" or protecting. Compare carefully what the subroutine does both times round, with each of the diagrams.



Now for the rest of that decision making routine EVALUTE. It contains a deliberate mistake - see if you can find it! (The program will still run perfectly smoothly even with the mistake still in.) If you can't sus it out on your own I'll tell you later on.

This routine is designed to compute a numerical value - a "priority" - for any individual move. Having done so it will compare this priority with those moves stored on the stack. If the new priority is <u>less</u>, it will forget this move and go on to explore a new one. If the new move is <u>equal</u> in priority it will be stored on the stack. If the new priority is <u>more</u> than those on the stack then the list will be abolished, and a new list started.

The registers in the routine have the following jobs:

A- a general purpose working register.
B- counts the number of items in the list. You may remember the CHOOSE routine earlier on relied on B containing this number of items.
C- a general purpose working register.
DE- a pointer which looks at the table of allowable directions of movement.
H- the direction being moved.
L- the low part of the address of the current square.

The routine begins at address 4E43:

```
CDF14D    EVALUATE  CALL SQUAREVAL      Check for danger and/or protection
C680                ADD 80              at current square.
322140              LD (INITIAL),A
```

11974C		LD DE,TABLE	Set pointer to start of table.
4D		LD C,L	Remember low part of the address of the current square for later use.
69	NXTMRND	LD L,C	Retrieve this value.
2640		LD H,40	Assign high part of this address.
1A	NXTDIR	LD A,(DE)	Select direction of movement.
1C		INC E	Move table pointer.
CB7E		BIT 7,(HL)	Check whether or not we are looking at a king.
2804		JR Z,ANYDIR	If so we can move in any direction.
CB7F		BIT 7,A	Check whether current direction is forward or backward.
20F6		JR NZ,NXTDIR	If backward, pick a new direction.
FE2E	ANYDIR	CP 2E	If this direction is 2E then we have
CAA04D		JP Z,KPCHKNC	covered all four directions.
C5		PUSH BC	Temporarily stack B - the number of items in the list of moves.
47		LD B,A	Store current direction temporarily.
81		ADD A,C	Find the address of the destination
6F		LD L,A	square in this direction.
7E		LD A,(HL)	Find the contents of this square.
60		LD H,B	The direction being moved is now stored in H, as required.
C1		POP BC	The number of choices of moves on the stack - B - is recovered.
FE80		CP 80	Is this destination square empty?
20E3	TEST	JR NZ,NXTMRND	If not, pick a new direction to examine.
ED537940		LD (SCANSQR),DE	Temporarily store the value of DE.

Note that while we need to temporarily store DE somewhere, we must not stack it, since we are shortly about to use the stack to examine our list. OLD ROM owners should interpret the address (SCANSQR) as 4020.

CDF14D		CALL SQUAREVAL	Check for danger and/or protection at destination square.

This is necessary because a move into danger is bad, and moving to protect another piece is good. Notice that by design the subroutine SQUAREVAL will not change the value of any register except A. One unfortunate flaw in the subroutine means that moving a king into danger will only generate the value five, rather than seven. Can you see why? Follow the subroutine through if you can't. Finally you should note that SQUAREVAL only requires L to be assigned initially, not HL. This is deliberate.

57	NEWPRI	LD D,A	Negate this quantity, since we do
3A2140		LD A,(INITIAL)	not want to move into danger, and we
92		SUB D	do want to move to protect another
57		LD D,A	piece. Add in the original square-value and store the result in D.
1E01		LD E,01	The number one is the number of steps
69		LD L,C	involved in this move.

We now have D containing the computed priority of this move, and E containing the number of steps in this move.

E3		EX (SP),HL	We now have H containing the priority of the list, and L containing the no. of steps for each move on the list.

```
A7                      AND A
ED52                    SBC HL,DE          Compare these two sets of quantities.
280D                    JR Z,EQUAL
19                      ADD HL,DE          Restore HL and the stack-top
E3                      EX (SP),HL
3013                    JR NC,FORGETIT     If computed priority is less, then
                                           do nothing.
ED7B7B40                LD SP,(LBASE)      Otherwise begin new list.
0600                    LD B,00            Zero items on list so far.
D5                      PUSH DE            Stack the priority and no. of steps.
1802                    JR NEWITEM
19         EQUAL        ADD HL,DE          Restore HL and the stack-top.
E3                      EX (SP),HL
04         NEWITEM      INC B              Increase no. of items in list.
E5                      PUSH HL
```

Now H contains the direction moved, and L the low part of the initial square. The top of the stack therefore now looks like this:

initial square	direction one	no. of steps	priority

↑ sp

This is not quite what we want - we want it to look like this:

no. of steps	priority	initial square	direction one

↑ sp

So we now want to swop the first and second bytes at the top of the stack with the third and fourth bytes. We want to do this without altering the position of the stack pointer, and without altering any of the registers. The following will achieve this - follow it through carefully -

```
33                      INC SP             Move the stack pointer to the
33                      INC SP             initial square. (final position)
E3                      EX (SP),HL         store initial square and direction 1.
3B                      DEC SP             Move the stack pointer back where it
3B                      DEC SP             came from.
E3                      EX (SP),HL         Store the number of steps and priority.
```

Note that even HL remains unchanged by this method. EVALUATE needs only two more instructions to complete it. These are

```
ED5B7940   FORGETIT     LD DE,(SCANSQR)    Restore the previous values of D
18B0                    JR NXTMRND         and E, and do the same for next
                                           direction.
```

As it stands the program will not test whether or not a computer's piece has reached the back row (and thus become a king). This is not a programming error, this is quite deliberate. The reason is that this is something I'd like you to do for yourself. Study the way in which the check on a human's piece is made - the low part of the destination address is compared with the low part of the address of the boundry between the back row and the second row - and make a similar test. You should find this a very simple addition to the program.

The EVALUATE routine is now complete. The whole program is now a closed structure - there are no holes in it now, no RET statements temporarily taking the place of subroutines that aren't there. If you now RUN the program (by typing RUN 4) it will actually make moves! Of course it won't do much else, but you should now be able to see how far we've progressed.

Oh - there is of course that deliberate mistake to think about. If you didn't notice it in the listing you probably noticed it by playing it. The problem is that the computer won't jump. As you can imagine this leads to a very poor game on its part.

The mistake is in the line labelled TEST. It currently says JR NZ,NXTMRND, which means that if a square in any particular direction is simply not empty then it will try a different direction. The line should read JR NZ,WHAT, where WHAT is a routine (which we haven't yet written) which is designed to decide whether the destination square contains a human's piece, whether a jump is possible - even whether or not a multiple jump is possible - and to evaluate the priority of whatever it finds.

Here is one such subroutine. It is not the only possible one, but a suggestion of one means of doing it. This particular version will cope only with single jumps, not with multiple jumps: The routine begins at 4E9B:

```
ED537940  WHAT   LD (SCANSQR),DE      Temporarily store the value of DE
57               LD D,A               Store the contents of the square
                                      we are now looking at in D.
E67F             AND 7F               Is it a human's piece?
FE27             CP 27
2806             JR Z,FOUND
ED5B7940         LD DE,(SCANSQR)      If not, retrieve the original value
1899             JR NXTMRND           of DE and resume the search.
3E81      FOUND  LD A,81              Assign A with either five or seven
CB12             RL D                 depending on whether or not we have
3F               CCF                  found a king.
17               RLA
17               RLA
57               LD D,A               Store this in D.
5C               LD E,H               Store the current direction in E.
7D               LD A,L               Find the next square in this direction.
84               ADD A,H
6F               LD L,A
2640             LD H,WKBOARD-low
7E               LD A,(HL)            Find the contents of this square.
63               LD H,E               Restore H to its previous value.
FE80             CP 80                Is this square empty?
2807             JR Z,JUMP
ED537940         LD DE,(SCANSQR)      If not, restore the original value
C34F4E           JP NXTMRND           of DE and resume the search.
CDF14D    JUMP   CALL SQUAREVAL       Check for danger and/or protection
                                      at destination square.
92               SUB D                Take contents of square into account.
18A2             JR NEWPRI            Check this new priority to see if it's
                                      worth stacking.
```

As you can see, the principle for finding a single jump is relatively straightforward. With this routine in place the computer will now play an adequate game of draughts, but although the human player is allowed to make multiple jumps, the computer will not. This addition I leave you to write yourself. I will, however give you a couple of hints.

First of all, the registers all have specific uses. All that is, except for A and C. These are as follows:

B - The number of choices of move available.
D - The priority of the current move.
E - The number of steps in the current move.
H - The direction being moved this step.
L - The low part of the address of the current square (within WKBOARD)

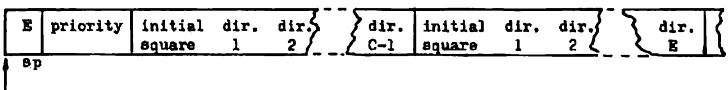
I suggest giving C a use too - it should be used to store <u>which step</u> of a multiple-step move we are currently examining. In other words, on the second step C will be two, on the third step C will be three, and so on. It is fairly easy to preserve the values of all of the registers by making proper use of the stack.

Nesting the subroutines and loops properly, so that the same routine is used to check for a third move as is used to check for a second move, is not as difficult as you might think - it merely requires a bit of positive thinking. It also has the advantage that, in theory, the computer can actually make twelve-fold jumps with no extra programming. The looping is not the biggest problem.

There are two problems which will face you. These are:

1) Having stored C-1 steps of the current move on the stack, how do we store step C? (ie how do we insert it into the middle of the stack)
2) Having established that the current move now stands at C steps, and can be increased no more, one of the following must happen: If C is less than E then the current move is abolished; if C is equal to E, the stack is left unchanged; if C is greater than E then the whole list of moves on the stack <u>except</u> the current move is abolished.

Let's take a look at the first problem first. Assuming C-1 steps are stacked, the situation we now have is this:



We wish to insert "direction C" between "direction C-1" and the initial square of the second move. The following subroutine will do just that, but follow it through very carefully because its mechanism is quite intricate.

```
C5       ADDASTEP  PUSH BC       The number of bytes at the top of the
D5                 PUSH DE       stack which need to be shifted down
E5                 PUSH HL       is C plus two, but once BC, DE, and HL
3E08               LD A,08       have been pushed onto the stack the
81                 ADD A,C       actual number is C plus eight.
210000             LD HL,0000
44                 LD B,H
4F                 LD C,A        This number is stored in BC.
39                 ADD HL,SP     HL points to the top of the stack.
54                 LD D,H
5D                 LD E,L
1B                 DEC DE        DE points to one byte below this.
EDB0               LDIR          Part of the stack is moved down.
3B                 DEC SP        The stack pointer is moved also.
E1                 POP HL
7C                 LD A,H
12                 LD (DE),A     The current direction is put in place.
D1                 POP DE
C1                 POP BC        The registers are retrieved.
0C                 INC C         C is increased to indicate that we
                                 are now at the next step.
```

You'll notice that the sequence LD HL,0000/ADD HL,SP is necessary because there is no such instruction as LD HL,SP (even though LD SP,HL is allowed). LDIR is used to shift the required part of the stack down one byte. The exact number of bytes to be shifted must first be very carefully calculated, and stored in BC in order that LDIR will work properly. Coincidently LDIR will leave DE finally pointing to just the right address for us to store the current direction. Since HL is at the top of the stack we may remove it, and load the current direction (H) into position, via A, <u>before</u> we remove DE and BC. Thus the stack pointer is still where we want it, and none of the values of any register (except A) have been changed.

The stack now looks like this:

E	priority	initial square	dir. 1	dir. 2	...	dir. C-1	dir. C	initial square	dir. 1	...	dir. E

↑ sp

Finally, C is incremented because we are now ready to examine the next step.

The two proceedures involved in the second problem may be solved by careful study of the above process. To abolish the current move is simple - DE is popped, the stack pointer is then incremented by the exact number of bytes, and DE is pushed back again. The second proceedure, that of abolishing the whole list except for the current move may be achieved by loading HL with the position within the stack of "direction C", DE with the contents of the variable LBASE, and then using LDDR, however, you'll have to do some thinking in order to work out BC (the number of bytes to be moved) and the new position of the stack pointer. If you understand how ADDASTEP works it will not be all that difficult to do.

With this problem to solve, I will leave you. It's not impossible I assure you. Finally, consider the length of this program so far - our addresses still begin with 4E, and we are allowed to go as far as 4FFF (although we need some left over for the screen and the stack). 1K draughts is quite, quite possible. With thought you may even be able to shorten it further.

<u>DOWNLOADING</u>

Although the program is only 1K it is currently stored in the <u>fourth</u> K. To download it into the <u>first</u> K the proceedure is this.

Change every address beginning with 4C to the corresponding address which begins 40. Do the same for 4D, changing it to 41, change 4E to 42, and 4F to 43.

Delete all lines of BASIC except the following:

<u>OLD ROM</u>

```
1 RANDOMISE USR(printboard)
2 INPUT A$
3 RANDOMISE USR(game)
4 GOTO 2
```

<u>NEW ROM</u>

```
1 INPUT A$
2 RAND USR game
```

(USE <u>ANY</u> FIVE DIGIT NUMBER FOR NOW)

Reserve enough space for the machine code using a <u>series</u> of REM statements from line 5 onwards. On the OLD ROM a REM statement with 46 characters after the word REM occupies exactly fifty bytes. On the NEW ROM a REM statement with 44 characters after the word REM occupies fifty bytes. The machine code will eventually overwrite not only the characters after the word REM, but the word REM itself and even the line numbers.

OLD ROM: type POKE 16463,-1
NEW ROM: type POKE 16535,-1
All of your REMs should disappear from the listing.

Now, using a machine code program, which you should store somewhere in the third K, copy all of the draughts program from address 4C97 onwards, down to 4097 onwards.

OLD ROM: copy the board printing routine to the point immediately after the draughts program proper finishes.

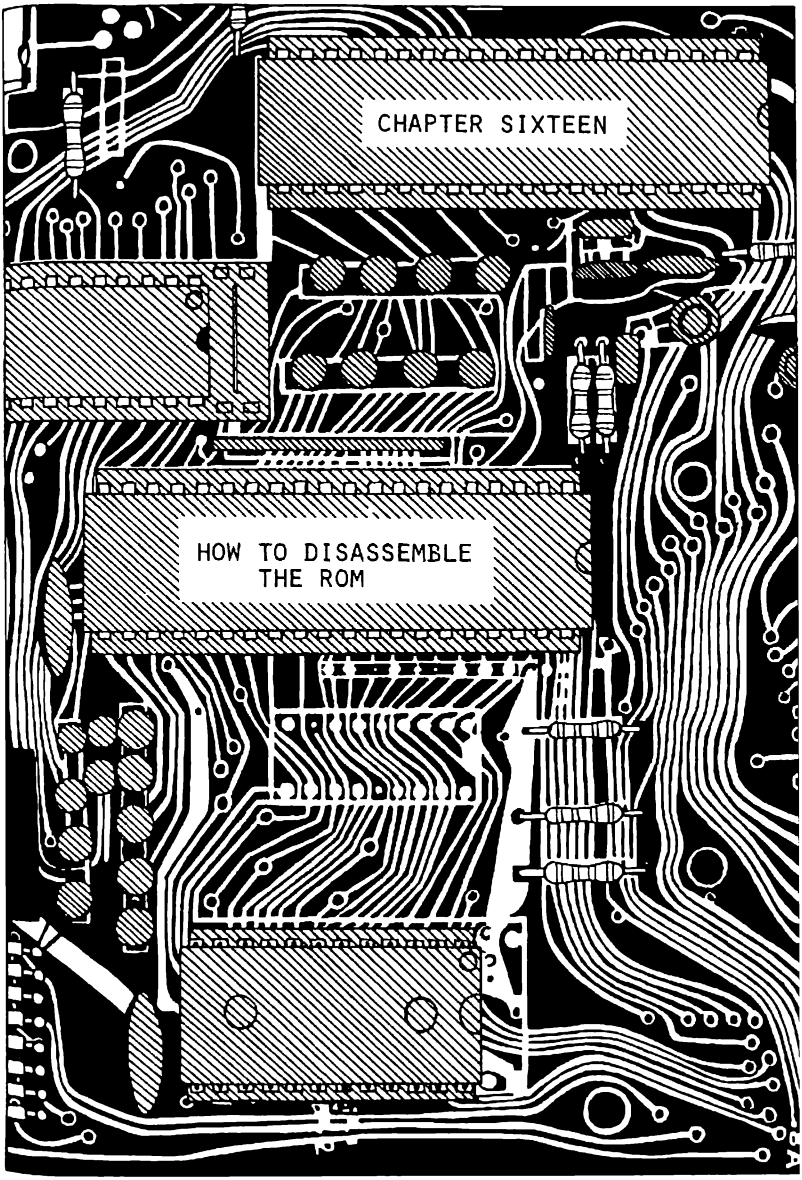
NEW ROM: DO NOT copy the board printing routine at all. Instead, leave it at 4C09, and replace the instruction RET by the following machine code program.

217D40	LD HL,FIRSTLINE	Fool the ROM into thinking that the
222940	LD (NXTLIN),HL	first line of program is about to
C30703	JP SAVE	be executed, then jump to the SAVE routine.

Start your cassette recorder up, so that it is recording, not playing, and type as a direct command RAND USR 19487. This should be done in the FAST mode. The program will then do the following tasks. 1) Print the playing board, 2) Specify that line one is about to be executed, and 3) SAVE the program, and the current display file (with the board pre-set-up) and the fact that line one is about to be executed. When you re-load from tape you will be in mid-program, with the first move (yours) about to be made.

The label "printboard" for the OLD ROM refers to the address at which the board printing routine is to be placed. The label "game" refers to the address 16612.

For the OLD ROM, the address WKBOARD should be changed to that of the board printing routine throughout. In this way the same space is effectively used twice. For the NEW ROM, the address WKBOARD should be left unchanged at 403C.



CHAPTER SIXTEEN

HOW TO DISASSEMBLE THE ROM

There are three "levels" at which we may disassemble, each slightly more sophisticated than the previous. The first two levels are not all that satisfactory, but they are very easy to program.

The first "level" we have already achieved - the USR routine HLIST which we saw earlier in the book will do this for us. That is, given an address such as 0808 it will produce an output like this:

```
0808 57
0809 ED
080A 4B
080B 39 T
080C 40
...
```

and so on. This is not really disassembly, although you can of course look these bytes up in the tables at the back of the book, but it's quite a time consuming task, and you're also very likely to get lost halfway through. The second "level" is not much better, but again is quite easy to program. What I'm talking about is an output something like this:

```
0808 57
0809 ED4B3940
080D 79
080E FE21
...
```

and so on. As you can see, each instruction has its component bytes listed out to exactly the right length. This produces a very pleasing display, and there is little or no chance of you "getting lost" when actually looking these bytes up in tables. The third "level" is the one we are actually aiming at - the one everybody wants. What we'd really like is an output like this:

```
0808 LD D,A
0809 LD BC,(4039)
080D LD A,C
080E CP 21
...
```

and so on. This <u>can</u> be quite easy to program - simply make the computer look up the appropriate words from a table instead of doing it ourselves - however this would take up rather a large amount of space just to store the table. Around 4K in fact. The method I will describe to you will allow such a program to fit in just 1K, but be warned: it's rather difficult. There is actually a "fourth level" of disassembly, which I won't even attempt to touch, but you may like to think about. Imagine an output like this:

```
PRINT     LD D,A
          LD BC,(S-POSN)
          LD A,C
          CP 21
          JR Z,EXIT
          ...
```

As I've said, I'm not even going to touch this one. The only extra it involves is storing yet another table, this time containing all of the labels used. Let's go back a bit now to something relatively simple. Let's consider a <u>slightly</u> improved version of HLIST which reaches the "second level" of disassembly, and works out the length of each instruction before printing it.

All we need is a table containing just two pieces of information for each byte. These are a) the number of bytes in an instruction beginning with this byte, and b) the number of bytes in an instruction beginning with DD or FD followed by this byte. As you know, some confusion may arise over those instructions beginning with CB or ED, but we don't actually need any tables or anything to cope with these provided we remember the following rules:

All instructions beginning CB are two bytes in length.

All instructions beginning DDCB or FDCB are four bytes in length.

All instructions beginning ED are two bytes in length, except for LD BC,(pq), LD DE,(pq), LD SP,(pq), LD (pq),BC, LD (pq),DE, and LD (pq),SP. The byte immediately after ED for these six instructions is 4B, 5B, 7B, 43, 53, or 73. In binary, all of these numbers have the form 01-- -011. No other instructions have this form.

There are <u>no</u> instructions beginning DDED or FDED.

Thus we need a table containing a very small amount of information relating to each byte. Firstly, those instructions which do <u>not</u> begin DD, ED, or FD can only be one, two, or three bytes in length. This means that to store the required information we only need <u>two bits</u>. Secondly those instructions which begin DD or FD can only be two, three, or four bytes in length, so ignoring the DD or FD itself this leaves one, two, or three bytes. Again we need only two bits. This makes four bits altogether, and we can thus represent the appropriate lengths for each byte by a single hexadecimal digit. Our program then will make use of the following table, called LENS. It should be stored such that each element of the table has the same high part of its address:

```
LENS      DEFB      5F 55 55 A5 55 55 55 A5
                    AF 55 55 A5 A5 55 55 A5
                    AF F5 55 A5 A5 F5 55 A5
                    AF F5 99 E5 A5 F5 55 A5
                    55 55 55 95 55 55 55 95
                    55 55 55 95 55 55 55 95
                    55 55 55 95 55 55 55 95
                    99 99 99 59 55 55 55 95
                    55 55 55 95 55 55 55 95
                    55 55 55 95 55 55 55 95
                    55 55 55 95 55 55 55 95
                    55 55 55 95 55 55 55 95
                    55 FF F5 A5 55 FE FF A5
                    55 FA F5 A5 55 FA F5 A5
                    55 F5 F5 A5 55 F5 FA A5
                    55 F5 F5 A5 55 F5 F5 A5
```

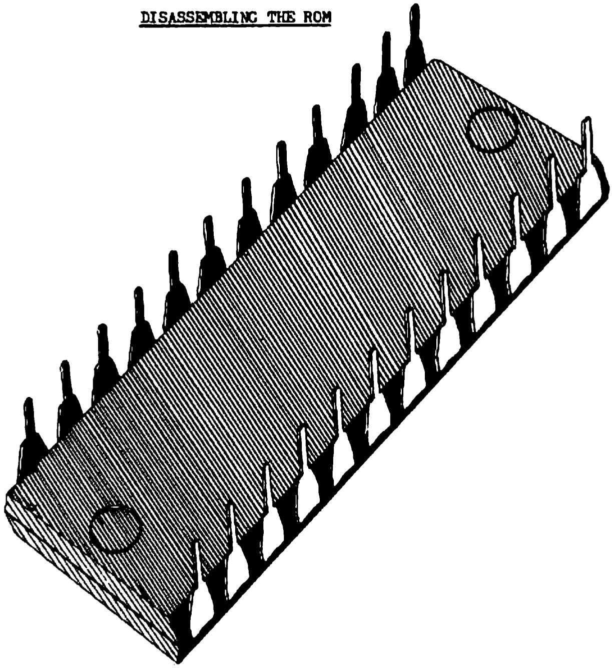
As you can see, there are sixteen rows, and sixteen hex digits in each row. Those instruction beginning with DD or FD which do not exist, such as DD00, have simply been assigned the appropriate number of bytes as if the DD/FD were not there.

The following program will "disassemble" to a string of bytes of the right length. It assumes that the table LENS exists, and it assumes that a subroutine HPRINT exists which prints the contents of the A register in hexadecimal without corrupting the other registers. This subroutine was in fact given earlier on in the book.

```
2B          START   DEC HL            HL is just  the address from which
23          NEXT    INC HL            we are disassembling.
3E76                LD A,76
D7                  RST 10            Print a newline.
7C                  LD A,H
CDhprint            CALL HPRINT       Print H in hex.
7D                  LD A,L
CDhprint            CALL HPRINT       Print L in hex.
AF                  XOR A
D7                  RST 10            Print a space.
0E00                LD C,00           C is just a flag to let us know
                                      whether or not an instruction
                                      begins with DD or FD.
7E          BYTE    LD A,(HL)         Obtain the byte to be disassembled.
FEDD                CP DD             Does it begin with either DD or FD?
2804                JR Z,DDFD
FEFD                CP FD
2007                JR NZ,NORM
CDhprint    DDFD    CALL HPRINT       If so, print "DD" or "FD" and look
23                  INC HL            at the next byte.
0C                  INC C             Change the flag C accordingly.
18F0                JR BYTE           Continue with next byte.
FEED        NORM    CP ED             Does the instruction begin ED?
201A                JR NZ,NOTED
CDhprint            CALL HPRINT       If so, print "ED" and look at the next
23                  INC HL            byte.
7E                  LD A,(HL)
E6C3                AND C3            Is it of the binary form 01-- -011?
FE43                CP 43
2004                JR NZ,CNE
0603                LD B,03           B counts the number of bytes to be
1802                JR THREE          printed after the byte ED.
0601        ONE     LD B,01
CDhprint    THREE   CALL HPRINT       Print the next B bytes.
23                  INC HL
7E                  LD A,(HL)
10F9                DJNZ THREE
18C2                JR NEXT           Continue with next byte.
E5          NOTED   PUSH HL           Temporarily store HL.
CB2F                SRA A             Divide A by two.
F5                  PUSH AF           Store the carry flag.
C6lens-low          ADD A,LENS-low    Find the reqired position in the table.
6F                  LD L,A
26lens-high         LD H,LENS-high
F1                  POP AF            Retrieve the carry flag.
7E                  LD A,(HL)
3804                JR C,DIG2         Use the carry flag to decide on which
1F                  RRA               digit from the table will be used.
1F                  RRA
1F                  RRA
1F                  RRA
0D          DIG2    DEC C             Use C to decide which two bits
2002                JR NZ,OK          to use.
1F                  RRA
1F                  RRA
E603        OK      AND 03            Put this number in B to use as
47                  LD B,A            a count.
E1                  POP HL            Retrieve theaddress of the byte to
2B                  DEC HL            be disassembled.
23          NXBYT   INC HL
7E                  LD A,(HL)
CDhprint            CALL HPRINT       print B bytes in hex.
10F9                DJNZ NXBYT
189E                JR NEXT           Continue with next byte.
```

Now we ascend to the "third level" - REAL disassembly in other words. However, I am not going to write the program for you this time round - you'll have to do it by yourself. I will explain precisely what it is you have to do in order to make a 1K disassembler, but the actual program itself must be your creation.

DISASSEMBLING THE ROM



The following is an algorithm which will enable you to dissasemble the hex codes into assembly, that is to change, for example, 69 to LD L,C, or from CB7E to BIT 7,(HL). One way would be to list a vast table - such as I have included in the appendices - but while alright for human beings it lacks the elegance of a well thought out computer program. The data alone would occupy around 4K. This algorithm will enable you to write your own machine language program occupying significantly less - two or even one K all told depending on how efficient your program is.

In this algorithm, the following conventions will be used:

r(0) means B, r(1) means C, r(2) means D, r(3) means E, r(4) means H, r(5) means L, r(6) means X, r(7) means A.

s(0) means BC, s(1) means DE, s(2) means Y, s(3) means SP.

q(0) means BC, q(1) means DE, q(2) means Y, q(3) means AF.

n(0) means 0, n(1) means 1; n(2) means 2, n(3) means 3, n(4) means 4, n(5) means 5, n(6) means 6; n(7) means 7.

c(0) means NZ, c(1) means Z, c(2) means NC, c(3) means C, c(4) means PO, c(5) means PE, c(6) means P, c(7) means M.

x(0) means ADD A,, x(1) means ADC A,, x(2) means SUB, x(3) means SBC A,, x(4) means AND, x(5) means XOR, x(6) means OR, x(7) means CP.

Define two variables, CLASS and INDEX, and initially let both of them equal zero.

Write the byte being disassembled in binary, and split it into three parts; F, G, and H. F consists of bits 7 and 6, G of bits 5, 4, and 3, and H of bits 2, 1, and 0. Thus to disassemble the byte 69 (binary 0110 1001) just split it into three parts thus: 01/101/001. In this particular case F is one, G is five, and H is one.

Next, split G into two parts; J and K; with J consisting of bits 2 and 1, and K just bit 0. If G then were binary 101 as above then split it like this: 10/1. In this case we would define J to be two, and K to be one.

Set aside an area of memory called DIS. This is to contain a STRING of unknown length. How you store this string is up to you. There are two different methods you could use - either terminate the data with an end-of-data character (any character will do, FF is as good as any), or begin the area DIS with one byte representing the number of characters of data there are in the string. (You only need one byte since DIS will never be more than 255 characters in length.) DIS should initially be an <u>empty</u> string, (ie containing no characters at all.)

The algorithm begins here.....

<u>If CLASS equals zero then the following applies;</u>

1) If the byte is 76 then complete dissasembled instruction is HALT.
2) If the byte is CB then let CLASS equal one and start again.
3) If the byte is ED then let CLASS equal two and start again.
4) If the byte is DD then let INDEX equal one and start again.
5) If the byte is FD then let INDEX equal two and start again.
6) If F equals zero then....
    - If H equals zero then....
        - If G greater than three then let DIS equal JR c(G-4),V.
        - If G less than four choose the Gth item in this list: NOP/EX AF,AF'/DJNZ V/JR V
    - If H equals one then...
        - If K is zero then let DIS equal LD s(J),VV
        - If K is one then let DIS equal ADD Y,s(J)
    - If H equals two then...
        - Let DIS equal LD plus the Gth item in this list: (BC),A/A,(BC)/(DE),A/A,(DE)/(VV),Y/Y,(VV)/(VV),A/A,(VV).
    - If H equals three then...
        - If K is zero then let DIS equal INC s(J)
        - If K is one then let DIS equal DEC s(J)
    - If H equals four then let DIS equal INC r(G)
    - If H equals five then let DIS equal DEC r(G)
    - If H equals six then let DIS equal LD r(G),V
    - If H equals seven then choose the Gth item from this list: RLCA/RRCA/RLA/RRA/DAA/CPL/SCF/CCF.

If F equals one then let DIS equal LD r(G),r(H).
If F equals two then let DIS equal x(G) r(H).
If F equals three then....

If H equals 0 then let DIS equal RET c(G)

If H equals one then...

If K is zero then let DIS equal POP q(J)

If K is one then choose the Jth item from this list: RET/EXX/JP (Y)/LD SP,Y.

If H equals two then let DIS equal JP c(G),VV

If H equals three then choose the Gth item from this list: JP VV/-/OUT (V),A/IN A,(V)/EX (SP),Y/EX DE,HL/DI/EI.

If H equals four then let DIS equal CALL c(G),VV

If H equals five then...

If K is zero then let DIS equal PUSH q(J).

IF K is one then let DIS equal CALL VV.

If H equals six then let DIS equal x(G) V.

If H equals seven then let DIS equal RST plus the Gth item in this list: 00/08/10/18/20/28/30/38.

<u>If CLASS equals one then the following applies:</u>

If F equals zero then choose the Gth item from this list: RLC/RRC/RL/RR/SLA/SRA/-/SRL and then add r(H).
If F equals one then let DIS equal BIT n(G),r(H).
If F equals two then let DIS equal RES n(G),r(H).
If F equals three then let DIS equal SET n(G),r(H).

<u>If CLASS equals two then the following applies:</u>

F cannot possibly equal zero.
If F equals one then....

If H equals zero then let DIS equal IN r(G),(C).

If H equals one then let DIS equal OUT (C),r(G).

If H equals two then...

If K equals zero then let DIS equal SBC HL,s(J).

If K equals one then let DIS equal ADC HL,s(J).

If H equals three then...

If K equals zero then let DIS equal LD (VV),s(J).

If K equals one then let DIS equal LD s(J),(VV).

If H equals four then let DIS equal NEG.

If H equals five then...

If K equals zero then let DIS equal RETN.

If K equals one then let DIS equal RETI.

If H equals six then choose the Gth item from this list: IM 0/-/IM 1/IM 2/-/-/-/-.

If H equals seven then choose the Gth item from this list: LD I,A/LD R,A/LD A,I/LD A,R/RRD/RLD/-/-.

If F equals two then choose the Hth item from this list: LD/CP/IN/OT/-/-/-/- and then add the Gth item from this list: I/D/IR/DR/-/-/-/-.
F cannot possibly be three.

<u>To compute the final output:</u>

If INDEX equals zero replace every Y by HL.
If INDEX equals one replace every Y by IX
If INDEX equals two replace every Y by IY

If INDEX equals zero replace every X by (HL)
If INDEX equals one replace every X by (IX+d) where d is defined by the next byte but one after the byte DD.
If INDEX equals two replace every X by (IY+d) where d is defined by the next byte but one after the byte FD.
(This does not apply if the X is preceeded by I)

Replace every V by the next byte in sequence (of those being disassembled).

DIS now contains the correctly disassembled instruction. This should now be printed to the screen.

It is possible to write a machine language program which disassembles things by using this algorithm. In fact it is possible to write such a program in just 1K. Surprizing as this may sound I should add that although it is possible, the program itself is rather complicated, and involves a completely new programming technique.

What I will do is to not actually write the program for you, but to give you hints and suggestions as to how it may be done. The program revolves around eight different subroutines, which are linked together by one MASTER subroutine which calls them all up in any required order. This is achieved as follows.

Somewhere in the program there should be a table called SUBTAB which contains eight different addresses - these are the addresses of the eight subroutines which control the program. The register-pair HL' (note the dash) will be pointing to a sequence of data which tells the MASTER subroutine which order it must call the others in. The data in this sequence is terminated by an item in which bit 7 is one. The data consists simply of numbers zero to seven. Zero calls subroutine zero, one calls subroutine one, and so on. Thus this number zero to seven determines exactly which subroutine the MASTER routine is to call.

So any item of data in this sequence looks, in binary, like this: 0--- -nnn for most items, or 1--- -nnn for the <u>last</u> item. (The part written nnn means the appropriate number zero to seven as described.) Now some of these eight subroutines will need to be supplied with DATA, which by coincidence will also need to be a number between zero and seven - if this number in binary is ddd then it makes sense to save space by storing this number amongst some of the unused bits of the subroutine-call, thus making it look, in binary, like this: 0-dd dnnn or 1-dd dnnn. We have now made use of every bit except bit 6. This isn't needed, so for sake of argument lets always make it zero. Any item of data in the sequence can then be 00dd dnnn, but the <u>last</u> byte must be 10dd dnnn.

I hope that didn't confuse you. To make things clear, suppose HL' points to an address at which is stored the sequence of data 00 01 22 83. This means that first of all subroutine zero is to be called, then subroutine one, then subroutine two (which will use the data binary 100 somewhere), then finally subroutine three. I say "finally" because bit 7 is set which means we are finished.

The master subroutine which will achieve this is as follows:

```
D9          MASTER    EXX
7E                    LD A,(HL)        Find byte of data, and increment
23                    INC HL           pointer.
D9                    EXX
5F                    LD E,A           Store this byte, in case bits 5, 4,
                                       and 3 contain data to be used in the
                                       appropriate subroutine.
E607                  AND 07           Isolate bits 2, 1, and 0.
17                    RLA              Multiply by two.
4F                    LD C,A           Store this number in the BC register
0600                  LD B,00          pair.
21 return             LD HL,RETURN     Specify the return address from each
E5                    PUSH HL          of the eight subroutines.
21 mastrads           LD HL,MASTRADS   Point HL to the start of the table
                                       which stores the eight subroutine
                                       call addresses.
09                    ADD HL,BC        Point HL to the required address.
4E                    LD C,(HL)        Store this address in the BC register
23                    INC HL           pair.
46                    LD B,(HL)
C5                    PUSH BC          Call this subroutine.
C9                    RET
7B          RETURN    LD A,E           If Bit 7 was not zero then continue
17                    RLA              with the next byte of data.
30E8                  JR NC,MASTER
```

You can learn a lot from studying this MASTER-SUBROUTINE. Can you see how the appropriate subroutine (one of eight) is called? First of all the label RETURN is pushed onto the stack. This means that if each of the eight routines ends with a RET instruction then control will jump to the label RETURN - just as if the subroutine had be accessed normally. To call the subroutine itself, the address of which was in the register-pair BC, we used PUSH BC followed by RET. Think carefully about how this works. The required address is pushed onto the stack, above the address RETURN. Then a RET instruction is executed. RET has the effect of popping the first number from the stack (the subroutine address) and jumping to that address. The first address left on the stack is now the address RETURN, which enables control to return correctly. All of this is necessary because there is no such instruction as CALL (BC) - in BASIC the statement GOSUB VARIABLE is allowed, but not in machine code. Another way we could have achieved the same as PUSH BC/RET is by using the sequence LD H,B/ LD L,C/JP (HL). Can you see why this does the same thing?

You may be wondering how the appropriate address came to be in HL' in the first place. There are two means by which this will be determined. Note that all of the alternative registers have specific jobs. These are:

BC'	The address of the byte to be disassembled.
D'	The variable INDEX.
E'	The variable CLASS.
HL'	Points to subroutine data.

The byte to be disassembled is located and stored in the D register by the means EXX/LD A,(BC)/INC BC/EXX/LD D,A. From this the quantities I called F, G, and H may later be discovered. Somewhere in the program there should be a table called TABLE containing twelve different addresses. HL' is simply read from this table. The twelve addresses correspond to the cases CLASS equals zero and F equals 0, 1, 2, or 3; CLASS equals one and F equals 0, 1, 2, or 3; and CLASS equals two and F equals 0, 1, 2, or 3.

The other way in which HL' may be determined is if subroutine zero is called. Subroutine zero is called by the data-byte 00. This will be immediately followed by eight different addresses corresponding to the cases H equals zero, up to H equals seven. Subroutine zero has the task of locating the appropriate address from this list and storing it in the register-pair HL'.

One subroutine you will need, (but not one of the eight central ones,) is a subroutine to add a single character to the end of the string DIS. Using the convention that the string begins at address DIS and is terminated by the byte FF, the string may be emptied by the sequence LD HL,DIS/LD (HL),FF. To add a character (held in the A register) the subroutine is

```
C5      ADDDIS  PUSH BC       Store the registers BC and HL so
E5              PUSH HL       that they won't be altered by the
                              subroutine.
0601            LD B,01       This is so that CPIR wont stop
                              because of BC.
21dis           LD HL,DIS     Find the start of the string.
F5              PUSH AF       Temporarily stack A.
3EFF            LD A,FF
EDB1            CPIR          Find the end of the string.
77              LD (HL),A     Insert a new end-of-string marker.
2B              DEC HL
F1              POP AF        Retrieve A.
77              LD (HL),A     Add this character.
E1              POP HL        Retrieve the remaining registers.
C1              POP BC
C9              RET           End of subroutine.
```

The eight subroutines you will need for this disassembly program are as follows:

SUBROUTINE 0 - SPLIT
This is the subroutine called by the byte 00. It is always the first subroutine called, if it is used at all. The byte 00 should be followed eight new addresses within the disassembler program. Located at these addresses are eight different sequences of data, which correspond to the cases H equals zero, H equals one, and so on up to H equals seven. One of these sequences is selected (according to H) and the data used to decide which of the eight subroutines should then be used.

SUBROUTINE 1 - LITERAL
The byte 01 (or 81 if it is the last subroutine-call in sequence) is followed by a series of characters, such as N O and P, which represent part or all of the disassembled instruction. The last character should have one of the unused bits (6 or 7) set, to indicate the fact that it is the last character. The subroutine should use one bit of data, with the meaning that if it is called by the byte 09 (or 89) then the literal data following should have a space inserted after the last character. This literal data is to be added to the end of the data storage area called DIS.

SUBROUTINE 2 - LIST-G
Means select the Gth item from the following list. The subroutine needs data to specify how many items there are in the following list. If there are four items the data 011 (3) is required, if there are eight items, the data 111 (7) is required, and so on, the data always being one less than the number of items in the list. For example the byte 3A (in binary 0/0/111/010 - meaning call subroutine 2 and provide it with the data 111) means select the Gth item from the following list of eight. The list could, for instance, be R, L, C, inverse A, R, R, C, inverse A, R, L, inverse A, R, R, inverse A,D, A, inverse A, C, P, inverse L, S, C, inverse F, C, C, inverse F. I've used 'inverse' to indicate the last character in an individual item. You don't have to do this - you can use any means you choose as long as it works. Thus if G (That is bits 5, 4, and 3 of the instruction being disassembled) were 5, the literal DAA would be added to the end of DIS. The next byte to be interpreted as data will be the byte after the inverse F.

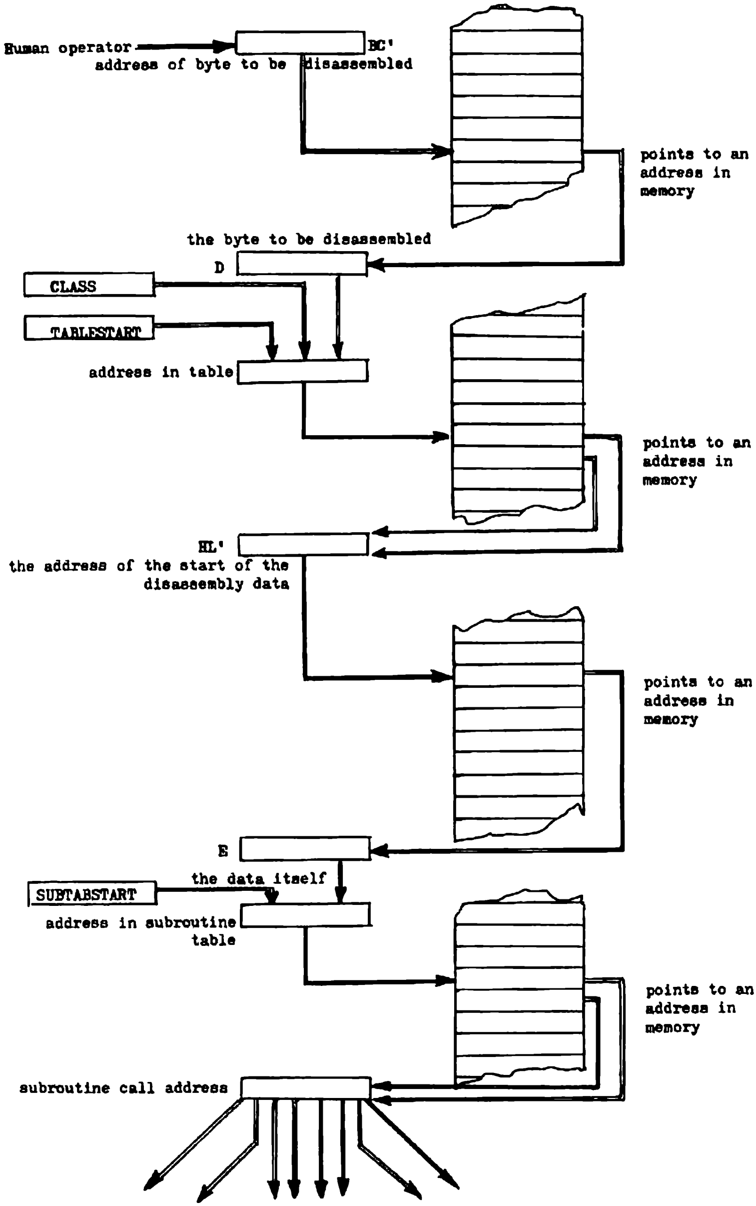
SUBROUTINE 3 - LIST-H
Means select the Hth item in the following list. Its explanation is exactly the same as that of subroutine 2.

SUBROUTINE 4 - SELECT-G
Again, three bits of data are required. Interpret as follows. If the data is 000 select r(G), if the data is 001 select s(G), if the data is 010 select q(G), if the data is 011 select n(G), if the data is 100 select c(G), and if the data is 110 select x(G). The item selected is to be added to the end of DIS.

SUBROUTINE 5 - SELECT-H
As subroutine 4, except that H is used instead of G.

SUBROUTINE 6 - SKIP
Resets bit 5 of E (the data-byte), and if the previous value of bit 5 was one skips over n bytes of data. The number n is determined by the immediately following byte. If bit 5 was zero this immediately following byte (which is only there to specify n) is ignored, and the next byte after is then interpreted as the next item of data.

SUBROUTINE 7 - K-SKIP
Replace bit 3 of E by bit 4, replace bit 4 by bit 5, and reset bit 5. Effectively this is the same as LET G equal J. Then if the previous value of bit 3 was one , n bytes are skipped over, as in subroutine six. This subroutine can be interpreted as IF K equals zero THEN.... otherwise IF K equals one then....


Human operator
BC'
address of byte to be disassembled
points to an address in memory
the byte to be disassembled
D
CLASS
TABLESTART
address in table
points to an address in memory
HL'
the address of the start of the disassembly data
points to an address in memory
E
the data itself
SUBTABSTART
address in subroutine table
points to an address in memory
subroutine call address

With these eight subroutines, which you will have to write yourself, you can disassemble every instruction. I will give you an example. Suppose CLASS is zero, and P is three. The first byte it has to interpret should be 00. This alters the value of HL' according to the quantity H, that is, bits 2, 1, and 0 of the byte being disassembled. Suppose now that H is one. HL' should now be pointing to the following sequence of data, listed here along with its meaning.

data	binary	meaning
07 05	0000 0111	KSKIP 5
09 35 34 B5	0000 1001	LITERAL POP (space)
94	1001 0100	SELECT-C·q (EXIT)
9A	1001 1010	LIST-C·4 (EXIT)
37 2A B9		RET
2A 3D BD		EXX
2F 35 00 16 3E 91		JP (Y)
31 29 00 38 35 1A BE		LD SP,Y

To represent strings of data here you can see I've used just the character codes, with the final character inversed to show that it is the last character. In other words EXX is written as 2A 3D BD rather than just 2A 3D 3D. It is of course very important to know where one string ends and the next begins.

If you follow through which subroutines have been called by the data and what they are supposed to do you'll see that in a total of only twenty-seven bytes we have said IF K equals zero then LET DIS equal POP q(J), IF K equals one then LET DIS equal the Jth item from this list: RET/EXX/JP (Y)/LD SP,Y. If this proceedure is continued for every instruction, following the algorithm I gave earlier in the chapter, you'll find that the data required for disassembly is now <u>significantly</u> LESS than 1K.

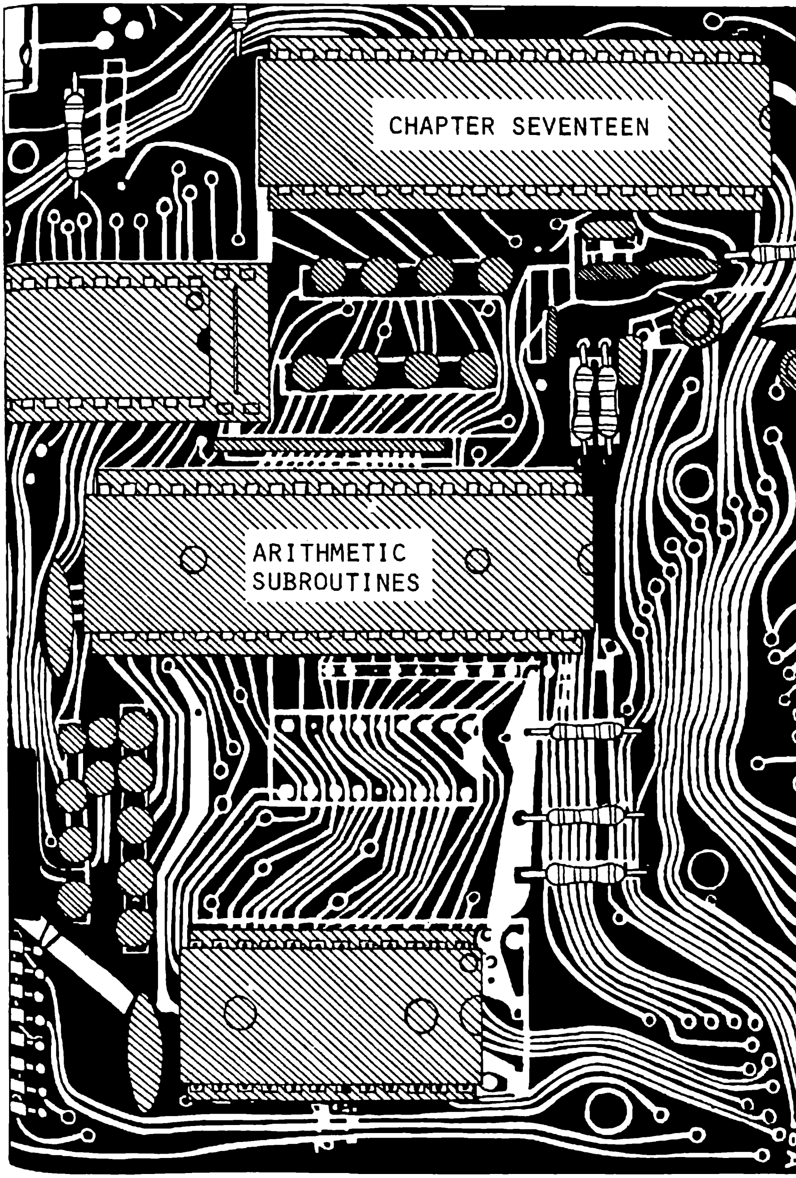
The entire disassembly program consists of initialising the variables CLASS and INDEX, assigning BC' (usually input by the human operator), finding the address HL' from tables, and then going into the master-routine. On exiting this it must then replace all V's, X's and Y's as defined earlier in this chapter, and then PRINT the result computed and go on to the next byte to be disassembled and treat it in the same way. The rest of the program consists of the eight subroutines, the table of addresses, and the data required for disassembly. The whole of this will occupy rather less than 1K.

However simple, or difficult, I may have made this program sound, you will undoubtedly find writing it a challenge. The vast majority of the program is data, and each address in every table must point to exactly the right byte. If you get any of it wrong it will be very difficult to trace.

You can improve the program too. I haven't used bit 6 of the data - you may be able to think of a use for it, for example it could indicate that a comma needs to be inserted, the choice is yours.

Like draughts, this program is so vast that even though the machine code listing itself will fit into 1K, you will need more than 1K in order for the machine code to be put there. Any editing program, BASIC or machine code, will take you above the 1K.

Good luck.



CHAPTER SEVENTEEN

ARITHMETIC SUBROUTINES

ARITHMETIC SUBROUTINES

This chapter is divided into two sections - one for the OLD, and one for the NEW ROM. We'll tackle the OLD ROM first because it's easier.

Numbers are represented in two bytes, and as such is it possible to store them in register pairs BC, DE, and HL. First of all we shall take a look at the five major arithmetic routines.

1). Addition. The address to call is OD3E, or more intellegibly, CALL ADD. The subroutine adds together the number stored in DE and the number stored in HL. The result is then placed in HL. This may be demonstrated by the following program:

```
113900    ADDDEMO   LD DE,0039
211100              LD HL,0011
CD3EOD              CALL ADD
C9                  RET
```

Here DE is loaded with the number fifty-seven, and HL with seventeen. On return to BASIC the result stored in HL should be fifty-seven plus seventeen, so the command PRINT USR(adddemo) should generate the number seventy-four.

2). Subtraction. Just the same - DE is subtracted from HL and the result stored in HL. The address is OD39. Thus to prove it:

```
113900    SUBDEMO   LD DE,0039
211100              LD HL,0011
CD390D              CALL SUB
C9                  RET
```

3). Multiplication. Up until now we have ignored multiplication completely, since there is no simple instruction which will multiply two numbers together. However, thanks to Uncle C, the ROM will do it for us. Simply CALL MULT, which is stored at address OD44, and as if by magic DE will be multiplied by HL, the result as usual being stored in HL. Watch out for what happens to BC and DE though! They're not unaltered.

4). Division. As you'd by now expect, the instruction CALL DIV will divide HL by DE (ignoring any remainder of course, since we are dealing in integers). The address of DIV is OD90

5). Powers. Is raising one number to the power of another going to be any more difficult? No of course not. With elegant simplicity the instruction CALL POWER (at ODOC) will do just that, raising HL to the power of DE, and putting the answer away in HL, using repeated multiplication to compute the answer.

One very important function is the RANDOM NUMBER GENERATOR. This is held at location OBED. To generate a random number between one and six, (say to simulate the roll of a die,) simply load HL with six and CALL RND. This is of course the same thing as RND(6). The number in the brackets should be placed in HL, and the final answer will end up in HL.

See if you can work out what this program does. What we're interested in is the number that it returns to BASIC.

```
211400    START     LD HL,0014
CDEDOB              CALL RND
110A00              LD DE,000A
CD440D              CALL MULT
116400              LD DE,0064
19                  ADD HL,DE
C9                  RET
```

Let's see if you got it right. HL is loaded with 14 and RND is called, so HL is replaced by a new value, RND(20). (Note that 14 (hex) is 20 (dec).) 0A is stored in DE, and the two are then multiplied together. We then have 10*RND(20). Finally 64 (hex) is added, giving 10*RND(20)+100.

We could use this routine in a games program. Suppose we needed to jump to a random destination. We could use the by now famous Tim Hartnell method of GOTO 10*RND(20)+100. Alternatively, if the above machine code were in a REM statement, say at address 16427, we could instead simply say GOTO USR(16427). This would do exactly the same job, except just a little bit faster.

We'll leave the OLD ROM now, and turn to the rather more complex field of arithmetic on the NEW ROM.

NEW ROM ARITHMETIC

The first and most important point to note is that NEW ROM numbers are stored as five bytes, not two, and so they can't fit into a register-pair as they stand. Nor are the numbers in simple form, for while it is true that zero is, as you'd expect, 00 00 00 00 00, it is not true that one is 00 00 00 00 01! In fact one is represented by 81 00 00 00 00. Here is a list of the Sinclair representation of the first few integers.

Decimal	Sinclair Form
0	00 00 00 00 00
1	81 00 00 00 00
2	82 00 00 00 00
3	82 40 00 00 00
4	83 00 00 00 00
5	83 20 00 00 00
6	83 40 00 00 00
7	83 60 00 00 00
8	84 00 00 00 00
9	84 10 00 00 00
10	84 20 00 00 00

...

and so on. There is a kind of pattern, but it's not instantly recognisable. Take a look at the negative numbers:

Decimal	Sinclair Form
-1	81 80 00 00 00
-2	82 80 00 00 00
-3	82 C0 00 00 00
-4	83 80 00 00 00
-5	83 A0 00 00 00
-6	83 C0 00 00 00
-7	83 E0 00 00 00
-8	84 80 00 00 00
-9	84 90 00 00 00
-10	84 A0 00 00 00

As you can see, you can instantly change a number from positive to negative just by adding 80 to the second byte. This doesn't apply to zero by the way - zero is represented uniquely to help speed the ROM up a little.

Knowing how the Sinclair Form is built up will slightly help your understanding of the ROM, so I will give here a brief explanation of how to turn decimal numbers into Sinclair numbers. It only takes a few simple steps.

STEP ONE: If the number is zero, then its Sinclair representation is 00 00 00 00 00.

STEP TWO: Ignoring the sign of the number, write it in binary (but without any leading zeroes). For example:

7	111
-10	1010
-4.25	100.01
0.325	0.011

Notice that the binary form has a BINARY point, not a DECIMAL point! 100.01 means one 4 plus no 2's plus no 1's (here we reach the binary point) plus no halves plus one quarter. The next column would have been an eighth.

STEP THREE is to work out a quantity called the EXPONENT. This is done as follows: If the part of the number to the left of the binary point is not zero then the exponent is the number of digits to the left of the point. If the part of the number to the left of the point is zero and the first digit after the point is one then the exponent is zero. If the part of the number to the left of the point is zero and the first digit after the point is zero, then count the number of zeroes between the point and the first 1 - the exponent is minus this number. The first byte of the Sinclair representation is 80 plus this exponent.

Decimal	Binary	Exponent	First byte of Sinclair Form
7	111	3	83
-10	1010	4	84
- 4.25	100.01	3	83
0.325	0.011	-1	7F

STEP FOUR: Now we can ignore the binary point altogether - that is what the exponent is for - to tell the computer where the point is supposed to go. So ignoring the point, write the binary form starting with the first 1 and then add sufficient zeroes to the right to make the whole thing thirty-two binary digits (bits) in length.

7	1110 0000 0000 0000 0000 0000 0000 0000
-10	1010 0000 0000 0000 0000 0000 0000 0000
-4.25	1000 1000 0000 0000 0000 0000 0000 0000
0.325	1100 0000 0000 0000 0000 0000 0000 0000

STEP FIVE. It is here that we remember the sign of the original number. If the original number was negative then do nothing. If the original number was positive then replace the first one by a zero. Thus:

7	0110 0000 0000 0000 0000 0000 0000 0000
-10	1010 0000 0000 0000 0000 0000 0000 0000
-4.25	1000 1000 0000 0000 0000 0000 0000 0000
0.325	0100 0000 0000 0000 0000 0000 0000 0000

STEP SIX - Now just change these numbers straight into hex, like so, making sure you remember to put the exponent byte at the start:

7	83 60 00 00 00
-10	84 A0 00 00 00
-4.25	83 88 00 00 00
0.325	7F 40 00 00 00

This is the form in which the ROM will be working. The largest exponent you may have is FF, so the largest positive number that can be stored is FF 7F FF FF FF. This turns out to be 1.7014118E38. (If you can't understand the "E" notation the E means "with the decimal point shifted (in the above case) 38 places to the right" In other words the number 170,141,180,000,000,000,000,000,000,000,000,000,000, which is a pretty vast quantity. It can still only store ten decimal places accurately though. The smallest positive number you can have (apart from zero) is 01 00 00 00 00, which happens to represent 2.9387359E-39. To you and me that's 0.000,000,000,000,000,000,000,000,000,000,000,000,002,938,735,9 which I'd say was pretty microscopic.

You can check all of this with the following BASIC program.

```
10 LET A=0
20 LET B=PEEK 16400+256*PEEK 16401
30 FOR I=1 TO 5
40 INPUT A$
50 POKE B+I,16*CODE A$+CODE A$(2)-476
60 NEXT I
70 PRINT A
```

The program sets up a variable A, and then overwrites its previous contents by POKEing into the variables area, one byte at a time. (That's a letter I in line 50, not a number 1). If you run it and input "82"/"40"/"00"/"00"/"00" (where / means newline) you'll find the number three printed. And so on.

An interesting little quirk is that if you input "00"/"80"/"00"/"00"/"00" (in theory this is minus-zero) the machine actually prints -C.6E-56 ! The letter C in mid-number, and an exponent of -56! Don't panic! This doesn't really happen in the ROM. We made it happen by POKEing something that doesn't make sense. The ROM does behave slightly more sensibly than human beings.

HOW TO USE FLOATING POINT NUMBERS PROPERLY

Having seen that a five byte number is too big to store in the registers the next question is undoubtedly "Well where does it store them then?" Answer - it stores them in an area of RAM called the CALCULATOR STACK, which works very much like the ordinary stack except for two points. 1) It can store both floating point numbers and strings, and 2) it is the right way up, not upside down like the machine stack.

To push a number stored in the BC register onto the calculator stack all you need to do is call up a subroutine in the ROM. CALL STACKBC, as I've called it, will change BC into five byte form as described above and then push this number onto the top of the calculator stack. You can do the same for a number stored in A (ie a number between 0 and 255) by calling STACKA. The addresses to call are: 1519 (STACKA) and 151C (STACKBC)

```
CALL STACKA     CD1915
CALL STACKBC    CD1C15
```

Incidently the first two instructions in the STACKA routine are LD C,A and LD B,00. It then leaps straight into STACKBC!

Conversely, to retrieve a number from the calculator stack we can CALL UNSTACK (address 0EA7), which removes a number from the calculator stack and stores it in the BC register.

Arithmetic is quite straightforward. The addresses are:

ADD	1754	addition
SUB	174B	subtraction
MULT	17C5	multiplication
DIV	1881	division

They work like this: The five-byte number stored at an address specified by HL (this means the number is stored in locations (HL),(HL)+1,(HL)+2, (HL)+3,and (HL)+4) is added to, multiplied by, divided by, or has a second number subtracted from it. The second number is stored at an address specified by DE. After the calculation the result is stored in the five bytes beginning at address HL.

To multiply together the two numbers at the top of the calculator stack one method would be as follows:

```
2A1C40          LD HL,(STKEND)
11FBFF          LD DE,FFFB
19              ADD HL,DE
E5              PUSH HL
221C40          LD (STKEND),HL
19              ADD HL,DE
D1              POP DE
CDC517          CALL MULT
```

Can you follow exactly what is going on? HL is loaded with the contents of the system variable STKEND - which gives the address of the first byte after the end of the calculator stack. DE is loaded with minus five, thus HL is decreased by five. This new value is loaded back into STKEND because we start off with two items on the stack and want to end up with only one. This is the address of one of the numbers to be multiplied. If you follow the listing through carefully you'll see that DE ends up with this value. First though HL is decreased by five again, to find the start of the other number to be multiplied.

To check that it really does work, run this program.

```
3E06    START   LD A,06
CD1915          CALL STACKA
3E07            LD A,07
CD1915          CALL STACKA
2A1C40          LD HL,(STKEND)
11FBFF          LD DE,FFFB
19              ADD HL,DE
E5              PUSH HL
221C40          LD (STKEND),HL
19              ADD HL,DE
D1              POP DE
CDC517          CALL MULT
CDA70E          CALL UNSTACK
C9              RET
```

Run it by typing PRINT USR start. What do you get?

But surely there must be easier ways to multiply six by seven. After all, the above program does look very complicated, and not something you'd easily remember. Well it's here that we really do start making full use of the ROM. The following program does exactly the same job, and I shall shortly explain why:

```
3E06            LD A,06
CD1915          CALL STACKA
3E07            LD A,07
CD1915          CALL STACKA
EF              RST 28
04              DEFB 04
34              DEFB 34
CDA70E          CALL UNSTACK
C9              RET
```

In the NEW ROM, RST 28 means "perform floating point arithmetic." The data that follows tells it precisely what calculations it's supposed to do. The byte 04 means multiply - all of the shuffling around of the calculator stack is done automatically. The byte 34 is used after a RST 28 instruction to indicate that there is no more data to come, and the next machine code instruction should follow.

The RST 28 data codes are ADD: 0F, SUB: 03, MULT: 04, and DIV: 05. Don't forget you'll need a byte 34 as well though, to end the data.

You may be wondering what happens if the number on the top of the calculator stack is not an integer between 0 and 65535 (the maximum value any two byte register can hold). Well my first answer would be "Try it for yourself and find out." Write a program that adds 8001 to 8001. Write a program that divides eight by three, then a program that divides seven by three. Write a program that subtracts five from zero, and another that subtracts a thousand from zero. But for those of you who are impatient I'll tell you anyway.

If the number at the top of the calculator stack is greater than 65535 then attempting to "unstack" it into BC will result in the program returning to BASIC - returning to command mode in fact - stopping with error code B (which means out of range)

If the number is a decimal then it will be rounded up or down (not just INTed) to the nearest whole number. If the decimal part is less than 0.5 it will be rounded down, and if the decimal part is greater than or equal to 0.5 it will be rounded up.

If the number is negative then error B will result, causing an immediate return to BASIC and stopping the program, if there is one.

RST 28 allows you to do much, much more than just simple arithmetic. All of the functions of the ZX81 are available to you. The data code for any particular function is just the character code of that function minus AB. For instance, the character code of SIN is C7. C7 minus AB is 1C. (If you don't believe me we'll do it in decimal - 199 minus 171 is 28.) This means we can find the SIN of the number at the top of the calculator stack using the sequence:

```
EF        RST 28
1C        DEFB 1C (SIN)
34        DEFB 34 (Exit).
```

To multiply two numbers (at the top of the calculator stack) together and then find the square root of the result we can use the sequence

```
EF        RST 28
04        DEFB 04 (MULT)
25        DEFB 25 (SQR)
34        DEFB 34 (EXIT)
```

If you're not absolutely convinced yet, run this program, which multiplies five by twenty, and then takes the square root.

```
3E05      LD A,05
CD1915    CALL STACKA
3E14      LD A,14
CD1915    CALL STACKA
```

```
EF          RST 28
04          DEFB 04 (MULT)
25          DEFB 25 (SQR)
34          DEFB 34 (EXIT)
CDA70E      CALL UNSTACK
C9          RET
```

You'll notice that this is the first time we've used more than one code in the RST 28 data. In fact you can use as many as you like, provided you end the list with 34.

To save you working it out for yourselves here is a list of the available functions that we are ready to use, together with their appropriate RST 28 code:

FUNCTION	CODE	FUNCTION	CODE
CODE	19	EXP	23
VAL	1A	INT	24
LEN	1B	SQR	25
SIN	1C	SGN	26
COS	1D	ABS	27
TAN	1E	PEEK	28
ASN	1F	USR	29
ACS	20	STR$	2A
ATN	21	CHR$	2B
LN	22	NOT	2C

Some of the entries in that list may surprize you. For instance we have the use of USR. This is very confusing - being allowed to use USR in the middle of a USR routine - but it's not really. Here's how it works. You've worked your way through a lot of RST 28 data, done a lot of calculation, and now you come across the code 29. What happens next is that the number at the top of the stack should be an integer between 0 and 65535 - or else you'll get an error B. This address is treated as a subroutine and CALLed. This subroutine will run exactly as you'd expect it to. When it's over (ie when a RET instruction is reached) the machine will go back to interpreting the RST 28 data from the next byte. USR will of course leave a new value at the top of the stack - the value held by BC at the end of the subroutine.

PEEK works in the same way, finding an address, PEEKing there, then pushing the result to the calculator stack.

All of the functions when used in this way will remove the number currently at the top of the calculator stack and replace it by a new one. For instance If the number at the top of the stack is 3.5 and the function INT is called, the 3.5 will be removed and replaced by a new value, 3.

The string functions CODE, VAL, and LEN, also CHR$ and STR$ need a small amount of explaining. You see, as well as storing numbers, the calculator stack can also store strings, so if you start off with the number 2000 on the top of this stack, and you then call STR$ (By using code 2A in a RST 28 instruction) then the item at the top of the calculator stack will now be the string "2000". You can demonstrate this with the following:

```
01D007      LD BC,07D0
CD1C15      CALL STACKBC
EF          RST 28
2A          DEFB 2A (STR$)
19          DEFB 19 (CODE)
34          DEFB 34 (EXIT)
CDA70E      CALL UNSTACK
C9          RET
```

This should produce the result of CODE STR$ 2000. Does it?

USING THE CALCULATOR'S MEMORY

If you take a quick glance at the manual you'll see that one of the system variables, MEMBOT, is thirty bytes long. This is the calculator's memory area. A quick calculation involving dividing by five, if you're up to it, shows that this leaves enough room to store six different five byte numbers. The six different areas of memory may each be used by RST 28 to store, or retrieve, numbers (but numbers only) from the top of the calculator stack. There are twelve different codes to achieve this - these are

```
C0        stores number in memory location 0
C1        stores number in memory location 1
C2        stores number in memory location 2
C3        stores number in memory location 3
C4        stores number in memory location 4
C5        stores number in memory location 5

E0        recalls number from memory location 0
E1        recalls number from memory location 1
E2        recalls number from memory location 2
E3        recalls number from memory location 3
E4        recalls number from memory location 4
E5        recalls number from memory location 5
```

Storing a number copies it from the top of the stack, and recalling a number simply places it at the end of the stack - it doesn't overwrite the previous top item.

Let's see how we can use this. Suppose we want to find SIN X+COS X. We must use the following technique. Assume that X is at the top of the stack.

```
EF        RST 28
C0        DEFB C0 (STOREØ)
1C        DEFB 1C (SIN)
E0        DEFB E0 (RCALLØ)
1D        DEFB 1D (COS)
0F        DEFB 0F (ADD)
34        DEFB 34 (EXIT)
```

Note that the SIN routine changes X into SIN X. When we again recall X there are now two items on the stack: SIN X and X. The COS routine changes X into COS X, so that the two items on the stack, are now SIN X, and COS X. The ADD routine will replace both of these by one single number - the answer we want - SIN X plus COS X.

We have now performed a fairly complex trigonometric function in just eight bytes!

Let's see how we can remove a floating point number from the stack without restricting ourselves to integers less than 65536. The way the ROM does it is like this:

```
2A1C40    LD HL,(STKEND)
2B        DEC HL
46        LD B,(HL)
2B        DEC HL
4E        LD C,(HL)
2B        DEC HL
56        LD D,(HL)
```

```
2B              DEC HL
5E              LD E,(HL)
2B              DEC HL
7E              LD A,(HL)
221C40          LD (STKEND),HL
```

As you can probably see for yourself, a five byte number is removed from the stack and stored in the registers A, E, D, C, and B. (In that order.) You can CALL this routine from address 13F4.

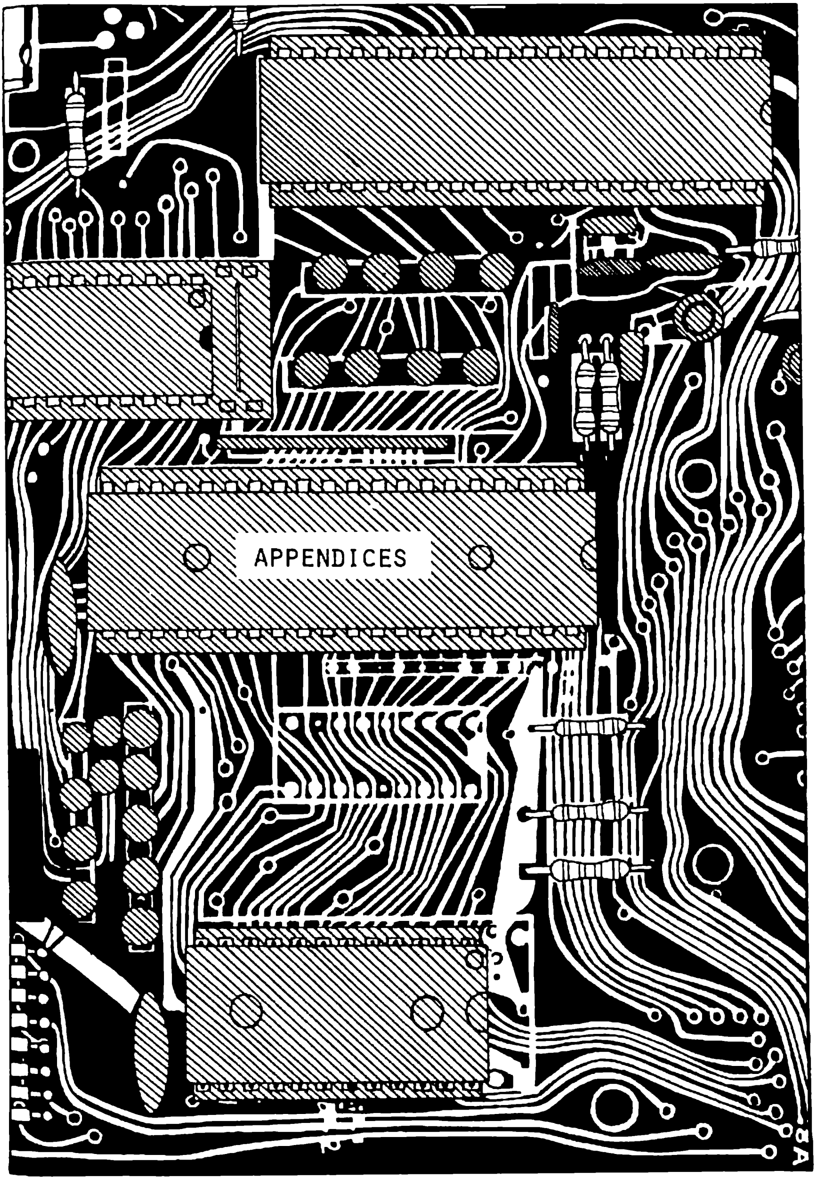
If the first item in the variable store is X then having popped SIN X plus COS X from the stack you can then store the result back in X as follows

```
2A1040          LD HL,(VARS)
23              INC HL
77              LD (HL),A
23              INC HL
73              LD (HL),E
23              INC HL
72              LD (HL),D
23              INC HL
71              LD (HL),C
23              INC HL
70              LD (HL),B
C9              RET
```

You can see that it takes more bytes to store the answer than it does to find it in the first place!

Let's see what else we can do with RST 28. We can use the logical functions AND and OR (that is BASIC AND and BASIC OR). Both of these are available from RST 28, having byte codes 08 and 07 respectively. Also you can SWOP the two numbers at the top of the stack. Code 01 will do this.

The following sequence will raise one number to the power of another. Can you see why? After RST 28: 01 22 04 23 34.


APPENDICES
8A

These appendices are designed to give you easy and quick reference to most of the things you'd want to look up.

A detailed list of the precise effect of each Z80 instruction may be found in chapter eight. This should be treated as a separate appendix.

The appendices are as follows:

APPENDIX ONE

HEXLD3

```
4082 F5
4083 E6
4084 F0
4085 1F 3
4086 1F 3
4087 1F 3
4088 1F 3
4089 C6
408A 1C 0
408B D7
408C F1
408D E6
408E 0F ?
408F C6
4090 1C 0
4091 D7
4092 C9
4093 82
4094 40
4095 95
4096 40
4097 78
4098 41
4099 78
409A 41
409B 2A E
409C 99
409D 40
409E 22 6
409F 97
40A0 40
40A1 54
40A2 5D
40A3 2A E
40A4 95
40A5 40
40A6 A7
40A7 ED
40A8 52
40A9 19 ;
40AA 30 K
40AB 1F 3
40AC 7C
40AD CD
40AE 82
40AF 40
40B0 7D
40B1 CD
40B2 82
40B3 40
40B4 AF
40B5 D7
40B6 7E
40B7 CD
40B8 82
40B9 40
40BA CB
40BB 78
40BC 20 4
40BD 04
40BE AF
40BF D7
40C0 7E
40C1 D7
40C2 3E Y
40C3 76
40C4 D7
40C5 23 7
40C6 22 6
40C7 95
40C8 40
40C9 18 /
40CA DB
40CB CF
40CC 14 =
40CD 2A E
40CE 10 (
40CF 40
40D0 23 7
40D1 46
40D2 23 7
40D3 CB
40D4 28 C
40D5 28 C
40D6 F4
40D7 ED
40D8 5B
40D9 95
40DA 40
40DB 23 7
40DC 7E
40DD 87
40DE 87
40DF 87
40E0 87
40E1 23 7
40E2 86
40E3 C6
40E4 24 8
40E5 12 >
40E6 13 <
40E7 ED
40E8 53
40E9 95
40EA 40
40EB E5
40EC 2A E
40ED 99
40EE 40
40EF ED
40F0 52
40F1 E1
40F2 30 K
40F3 04
40F4 ED
40F5 53
40F6 99
40F7 40
40F8 10 (
40F9 E1
40FA C9
40FB 2A E
40FC 99
40FD 40
40FE ED
40FF 5B
4100 93
4101 40
4102 A7
4103 ED
4104 52
4105 22 6
4106 97
4107 40
4108 44
4109 4D
410A C9
410B 2A E
410C 10 (
410D 40
410E 11 )
410F 06
4110 00
4111 19 ;
4112 E5
4113 2A E
4114 93
4115 40
4116 ED
4117 4B
4118 97
4119 40
411A ED
411B B0
411C C9
411D 2A E
411E 10 (
411F 40
4120 11 )
4121 06
4122 00
4123 19 ;
4124 ED
4125 5B
4126 93
4127 40
4128 ED
4129 4B
412A 97
412B 40
412C ED
412D B8
412E C9
412F 2A E
4130 10 (
4131 40
4132 23 7
4133 4E
4134 23 7
4135 46
4136 CB
4137 28 C
4138 CB
4139 19 ;
413A 20 4
413B 02
413C CF
413D 15 +
413E C5
413F 2A E
4140 99
4141 40
4142 ED
4143 5B
4144 95
4145 40
4146 A7
4147 ED
4148 52
4149 23 7
414A 44
414B 4D
414C E1
414D ED
414E 5B
414F 99
4150 40
4151 19 ;
4152 22 6
4153 99
4154 40
4155 EB
4156 ED
4157 B8
4158 CD
4159 CD
415A 40
415B C9
415C 2A E
415D 99
415E 40
415F ED
4160 5B
4161 97
4162 40
4163 D5
4164 A7
4165 ED
4166 52
4167 44
4168 4D
4169 E1
416A 23 7
416B ED
416C 5B
416D 95
416E 40
416F ED
4170 B0
4171 1B .
4172 ED
4173 53
4174 99
4175 40
4176 CF
4177 12 >
```

```
 10 PRINT " LIST "
 20 GOSUB 600
 30 RUN USR 16539
100 PRINT "WRITE"
110 GOSUB 600
120 INPUT A$
130 PRINT A$;"  ";
140 RAND USR 16589
150 GOTO 120
200 PRINT "INSERT"
210 GOSUB 600
220 INPUT A$
230 PRINT A$;"  ";
240 RAND USR 16667
250 GOTO 220
300 PRINT "DELETE"
310 GOSUB 600
320 LET A=16535
330 GOSUB 610
340 RUN USR 16732
400 DIM O$(USR 16635)
410 RAND USR 16651
420 SAVE "HEXLD3"
500 RAND USR 16669
510 CLEAR
520 STOP
600 LET A=16533
610 PRINT "ADDRESS ";
620 INPUT A$
630 PRINT A$
640 POKE A+1,16*CODE A$+CODE A$
(2)-476
650 POKE A,16*CODE A$(3)+CODE A
$(4)-476
660 CLEAR
670 RETURN
```

APPENDIX TWO

OLD ROM SYSTEM VARIABLES:

Decimal	Hex	Name
16384	4000	ERR.NR
16385	4001	FLAGS
16386	4002	PPC
16388	4004	E.ADDR
16390	4006	E.PPC
16392	4008	VARS
16394	400A	E.LINE
16396	400C	D.FILE
16398	400E	DF.EA
16400	4010	DF.END
16402	4012	DF.SZ
16403	4013	S.TOP
16405	4015	X.PTR
16407	4017	OLDPPC
16409	4019	FLAGX
16410	401A	T.ADDR
16412	401C	SEED
16414	401E	FRAMES
16416	4020	V.ADDR
16418	4022	ACC
16420	4024	S.POSN
16422	4026	CH.ADD

NEW ROM SYSTEM VARIABLES:

Decimal	Hex	Name
16384	4000	ERR.NR
16385	4001	FLAGS
16386	4002	ERR.SP
16388	4004	RAMTOP
16390	4006	MODE
16391	4007	PPC
16393	4009	VERSN
16394	400A	E.PPC
16396	400C	D.FILE
16398	400E	DF.CC
16400	4010	VARS
16402	4012	DEST
16404	4014	E.LINE
16406	4016	CH.ADD
16408	4018	X.PTR
16410	401A	STKBOT
16412	401C	STKEND
16414	401E	BERG
16415	401F	MEM
16417	4021	SPARE1
16418	4022	DF.SZ
16419	4023	S.TOP
16421	4025	LAST.K
16423	4027	DB.ST
16424	4028	MARGIN
16425	4029	NXTLIN
16427	402B	OLDPPC
16429	402D	FLAGX
16430	402E	STRLEN
16432	4030	T.ADDR
16434	4032	SEED
16436	4034	FRAMES
16438	4036	COORDS
16440	4038	PR.CC
16441	4039	S.POSN
16443	403B	CDFLAG
16444	403C	PRBUFF
16477	405D	MEMBOT
16507	407B	SPARE2

NEW ROM MEMORY ORGANISATION

4000	system variables
407D	program
(D.FILE)	screen
(VARS)	variables
(E.LINE)	edit line
(STKBOT)	calculator stack
(STKEND)	spare
SP	machine stack
(ERR.SP)	GOSUB stack
(RAMTOP)	reserved area

OLD ROM MEMORY ORGANISATION

4000	system variables
4028	program
(VARS)	variables
(E.LINE)	edit line
(D.FILE)	screen
(DF.END)	spare
SP	machine stack

SYSTEM VAR.	OLD ROM ADDR	NEW ROM ADDR	NO. OF BYTES	PURPOSE
ACC	4022	-	2	Value of last expression
BERG	-	401E	1	Used by floating point calculator
CDFLAG	-	403B	1	Flags relating to FAST/SLOW mode
CH.ADD	4026	4016	2	Address of the next character to interpret
COORDS	-	4036	2	Coordinates of last point PLOTted
D.FILE	400C	400C	2	Address of start of display file
DB.ST	-	4027	1	Debounce status of keyboard
DEST	-	4012	2	Address of variable being assigned
DF.CC	-	400E	2	Address of print position within display file
DF.EA	400E	-	2	Address of start of lower part of screen
DF.END	4010	-	2	Address of end of display file
DF.SZ	4012	4022	2	Number of lines in lower part of screen
E.ADDR	4004	-	2	Address of cursor in edit line
E.LINE	400A	4014	2	Address of start of edit line
E.PPC	4006	400A	2	Line number of line with cursor
ERR.NR	4000	4000	1	Current report code minus one
ERR.SP	-	4002	2	Address of top of GOSUB stack
FLAGS	4001	4001	1	Various flags
FLAGX	4019	402D	1	Various flags
FRAMES	401E	4034	2	Updated once for every TV frame displayed
LAST.K	-	4025	2	Keyboard scan taken after the last TV frame
MARGIN	-	4028	1	Number of blank lines above or below picture
MEM	-	401F	2	Address of start of calculators memory area
MEMBOT	-	405D	1E	Area which may be used for calculator memory
MODE	-	4006	1	Current cursor mode
NXTLIN	-	4029	2	Address of next program line to be executed
OLDPPC	4017	402B	2	Line number to which CONT/CONTINUE jumps
PPC	4002	4007	2	Line number of line being executed
PR.CC	-	4038	1	Address of LPRINT position (High part assumed 40)
PRBUFF	-	403C	21h	Buffer to store LPRINT output
RAMTOP	-	4004	2	Address of reserved area (not wiped out by NEW)
S.POSN	4024	4039	2	Coordinates of print position
S.TOP	4013	4023	2	Line number of line at top of screen
SEED	401C	4032	2	Seed for random number generator
SPARE1	-	4021	1	One spare byte
SPARE2	-	407B	2	Two spare bytes
STKBOT	-	401A	2	Address of calculator stack
STKEND	-	401C	2	Address of end of calculator stack
STRLEN	-	402E	2	Information concerning assigning of strings
T.ADDR	401A	4030	2	Address of next item in syntax table
V.ADDR	4020	-	2	Address of variable name to be assigned
VARS	4008	4010	2	Address of start of variables area
VERSN	-	4009	1	First system variable to be SAVEd
X.PTR	4015	4018	2	Address of character preceeding syntax error marker

APPENDIX THREE

ORDINARY	0	1	2	3
0	NOP	LD BC,mn	LD (BC),A	INC BC
1	DJNZ e	LD DE,mn	LD (DE),A	INC DE
2	JR NZ,e	LD HL,mn	LD (pq),HL	INC HL
3	JR NC,e	LD SP,mn	LD (pq),A	INC SP
4	LD B,B	LD B,C	LD B,D	LD B,E
5	LD D,B	LD D,C	LD D,D	LD D,E
6	LD H,B	LD H,C	LD H,D	LD H,E
7	LD (HL),B	LD (HL),C	LD (HL),D	LD (HL),E
8	ADD A,B	ADD A,C	ADD A,D	ADD A,E
9	SUB B	SUB C	SUB D	SUB E
A	AND B	AND C	AND D	AND E
B	OR B	OR C	OR D	OR E
C	RET NZ	POP BC	JP NZ,pq	JP pq
D	RET NC	POP DE	JP NC,pq	OUT (n),A
E	RET PO	POP HL	JP PO,pq	EX (SP),HL
F	RET P	POP AF	JP P,pq	DI

	4	5	6	7
0	INC B	DEC B	LD B,n	RLCA
1	INC D	DEC D	LD D,n	RLA
2	INC H	DEC H	LD H,n	DAA
3	INC (HL)	DEC (HL)	LD (HL),n	SCF
4	LD B,H	LD B,L	LD B,(HL)	LD B,A
5	LD D,H	LD D,L	LD D,(HL)	LD D,A
6	LD H,H	LD H,L	LD H,(HL)	LD H,A
7	LD (HL),H	LD (HL),L	HALT	LD (HL),A
8	ADD A,H	ADD A,L	ADD A,(HL)	ADD A,A
9	SUB H	SUB L	SUB (HL)	SUB A
A	AND H	AND L	AND (HL)	AND A
B	OR H	OR L	OR (HL)	OR A
C	CALL NZ,pq	PUSH BC	ADD A,n	RST 00
D	CALL NC,pq	PUSH DE	SUB n	RST 10
E	CALL PO,pq	PUSH HL	AND n	RST 20
F	CALL P,pq	PUSH AF	OR n	RST 30

AFTER CB

	0	1	2	3	4	5	6	7
0	RLC B	RLC C	RLC D	RLC E	RLC H	RLC L	RLC (HL)	RLC A
1	RL B	RL C	RL D	RL E	RL H	RL L	RL (HL)	RL A
2	SLA B	SLA C	SLA D	SLA E	SLA H	SLA L	SLA (HL)	SLA A
3	-	-	-	-	-	-	-	-
4	BIT 0,B	BIT 0,C	BIT 0,D	BIT 0,E	BIT 0,H	BIT 0,L	BIT 0,(HL)	BIT 0,A
5	BIT 2,B	BIT 2,C	BIT 2,D	BIT 2,E	BIT 2,H	BIT 2,L	BIT 2,(HL)	BIT 2,A
6	BIT 4,B	BIT 4,C	BIT 4,D	BIT 4,E	BIT 4,H	BIT 4,L	BIT 4,(HL)	BIT 4,A
7	BIT 6,B	BIT 6,C	BIT 6,D	BIT 6,E	BIT 6,H	BIT 6,L	BIT 6,(HL)	BIT 6,A
8	RES 0,B	RES 0,C	RES 0,D	RES 0,E	RES 0,H	RES 0,L	RES 0,(HL)	RES 0,A
9	RES 2,B	RES 2,C	RES 2,D	RES 2,E	RES 2,H	RES 2,L	RES 2,(HL)	RES 2,A
A	RES 4,B	RES 4,C	RES 4,D	RES 4,E	RES 4,H	RES 4,L	RES 4,(HL)	RES 4,A
B	RES 6,B	RES 6,C	RES 6,D	RES 6,E	RES 6,H	RES 6,L	RES 6,(HL)	RES 6,A
C	SET 0,B	SET 0,C	SET 0,D	SET 0,E	SET 0,H	SET 0,L	SET 0,(HL)	SET 0,A
D	SET 2,B	SET 2,C	SET 2,D	SET 2,E	SET 2,H	SET 2,L	SET 2,(HL)	SET 2,A
E	SET 4,B	SET 4,C	SET 4,D	SET 4,E	SET 4,H	SET 4,L	SET 4,(HL)	SET 4,A
F	SET 6,B	SET 6,C	SET 6,D	SET 6,E	SET 6,H	SET 6,L	SET 6,(HL)	SET 6,A

ORDINARY	8	9	A	B
0	EX AF,AF'	ADD HL,BC	LD A,(BC)	DEC BC
1	JR e	ADD HL,DE	LD A,(DE)	DEC DE
2	JR Z,e	ADD HL,HL	LD HL,(pq)	DEC HL
3	JR C,e	ADD HL,SP	LD A,(pq)	DEC SP
4	LD C,B	LD C,C	LD C,D	LD C,E
5	LD E,B	LD E,C	LD E,D	LD E,E
6	LD L,B	LD L,C	LD L,D	LD L,E
7	LD A,B	LD A,C	LD A,D	LD A,E
8	ADC A,B	ADC A,C	ADC A,D	ADC A,E
9	SBC A,B	SBC A,C	SBC A,D	SBC A,E
A	XOR B	XOR C	XOR D	XOR E
B	CP B	CP C	CP D	CP E
C	RET Z	RET	JP Z,pq	●
D	RET C	EXX	JP C,pq	IN A,(n)
E	RET PE	JP (HL)	JP PE,pq	EX DE,HL
F	RET M	LD SP,HL	JP M,pq	EI

	C	D	E	F
0	INC C	DEC C	LD C,n	RRCA
1	INC E	DEC E	LD E,n	RRA
2	INC L	DEC L	LD L,n	CPL
3	INC A	DEC A	LD A,n	CCF
4	LD C,H	LD C,L	LD C,(HL)	LD C,A
5	LD E,H	LD E,L	LD E,(HL)	LD E,A
6	LD L,H	LD L,L	LD L,(HL)	LD L,A
7	LD A,H	LD A,L	LD A,(HL)	LD A,A
8	ADC A,H	ADC A,L	ADC A,(HL)	ADC A,A
9	SBC A,H	SBC A,L	SBC A,(HL)	SBC A,A
A	XOR H	XOR L	XOR (HL)	XOR A
B	CP H	CP L	CP (HL)	CP A
C	CALL Z,pq	CALL pq	ADC A,n	RST 08
D	CALL C,pq	●	SBC A,n	RST 18
E	CALL PE,pq	●	XOR n	RST 28
F	CALL M,pq	●	CP n	RST 38

AFTER CB

	8	9	A	B	C	D	E	F
0	RRC B	RRC C	RRC D	RRC E	RRC H	RRC L	RRC (HL)	RRC A
1	RR B	RR C	RR D	RR E	RR H	RR L	RR (HL)	RR A
2	SRA B	SRA C	SRA D	SRA E	SRA H	SRA L	SRA (HL)	SRA A
3	SRL B	SRL C	SRL D	SRL E	SRL H	SRL L	SRL (HL)	SRL A
4	BIT 1,B	BIT 1,C	BIT 1,D	BIT 1,E	BIT 1,H	BIT 1,L	BIT 1,(HL)	BIT 1,A
5	BIT 3,B	BIT 3,C	BIT 3,D	BIT 3,E	BIT 3,H	BIT 3,L	BIT 3,(HL)	BIT 3,A
6	BIT 5,B	BIT 5,C	BIT 5,D	BIT 5,E	BIT 5,H	BIT 5,L	BIT 5,(HL)	BIT 5,A
7	BIT 7,B	BIT 7,C	BIT 7,D	BIT 7,E	BIT 7,H	BIT 7,L	BIT 7,(HL)	BIT 7,A
8	RES 1,B	RES 1,C	RES 1,D	RES 1,E	RES 1,H	RES 1,L	RES 1,(HL)	RES 1,A
9	RES 3,B	RES 3,C	RES 3,D	RES 3,E	RES 3,H	RES 3,L	RES 3,(HL)	RES 3,A
A	RES 5,B	RES 5,C	RES 5,D	RES 5,E	RES 5,H	RES 5,L	RES 5,(HL)	RES 5,A
B	RES 7,B	RES 7,C	RES 7,D	RES 7,E	RES 7,H	RES 7,L	RES 7,(HL)	RES 7,A
C	SET 1,B	SET 1,C	SET 1,D	SET 1,E	SET 1,H	SET 1,L	SET 1,(HL)	SET 1,A
D	SET 3,B	SET 3,C	SET 3,D	SET 3,E	SET 3,H	SET 3,L	SET 3,(HL)	SET 3,A
E	SET 5,B	SET 5,C	SET 5,D	SET 5,E	SET 5,H	SET 5,L	SET 5,(HL)	SET 5,A
F	SET 7,B	SET 7,C	SET 7,D	SET 7,E	SET 7,H	SET 7,L	SET 7,(HL)	SET 7,A

AFTER DD

	0	1	2	3	4	5
0	-	-	-	-	-	-
1	-	-	-	-	-	-
2	-	LD IX,mn	LD (pq),IX	INC IX	-	-
3	-	-	-	-	INC (IX+d)	DEC (IX+
4	-	-	-	-	-	-
5	-	-	-	-	-	-
6	-	-	-	-	-	-
7	LD (IX+d),B	LD (IX+d),C	LD (IX+d),D	LD (IX+d),E	LD (IX+d),H	LD (IX+d
8	-	-	-	-	-	-
9	-	-	-	-	-	-
A	-	-	-	-	-	-
B	-	-	-	-	-	-
C	-	-	-	-	-	-
D	-	-	-	-	-	-
E	-	POP IX	-	EX (SP).IX	-	PUSH IX
F	-	-	-	-	-	-

AFTER DD

	6	7	8	9	A	B	C	D	E
0	-	-	-	ADD IX,BC	-	-	-	-	-
1	-	-	-	ADD IX,DE	-	-	-	-	-
2	-	-	-	ADD IX,IX	LD IX,(pq)	DEC IX	-	-	-
3	LD (IX+d),n	-	-	ADD IX,SP	-	-	-	-	-
4	LD B,(IX+d)	-	-	-	-	-	-	-	LD C,(IX+d)
5	LD D,(IX+d)	-	-	-	-	-	-	-	LD E,(IX+d)
6	LD H,(IX+d)	-	-	-	-	-	-	-	LD L,(IX+d)
7	-	LD (IX+d),A	-	-	-	-	-	-	LD A,(IX+d)
8	ADD A,(IX+d)	-	-	-	-	-	-	-	ADC A,(IX+d)
9	SUB (IX+d)	-	-	-	-	-	-	-	SBC (IX+d)
A	AND (IX+d)	-	-	-	-	-	-	-	XOR (IX+d)
B	OR (IX+d)	-	-	-	-	-	-	-	CP (IX+d)
C	-	-	-	-	-	●	-	-	-
D	-	-	-	-	-	-	-	-	-
E	-	-	-	JP (IX)	-	EX DE,IX	-	-	-
F	-	-	-	LD SP,IX	-	-	-	-	-

AFTER ED

	8	9	A	B	C	D	E	F
0	-	-	-	-	-	-	-	-
1	-	-	-	-	-	-	-	-
2	-	-	-	-	-	-	-	-
3	-	-	-	-	-	-	-	-
4	IN C,(C)	OUT (C),C	ADC HL,BC	LD BC,(pq)	-	RETI	-	LD R,A
5	IN E,(C)	OUT (C),E	ADC HL,DE	LD DE,(pq)	-	-	IM 2	LD A,R
6	IN L,(C)	OUT (C),L	ADC HL,HL	-	-	-	-	RLD
7	IN A,(C)	OUT (C),A	ADC HL,SP	LD SP,(pq)	-	-	-	-
8	-	-	-	-	-	-	-	-
9	-	-	-	-	-	-	-	-
A	LDD	CPD	IND	OUTD	-	-	-	-
B	LDDR	CPDR	INDR	OTDR	-	-	-	-
C	-	-	-	-	-	-	-	-
D	-	-	-	-	-	-	-	-
E	-	-	-	-	-	-	-	-
F	-	-	-	-	-	-	-	-

AFTER FD

	0	1	2	3	4	5
0	-	-	-	-	-	-
1	-	-	-	-	-	-
2	-	LD IY,mn	LD (pq),IY	INC IY	-	-
3	-	-	-	-	INC (IY+d)	DEC (IY+d)
4	-	-	-	-	-	-
5	-	-	-	-	-	-
6	-	-	-	-	-	-
7	LD (IY+d),B	LD (IY+d),C	LD (IY+d),D	LD (IY+d),E	LD (IY+d),H	LD (IY+d),L
8	-	-	-	-	-	-
9	-	-	-	-	-	-
A	-	-	-	-	-	-
B	-	-	-	-	-	-
C	-	-	-	-	-	-
D	-	-	-	-	-	-
E	-	POP IY	-	EX (SP),IY	-	PUSH IY

AFTER FD

	6	7	8	9	A	B	C	D	E	F
0	-	-	-	ADD IY,BC	-	-	-	-	-	-
1	-	-	-	ADD IY,DE	-	-	-	-	-	-
2	-	-	-	ADD IY,IY	LD IY,(pq)	DEC IY	-	-	-	-
3	LD (IY+d),n	-	-	ADD IY,SP	-	-	-	-	-	-
4	LD B,(IY+d)	-	-	-	-	-	-	-	LD C,(IY+d)	-
5	LD D,(IY+d)	-	-	-	-	-	-	-	LD E,(IY+d)	-
6	LD H,(IY+d)	-	-	-	-	-	-	-	LD L,(IY+d)	-
7	-	LD (IY+d),A	-	-	-	-	-	-	LD A,(IY+d)	-
8	ADD A,(IY+d)	-	-	-	-	-	-	-	ADC A,(IY+d)	-
9	SUB (IY+d)	-	-	-	-	-	-	-	SBC (IY+d)	-
A	AND (IY+d)	-	-	-	-	-	-	-	XOR (IY+d)	-
B	OR (IY+d)	-	-	-	-	-	-	-	CP (IY+d)	-
C	-	-	-	-	-	●	-	-	-	-
D	-	-	-	-	-	-	-	-	-	-
E	-	-	-	JP (IY)	-	EX DE,IY	-	-	-	-
F	-	-	-	LD SP,IY	-	-	-	-	-	-

	0	1	2	3	4	5	6	7
0	-	-	-	-	-	-	-	-
1	-	-	-	-	-	-	-	-
2	-	-	-	-	-	-	-	-
3	-	-	-	-	-	-	-	-
4	IN B,(C)	OUT (C),B	SBC HL,BC	LD (pq),BC	NEG	RETN	IM 0	LD I,A
5	IN D,(C)	OUT (C),D	SBC HL,DE	LD (pq),DE	-	-	IM 1	LD A,I
6	IN H,(C)	OUT (C),H	SBC HL,HL	-	-	-	-	RRD
7	-	-	SBC HL,SP	-	-	-	-	-
8	-	-	-	-	-	-	-	-
9	-	-	-	-	-	-	-	-
A	LDI	CPI	INI	OUTI	-	-	-	-
B	LDIR	CPIR	INIR	OTIR	-	-	-	-
C	-	-	-	-	-	-	-	-
D	-	-	-	-	-	-	-	-
E	-	-	-	-	-	-	-	-
F	-	-	-	-	-	-	-	-

	AFTER DDCBdd		AFTER FDCBdd	
	6	E	6	E
0	RLC (IX+d)	RRC (IX+d)	RLC (IY+d)	RRC (IY+d)
1	RL (IX+d)	RR (IX+d)	RL (IY+d)	RR (IY+d)
2	SLA (IX+d)	SRA (IX+d)	SLA (IY+d)	SRA (IY+d)
3	-	SRL (IX+d)	-	SRL (IY+d)
4	BIT 0,(IX+d)	BIT 1,(IX+d)	BIT 0,(IY+d)	BIT 1,(IY+d)
5	BIT 2,(IX+d)	BIT 3,(IX+d)	BIT 2,(IY+d)	BIT 3,(IY+d)
6	BIT 4,(IX+d)	BIT 5,(IX+d)	BIT 4,(IY+d)	BIT 5,(IY+d)
7	BIT 6,(IX+d)	BIT 7,(IX+d)	BIT 6,(IY+d)	BIT 7,(IY+d)
8	RES 0,(IX+d)	RES 1,(IX+d)	RES 0,(IY+d)	RES 1,(IY+d)
9	RES 2,(IX+d)	RES 3,(IX+d)	RES 2,(IY+d)	RES 3,(IY+d)
A	RES 4,(IX+d)	RES 5,(IX+d)	RES 4,(IY+d)	RES 5,(IY+d)
B	RES 6,(IX+d)	RES 7,(IX+d)	RES 6,(IY+d)	RES 7,(IY+d)
C	SET 0,(IX+d)	SET 1,(IX+d)	SET 0,(IY+d)	SET 1,(IY+d)
D	SET 2,(IX+d)	SET 3,(IX+d)	SET 2,(IY+d)	SET 3,(IY+d)
E	SET 4,(IX+d)	SET 5,(IX+d)	SET 4,(IY+d)	SET 5,(IY+d)
F	SET 6,(IX+d)	SET 7,(IX+d)	SET 6,(IY+d)	SET 7,(IY+d+

APPENDIX FOUR

INSTRUCTIONS		FLAGS
Opcode	Hexcode	S Z - H - P N C
ADC A,r	table 1	@ @ - @ - @ 0 @
ADC HL,s	table 2	@ @ - @ - @ 0 @
ADD A,r	table 1	@ @ - @ - @ 0 @
ADD HL,s	table 2	- - - @ - - 0 @
ADD IX,s	table 2	- - - @ - - 0 @
ADD IY,s	table 2	- - - @ - - 0 @
AND r	table 1	@ @ - 1 - @ 0 0
BIT b,r	table 1	? @ - 1 - @ 0 0
CALL pq	CDqqpp	- - - - - - - - -
CALL c,pq	table 3	- - - - - - - - -
CCF	3F	- - - x - - 0 @
(the H flag becomes the previous value of the C flag)		
CP r	table 1	@ @ - @ - @ 1 @
CPI	EDA1	@ x - @ - x 1 -
CPD	EDA9	@ x - @ - x 1 -
CPIR	EDB1	@ x - @ - x 1 -
CPDR	EDB9	@ x - @ - x 1 -
(Z becomes 1 if BC becomes zero, P/V becomes 1 if A = (HL-1))		
CPL	2F	- - - 1 - - 1 -
DAA	27	@ @ - @ - @ - @
DEC r	table 1	@ @ - @ - @ 1 -
DEC s	table 2	- - - - - - - - -
DI	F3	- - - - - - - - -
DJNZ e	10ee	- - - - - - - - -
EI	FB	- - - - - - - - -
EX AF,AF'	08	- - - - - - - - -
EX DE,HL	EB	- - - - - - - - -
EX (SP),HL	E3	- - - - - - - - -
EX (SP),IX	DDE3	- - - - - - - - -
EX (SP),IY	FDE3	- - - - - - - - -
EXX	D9	- - - - - - - - -

INSTRUCTIONS		FLAGS
Opcode	Hexcode	S Z - H - P N C
HALT	76	- - - - - - - - -
IM 0	ED46	- - - - - - - - -
IM 1	ED56	- - - - - - - - -
IM 2	ED5E	- - - - - - - - -
INC r	table 1	@ @ - @ - @ 0 -
INC s	table 2	- - - - - - - - -
IN A,(n)	DBnn	- - - - - - - - -
IN r,(C)	table 1	@ @ - @ - @ 0 -
INI	EDA2	? x - ? - ? 1 -
IND	EDAA	? x - ? - ? 1 -
(Z becomes 1 if B becomes zero)		
INIR	EDB2	? 1 - ? - ? 1 -
INDR	EDBA	? 1 - ? - ? 1 -
JP pq	C3qqpp	- - - - - - - - -
JP c,pq	table 3	- - - - - - - - -
JP (HL)	E9	- - - - - - - - -
JP (IX)	DDE9	- - - - - - - - -
JP (IY)	FDE9	- - - - - - - - -
JR e	18ee	- - - - - - - - -
JR c,e	table 3	- - - - - - - - -
LD (BC),A	02	- - - - - - - - -
LD A,(BC)	0A	- - - - - - - - -
LD (DE),A	12	- - - - - - - - -
LD A,(DE)	1A	- - - - - - - - -
LD I,A	ED47	- - - - - - - - -
LD R,A	ED4F	- - - - - - - - -
LD A,I	ED57	@ @ - 0 - x 0 -
LD A,R	ED5F	@ @ - 0 - x 0 -
(P/V is set to interrupt storage flag)		
LD SP,HL	F9	- - - - - - - - -
LD SP,IX	DDF9	- - - - - - - - -
LD SP,IY	FDF9	- - - - - - - - -

INSTRUCTIONS		FLAGS
Opcode	Hexcode	S Z - H - P N C
LD r,r	table 1	- - - - - - - - -
LD s,mn	table 2	- - - - - - - - -
LD A,(pq)	3Aqqpp	- - - - - - - - -
LD s,(pq)	table 2	- - - - - - - - -
LD (pq),A	32qqpp	- - - - - - - - -
LD (pq),s	table 2	- - - - - - - - -
LDI	EDA0	- - - 0 - x 0 -
LDD	EDA8	- - - 0 - x 0 -
(P/V becomes 0 if BC becomes 0)		
LDIR	EDB0	- - - 0 - 0 0 -
LDDR	EDB8	- - - 0 - 0 0 -
NEG	ED44	@ @ - @ - @ 1 @
NOP	00	- - - - - - - - -
OR r	table 1	@ @ - 0 - @ 0 0
OUT (n),A	D3nn	- - - - - - - - -
OUT (C),r	table 1	- - - - - - - - -
OUTI	EDA3	? x - ? - ? 1 -
OUTD	EDAB	? x - ? - ? 1 -
(Z becomes 1 if B becomes zero)		
OTIR	EDB3	? 1 - ? - ? 1 -
OTDR	EDBB	? 1 - ? - ? 1 -
POP AF	F1	x x x x x x x x
(Flags are determined by the byte at the top of the stack)		
POP s	table 2	- - - - - - - - -
PUSH AF	F5	- - - - - - - - -
PUSH s	table 2	- - - - - - - - -
RES b,r	table 1	- - - - - - - - -
RET	C9	- - - - - - - - -
RET c	table 3	- - - - - - - - -
RETN	ED45	- - - - - - - - -
RETI	ED4D	- - - - - - - - -
RLA	17	- - - 0 - - 0 @
RL r	table 1	@ @ - 0 - @ 0 @
RLCA	07	

INSTRUCTIONS		FLAGS
Opcode	Hexcode	S Z - H P N C
RES b,r	table 1	- - - - - - - - -
RET	C9	- - - - - - - - -
RET c	table 3	- - - - - - - - -
RETN	ED45	- - - - - - - - -
RETI	ED4D	- - - - - - - - -
RLCA	07	- - - 0 - - 0 @
RRCA	0F	- - - 0 - - 0 @
RLA	17	- - - 0 - - 0 @
RRA	1F	- - - 0 - - 0 @
RLC r	table 1	@ @ - 0 - @ 0 @
RRC r	table 1	@ @ - 0 - @ 0 @
RL r	table 1	@ @ - 0 - @ 0 @
RR r	table 1	@ @ - 0 - @ 0 @
RRD	ED67	@ @ - 0 - @ 0 -
RLD	ED6F	@ @ - 0 - @ 0 -
RST 00	C7	- - - - - - - - -
RST 08	CF	- - - - - - - - -
RST 10	D7	- - - - - - - - -
RST 18	DF	- - - - - - - - -
RST 20	E7	- - - - - - - - -
RST 28	EF	- - - - - - - - -
RST 30	F7	- - - - - - - - -
RST 38	FF	- - - - - - - - -
SBC A,r	table 1	@ @ - @ - @ 1 @
SBC HL,s	table 2	@ @ - @ - @ 1 @
SCF	37	- - - 0 - - 0 1
SET b,r	table 1	- - - - - - - - -
SLA r	table 1	@ @ - 0 - @ 0 @
SRA r	table 1	@ @ - 0 - @ 0 @
SRL r	table 1	@ @ - 0 - @ 0 @
SUB r	table 1	@ @ - @ - @ 1 @
XOR r	table 1	@ @ - 0 - @ 0 0

TABLE ONE

r	B	C	D	E	H	L	(HL)	A	(IX+d)	(IY+d)	n
ADD A,r	80	81	82	83	84	85	86	87	DD86dd	FD86dd	C6nn
ADC A,r	88	89	8A	8B	8C	8D	8E	8F	DD8Edd	FD8Edd	CEnn
AND r	A0	A1	A2	A3	A4	A5	A6	A7	DDA6dd	FDA6dd	E6nn
BIT 0,r	CB40	CB41	CB42	CB43	CB44	CB45	CB46	CB47	DDCBdd46	FDCBdd46	-
BIT 1,r	CB48	CB49	CB4A	CB4B	CB4C	CB4D	CB4E	CB4F	DDCBdd4E	FDCBdd4E	-
BIT 2,r	CB50	CB51	CB52	CB53	CB54	CB55	CB56	CB57	DDCBdd56	FDCBdd56	-
BIT 3,r	CB58	CB59	CB5A	CB5B	CB5C	CB5D	CB5E	CB5F	DDCBdd5E	FDCBdd5E	-
BIT 4,r	CB60	CB61	CB62	CB63	CB64	CB65	CB66	CB67	DDCBdd66	FDCBdd66	-
BIT 5,r	CB68	CB69	CB6A	CB6B	CB6C	CB6D	CB6E	CB6F	DDCBdd6E	FDCBdd6E	-
BIT 6,r	CB70	CB71	CB72	CB73	CB74	CB75	CB76	CB77	DDCBdd76	FDCBdd76	-
BIT 7,r	CB78	CB79	CB7A	CB7B	CB7C	CB7D	CB7E	CB7F	DDCBdd7E	FDCBdd7E	-
CP r	B8	B9	BA	BB	BC	BD	BE	BF	DDBEdd	FDBEdd	FEnn
DEC r	05	0D	15	1D	25	2D	35	3D	DD35dd	FD35dd	-
IN r,(C)	ED40	ED48	ED50	ED58	ED60	ED68	-	ED78	-	-	-
INC r	04	0C	14	1C	24	2C	34	3C	DD34dd	FD34dd	-
LD B,r	40	41	42	43	44	45	46	47	DD46dd	FD46dd	06nn
LD C,r	48	49	4A	4B	4C	4D	4E	4F	DD4Edd	FD4Edd	0Enn
LD D,r	50	51	52	53	54	55	56	57	DD56dd	FD56dd	16nn
LD E,r	58	59	5A	5B	5C	5D	5E	5F	DD5Edd	FD5Edd	1Enn
LD H,r	60	61	62	63	64	65	66	67	DD66dd	FD66dd	26nn
LD L,r	68	69	6A	6B	6C	6D	6E	6F	DD6Edd	FD6Edd	2Enn
LD (HL),r	70	71	72	73	74	75	-	77	-	-	36nn
LD A,r	78	79	7A	7B	7C	7D	7E	7F	DD7Edd	FD7Edd	3Enn
LD (IX+d),r	DD70 dd	DD71 dd	DD72 dd	DD73 dd	DD74 dd	DD75 dd	-	DD77 dd	-	-	DD36 ddnn
LD (IY+d),r	FD70 dd	FD71 dd	FD72 dd	FD73 dd	FD74 dd	FD75 dd	-	FD77 dd	-	-	FD36 ddnn
OR r	B0	B1	B2	B3	B4	B5	B6	B7	DDB6dd	FDB6dd	F6nn
OUT (C),r	ED41	ED49	ED51	ED59	ED61	ED69	-	ED79	-	-	-
RL r											
RES 0,r	CB80	CB81	CB82	CB83	CB84	CB85	CB86	CB87	DDCBdd86	FDCBdd86	-
RES 1,r	CB88	CB89	CB8A	CB8B	CB8C	CB8D	CB8E	CB8F	DDCBdd8E	FDCBdd8E	-
RES 2,r	CB90	CB91	CB92	CB93	CB94	CB95	CB96	CB97	DDCBdd96	FDCBdd96	-
RES 3,r	CB98	CB99	CB9A	CB9B	CB9C	CB9D	CB9E	CB9F	DDCBdd9E	FDCBdd9E	-
RES 4,r	CBA0	CBA1	CBA2	CBA3	CBA4	CBA5	CBA6	CBA7	DDCBddA6	FDCBddA6	-
RES 5,r	CBA8	CBA9	CBAA	CBAB	CBAC	CBAD	CBAE	CBAF	DDCBddAE	FDCBddAE	-
RES 6,r	CBB0	CBB1	CBB2	CBB3	CBB4	CBB5	CBB6	CBB7	DDCBddB6	FDCBddB6	-
RES 7,r	CBB8	CBB9	CBBA	CBBB	CBBC	CBBD	CBBE	CBBF	DDCBddBE	FDCBddBE	-

r	B	C	D	E	H	L	(HL)	A	(IX+d)	(IY+d)	n
RLC r	CB00	CB01	CB02	CB03	CB04	CB05	CB06	CB07	DDCBdd06	FDCBdd06	-
RRC r	CB08	CB09	CB0A	CB0B	CB0C	CB0D	CB0E	CB0F	DDCBdd0E	FDCBdd0E	-
RL r	CB10	CB11	CB12	CB13	CB14	CB15	CB16	CB17	DDCBdd16	FDCBdd16	-
RR r	CB18	CB19	CB1A	CB1B	CB1C	CB1D	CB1E	CB1F	DDCBdd1E	FDCBdd1E	-
SET 0,r	CBC0	CBC1	CBC2	CBC3	CBC4	CBC5	CBC6	CBC7	DDCBddC6	FDCBddC6	-
SET 1,r	CBC8	CBC9	CBCA	CBCB	CBCC	CBCD	CBCE	CBCF	DDCBddCF	FDCBddCE	-
SET 2,r	CBD0	CBD1	CBD2	CBD3	CBD4	CBD5	CBD6	CBD7	DDCBddD6	FDCBddD6	-
SET 3,r	CBD8	CBD9	CBDA	CBDB	CBDC	CBDD	CBDE	CBDF	DDCBddDE	FDCBddDE	-
SET 4,r	CBE0	CBE1	CBE2	CBE3	CBE4	CBE5	CBE6	CBE7	DDCBddE6	FDCBddE6	-
SET 5,r	CBE8	CBE9	CBEA	CBEB	CBEC	CBED	CBEE	CBEF	DDCBddEE	FDCBddEE	-
SET 6,r	CBF0	CBF1	CBF2	CBF3	CBF4	CBF5	CBF6	CBF7	DDCBddF6	FDCBddF6	-
SET 7,r	CBF8	CBF9	CBFA	CBFB	CBFC	CBFD	CBFE	CBFF	DDCBddFE	FDCBddFE	-
SUB A,r	90	91	92	93	94	95	96	97	DD96dd	FD96dd	D6nn
SBC A,r	98	99	9A	9B	9C	9D	9E	9F	DD9Edd	FD9Edd	DEnn
SLA r	CB20	CB21	CB22	CB23	CB24	CB25	CB26	CB27	DDCBdd26	FDCBdd26	-
SRA r	CB28	CB29	CB2A	CB2B	CB2C	CB2D	CB2E	CB2F	DDCBdd2E	FDCBdd2E	-
SRL r	CB38	CB39	CB3A	CB3B	CB3C	CB3D	CB3E	CB3F	DDCBdd3E	FDCBdd3E	-
XOR r	A8	A9	AA	AB	AC	AD	AE	AF	DDAEdd	FDAEdd	EEnn

TABLE TWO

s	BC	DE	HL	SP	IX	IY
ADC HL,s	ED4A	ED5A	ED6A	ED7A	-	-
ADD HL,s	09	19	29	39	-	-
ADD IX,s	DD09	DD19	-	DD39	DD29	-
ADD IY,s	FD09	FD19	-	FD39	-	FD29
DEC s	0B	1B	2B	3B	DD2B	FD2B
INC s	03	13	23	33	DD23	FD23
LD s,mn	01nnmm	11nnmm	21nnmm	31nnmm	DD21nnmm	FD21nnmm
LD s,(pq)	ED4Bqqpp	ED5Bqqpp	2Aqqpp	ED7Bqqpp	DD2Aqqpp	FD2Aqqpp
LD (pq),s	ED43qqpp	ED53qqpp	22qqpp	ED73qqpp	DD22qqpp	FD22qqpp
POP s	C1	D1	E1	-	DDE1	FDE1
PUSH s	C5	D5	E5	-	DDE5	FDE5
SBC HL,s	ED42	ED52	ED62	ED72	-	-

TABLE THREE

c	NZ	Z	NC	C	PO	PE	P	M
CALL c,pq	C4qqpp	CCqqpp	D4qqpp	DCqqpp	E4qqpp	ECqqpp	F4qqpp	FCqqpp
JP c,pq	C2qqpp	CAqqpp	D2qqpp	DAqqpp	E2qqpp	ECqqpp	F2qqpp	FAqqpp
JR c,e	20ee	28ee	30ee	38ee	-	-	-	-
RET c	C0	C8	D0	D8	E0	E8	F0	F8

APPENDIX FIVE

	0	1	2	3	4	5	6	7
0	space space	"						
1	((	))	- >	+ <	* =	/ +	= -	> *
2	4 4	5 5	6 6	7 7	8 8	9 9	A A	B B
3	K K	L L	M M	N N	O O	P P	Q Q	R R
4	? RND	? PI	? INKEY$	? ?	? ?	? ?	? ?	? ?
5	? ?	? ?	? ?	? ?	? ?	? ?	? ?	? ?
6	? ?	? ?	? ?	? ?	? ?	? ?	? ?	? ?
7	up up	down down	left left	right right	HOME GRPHCS	EDIT EDIT	NEWLINE NEWLINE	RUBOUT RUBOUT
8								
9								
A								
B								
C	? ""	? AT	? TAB	? ?	? CODE	? VAL	? LEN	? SIN
D	? SQR	? SGN	? ABS	? PEEK	" USR	THEN STR$	TO CHR$	; NOT
E	AND STEP	OR LPRINT	** LLIST	= STOP	< SLOW	> FAST	LIST NEW	RETURN SCROLL
F	? LIST	? LET	? PAUSE	? NEXT	? POKE	? PRINT	? PLOT	? RUN

First row - OLD ROM characters
Second row - NEW ROM characters

	8	9	A	B	C	D	E	F
0	■ ■	■ ■	■ ■	■ "	£ £	$ $	: :	? ?
1	< /	; ;	, ,	. .	Ø Ø	1 1	2 2	3 3
2	C C	D D	E E	F F	G G	H H	I I	J J
3	S S	T T	U U	V V	W W	X X	Y Y	Z Z
4	? ?	? ?	? ?	? ?	? ?	? ?	? ?	? ?
5	? ?	? ?	? ?	? ?	? ?	? ?	? ?	? ?
6	? ?	? ?	? ?	? ?	? ?	? ?	? ?	? ?
7	? K/L	? FNCTION	? ?	? ?	? ?	? ?	? number	? cursor
8	■ ■	■ ■	■ ■	■ ■	■ ■	■ ■	■ ■	■ ■
9	■ ■	■ ■	■ ■	■ ■	■ ■	■ ■	■ ■	■ ■
A	■ ■	■ ■	■ ■	■ ■	■ ■	■ ■	■ ■	■ ■
B	■ ■	■ ■	■ ■	■ ■	■ ■	■ ■	■ ■	■ ■
C	? COS	? TAN	? ASN	? ACS	? ATN	? LN	? EXP	? INT
D	, **	) OR	(AND	NOT <=	- >=	+ <>	* THEN	/ TO
E	CLS CONT	DIM DIM	SAVE REM	FOR FOR	GO TO GOTO	POKE GOSUB	INPUT INPUT	RANDOMISE LOAD
F	STOP SAVE	CONTNUE RAND	IF IF	GO SUB CLS	LOAD UNPLOT	CLEAR CLEAR	REM RETURN	? COPY

A FAREWELL PROGRAM

(NEW ROM users only!)

This program looks particularly effective when run in the SLOW mode. I'm not telling you what it does - feed it in and find out....

BASIC: 1 REM one hundred and sixty one characters
2 RAND USR 16514

MACHINE CODE: (To be written to address 4082 - decimal 16514):

```
21B940      LD HL,40B9     01FF0001    DEFB 01 FF 00 01
7E          LD A,(HL)      0100FFFF    DEFB 01 00 FF FF
23          INC HL         FE050C4E    DEFB FE 05 0C 4E
1F          RRA            7C54004E    DEFB 7C 54 00 4E
F5          PUSH AF        7C55B4AE    DEFB 7C 55 B4 AE
D7          RST 10         B2B2B1B1    DEFB B2 B2 B1 B1
F1          POP AF         B1B1B1B1    DEFB B1 B1 B1 B1
30F8        JR NC,F8       B1B1B0B4    DEFB B1 B1 B0 B4
0680        LD B,80        B5B5B3B3    DEFB B5 B5 B3 B3
3D          DEC A          B3B3B5B3    DEFB B3 B3 B5 B3
20FD        JR NZ,FD       B3B5B3B3    DEFB B3 B5 B3 B3
10FB        DJNZ FB        B5B3B3B3    DEFB B5 B3 B3 B3
01240A      LD BC,0A24     B3B0B0B0    DEFB B3 B0 B0 B0
C5          PUSH BC        B0B1B1B1    DEFB B0 B1 B1 B1
E5          PUSH HL        B0B1B0B1    DEFB B0 B1 B0 B1
CDB60B      CALL 0BB6      B1AEB3B3    DEFB B1 AE B3 B3
C1          POP BC         B5AFB0B0    DEFB B5 AF B0 B0
0A          LD A,(BC)      B1AEB1B0    DEFB B1 AE B1 B0
6F          LD L,A         B1B3B3B3    DEFB B1 B3 B3 B3
2640        LD H,40        B3B0B0B1    DEFB B3 B0 B0 B1
5E          LD E,(HL)      B1B3B3B3    DEFB B1 B3 B3 B3
2C          INC L          B1B1B0B0    DEFB B1 B1 B0 B0
56          LD D,(HL)      AEB2B1B2    DEFB AE B2 B1 B2
E1          POP HL         B1B2B1B2    DEFB B1 B2 B1 B2
C8          RET Z          B2B2B4B5    DEFB B2 B2 B4 B5
19          ADD HL,DE      B5B5B4B5    DEFB B5 B5 B4 B5
C5          PUSH BC        B5B4B5B5    DEFB B5 B4 B5 B5
4D          LD C,L         B4B5B5B4    DEFB B4 B5 B5 B4
44          LD B,H         B5B5AEB7    DEFB B5 B5 AE B7
E1          POP HL         FF          DEFB FF
23          INC HL
18E9        JR E9
```

MASTERING MACHINE CODE ON YOUR ZX 81

TONI BAKER

The ZX-81 computer from Sinclair Research, Ltd., is an exciting new breakthrough in personal computing. About the size of this book, it uses your television set to display programs and any cassette recorder to save programs. Though it can be used for games, for home recordkeeping, and for business functions, it is not "for" any of these uses. Because it is the least expensive, most complete, and most powerful computer of its size on the market, it is an ideal "first computer" to introduce adults and children to the world of computing.

This comprehensive, yet easy-to-understand handbook leads the programmer gently from the BASIC language into ZX-81 machine code, permitting much faster execution of programs and more efficient use of memory. Discover the internal secrets of the ZX-81 machine, and extend your programming capabilities!

Other books from Reston on the ZX-81:

The ZX-81 Pocket Book by Trevor Toms

49 Explosive Games for The ZX-81 by Tim Hartnell

Making the Most of Your ZX-81 by Tim Hartnell

Information about ZX-81 may be obtained from the National ZX Users Group, 599 Adamsdale Rd., N. Attleboro, Mass. 02760.

RESTON PUBLISHING COMPANY, INC.
A Prentice-Hall Company
Reston, Virginia

0-8359-4261-9