"KOBRAHSOFT"
"Pleasant View",
Hulme Lane, Hulme,
Nr. Longton, Stoke-on- Trent,
Staffs. ST3 5BH.
ENGLAND.

TEL:- 078 130 5244.

Dear Customer,

many thanks for your valued order for our "SPECTRUM Z80 MACHINE CODE COURSE. We hope you will find it interesting and informative. We enclose the complete course consisting of 12 newsletters and exercises, PLUS a cassette containing our "KD1 DISASSEMBLER" and "KA1 ASSEMBLER", which we feel are extremely useful and must make our course the best value for money available! Remember, if you get stuck, or if you have any enquiries about any part of the course, please ring us on the above number, or write to the above address. If you write, PLEASE enclose an S.A.E for our reply. We thank you again for buying our course, and we remain,

Yours Sincerely,

"KOBRAHSOFT SOFTWARE".

"KOBRAHSOFT Z80 MACHINE CODE COURSE"

CHAPTER (1)

General Introduction.

When you switch on your Spectrum, and program it in Basic, the following sequence occurs:-

PROGRAMMER (Basic Instructions) >>> BASIC INTERPRETER >>> Produces Machine Code >>> C.P.U >>> Executes Instructions

i.e. your Basic instructions are broken down by the Basic Interpreter into Machine Code, which the C.P.U executes to make your program work. Let us examine these terms:-

BASIC INSTRUCTIONS:- you type these in from the keyboard i.e. LET A=1; PRINT "Hello", etc. These are mostly similar to English
BASIC INTERPRETER:- this is a program in your Spectrum which "decodes" your Basic instructions into Machine Code instructions which the C.P.U can "understand", then execute.
C.P.U :- the Central Processing Unit or "brain" of your Spectrum - it is the Z80A Microprocessor. As you may expect, being an electronic device, it cannot understand English - only Machine Code.
Machine Code :- the language the C.P.U understands - it is simply a series of electronic signals which represent a series of ordinary numbers. The sequence of the numbers tell the C.P.U what to do. It knows what to do for a certain sequence of numbers, because it is pre-programmed with a set of Machine Code instructions - there are over 200 for the Z80!

Points to Note:-
----

(1). Each Basic instruction has to be decoded into Machine Code by the Basic Interpreter - this takes time, and is why programs written in Machine Code run upto 60 times faster than those written in Basic.
(2). The Basic Interpreter is contained in a ROM in your Spectrum's memory.
(3). Memory - Your Spectrum can be thought of as having a "memory" consisting of 65535 "boxes" - each "box" can hold a number from 0 to 255. The values 65535 and 0 to 255 are a characteristic of, and are fixed by, the C.P.U.
(4). ROM - means "Read Only Memory" i.e. you can read from it using PEEK to find the number in a box, but you can't write to it, using POKE to alter it by putting a new number in a box.
(5). PEEK - if you type PRINT PEEK (memory location) - this gives the number in that location i.e. PRINT PEEK 0 gives 243 the first number in the ROM.
(6). If you type POKE location, number - you can put a new number in a box - but only if it is in the RAM.
(7). RAM - This means Random Access Memory - the rest of your Spectrum's memory apart from the ROM. The RAM CAN be changed using POKE to put a new number in a box.

Why use Machine Code?:-

Advantages.
----

(1). Faster execution of your program.
(2). More efficient use of memory.
(3). Freedom from the Operating System.
i.e. you can talk directly to the C.P.U. i.e.:-

PROGRAMMER >>> Machine Code >>> C.P.U >>> Executes program.

Disadvantages.
----

(1). Being a simple sequence of numbers, mistakes are easy to make, one wrong number can give an entirely different instruction!
(2). It cannot readily be adapted to other computers.
(3). Arithmetic calculations are very difficult in Machine Code.

As you can imagine, a program consisting of a series of numbers is hard to understand. Thus, the program can be written in "ASSEMBLY LANGUAGE" - this uses abbreviations or mnemonics (pronounced "MEEMONICS") to relate the instructions to English i.e. it uses:- LD - means LOAD; INC - means INCREMENT; SUB - means SUBTRACT, and so on. The mnemonics are entered into a program called an "ASSEMBLER" - this then produces the machine code (numbers) to be executed by the C.P.U i.e. an Assembler is a convenient way of generating machine code. Similarly, the reverse is a "DISASSEMBLER" i.e. it converts machine code back to mnemonics for easier understanding of a program. The course now INCLUDES our "KD1 DISASSEMBLER" AND "KA1 ASSEMBLER", making it very comprehensive. For FULL details of KD1 and KA1, see Chapter 6.

As stated earlier, machine code is simply a sequence of numbers which represent instructions which the C.P.U can execute. As you may suspect, since the computer is an electronic device, the numbers are represented by electronic signals. We can imagine a series of switches - these can either be ON (shown by a ONE), or OFF (shown by a ZERO). i.e.:-

1	0	0	1	1	1	0	1	0
ON	OFF	OFF	ON	ON	ON	OFF	ON	OFF

The Z80 C.P.U is an EIGHT-BIT device i.e. it groups these "switches" together in blocks of eight. Each "switch" is called a BIT. Each group of 8 BITS = 1 BYTE or one memory location or "box". Incidentally, each group of 4 BITS is called a "NIBBLE" (truly!). The above sequence of 1,0,0,1 etc is called the BINARY sequence. Consider again a group of 8 bits (=1 BYTE). These bits are numbered conventionally thus:- 7,6,5,4,3,2,1,0 i.e. each byte has 8 bits numbered 0 to 7 from right to left (not 1 to 8 as you might expect). The bits can now be related to numbers using the BINARY system thus:-

7	6	5	4	3	2	1	0	: BIT NUMBER
128	64	32	16	8	4	2	1	: Decimal Number

or, more simply, the BINARY system is based on the number 2 (similarly to the usual DECIMAL system, which is based on the number 10) i.e. the decimal number which each bit can represent if = 1 (or is ON or SET, as we say) is the number 2 raised "to the power" of the bit number. Take bit number 2, if set - this represents the number 2X2 = 4. Similarly, for bit 5 = 2X2X2X2X2 (5 times) = 32 and so on. Bit 0 is a special case - if it = 0 or OFF (or RESET) - it represents 0. If = 1 or ON (or SET) it represents 1 (since 2 to the power of 0 = 1 - this is a mathematical fact - don't worry about it!). Thus, we can now say:-

7	6	5	4	3	2	1	0	- BIT NUMBER
128	64	32	16	8	4	2	1	- DECIMAL NUMBER
2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	- 2 TO THE POWER OF

We can now see that if ALL the BITS are ON, or SET (=1), we get:-

7	6	5	4	3	2	1	0	- BIT NUMBER.
1	1	1	1	1	1	1	1	- ALL Bits SET.
128	64	32	16	8	4	2	1	- Decimal Numbers.

i.e. the total number the byte can represent is thus:- 128+64+32+16+8+4+2+1=255! Thus, the biggest number that can be represented by 8 bits or 1 BYTE is 255. We can take this further - consider 2 BYTES. Here we have:-

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	- BIT NO.
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	- ALL SET
32768	16384	8192	4096	2048	1024	512	256	128	64	32	16	8	4	2	1	- Decimal Nos.

Add all these together, and you get 65535 - the biggest number which can be represented by 2 bytes. It is important that you understand this BINARY representation. Try counting in BINARY thus:-

0	0	0	0	0	0	0	0	=	0
0	0	0	0	0	0	0	1	=	1
0	0	0	0	0	0	1	0	=	2
0	0	0	0	0	0	1	1	=	3
0	0	0	0	0	1	0	0	=	4
0	0	0	0	0	1	0	1	=	5
0	0	0	0	0	1	1	0	=	6
0	0	0	0	0	1	1	1	=	7

and so on. You may now see that numbers are expressed in BINARY according to electrical "switches" being ON (SET), or OFF (RESET). Thus, we can have:- 0 0 1 1 1 1 1 1 = 63 = CCF - Complement Carry Flag - a machine code instruction. There are over 200 machine code instructions which the C.P.U is pre-programmed to recognise - we will send you a list of these later in the course. The Binary system is also used, since it can readily be related to another number system called HEXADECIMAL numbering. Here, the sequence is based on the number 16. Here, we get:-

0 1 2 3 4 5 6 7 8 9 A B C D E F

i.e. A = 10, B = 11, C = 12, D = 13, E = 14, F = 15. The letters used are simply a convention. Thus, 15 in Hex. = F. 11 = B. We can count further in Hex. thus:-
FFF

```
16X16X16    16X16    16X1    16X0
 =4096      =256     =16     0 to F
   0          0       0        F      -  (15 Decimal)
                              + 1         +1
----------------------------------------  ----
                      1        0          16
```

Thus, Hexadecimal, (usually shortened to "Hex"), numbers increase in multiples of 16 i.e. 16X16 (256), 16X16X16 (4096), and so on. For example, what is 63 in Hex? To calculate this, we divide by the largest of the multiples - here 16 (the next - 256 is too big), this gives the first digit as 3, with a remainder of 15. But 15 = F, so 63 = 3FH. NOTE:- All Hex numbers usually have a suffix "H" to show they are in Hex. Thus, here we have 3FH. Another example is:- What is 681 in Hex? Here, we again start by dividing the decimal number by the largest Hex multiple, which in this case is 256 (16X16). This gives a first digit of 2, with a remainder of 169. We now divide the remainder by the next lowest multiple i.e. 16 - this gives the next digit as 10, with a remainder of 9. Remember, 10 in Hex = A. Thus, the answer is: 681 Decimal = 2A9H. The BINARY system relates to the Hex system thus:- EACH NIBBLE (group of 4 BITS) IN A BYTE, REPRESENTS ONE HEX DIGIT. Consider the example 63 above. In BINARY this is:-

```
  7   6   5   4   3   2   1   0 - Byte Number.
  0   0   1   1   1   1   1   1 - Binary Number.
128  64  32  16   8   4   2   1 - Decimal Numbers.
```

To rapidly relate to Hex, we divide the 8 bits of the byte into 2 nibbles thus:-

```
                  ^
  7   6   5   4  ^ 3   2   1   0 - Byte Number.
  0   0   1   1  ^ 1   1   1   1 - Binary Number.
  0   0   2   1  ^ 8   4   2   1 - Decimal Numbers.
      2+1 = 3    ^ 8+4+2+1 = 15 =FH - Hex Number.
```

The only difference is that, by splitting the byte, we now have 2 nibbles, numbered thus:-

```
                  ^
  7   6   5   4  ^ 3   2   1   0 - Byte Number.
  8   4   2   1  ^ 8   4   2   1 - Decimal Numbers.
                  ^
```

i.e. the largest Hex number we can represent in one byte is:-

```
                  ^
  7   6   5   4  ^ 3   2   1   0 - Byte Number.
  1   1   1   1  ^ 1   1   1   1 - Binary Number.
  8   4   2   1  ^ 8   4   2   1 - Decimal Numbers.
8+4+2+1 = 15 = FH  ^8+4+2+1 = 15 = FH - Hex Number.
          i.e. F   ^   F
```

i.e. = FFH = 255 Decimal. Similarly, for 2 bytes, the largest number is FFFFH = 65535 Decimal. This simple way to interchange between Binary and Hex is useful, since most Assemblers and Disassemblers display the machine code in Hex numbers. Also, these can now be converted to Binary, so you can see the state of each individual bit in the byte.

The Structure of the C.P.U.

We will now consider in more detail the structure of the "brain" of your Spectrum - the Central Processing Unit.
We can consider a machine code program to be a sequence of instructions for the C.P.U to perform particular tasks. In the Z80 C.P.U, all instructions are represented internally as single or multiple bytes. Instructions represented by one byte are called "SHORT" instructions; longer ones are represented by two or more bytes. Because the Z80 is an 8-bit processor, it can only deal with one byte at a time - if it requires more than one byte, it must fetch bytes successively from memory. Thus, a single byte instruction is usually executed more rapidly than a 2 or 3 byte one. This is why it is best to write your machine code program using as many single byte instructions as possible.

The structure of the Z80 can be divided into 5 main parts i.e:-

(1). THE CONTROL UNIT.
(2). THE INSTRUCTION REGISTER.
(3). THE PROGRAM COUNTER.
(4). THE ARITHMETIC - LOGIC UNIT (A.L.U).
(5). THE 24 USER - REGISTERS.

We will now consider each in more detail:-

THE CONTROL UNIT.

The Control Unit is the supervisor for the C.P.U's processing. Its task is to time and coordinate the input, processing and output of any task the C.P.U is performing.

THE INSTRUCTION REGISTER.

The Instruction Register is where the CPU stores the current instruction which it is about to execute. Remember, your program is a sequence of instructions. To execute the instructions, the Control Unit must fetch each instruction in turn from memory, and place it in the Instruction Register.

THE PROGRAM COUNTER.

This is a location in the CPU where THE ADDRESS OF THE NEXT INSTRUCTION TO BE EXECUTED is stored.

THE ARITHMETIC AND LOGIC UNIT (A.L.U).

This is the calculator inside the CPU. It can only perform simple arithmetic, i.e. addition, subtraction, incrementation (adding 1), decrementation (subtracting 1), NOT multiplication or division!

THE USER REGISTERS.

There are 24 user registers in the CPU. They are locations which can hold one byte or two bytes. They are usually shown as:

```
A  ^  F
B  ^  C
D  ^  E
H  ^  L
   IX
   IY
```

The use of the letters is pure convention. The important points are:-

The registers AF, BC, DE, and HL are usually paired together. However, they can ALSO be used SEPARATELY as individual registers i.e. B, C, D, E, H, and L. When used thus, each register can represent 8 bits or 1 BYTE. When "paired", they can represent 16 bits or 2 bytes. Also, the arrangement of letters in the pairs tells us which register holds the high part (i.e. HIGH ORDER BYTE - H.O.B), and which the low part (LOW ORDER BYTE - L.O.B). Thus, for example, using the Hex number 7EEAH, this is 2 bytes remember, the H.O.B is 7EH, the L.O.B is EAH. Thus, if this was contained in the HL register pair, the H register would contain 7EH, the L register EAH. Similarly, for AF, BC, and DE - A,B, and D contain the H.O.B; F,C, and E contain the L.O.B. However, the AF register pair is a special case. Here, the A and F registers are ALWAYS used SEPARATELY. The A register has a special name - it is called the ACCUMULATOR. It is always used to perform arithmetic functions. Also, the F register is known as the FLAGS register. This is used thus:- the 8 bits in the byte of the register are used as "FLAGS", i.e. indicators as to whether the result of a certain operation is negative, or positive; zero or not zero etc; according to which bits in the byte are SET (=1) or RESET (=0). Thus, the A and F registers are ALWAYS used singly as single 8 bit registers, each with a special function. The HL register pair is also more important than the BC and DE pairs. This is because certain 16 bit arithmetic operations can only be performed using HL. Because of this, general register pair operations will usually be faster using the HL register pair - use it preferentially where you can! The IX and IY register pairs are also a special case. They are called INDEX REGISTERS i.e. they contain an address (ALWAYS 16 bit) to which can be added offsets i.e. IX + 1, IY + 2, etc - hence the name INDEX registers.

The CPU also has an ALTERNATE REGISTER SET. This can be likened to a mirror image set of the AF, BC, DE and HL registers. They are used mainly as a temporary storage place for data, which can then readily be put back into the usual registers with a simple "exchange" command. They can thus only be used in exchange - there are no separate instructions for them. There are 3 more important registers - these are:- THE STACK POINTER (SP). This contains the address of the stack in memory. The stack is a series of memory locations (usually in RAM), where the CPU stores numbers temporarily. Liken it to the stack used in an office i.e. that metal spike onto which are placed bills, etc. The only difference being that in the computer, it can be thought of as hanging from the ceiling i.e. as the stack grows, it grows DOWNWARD from HIGH to LOWER memory locations. It is ALWAYS used as a 16 bit register PAIR. Numbers are stored by PUSHING them onto the stack, then retrieved by POPPING them off (see later).

Another register is the I Register or Interrupt Vector register. It is usually used to hold the base address of a table of addresses for handling different responses to an interrupt - this is only very rarely used. The final register is the R or Memory Refresh register. This is mainly used for obtaining random numbers from 0 to 255.

We hope you have understood the important points in this first part of the Z80 course. To help, please find on a separate sheet a few exercises to try - the answers are on the reverse side - no peeking!

"KOBRAHSOFT Z80 MACHINE CODE COURSE"

CHAPTER (1) EXERCISES

We hope these few exercises will help to illustrate the points dealt with in the first chapter of our Z80 course. We suggest you try them, then check with the answers which we have printed on the opposite side. Also, make a few of your own - the more practice you get the easier it will become. Remember, if you get stuck, or if you have any questions about any part of this course, don't hesitate to write to us. PLEASE enclose an S.A.E. for our reply!

(A). What are the following Decimal numbers expressed in BINARY notation?

(1). 9.
(2). 24.
(3). 68.
(4). 149.
(5). 217.

(B). What are the following Decimal numbers expressed in HEX notation?

(1). 121.
(2). 249.
(3). 42.
(4). 98.
(5). 743.

For the answers, see overleaf - no peeping!

"KOBRAHSOFT Z80 MACHINE CODE COURSE"

CHAPTER (2)

USING THE REGISTERS.

We will now consider the use of the REGISTERS in machine code, by considering a few simple machine code instructions i.e. LD HL,nn. These are the "mnemonics" (remember?) for:- "Load the HL register pair with the 2 byte number represented by nn". The machine code instructions are:-

In DECIMAL: 33,n,n or 33,nL,nH.
In HEX : 21,n,n or 21,nL,nH.

-not as you might have expected:- 33,nH,nL; where nH is the HIGH ORDER BYTE (the larger part of the number), and nL is the LOW ORDER BYTE (the smaller part of the number) e.g. for the Hex number 7D06 - 7D is the H.O.B, 06 or 6 is the L.O.B. As you can see, this is a THREE BYTE instruction i.e. 33 (the instruction to LOAD HL with a 2 byte number) - the number consisting of the 2 bytes that follow i.e. n,n or more specifically, nL,nH. The numbers are nL,nH and NOT nH,nL as you would expect, because: IN THE Z80 CPU, NUMBERS ARE STORED LOW BYTE FIRST, HIGH BYTE SECOND! i.e. the reverse to what you would expect. This is simply a convention, but it is MOST IMPORTANT. Thus, for the Hex number 7D06, we would have:-

MNEMONICS	Z80 INSTRUCTIONS
LD HL,7D06	33,6,125 - in DECIMAL.
LD HL,7D06	21,6,7D - in Hex.

This is how the instruction MUST be entered in a machine code program. When executed, the result will give:

	H Register	L Register
In Hex:	7D	06

i.e. the result you wanted. YOU must enter the correct instruction sequence so that the result is to be what you want. Consider, if you had written:- (in Hex) 21,7D,06 - you would get as a result:-

H Register	L Register
06	7D

- the reverse to what you wanted! It is most important that you understand how to enter numbers as Z80 instructions correctly, to get the result you want. Similarly:- (In Hex) for the number 7D06:-

MNEMONICS	Z80 INSTRUCTIONS
LD BC,nn	1,nL,nH OR 1,6,7D - gives B=7D; C=06.
LD DE,nn	11,nL,nH OR 11,6,7D - gives D=7D; E=06.

We will now write our FIRST machine code program to show you how fast it really is! A good example uses the LDIR command, which moves blocks of memory around. We will discuss this in greater detail later. We will load the SCREEN$ (code for the picture) from any of your games to a convenient location; e.g. 32000 (7D00 Hex); then use the LDIR command to move ALL 6912 bytes into the screen area at 16384 (4000 Hex). Firstly, choose a game with a good picture - we will use for our example "EXPLODING FIST". ANY picture will do - but ensure that it is suitable i.e. it loads at normal speed, and is not "protected" - the Melbourne House games are good in this respect. To check, type LOAD""CODE 16384 (ENTER) and PLAY the game tape from the start. Usually the Basic loader appears first, then in a suitable game, the SCREEN$ should then load to give you a picture. Now, load the picture to 32000 by typing LOAD""CODE 32000 (ENTER). Instead of a picture now, the SCREEN$ code will simply load to address 32000. We can put our machine code program at, say, 30000. The mnemonics for the program are:-

MNEMONICS	Z80 INSTRUCTIONS (Hex)	Z80 INSTRUCTIONS (Decimal)
LD HL,7D00	21,0,7D	33,0,125
LD DE,4000	11,0,40	17,0,64
LD BC,1B00	1,0,1B	1,0,27
LDIR	ED,B0	237,176
RET	C9	201

i.e. the program consists of the sequence:- 33,0,125,17,0,64,1,0,27,237,176,201 - only 12 numbers! The HL register contains the start address of the block to be moved - here = 7D00 Hex = 32000 Dec. The DE register contains the destination address - here = 4000H = 16384 Dec. The BC register contains the number of bytes to be moved - here = 6912 Dec = 1B00 Hex. The instruction for LDIR are the numbers ED, B0 Hex = 237,176 Dec. The RET or Return ensures we return to Basic. The instruction for RETurn is C9 Hex = 201 Dec. We can POKE the 12 numbers for the routine starting at 30000 using the following Basic Loader (NOTE:- This could also be done using an ASSEMBLER - see earlier). Type in:-

```
10: FOR A=30000 TO 30011: INPUT B: POKE A,B: NEXT A
```

This program waits for you to input a number, then POKES it to the appropriate address. RUN it, and enter the 12 numbers. Your first machine code program now resides at address 30000 - 30011. If you wish to examine a range of memory locations, we will use the following program (we shall call it "PEEK LINE"):-

Type 10 (ENTER) to remove the previous program, then type in:-

10: FOR A=30000 TO 30011: PRINT A,: PRINT PEEK A: NEXT A

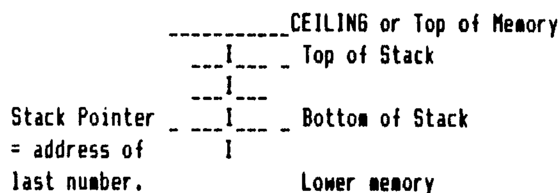
RUN it, and you will see your machine code program! How do we execute this program? All machine code routines can be executed from Basic using the USR command. So here, we type RANDOMISE USR 30000 (ENTER). On pressing ENTER, the whole picture (6912 bytes) is moved from address 32000 to the screen in a flash! Such is the speed of machine code! If you want to repeat the routine to impress your friends, type 10 (ENTER) to remove "PEEK LINE", then enter this small Basic program:-

10: RANDOMISE USR 30000
20: GOTO 20

RUN it, and the picture will appear in a flash! Line 20 stops the computer from printing the usual "O.K." message at the bottom of the screen, and spoiling the picture. To re-run the program, press "BREAK"; type CLS (ENTER); then type RUN (ENTER) again, and so on. This illustrates the beauty of machine code - whilst running your machine code routine, YOU are in TOTAL control of the computer, not the reverse! But REMEMBER, ONE wrong number can give the CPU a completely different instruction from the one you intended, which usually "CRASHES" the computer. This is simply the term used when there is no response from the keyboard - don't worry, no damage is done! It is simply resolved by switching your computer off, then (after a few seconds) on again - or by pressing the RESET switch if you have one. This is said to RESET the computer, or start it anew. Incidentally, this can also be done (if you have keyboard response and wish to reset the computer) by typing RANDOMISE USR 0 (ENTER). This calls the ROM routine at address 0, and resets your computer - just as if you were switching it on in the morning.

THE STACK AND STACK POINTER (S.P.).

As stated earlier, the stack is an area of memory which the CPU uses for the temporary storage of numbers. The Stack Pointer (S.P.) is a 2 byte register which contains the current address of the last number on the stack. Liken it to that office "spike" on which you stick bills, etc. However, in the Spectrum, the spike can be imagined to "hang from the ceiling", thus:-



i.e. it grows downwards (towards lower memory). A number is saved onto the stack using the PUSH instruction. It is retrieved using the POP instruction. All numbers saved on the stack are 2 byte numbers i.e. you CAN have PUSH HL - save the value contained in the HL register on the stack. You CANNOT have PUSH D - save the contents of a single register. Can you see that when you PUSH a number onto the stack, the address of the S.P. DECREASES by 2 bytes? (i.e. SP=SP-2). Similarly, a POP INCREASES S.P. by 2 bytes (SP=SP+2). PUSH and POP are a quick way to exchange numbers between registers i.e. if you PUSH HL, then POP DE, the number in HL is transferred to DE! You MUST always know the location of the stack in memory. If you overwrite the stack (i.e. load other bytes over it), the Spectrum will crash, since the CPU won't know where its next number will come from! The position of the stack is fixed from Basic using the CLEAR instruction i.e. CLEAR 60000 means the stack starts at 59999 (CLEAR value - 1). In machine code it is fixed using the LD SP,nn instruction. Where nn is the 2 byte address you choose. Remember, you must always REVERSE your sequence of PUSHes and POPs to retrieve the correct numbers you need i.e.

PUSH AF
PUSH BC
PUSH HL

reversed gives:-

POP HL
POP BC
POP AF

- to retrieve the original values.

USING THE REGISTERS TO ADDRESS MEMORY.

There are FIVE main ways in which data can be transferred from one register to another, or from a register to memory i.e.:-

(1). Immediate addressing.
(2). Register addressing.
(3). Register indirect addressing.
(4). Extended addressing.
(5). Indexed addressing.

Don't worry too much about the complex names - these are only used as a convention. Taking each one in turn:-

IMMEDIATE ADDRESSING.

The general form for this is:- LD r,n (or other instruction, we are using LD as an example only). Where r is any 8-bit (single byte) register e.g. A,H,L etc, and n is any 8-bit (single byte) number e.g. 0 - 255. We thus have an interaction between a REGISTER and DATA. An example is:- LD A,255 (Dec.) or LD A,FF (Hex). The machine code instructions are:- 62,255 (Dec.) or 3E,FF (Hex). Note that the actual data is a part of the instruction - this means that the CPU can execute the instruction IMMEDIATELY - it doesn't need to look in memory to find more information in order to perform the instruction. The general format is:- 1st Byte:- instruction code (or opcode) - this tells the computer what the instruction is i.e. LD A; LD H; etc. 2nd Byte:- the actual data byte. Since this is a single byte - only numbers in the range 0-255 can be used.

REGISTER ADDRESSING.

The general form is:- LD r,r. (or other instructions) - i.e. an exchange between one register and another. An example is: LD A,B; or LD H,E etc. NOTE:- We CANNOT have LD H,F - since F the FLAG register is a special case (see earlier). The instruction LD H,E means LOAD H with the contents of E. Thus if H contains 7D (Hex) and E contains 3F (Hex). After execution of the instruction we would have:- H contains 3F, and E contains 3F. These instructions only need ONE byte, thus they are fast and are to be used wherever possible.

REGISTER INDIRECT ADDRESSING.

This mode is a little more complex. The general format is:- LD (rr),A or LD A,(rr) or LD (HL),n. Here we have the transfer of data between the CPU and a memory location whose address is contained in a 16-bit (2 byte) register pair i.e. HL, DE etc. NOTE:- rr is used to show such a REGISTER PAIR. Consider:- LD (rr),A. Suppose rr is the HL register pair, and it contains the number 7D00H (the suffix "H" from now on will indicate a HEX number - no suffix will indicate a Decimal number; this is the usual convention used). Suppose the Accumulator A contains the number 7EH. The instruction LD (HL),A means take the value in A, and put it into the address contained in HL. Thus, after execution, the location 7D00H would contain 7EH, HL would still contain 7D00H, and A 7EH. For LD A,(rr) - suppose rr was HL, which contained 7F00H; if location 7F00H contained the number 4AH, and A contained 63H - after execution, A would contain 4AH. For the example:- LD (HL),n. Suppose HL held the number 8000H and n (a single byte number) was 4DH. If the location 8000H contained, say, 2DH; after execution it would contain 4DH. This mode of addressing is faster than ordinary indirect addressing, since the CPU need not fetch the address from memory.

EXTENDED ADDRESSING.

The general format is:-

```
        LD A, (nn)  or  LD (nn),A
   or:- LD HL,(nn)  or  LD (nn),HL
```

where nn represents a 2 byte (16-bit) address in memory. In this mode, the instructions from the program supply the CPU with an address specified by these 2 bytes.

If single registers i.e. A,H,D etc are involved, there will be 3 instructions. With register pairs e.g. HL,DE etc, there will be 4 instructions.

```
E.g. Consider:- LD A,(nn) - the general instruction format is:- 3A,nL,nH.
                LD (nn),A - the general instruction format is:- 32,nL,nH.
               LD HL,(nn) - the general instruction format is:- ED,6B,nL,nH.
               LD (nn),HL - the general instruction format is:- ED,63,nL,nH.
```

or, in general,

Byte 1:- Opcode - tells the CPU what type of instruction to expect.
Byte 2:- Possible additional opcode if register pairs are involved.
Byte 3:- Low order byte of the 2 byte address.
Byte 4:- High order byte of the 2 byte address.

INDEXED ADDRESSING.

This, as you might expect, involves the use of the Index Registers IX and IY. The general format is:-

LD r, (IX+d) or LD r, (IY+d) or LD (IX+d),r or LD (IY+d),r.

Where r is a single byte register e.g. A,B,H,D etc; d is a single byte number for the displacement from the start address. The CPU adds this number d to the contents of the IX or IY register to find the required address. One typical usage for this type of addressing technique is the manipulation of tables of data. e.g. you can set the address in the IX or IY registers to be the start address of a table of data. You can then specify any particular byte to which you want by adding the value d.

e.g.:-

LD IX, TABLESTART
LD A, (IX+5)

This will refer to the 5th byte from the start of the table, and it will be placed in the Accumulator.

The general format is:-

Byte 1:- opcode - tells the CPU what type of instruction.
Byte 2:- opcode - as above.
Byte 3:- displacement d.

Indexed addressing is slow because the CPU must perform an addition in order to obtain the effective address. However, it is very flexible since the same instruction can refer to all the elements in a table simply by altering the value of d.

NOTE:- You can combine immediate addressing (i.e. specifying the number you want loaded) with external addressing (i.e. specifying the address to be loaded by using a register pair). This is called "IMMEDIATE EXTERNAL ADDRESSING". Unfortunately, you can only use the HL register pair and the general format is thus:- LD (HL),n. This is very useful, since it allows you to directly fill a memory location with a number without having to load the number into a register. Consider:- if HL contains the address 7D00H, with the instruction LD (HL),FFH, you can put the value FFH in address 7D00H! The instruction is only 2 bytes long i.e. 36H,n. Where n is a single byte number.

A similar combination is possible using the Index Registers IX and IY. This is called "IMMEDIATE INDEXED ADDRESSING". The instructions take the form of:-

LD (IX+d),n
LD (IY+d),n

THE FLAGS REGISTER F.

As we stated earlier, the F or "FLAGS" register is used to indicate the existence of certain conditions. The Flag register is an 8-bit register and, of the 8 bits, 6 are used as "flags". These are as follows:-

Bit No.:-	7	6	5	4	3	2	1	0
FLAG :-	SIGN FLAG (S)	ZERO FLAG (Z)	NOT USED	HALF CARRY FLAG (H)	NOT USED	PARITY and OVERFLOW FLAG (P/V)	NEGATE or SUBTRACT FLAG (N)	CARRY FLAG (C)

NOTE:- These flags all relate to the condition of the number in the A register (Accumulator). We shall briefly discuss each flag in turn - we will discuss their uses more fully later.

THE ZERO (Z) FLAG.

This flag will be set i.e. the bit which represents the flag (bit 6) will be set or = 1 if, as a result of an arithmetic operation, the contents of the A register are zero; otherwise it is reset or = 0.

THE SIGN FLAG (S).

This is very similar in use to the zero flag above i.e. if the contents of A are positive - the flag is set; if negative it is reset.

THE CARRY FLAG.

This is one of the most important of the flags. Briefly, it is set if the result of an arithmetical operation would give an "underflow" i.e. 200-201 gives 255! This is best seen by thinking of the carry bit as a 9th bit of the A register thus:-

Number	Carry Bit	Number in Binary Form
200	-	1 1 0 0 1 0 0 0
- 201	-	- 1 1 0 0 1 0 0 1
=====		==================
255	1 (set)	1 1 1 1 1 1 1 1

A similar situation arises with an "overflow" i.e. 132+135=267 with the carry set.

PARITY/OVERFLOW FLAG (P/V).

This flag is set when, after an arithmetic operation, there are an EVEN number of SET bits in the result.

SUBTRACTION OR NEGATE FLAG (N).

This flag is set if the last operation was a subtraction.

HALF-CARRY FLAG (H).

This flag is set similarly to the carry flag, but only if the overflow occurs from the 5th bit - not the 9th!

Now for the good news - no exercises this week! We think this chapter should keep you thinking for a while!

"KOBRAHSOFT Z80 MACHINE CODE COURSE"

CHAPTER (3)

This month we give you a list of the MORE COMMON Z80 mnemonics and their representations in Hex numbers. Do not worry if you don't understand them - we will deal with each group in more detail later.

MNEMONIC	HEX NO	MNEMONIC	HEX NO	MNEMONIC	HEX NO	MNEMONIC	HEX NO
ADC HL,BC	ED 4A	INC A	3C	LD IX,nn	DD 21 xx xx	SUB n	93
ADC HL,DE	ED 5A	INC BC	03	LD IY,(ADDR)	FD 2A xx xx	XOR A	AF
ADC HL,SP	ED 7A	INC DE	13	LD IY,nn	FD 21 xx xx	XOR n	EE xx
ADD A,(HL)	86	INC HL	23	LD L,n	2E xx		
ADD A,(IX+dis)	DD 86 xx	INC IX	DD 23	LD SP,(ADDR)	ED 7B xx xx		
ADD A,(IY+dis)	FD 86 xx	INC IY	FD 23	LD SP,nn	31 xx xx		
ADD A,n	C6 xx	INC SP	33	LD SP,HL	F9		
ADD HL,BC	9	INCR	ED BA	LD SP,IX	DD F9		
ADD HL,DE	19	INIR	ED B2	LD SP,IY	FD F9		
ADD HL,HL	29	JP (HL)	E9	LDD	ED A8		
ADD HL,SP	39	JP (IX)	DD E9	LDDR	ED B8		
ADD IX,BC	DD 09	JP ADDR	C3 xx xx	LDI	ED A0		
ADD IX,DE	DD 19	JP C,ADDR	DA xx xx	LDIR	ED B0		
ADD IX,IX	DD 29	JP NC,ADDR	D2 xx xx	OR (HL)	B6		
ADD IX,SP	DD 39	JP NZ,ADDR	C2 xx xx	OR A	B7		
ADD IY,BC	FD 09	JP Z,ADDR	CA xx xx	OR n	F6 xx		
ADD IY,DE	FD 19	JR C,dis	38 xx	OTDR	ED BB		
ADD IY,IY	FD 29	JR dis	18 xx	OTIR	ED B3		
ADD IY,SP	FD 39	JR NC,dis	30 xx	OUT (C),A	ED 79		
AND (HL)	A6	JR NZ,dis	20 xx	OUT port,A	D3 port		
AND A	A7	JR Z,dis	28 xx	POP AF	F1		
AND n	E6 xx	LD (ADDR),A	32 xx xx	POP BC	C1		
BIT 0,A	CB 47	LD (ADDR),BC	ED 43 xx xx	POP DE	D1		
CALL ADDR	CD xx xx	LD (ADDR),DE	ED 53 xx xx	POP HL	E1		
CALL C,ADDR	DC xx xx	LD (ADDR),HL	22 xx xx	POP IX	DD E1		
CALL NC,ADDR	D4 xx xx	LD (BC),A	02	POP IY	FD E1		
CALL NZ,ADDR	C4 xx xx	LD (DE),A	12	PUSH AF	F5		
CALL Z,ADDR	CC xx xx	LD (HL),A	77	PUSH BC	C5		
CP (HL)	BE	LD (HL),n	36 xx	PUSH DE	D5		
CP A	BF	LD (IX+dis),A	DD 77 xx	PUSH HL	E5		
CP n	FE xx	LD (IX+dis),n	DD 36 xx xx	PUSH IX	DD E5		
CPDR	ED B9	LD A,(ADDR)	3A xx xx	PUSH IY	FD E5		
CPIR	ED B1	LD A,(BC)	0A	RES 0,A	CB 87		
DEC (HL)	35	LD A,(DE)	1A	RET	C9		
DEC A	3D	LD A,(HL)	7E	RET C	D8		
DEC BC	0B	LD A,(IX+dis)	DD 7E xx	RET NC	D0		
DEC DE	1B	LD A,n	3E xx	RET NZ	C0		
DEC HL	2B	LD B,n	06 xx	RET Z	C8		
DEC IX	DD 2B	LD BC,(ADDR)	ED 4B xx xx	RL A	CB 17		
DEC IY	FD 2B	LD BC,nn	01 xx xx	RR A	CB 1F		
DEC SP	3B	LD C,n	0E xx	RST 08	CF		
DI	F3	LD D,n	16 xx	SBC HL,BC	ED 42		
DJNZ dis	10 xx	LD DE,(ADDR)	ED 5B xx xx	SBC HL,DE	ED 52		
EI	FB	LD DE,nn	11 xx xx	SBC HL,HL	ED 62		
EX (SP),HL	E3	LD E,n	1E xx	SBC HL,SP	ED 72		
EX AF,A'F'	08	LD H,n	26 xx	SCF	37		
EX DE,HL	EB	LD HL,(ADDR)	2A xx xx	SET 0,A	CB C7		
IN A,port	DB xx	LD HL,nn	21 xx xx	SLA A	CB 27		
INC (HL)	34	LD IX,(ADDR)	DD 2A xx xx	SRA A	CB 2F		

We must stress that this is not a FULL list - for the full list, please consult any of the Z80 reference works which are usually available at your local library. A good book we can recommend is:- "PROGRAMMING THE Z80" by Rodnay Zaks. This gives a FULL list of ALL the Z80 mnemonics, together with a lot of other useful information.

NOTE:- In the above list:-

n - represents a SINGLE BYTE number.
nn - represents a TWO BYTE number.
xx - represents a SINGLE BYTE number (2 Hex digits).
xx xx - represents a TWO BYTE number (4 Hex digits).

Information Representation in the Spectrum.

As we stated earlier, all information in your Spectrum is stored as groups of BITS. A BIT stands for Binary digIT i.e. a "0" or a "1". Because of its electronic nature, your Spectrum can only represent data using this two-state or BINARY system. Just to remind you, the rules for ADDING binary numbers are:-

0 + 0 = 0
0 + 1 = 1
1 + 0 = 1
1 + 1 = (1) 0

- where (1) denotes a "carry" of 1 i.e.

```
  0 0 1 1  (= 3 decimal)
+ 0 0 0 1  (= 1 decimal)
-------------------------
= 0 1 0 0  (= 4 decimal)
```

As stated earlier, we can thus represent numbers from 00000000 to 11111111 i.e. 0 to 255 in 8 bits in binary. There are 2 problems here:- (1) we are only representing POSITIVE numbers. (2) The magnitude of these numbers is limited to 255 if we use only 8 bits. Consider each of these problems in turn:-

SIGNED BINARY.

In a signed binary representation, the left-most bit is used to indicate the sign of the number. As a convention, "0" is used to denote a POSITIVE number, while "1" is used to denote a NEGATIVE number. Now, 11111111 will represent -127, while 01111111 will represent +127. We can now represent positive and negative numbers, but we have reduced the maximum range to 127. e.g. 00000001 represents +1 (the leading 0 means "+", followed by 0000001 = 1). Also, 10000001 is -1 (the leading "1" means "-", followed by 0000001 = 1).

Now consider the MAGNITUDE problem. In order to represent larger numbers, we will have to use a larger number of bits e.g. if we use 16 bits (2 bytes), we can represent numbers from -32768 to +32768 in signed binary. Bit 15 is used for the sign, and the remaining 15 bits (14 to 0) are used for the magnitude.

Consider using signed binary for a simple addition i.e.:- let us add -5 and +7:-

```
              +7 is:- 00000111
              -5 is:- 10000101
                      --------
The sum is:-          10001100 or -12!
```

This is incorrect! It should be +2! i.e. the binary addition of signed numbers does not work correctly. The solution to this problem is called the "two's complement" system, which will be used instead of the signed binary system. In order to introduce two's complement, let us first introduce an intermediate step:- one's complement.

One's Complement.

In this system, all positive integers (whole numbers) are represented in their correct binary format. For example, +3 is represented as usual by 00000011. However, its complement -3 is obtained by complementing or swapping every bit in the original representation i.e. each 0 is transformed into a 1, and each 1 is transformed into a 0. Thus, in this example, the one's complement representation of -3 will be 11111100. Also, +2 is 00000010; -2 is 11111101. Note that here, positive numbers start with a 0 on the left, negative with a 1 on the left.

As a check, let us add minus 4 and plus 6:-

```
             -4 is:- 11111011
             +6 is:- 00000110
                     --------
The sum is:- (1)     00000001
```

where (1) indicates a carry. The correct result is 2 or 00000010. i.e. it did not work. This is overcome using:-

Two's Complement.

In this system, positive numbers are still shown, as usual, in signed binary, just as in one's complement. The difference lies in the representation of NEGATIVE numbers. A negative number in two's complement is obtained by first computing the one's complement, and then ADDING ONE. Consider:- +3 is shown in signed binary by 00000011. It's one's complement is 11111100. The two's complement is this PLUS 1 or 11111101. Consider:-

```
 3 is:- 00000011
+5 is:- 00000101
-----------------
=8    = 00001000
```

The result is correct! i.e. we have found a way to represent negative numbers - we now know how to use this convention if we wish to enter numbers of specific sign for arithmetical calculations, etc.

USING THE MNEMONICS.

We will now consider the use of the various groups of mnemonics in machine code programs. As we mentioned earlier, all machine code programs are called from Basic using the "USR" command. This can take the form of RANDOMIZE USR, PRINT USR, RUN USR etc. PRINT USR is a special case, since, on returning to Basic from a machine code routine it prints the value in the BC register pair at that time. When the Spectrum operating system encounters this "USR" function, it loads the BC register pair with the number specified in the USR command. We can check this thus:- Type:- POKE 32000,201 (ENTER). Now type PRINT USR 32000 (ENTER). The number 32000 appears. You have executed another machine code program. It works thus:- when you POKE the value 201 to address 32000 - this is the mnemonic for RETurn i.e. return to Basic. (See list above). Typing the PRINT USR 32000 command loads 32000 into the BC register pair - this is printed on returning to Basic! A further refinement would be:- POKE 32000,14; POKE 32001,240; POKE 32002,201. These numbers represent the machine code program:-

```
LD C,240
RET
```

Thus, type PRINT USR 32000. The number 32240 appears! This is because you have loaded the C register with 240 (F0H) - the B register is unaltered - thus the new number in BC is 32000 + 240 = 32240. Experiment by putting various other numbers in the C register i.e. change the number at 32001, and note the results. Remember, you MUST always include a RETurn, otherwise you will not return to Basic and the computer will crash!

Another example of an 8 bit loading instruction is:-

```
LD A,B
```

This would read as: "Load the A (Accumulator) register with the contents of the B register". We can have similar instructions involving the other registers i.e. LD C,A; LD E,H etc - even LD A,A! Remember, the F or Flags register is a special case, so there are no instructions involving it singly e.g. LD F,A is not permitted. A general representation of this type of instruction is:-

```
LD r,r
```

- where r represents any 8 bit register EXCEPT F. Remember, this is called REGISTER ADDRESSING (see earlier) or the transfer of information from one register to another.

We can also load numbers into a register, e.g.:-

```
LD D,D7H
```

- where this means "load the D register with the number D7 in Hex". (215 decimal). The number for the mnemonic LD D,n or LD D with a number, is 16H, followed by the number; or 16,n. Thus, the instruction for LD D,D7H is:- 16H,D7H or 22,215 in decimal. This would put the number 215 into the D register.

You may recall this is known as immediate addressing. Again, you can do this with any of the registers, with any numbers - the limitation of course being the size of the numbers i.e. 0-255 since you have only 8 bits. A shorthand representation of this is:-

LD r,n

where r indicates any register and n any SINGLE BYTE number.

We now need to know how to put numbers into memory locations - after all, we have only so many registers! The general mnemonic to do this is:-

LD A,(nn)

Remember that the brackets mean "the contents of the location pointed to by the number nn". Where nn is a TWO BYTE (16 bit) number. Let us clarify this, as it is a difficult concept, but one which it is most important that you understand clearly. Suppose the number is, for example, 32000 (7D00H). Thus, we have:-

LD A,(7D00H)

Suppose location 32000 contained the number 200; then the above instruction would result in the number 200 being placed in the A register. Other points to note about this instruction are:-

(1). You can only use it with the A register.
(2). YOU must supply the memory location as a 2 byte (16 bit) number.

The reverse instruction is also valid i.e.:-

LD (nn),A

Here, the number in the A register is placed in the memory location addressed by the number nn - in the above case, if A contained the number 175, the instruction would result in this number being placed in location 32000.

Remember, these 2 instructions only apply to the A register - there are other instructions for the other registers, but none quite as clear as these. Again, it's because A is a special register (the Accumulator).

As you can see, these 2 instructions are very powerful. The instruction LD A,(nn) allows us to get a number from any memory location, and place it in the A register. The instruction LD (nn),A allows us to put a number in any memory location from the A register.

Another form of instruction uses register indirect addressing. These are of the general form:-

LD r,(HL)
LD A,(BC)
LD A,(DE)

These read as:- "Load the register r with the contents of the memory location pointed to by HL".
"Load A with the contents of the memory location pointed to by BC".
"Load A with the contents of the memory location pointed to by DE".

Note that by using the HL register pair as the pointer to our memory location, we can load to ANY register - even H or L - but using BC or DE, we can only load into the Accumulator. This is because the HL register pair the favoured register pair in the same way that the A register is the favoured single register. Again, the reverse instructions apply i.e.:- LD (HL),r; LD (BC),A; LD (DE),A.

Alternatively, we could use the index registers IX and IY to point to the memory location i.e.

LD r,(IX+d)
LD r,(IY+d)

Here, r is again any register, and d is the displacement from the address pointed to by IX or IY. NOTE:- This is a single byte number d - NOT the D register. It is usually used for addressing tables containing lists of data. The reverse instructions also apply i.e.:- LD (IX+d),r; LD (IY+d),r.

We can also have Immediate External Addressing of the form:-

LD (HL),n

Unfortunately, this ONLY uses HL. With the index registers, we can have Immediate Indexed Addressing of the form:-

LD (IX+d),n and LD (IY+d),n

Now, please try the examples to see if you have understood this chapter.

"KOBRAHSOFT Z80 MACHINE CODE COURSE"

CHAPTER (3) EXERCISES

(1). What is +15 in Signed Binary?

(2). What is +65 in Signed Binary?

(3). What is -27 in Signed Binary?

(4). What is -110 in Signed Binary?

(5). What is +7 in 1's Complement?

(6); What is -65 in 1's Complement?

(7). What is -42 in 2's Complement?

Answers overleaf!

ANSWERS.

(1). Answer = 00001111.

(2). Answer = 01000001.

(3). Answer = 10011011.

(4). Answer = 11101110.

(5). Answer = 00000111.

(6). Answer = 10111110.

(7). Here, +42 in 1's Complement is:- 00101010. Complementing to get the negative number gives:- 11010101. To get the 2's Complement, we must add 1 i.e.:-

```
  11010101
+ 00000001
  --------
  11010110 - the answer.
```

"KOBRAHSOFT Z80 MACHINE CODE COURSE"

CHAPTER (4)

We continue this month by examining the use of more groups of mnemonics. The next being:-

The INCrement and DECrement Group.

The general forms of the instructions are:-

INC r - where r is a single byte (8-bit register).
DEC r
INC rr - where rr is a 2 byte (16-bit REGISTER PAIR).
DEC rr

INC is short for INCREMENT or "increase the value by ONE"; DEC is short for DECREMENT or "decrease the value by ONE". Note, again, how the register containing an 8-bit number is represented by a SINGLE letter, while the register containing a 16-bit number is represented by TWO letters. This is the standard convention which you will find over and over again.

Thus, we can have:-

INC H, INC D, INC A, INC L, INC E, INC B etc.
INC HL, INC DE, INC BC, INC IY, INC IX etc.

In each case, the meaning is "Increase the value of the number in the register or register pair by ONE". Thus, if the H register contained, say, 55H (85 decimal); after executing the INC H instruction, it would contain 56H (86 Decimal).

Suppose the register pair DE contained the number 2FA2H (can you convert this to decimal? - it is in fact 12194 Decimal) - on executing an INC DE instruction, it would contain 2FA3H (12195 Decimal). NOTE that with a register pair and the corresponding 16-bit number, the LOW ORDER BYTE is incremented ONLY i.e. the byte in the L Register.

Similarly, we can also have:-

DEC H, DEC D, DEC A, DEC L, DEC E, DEC B etc
DEC HL, DEC DE, DEC BC, DEC IY, DEC IX etc.

where the meaning is "Decrease the value of the number in the register or register pair by ONE". Using the above examples, if the H register contained 55H (85 Decimal); after executing the DEC H instruction, it would contain 54H (84 Decimal).

Again, for the DE register pair, if it contained 2FA2H (12194 Decimal); after the DEC DE instruction it would contain 2FA1H (12193 Decimal).

We can also increase the value of any memory location, using an instruction such as:-

INC (HL)

REMEMBER - this has a totally different meaning from INC HL. The instruction INC (HL) means "Increase the value of the number contained in the location in HL by ONE". Thus, if HL contained 5800H (22528 Decimal), and suppose the location 5800H contained the number 15H (21 Decimal); after the instruction INC (HL), HL would STILL contain 5800H, but the MEMORY LOCATION 5800H would NOW contain 16H (22 Decimal). A shorter way of stating this is " Increase the contents of the memory location pointed to by HL by one" i.e. the brackets in INC (HL) mean "the CONTENTS of the memory location contained in HL".

NOTE:- While INC HL acts on the 16-bit number in HL, INC (HL) acts on the 8-bit number stored in the location contained in the HL register pair.

Similarly, we can also have:-

DEC (HL)

- with the same meaning, only DECREASING the value. We can also have, for the Index Registers:-

INC (IX+d)
INC (IY+d) - where d is the displacement
DEC (IY+d) from the start address.
DEC (IY+d)

NOTE:- ONLY INCrease and DECrease operations on 8-bit numbers affect the FLAGS. The same operations on 16-bit numbers DOES NOT AFFECT THE FLAGS.

Since the Flags are affected when 8-bit numbers are involved, we will now review the operation of the flags:-

SIGN FLAG:- This flag will be SET (= 1), if bit 7 of the 8-bit RESULT is 1.

ZERO FLAG:- This flag will be SET (= 1), if the 8-bit RESULT is ZERO.

OVERFLOW FLAG:- This flag will be SET (= 1), if the CONTENTS OF BIT 7 of the 8-bit number is changed by the operation.

HALF-CARRY FLAG:- This flag will be SET (= 1), if there is a CARRY INTO, or a BORROW FROM bit 4 of the 8-bit number.

NEGATE FLAG:- This flag is SET (= 1), if the last instruction was a subtraction. Thus, it is NOT SET (i.e.=0) for "INC" and SET (= 1), for "DEC".

SINGLE BYTE (8-BIT) ARITHMETIC.

We give below a table of the operations for 8-bit arithmetic. As you will see, these are of THREE main types i.e. ADDING, SUBTRACTING and COMPARING:-

MNEMONIC	NO. OF BYTES	TIME TAKEN	EFFECT ON THE FLAGS					
			C	Z	PV	S	N	H
ADD A,register	1	4	X	X	X	X	0	X
ADD A,number	2	7	X	X	X	X	0	X
ADD A, (HL)	1	7	X	X	X	X	0	X
ADD A, (IX+d)	3	19	X	X	X	X	0	X
ADD A, (IY+d)	3	19	X	X	X	X	0	X
ADC A,register	1	4	X	X	X	X	0	X
ADC A,number	2	7	X	X	X	X	0	X
ADC A, (HL)	1	7	X	X	X	X	0	X
ADC A, (IX+d)	3	19	X	X	X	X	0	X
ADC A, (IY+d)	3	19	X	X	X	X	0	X
SUB register	1	4	X	X	X	X	1	X
SUB number	2	7	X	X	X	X	1	X
SUB (HL)	1	7	X	X	X	X	1	X
SUB (IX+d)	3	19	X	X	X	X	1	X
SUB (IY+d)	3	19	X	X	X	X	1	X
SBC A,register	1	4	X	X	X	X	1	X
SBC A,number	2	7	X	X	X	X	1	X
SBC A,(HL)	1	7	X	X	X	X	1	X
SBC A, (IX+d)	3	19	X	X	X	X	1	X
SBC A, (IY+d)	3	19	X	X	X	X	1	X
CP register	1	4	X	X	X	X	1	X
CP number	2	7	X	X	X	X	1	X
CP (HL)	1	7	X	X	X	X	1	X
CP (IX+d)	3	19	X	X	X	X	1	X
CP (IY+d)	3	19	X	X	X	X	1	X

NOTE:- Here, the flags notation represents:-

X - indicates flag is altered by the operation.
0 - indicates flag is set to 0 (RESET).
1 - indicates flag is set to 1 (SET).

The time taken is expressed in "T" States, where 1 "T" State = 0.5 millionths of a second!

The point to remember is that:- ALL 8-bit arithmetic operations must be performed using the A Register (Accumulator). This is such a strong convention in Z80 machine code mnemonics, that the abbreviation "A" is even omitted in some mnemonics. That is, to subtract "B" from "A", we would expect to see:-

SUB A,B

- whereas we actually find:-

SUB B

Despite this limitation on all arithmetical instructions being restricted to the A register, the Z80 language is still very versatile in what we can actually add to whatever number we have in the Accumulator i.e. we can have:-

```
      ADD A,r   - Add any single register to A.
      ADD A,n   - Add any 8-bit number to A.
    ADD A,(HL)  - Add the 8-bit number in the location whose address is in HL.
  ADD A,(IX+d)  - Add the 8-bit number in the location whose address is in IX+d.
  ADD A,(IY+d)  - Add the 8-bit number in the location whose address is in IY+d.
```

As you can see, we have a very versatile range of possible numbers we can add to whatever number is stored in the Accumulator - any number, any register, and virtually any way we care to define a memory location.
The one which is missing is:-

ADD A,(nn)

- where we define the address in the course of the program. We can get around this problem by writing:-

```
LD HL,nn
ADD A,(HL)
```

Note, again, the favoured role of the HL register pair! We CANNOT specify the memory location using the BC or DE register pairs.
The other limitation implicit in all this is also the inherent limitation of 8-bit numbers which can only hold values up to 255, as we have already seen. For example, if we execute the instructions:-

```
LD A,80H
ADD A,81H
```

- we get a result of 1 in the Accumulator, but the carry flag will be set to indicate that the total of the two numbers exceeded 255 i.e.

```
  80H
+ 81H
 -----
 101H  - since 8 + 8 = 16 or 10H.
```

- the extra 1 of the 101H having the effect of setting the carry flag. This would normally go unnoticed, but for the fact that there is another instruction which also takes into account any setting of the carry flag i.e.:- "ADC" or "add with carry". Here, the overflow is recorded by the carry flag being set. Using this instruction, we can add numbers greater than 255 together with a chaining operation i.e.:- Suppose we want to add 3E8H (1000 Decimal) to 7D0H (2000 Decimal), then store the result in the BC register pair:-

```
 LD A,E8H   - Lower part of first number.
ADD A,D0H   - Lower part of second number.
 LD C,A     - Store result in C register.
 LD A,3H    - Higher part of first number.
ADC A,7H    - Higher part of second number.
 LD B,A     - Store result in B register.
```

Here, after the first addition of E8H and D0H, we will have the carry flag set because the result was greater than 255. Also, the Accumulator will contain B8H - check this for yourself. You might expect the second addition of 3+7 (Decimal) to give 10 (Decimal) or AH. In fact, we get 11 (Decimal) or BH - because of the carry being set. Thus, the final result is BB8H or 3000 Decimal! We could repeat this process to consider any size of number, and store the result in memory rather than in a register pair.

8-Bit Subtraction.

This is exactly the same as 8-bit addition. Again, two sets of commands are available, one for ordinary subtraction, and one for subtraction with carry i.e.:-

SUB s - Subtract s.
SBC s - Subtract s with carry.

Again, the notation "s" refers to the same range of possible operands as for the add instruction.
Another important operation is:-

COMPARING TWO 8-BIT NUMBERS.

When we "compare" two numbers, if they are the same we say they are "equal". Another way of expressing this is to say the difference between them is zero e.g. compare 6 with 6 - they are equal. What if the second number is greater than the first? For example compare 6 with 8? Here, on subtracting 6 - 8, we see the result of - 2 is negative. Similarly, compare 8 with 6. Here, the subtraction of 8 - 6 = 2 i.e. the result is positive. We can see that the compare operation is essentially one of subtraction. We can thus devise a machine code operation for "compare", using the subtract instructions and the flags - especially the carry and zero flags.

Suppose we wish to compare a range of numbers with the number 5. We can use the instructions:-

LD A,5 - the number we have (in the Accumulator).
SUB N - the number being compared.

This gives the following results:-

If N = 5 (N equals 5) - The zero flag will be set (=1), carry flag reset (=0).
If N < 5 (N less than 5) - The zero flag is reset; carry flag reset.
If N > 5 (N greater than 5) - The zero flag is reset; carry flag set.

We can thus see that the test for equality is the zero flag, and the test for "greater than" will be the carry flag being set. Also, the test for "less than" is both flags being reset.

The only problem with this method is that the contents of the Accumulator have been changed when using the SUB N instruction.

Fortunately, this can be avoided using the "CP s" instruction, or "compare with s". NOTE:- This compares the operand "s", which can be a number n, or another register etc, with the number in the Accumulator - the important fact being that THE CONTENTS OF A ARE UNCHANGED BY THE OPERATION. The only effect is on the flags. Thus, for example, "CP 5" means "compare the number in the Accumulator with 5". Also, "CP B" - means "compare the contents of A with the contents of B", and so on.

Thus, remember that "compare" is exactly the same as "subtract" with the important difference that using "compare" leaves the contents of the Accumulator UNCHANGED.

SUMMARY.

8-bit arithmetic using the Z80 is limited to:-

ADDITION.
SUBTRACTION.
COMPARISON.

Also, it can ONLY be performed using the ACCUMULATOR.

Because of the nature of 8-bit numbers i.e. they can only hold values up to 255, we must always consider any overflow i.e. any result with a value greater than 255.

The CARRY flag is the important flag which we can use to inform us of any overflow.

Additional instructions i.e. add with carry and subtract with carry, allow us to chain arithmetical operations to deal with overflow.

"KOBRAHSOFT Z80 MACHINE CODE COURSE"

CHAPTER (5)

LOGICAL OPERATORS.

In this Chapter, we will look at the Logical Operator instructions available to the Z80. There are THREE of these i.e.:-

AND
OR
XOR

The reason these operations are important is that they operate on the individual bits of a single byte number. Consider the first - the "AND" operation:-

BIT A	BIT B	RESULT OF BIT A "AND" BIT B
0	0	0
1	0	0
0	1	0
1	1	1

- we can see that the result of an "AND" operation is to give us a 1 only if A AND B BOTH contained 1.

Thus, in machine code, if you "AND" two numbers, the result is what you would get if you "AND"ed each of the individual bits of the two numbers.

You may well ask, what is the point of such an operation?

The answer is the "AND" operation is very useful in that it allows us to mask a byte so that it is altered to contain only certain bits i.e. if, for example, we wanted to limit a particular number to lie in the range 0 - 7, we need to specify that ONLY bits 0 - 2 contain information - since if bit 3 contained information, the number would be at least 8! Thus, we have:-

```
0   0   0   0       0   1   0   1  =  5
(----------------)
These bits must be 0
```

Thus, if we take a number whose value we do not know, and apply the "AND" operation with 7, the result will be a number which lies in the range 0 - 7. e.g.:-

```
                  0   1   1   0     1   0   0   1  =  105
                  0   0   0   0     0   1   1   1  =  7  (the mask)
                  ---------------------------------------
result of "AND"   0   0   0   0     0   0   0   1  =  1  (in range 1 - 7).
```

NOTE:- the Z80 ONLY ALLOWS THE "AND" OPERATION TO TAKE PLACE IN THE ACCUMULATOR.

The Accumulator can be "AND"ed with an 8-bit number i.e. n; any of the other 8-bit registers e.g. B, C, D, etc; with (HL); with (IX+d); or with (IY+d). The instructions are shortened so the Accumulator need not be specifically mentioned i.e.:-

```
   AND 7 -  means "AND" the number in the Accumulator with 7
   AND E -  means "AND" the number in the Accumulator with the number in the E register.
AND (HL) -  means "AND" the number in the Accumulator with the number pointed to by (HL).
```

- the Accumulator is not mentioned in the instruction.

The same range of possibilities and the same restriction to using the Accumulator also apply to the other two operations, i.e. "OR" and "XOR".

The "OR" operation is very similar in concept to the "AND" operation i.e.:-

BIT A	BIT B	BIT A "OR" BIT B
0	0	0
0	1	1
1	0	1
1	1	1

It is obvious that the result of an "OR" operation is to give a 1 if either A or B contained 1.
Again, you may ask, what is the point of such an operation?
The "OR" operation is very useful in that it allows us to set any bits in a number i.e. make them = 1.
If, for example, we wished to ensure that a number was odd, then clearly we have to set bit 0. i.e.:-

```
LD A, Number
OR 1           - make number odd.
```

The above two lines would be a typical assembly listing.

The concept of "XOR" - pronounced "exclusive or" - is also easy to understand, but its actual use in programming is more limited.

The result of "XOR" is a 1 only if one of A or B contains a 1 i.e. the result is the same as for the "OR" operation in all cases except when BOTH A AND B contain 1 i.e. XOR = OR - AND i.e.:-

BIT A	BIT B	BIT A "XOR" BIT B
0	0	0
1	0	1
0	1	1
1	1	0

The last thing we must consider is the effect these operations have on the flags i.e.:-

Zero Flag:- This flag will be on (=1) if the result is zero.

Sign Flag:- This flag will be on (=1) if bit 7 of the result is set (=1).

Carry Flag:- This flag will be off (=0) after "AND", "OR", and "XOR". i.e. carry is reset.

Parity Flag:- This flag will be on (=1) if there is an even number of bits in the result i.e.:-

0 1 1 0 1 1 1 0 = OFF.
0 1 1 0 1 0 1 0 = ON.

Half-Carry Flag:- This flag will be off (=0) after "AND", "OR" and "XOR".

Subtract Flag:- This flag will be off (=0) after "AND", "OR" and "XOR".

Use of Logical Operations on the Flags:-

There is a special case for these Logical Operations - where the Accumulator operates on itself i.e.:-

```
AND A  -  A is unchanged, carry flag is cleared.
OR A   -  A is unchanged, carry flag is cleared.
XOR A  -  A is set to 0, carry flag is cleared.
```

These instructions are often popular because they require only ONE byte to do what might otherwise require two, such as LD A,0.

The carry flag often needs to be cleared, e.g. as a matter of routine before using any of the arithmetic operations such as:-

ADC - Add with carry.
SBC - Subtract with carry.

- this is easily done by using the instruction AND A, without affecting the contents of any of the registers.

SUMMARY OF THE EFFECTS OF LOGICAL OPERATORS ON THE FLAGS.

Mnemonic	Bytes	Time Taken	C	Z	PV	S	N	H
AND Register	1	4	0	X	X	X	0	1
AND Number	2	7	0	X	X	X	0	1
AND (HL)	1	7	0	X	X	X	0	1
AND (IX+d)	3	19	0	X	X	X	0	1
AND (IY+d)	3	19	0	X	X	X	0	1
OR Register	1	4	0	X	X	X	0	0
OR Number	2	7	0	X	X	X	0	0
OR (HL)	1	7	0	X	X	X	0	0
OR (IX+d)	3	19	0	X	X	X	0	0
OR (IY+d)	3	19	0	X	X	X	0	0
XOR Register	1	4	0	X	X	X	0	0
XOR Number	2	7	0	X	X	X	0	0
XOR (HL)	1	7	0	X	X	X	0	0
XOR (IX+d)	3	19	0	X	X	X	0	0
XOR (IY+d)	3	19	0	X	X	X	0	0

(The "Effect on Flags" heading spans columns C, Z, PV, S, N, H.)

The notation used is:-

X - means flag is altered by the operation.
0 - means flag is set to 0.
1 - means flag is set to 1.
- - means flag is unaffected.

USING 16-BIT NUMBERS.

So far, we have only mainly dealt with single byte (8-bit) numbers. We know that the register pairs i.e. HL, DE, BC are capable of holding 16-bit numbers which can go to 65535.

The important point to remember is that instructions using 16-bit numbers will always be much slower than 8-bit - this is because the instructions using 16-bit numbers contain 3 or more bytes which have to be fetched from memory.

The addressing modes available using 16-bit numbers are:-

Immediate Extended Addressing.

An example of this is the instruction:-

LD rr,nn

This is the equivalent of 8-bit Immediate Addressing. It is merely Immediate Addressing extended so as to accomodate 16-bit data transfer.

As stated earlier, instructions that operate on 16-bit numbers are longer and slower than those for 8-bits. The format for Immediate Extended Addressing is thus:-

Byte 1 Instruction

Byte 2 n1 - L.O.B of the number.

Byte 3 n2 - H.O.B of the number.

We use this type of addressing instruction to define the contents of a register pair, for example a pointer to a memory location.

Register Addressing.

You may recall that Register Addressing is the name we give to an instruction if the value we want to manipulate is stored in one of the registers.

The same holds true for 16-bit instructions, except that there are only a few instructions of this type in the CPU's repertoire. These are mainly relating to arithmetical operations, and are very limited in the register combinations allowed. e.g.:-

ADD HL,BC

We will mention again the preference the CPU has for its HL register pair. Some instructions can ONLY be carried out by this register pair. This is especially true of arithmetical operations, as we shall see later.

Register Indirect Addressing.

This is the name we give to instructions where the value we want is in memory, and the address of the memory location is held by a register pair e.g.:- JP (HL).

Extended Addressing.

This is similar to Register Indirect Addressing, but the value you want is not in a register pair, but in a pair of memory locations e.g.:- LD HL,(nn) - where nn must be specified at the program stage.

Examples.

(1). Immediate Extended Addressing:-

Try this small machine code program:-

Decimal Nos.	Hex Nos.	Mnemonics	Comments
1, 15, 0	01,0F,00	LD BC,FH	Load BC with FH (15 Decimal).
201	C9	RET	Return to Basic.

POKE the numbers in to a suitable location e.g. 32000 - 32003. RUN the program with:- PRINT USR 32000 (ENTER). The result is 15 i.e. the number in the BC register pair.

(2). Register Addressing:-

Try:-			
33,0,64	21,00,40	LD HL,4000H	Load HL with 4000H (16384 Decimal).
1,15,0	01,0F,00	LD BC,FH	Load BC with FH (15 Decimal).
9	09	ADD HL,BC	Add the two numbers
201	C9	RET	Return to Basic.

POKE the numbers in, and RUN with PRINT USR 32000. The result is 15! Why not 16384 +15? We did add the two numbers, but the result is stored in the HL register pair! To see the correct total, consider:-

(3). Extended Addressing:-

Try:-			
33,0,64	21,00,40	LD HL,4000H	Load HL with 4000H (16384 Decimal).
1,15,0	01,0F,00	LD BC,FH	Load BC with FH (15 Decimal).
9	09	ADD HL,BC	Add the two.
34,100,125	22,64,7D	LD (7D64H),HL	Put HL value in 32100 and 32101.
237,75,100,125	ED,4B,64,7D	LD BC,(7D64H)	Get value of BC from 32100 and 32101.
201	C9	RET	Return to Basic.

POKE the numbers in, and RUN with PRINT USR 32000. The correct result is 400FH (16399 Decimal). A better way would be to use the PUSH and POP instructions, but this illustrates another way which should be clearer.

"KOBRAHSOFT Z80 MACHINE CODE COURSE"

CHAPTER (6)

KOBRAHSOFT KD1 DISASSEMBLER

INSTRUCTIONS FOR USE

Introduction.

In this month's newsletter we have enclosed a copy of our "KSFT KD1 DISASSEMBLER" (which, incidentally, we used to sell separately for £5.95!). KD1 is an efficient and easy-to-use disassembler for your Spectrum. It can be used to list a disassembly of the Z80 mnemonics of any program; either to your T.V. screen, or to a ZX Printer. It also allows easy interchange to and from Basic, and also contains a Number Converter to convert (a) Hexadecimal to Decimal numbers, (b) Decimal to Hexadecimal numbers. Also, it contains the facility to Inspect and/or modify any section of memory - allowing you to easily enter machine code as a series of Hex numbers.

LOADING.

For the 48K Spectrum, type:- LOAD "" CODE (ENTER). For the 128K Spectrum, select 128K Basic from the Tape Loader Menu, then type:- LOAD "" CODE (ENTER) as for the 48K Spectrum. KD1 will also run on the 128K Spectrum in 48K Mode, again being loaded using:- LOAD "" CODE (ENTER).

The program will then run, displaying the message:- "To Start: Press BREAK". On pressing the BREAK key, KD1 goes into the Command Mode - with a flashing cursor at the bottom of the screen. At this stage, the following commands are available:-

Commands.

(1). Number Converter. (N).

To access the Number Converter, press "N". The message:- HEX or DEC? is displayed. To convert a Hex to a Decimal number, press "H". Now, enter your Hex Number. NOTE:- Any leading zeros MUST be included i.e. enter 38 Hex as 0038, etc. Any normal Hex letters (A - F) may also be entered. On pressing ENTER, the result is shown.

To convert Decimal to Hex, press "N", then "D" for Decimal. Now, enter your Decimal number WITHOUT leading zeros. Pressing ENTER gives the result.

(2). Inspection of Memory, and/or Insertion of Hex Numbers. (I).

By pressing "I", you have two functions available:-

(a) Inspection of memory - after pressing "I", the letter "M" appears. If you now enter a four digit Hex number (again including leading zeros), on typing the fourth digit, the address is shown, together with a two digit number to its right. This is the content of that memory location. On pressing ENTER again, the next location is shown in a similar way. Thus, by continually pressing ENTER, a range of locations can be studied. To go back to the Command Mode, press "X".

(b) Insertion of Hex numbers in memory - after pressing "I", and typing in your memory location, you can change its contents by typing in a 2 digit Hex number (again using leading zeros). For example, suppose you press "I", then type 7D00 to see the contents of location 7D00 Hex. Suppose this shows 00. If you now type, say, F3; this number is shown and is inserted in that location - check by going back to Command Mode (press "X"), press "I", then 7D00, followed by ENTER - you will see F3 at 7D00! This is a convenient way to enter a machine code program as Hex numbers. Again to return to Command Mode, press "X".

(3). Disassembler. (D).

NOTE:- Any section of memory may be disassembled EXCEPT that occupied by KD1. Any attempt to disassemble this area will produce the message:- "INVALID ADDRESS". To enter the disassembler mode, press "D". Next, type (in 4 Hex digits) your required start address. You can now also similarly specify a 4 Hex digit end address. If you don't, the only difference is that KD1 will continue its disassembly up to 65535 (FFFFH), which is probably inconvenient, especially when outputting to your printer! In the case of no specified end address - press ENTER. You are then asked if you require a printout to the ZX Printer. If yes - press "/", if no - press "N", when the listing will appear on the screen. To list more - press ENTER. To end the listing - press "F". When an end address is specified - ENTER is not needed - the enquiry PRINT? appears on pressing the last digit. NOTE:- To get "/", press Symbol Shift and "V" keys. If you do not specify an end address, you can stop a continual printout to a ZX Printer by pressing "BREAK". This will return you to Basic, with the usual error message.

(4). Return to Basic. (P).

For a NORMAL return to Basic, press "R" while in Command Mode - you will be returned to Basic. To re-enter KD1 from Basic, type:- RANDOMISE USR 59625 (RUN USR 59625 on the 128K Spectrum), then press "BREAK". This will take you back into the Command Mode.

As you have seen, KD1 is an easy-to-use, but quite powerful disassembler. We suggest you try it on any sources of machine code you can get hold of. As an exercise, why not try disassembling your Spectrum's ROM? This is easily done by using a start address of 0, or 0000! Also, check any Basic Loaders in any of your games - if on listing they contain a REM which gives a lot of unusual characters - it is likely to contain machine code. Try disassembling the area around the start of Basic i.e. 5CCBH or 5D05H. You could also try entering the "Screen Mover" routine we showed earlier which uses the block move command LDIR.

"KA1 EDITOR/ASSEMBLER"

INTRODUCTION.

KA1 is a multi-purpose Editor/Assembler program which allows you to enter and edit the source code (mnemonics and label names), produce the object code (the Hex machine code), and create hard copy on the ZX Printer, or cassette copies on tape. It operates in two distinct parts. Firstly the EDITOR allows entry of the Source Code by means of line numbers in a similar fashion to a Basic program, and allows editing of that source code. In the second part of the operation, the Source Code is assembled into the actual Hex code (object code).

LOADING KA1.

For the 48K Spectrum:- Type LOAD "" and PLAY in the program.

For the 128K Spectrum:- Select 48K Basic, and proceed as for 48K Spectrum.

THE EDITOR.

Having loaded KA1, the program will ask whether you wish to use NEW TEXT or to CONTINUE with existing text. As at the start there will be no text (listing) in the text buffer, press "N" in response to the prompt. (See later for the CONTINUE option). The message will disappear, and the cursor will appear in the lower left corner of the screen. KA1 is now ready to use.

USING THE KEYBOARD IN KA1.

As the number of commands accessible from KA1 is much smaller than the number from Basic, the keyboard operates in a simplified manner. EXTENDED mode is not available, nor is the GRAPHICS mode, or CAPS LOCK mode. KA1 knows when lower case letters may be needed, and displays capitals at all other times. Wherever possible the KA1 commands are accessed by the same keys as the equivalent keywords in Basic. These command names are shown below, and are only available when the cursor is at the left hand side of the bottom line:-

LIST:- on Key K.
EDIT:- on Key 1 with Caps Shift.
SAVE:- on Key S.
LOAD:- on Key J.
NEW :- on Key A.
CLEAR:- on Key X.
RETURN:- on Key Y.

Other KA1 commands that have no Basic equivalent or are not Basic keywords are as follows, and are also only accessible with the cursor at the left end of the line:-

RENUM:- use Key N.
VERIFY:- use Key V.
AUTO:- use Key D.
ASSEMBLE:- use Key R.

Cursor control functions are accessible wherever the cursor is positioned, and are:-

<-- on Key 5 with Caps Shift.
--> on Key 8 with Caps Shift.
DELETE on Key 0 with Caps Shift.
ENTER on Key ENTER.

All other alpha-numeric keys operate as normal, giving the appropriate letter or number. Use of the Symbol Shift with a key will give the red symbol shown on that key top, providing it is a single character symbol i.e. "!" is available from Key 1 plus Symbol Shift, but using Key "U" plus Symbol Shift will produce no result since "OR" is not a single character symbol. Symbols normally accessed by the use of the Extended Mode in Basic are not accessible from KA1. The only time you will need to use CAPS SHIFT to access a capital letter is when you are entering a message line in KA1. At all other times, KA1 automatically displays capital letters.

THE SCREEN DISPLAY.

The screen display of KA1 is divided into fields. This produces an ordered form of listing that is very easily readable.

The cursor control functions automatically recognise the boundaries between the fields, making entry of a line into the text buffer a simple operation.

ENTERING A LINE.

Only ONE instruction per line is allowed, and the line must contain a line number between 0 and 9999. The program line may then next contain a LABEL to identify the instruction in this line. The 4 fields are:-

Line No.	-	1st 4 characters + 1 space.
Label Name	-	Next 5 characters + 1 space.
Operation Name	-	Next 4 characters + 1 space.
Operands	-	Rest of line.

With the cursor to the left, type a line number e.g. 10. Press SPACE, this advances the cursor to the label field. For no label press SPACE again to go to the Op Name field. All programs MUST start with an ORIGIN address, defined by ORG. Type ORG, press SPACE to move to the Operand Field. Type 7D00H to specify 32000 Decimal. Press ENTER and the line is complete. Using the 4 fields into which each line is divided, always enter a line number; then a label if needed (max. 5 chars); enter the Op Name (CALL, LD etc); enter the operand into the last field. With 2 operands, they must be separated by a comma, e.g.:-

```
0010        ORG   7D00H
0020        LD    A,20H
0030        END
```

NOTE:- Each program MUST have an END instruction.

EDITOR COMMANDS.

(1). LIST.

Press the K key, then ENTER. For more listing, press SPACE. Press "K" then a line number then ENTER lists from that line. To list to a ZX Printer, press "K", then / (Symbol Shift and Key "V") - ENTER lists to the printer.

(2). AUTO.

Press Key "D" - gives AUTO. Type a line number then ENTER. The line numbers will be automatically displayed and increase by 10 each time you enter a line.

(3). EDIT.

Press Caps Shift and Key 1 - gives EDIT. Adding a line number edits that line. Edit as for Basic.

(4). NEW.

Press the "A" Key and ENTER. This clears all the text from the text buffer.

(5). RETURN.

Press "Y" then ENTER - returns you to Basic. To return to KA1, type R.USR 58926 ENTER.

(6). RENUM.

Press "N" - gives RENUM. Now, type the new interval between lines (1 to 99). Press ENTER and the program is renumbered.

THE ASSEMBLER.

The assembler part of KA1 converts the mnemonics into Hex code. The mnemonics in the listing are called the Source Code, the assembled Hex code is the Object Code.

OPERANDS.

NUMBERS.

KA1 will accept Decimal or Hex numbers, and defaults to Decimal. Numbers MUST start with a numeric digit i.e. 0 to 9, and Hex numbers must have a suffix H. If a Hex number starts with a letter (A to F), it must be preceded by a zero i.e. D000 Hex must be written 0D000H, and FF Hex as 0FFH.

JR/DJNZ.

Such jumps as these must be within the range +127 to -128.

LABELS.

Any label can be used, to a maximum length of 5 characters.

ASSEMBLER DIRECTIVES.

ORG.

This specifies the start address in memory where the code will reload with LOAD""CODE. ORG must be the first line you enter

END.

This specifies the end of the program and MUST be included.

DEFB.

Defines a single byte at the current address. Must be in the range 0-255.

DEFW.

Defines the next two bytes at the current address in the range 0 - 65535.

COMMANDS.

ASSEMBLE.

Press "R" - gives ASSEMBLE. 3 choices are available:-

(1). Press "R" then ENTER. This gives an assembly with no display and is the fastest.

(2). Press "R" then Symbol Shift and "V" (gives /). Pressing ENTER gives an assembly to the ZX Printer.

(3). Press "R" then "S" then ENTER - gives an assembly to the screen in full detail. PAUSE with "M".

CLEAR.

Press "X" then ENTER. This clears the text buffer.

CONTINUE.

Pressing "C" in response to "New Text or Continue?" does not alter the text when returning from Basic. "N" clears it.

CASSETTE ROUTINES.

SAVE.

You can save either the Text or Object buffer to tape by pressing "S" i.e.:-

(1). TEXT BUFFER:- The form is SAVE"name" T. Saves the text to tape.

(2). OBJECT BUFFER:- The form is SAVE"name"C. Saves the object buffer as bytes to tape.

VERIFY.

Press "V". The form is VERIFY"name". There is no need to specify text or code.

LOAD.

The KA1 will ONLY load TEXT files. 2 options are available:-

(1). Press "J". The form is LOAD"name"N. The "N" means NEW - the existing text buffer is cleared, as is the label table. "N" need not be specified.

(2). Again press "J". The form is LOAD"name"C. Here "C" means CLEAR - the text buffer is cleared, but the label table is protected.

We hope you enjoy using your KA1 assembler, and find it useful in writing machine code routines.

"KOBRAHSOFT Z80 MACHINE CODE COURSE"

CHAPTER (7)

USING 16-BIT NUMBERS.

We will now discuss the use of 16-bit (2 byte) numbers in machine code.

As we discussed earlier, the largest number that can be represented using 8 bits is 255. Clearly, this would be a serious limitation, particularly if we wanted to specify an address in the region 256 to 65535. Thankfully there is a way around this problem in the Z80 chip, using 16-bit (2 byte) numbers to represent addresses in the range 256 to 65535.

SPECIFYING ADDRESSES USING 16-BIT NUMBERS.

PLEASE NOTE:- All addresses MUST be specified by a 16-bit number, even if it is in the range 0 to 255. This is because the CPU is set up to recognise that a number is not to be taken as an address unless it is specified by 2 bytes of 8 bits each. It can be regarded as a convention, if you like.

You may remember that this was implied by the use of the instruction:-

LD A, (nn)

You must also remember that 16-BIT NUMBERS ARE STORED IN REGISTER PAIRS HIGH NUMBER FIRST, LOW NUMBER SECOND.

In complete contrast to this is the convention of:-

STORING 16-BIT NUMBERS IN MEMORY.

Here, the convention used is COMPLETELY OPPOSITE TO THAT ABOVE, i.e. WHEN LOADING 16-BIT NUMBERS INTO MEMORY, THE LOW BYTE IS ALWAYS STORED FIRST IN MEMORY!

Consider the example of placing the contents of the HL register pair into memory. Suppose that the HL register pair contains the number 7D00H or 32000 Decimal. As we have stated, in the registers, H contains the HIGH byte, L the LOW byte. (Similarly, for BC, B contains the HIGH byte, C the LOW byte; also for DE and AF). Thus we can show this as:-

H	L	
7D	00	(Hex).

This result would be achieved, for example, by executing the instruction:-

LD HL,7D00H

Now suppose we wish to place this number in memory at, say, address 8000H or 32768 Decimal. This can be done using the instruction:-

LD (Address), HL

where (Address) is (8000H). Remember, the brackets mean 'the location pointed to'. On executing this instruction, we would find:-

Memory Location	Contents	
32768	00	(Decimal)
32769	125	(Decimal)

i.e. LOW byte FIRST, HIGH byte SECOND! Again, this is simply a convention which means it is the way the Z80 has been programmed, but it is MOST IMPORTANT that you remember it. Again it is :-

IN REGISTERS:- HIGH byte FIRST.
IN MEMORY AND PROGRAMS:- LOW byte FIRST.

Thus, please remember when using 16 bit numbers, ALWAYS CONSIDER CAREFULLY THE ORDER OF THE BYTES.

We will use the above as an example. The whole program is:-

```
LD HL,7D00H
LD (8000H), HL
RET
```

Remember, don't forget the RET instruction - this is to ensure we return to Basic! The machine code instructions are:-

Mnemonics	Instructions (Hex. Nos.)	Decimal Nos.
LD HL,7D00H	21,00,7D	33,00,125
LD (8000H), HL	22,00,80	34,00,128
RET	C9	201

NOTE:- This is the way the program instructions must be stored in memory. The program can be entered into memory:-

(1). By POKING in the values to a suitable location using a Basic program such as:-

```
10: FOR A=30000 TO 30006:INPUT B:POKE A,B:NEXT A
```

(2). Use the KD1 Disassembler we supplied in the last newsletter. Use the "I" facility to insert the Hex. numbers starting at address 7530H (30000 Decimal).

Having entered the program into memory, run it by typing RANDOMIZE USR 30000 (or RUN USR 30000 on the 128K Spectrum). NOTE:- The 128K Spectrum sometimes won't accept the RANDOMIZE USR command.

Examine the memory locations at 8000H and 8001H (32768 and 32769) using PRINT PEEK 32768, then PRINT PEEK 32769. You will find:- 32768=0; 32769=125 (if you have done it correctly!).

We apologise for going into so much detail, but this is a difficult concept which MUST be well understood to avoid mistakes.

OTHER 16-BIT LOAD INSTRUCTIONS.

As well as using the register pairs, we can also load 16-bit numbers into the Index Registers IX and IY using:-

```
LD IX, nn
LD IY, nn
```

We can also interchange numbers to and from memory and register pairs using:-

```
LD (nn),rr
LD (nn), IX
LD (nn), IY
```

where nn is a 16-bit number, rr is a register pair.

We can also have:-

```
LD rr, (nn)
LD IX, (nn)
LD IY, (nn)
```

- the reciprocal of the above instructions.

EXERCISE:-

Check your understanding of the previous examples by finding the contents of the System Variable PROG, which is a two byte number stored at address 23635 (5C53H). This will be either 23755 (5CCBH), or 23813 (5D05H), depending on whether a microdrive is fitted or not.

We can do this from Basic using:-

```
PRINT PEEK 23635 + 256*PEEK 23636
```

We will now do the same thing using the following machine code program:-

Mnemonics	Machine Code Instructions (Hex and Dec Nos.)	
LD BC, (5C53H)	ED,4B,53,5C	237,75,83,92
RET	C9	201

Again enter the numbers into memory as before, and execute the program by typing PRINT USR 30000. The number 23755 or 23813 should appear on your screen.

The reason for this is that, when a PRINT USR command is used, on returning to Basic, the number in the BC register pair is printed. In this case it is our answer - the address contained in the System Variable PROG.

NOTE:- The LD BC,(nn) instruction is a FOUR BYTE instruction!

EXERCISES.

Use a similar method to the above to determine the values of the addresses stored in other System Variables (see list on P. 173 of your Spectrum Manual. Remember - some are only one byte numbers! Check your answers using the PRINT PEEK method. Also, POKE a few values into different memory locations and see if you get the correct results using the above methods.

16-BIT ARITHMETIC.

In this we again have a favoured register pair - the HL register pair. However, we do not have such a wide range of options as in 8-bit arithmetic!

16-BIT ADDITION.

The possible operations are:-

```
ADD HL,BC
ADD HL,DE
ADD HL,HL
ADD HL,SP
```

These are the only options! Note that we cannot add an absolute number to HL e.g. "ADD HL,nn" is not permitted. We can get around this using:-

```
LD DE,nn
ADD HL,DE
```

However, this uses up four 8-bit registers - not very elegant! Also, there is no addition between HL and IX and IY. Niether is there any LOAD instruction which allows you to transfer the contents of the Index Registers to BC or DE, so we must use:-

```
PUSH IX
POP DE
ADD HL,DE
```

i.e. the contents of IX are PUSHed onto the stack, and POPped off into the DE register pair. This is then added to HL with the last instruction.

EFFECT ON THE FLAGS.

The main flag affected is the Carry flag. (The subtract flag is also affected, but its use is very limited). The carry flag will be set if there is any overflow from the high bit of the H register - any overflow from the L register automatically goes into the H register in any 16-bit calculation. We can also have "ADD WITH CARRY" or "ADC", which operates in a similar manner to "ADD" in 8-bit arithmetic, with the same register pairs i.e. :-

```
ADC HL,BC
ADC HL,DE
ADC HL,HL
ADC HL,SP
```

16-BIT SUBTRACTION.

Note that you cannot have subtraction without carry. Thus, you must always check the status of the carry flag before any 16-bit subtraction operation. It is usually best to include an instruction to clear the carry flag first - e.g. an "AND A" instruction will do this without altering the contents of the Accumulator (see Logical Operators earlier). The allowed instructions are:-

```
SBC HL,BC
SBC HL,DE
SBC HL,HL
SBC HL,SP
```

EXERCISE:-

Using the System Variable STKEND (start of free memory space), write a simple program to calculate the value for the amount of free space. (Hint:- Load HL with the value in STKEND, then subtract SP - the NEGATIVE result will be the amount of free space). See later for the solution.

EFFECT OF 16-BIT CARRY ARITHMETIC ON THE FLAGS.

Three other flags are affected by the "ADC" and "SBC" instructions which were not affected by the simple "ADD" instructions. These are the ZERO, SIGN and OVERFLOW flags. Each is set according to the result of the operation.

INDEX REGISTER ARITHMETIC.

NOTE:- The Index Registers are limited to ONLY ADDITION WITHOUT CARRY! Also, the range of registers which can be added to them is very limited i.e.:-

```
ADD IX,BC
ADD IY,BC
ADD IX,DE
ADD IY,DE
ADD IX,IX
ADD IY,IY
ADD IX,SP
ADD IY,SP
```

SOLUTION TO MEMORY LEFT EXERCISE:-

First, we must load HL with the contents of the System Variable STKEND i.e. :- LD HL, (STKEND).
NOTE:- The value of STKEND is 5C65H (23653 Decimal). Now we must subtract SP using SBC HL,SP. However, REMEMBER we must clear the carry flag first using AND A!

However, because the Stack Pointer is (usually) higher in memory than the top of your program, the result will be negative. To convert this to a positive value, we can use the BC register pair (we could also use DE). Firstly, we must shift HL to BC - but there is no appropriate LOAD instruction, but we can get around this using the stack by using a PUSH then a POP thus:-

```
PUSH HL
POP BC
```

- this keeps HL the same, but transfers its contents to BC. To get HL = -BC i.e. the negative value of BC, we must subtract BC from HL TWICE. Remember, though, that the carry flag has just been set by the SBC HL,SP (since SP is greater than HL), so we must clear it again using "AND A". HL now contains the negative value of its previous contents or the positive number of bytes left.To get HL back into BC (to get a result from the USR function) we can PUSH HL, POP BC, and return with RET i.e.:-

```
LD HL, (STKEND)
AND A
SBC HL,SP
PUSH HL
POP BC
AND A
SBC HL,BC
SBC HL,BC
PUSH HL
POP BC
RET
```

"KOBRAHSOFT Z80 MACHINE CODE COURSE"

CHAPTER (8)

USING JUMPS AND LOOPS IN MACHINE CODE.

A jump in machine code is the Basic equivalent of the "GOTO" instruction, i.e. the program jumps to the line in Basic, or the specified address in machine code. This is a very powerful instruction, but you MUST ensure that there is a meaningful machine code instruction at the jump address, otherwise unpredictable results will follow!

There are TWO types of "jump" instruction available in machine code; these are:-

(1). The ABSOLUTE jump.

(2). The RELATIVE or JUMP RELATIVE jump.

The ABSOLUTE jump can be summarised as:-

JP xx xx

- where the JP is the JUMP instruction, and xx xx is a two-byte (16 bit) number. Since the whole of the Spectrum memory can be addressed by a 16 bit number, the ABSOLUTE jump allows us to jump to ANY position in the Spectrum's memory!

The jump can also be dependent on the status of one of the flags, e.g. the carry flag. We can thus have:-

JP cc, nn

or:- JP Z, 0000

This would read as "jump if zero to address zero". This is the address the Spectrum jumps to when you switch on - the computer is said to RESET. We can illustrate this with the following simple machine code program. Type POKE 32000,195; POKE 32001,0; POKE 32002,0. This puts the 3 bytes 195,0,0 at 32000. This is the simple instruction JP 0000, or RESET. Activate by typing RANDOMISE USR 32000 (ENTER) - the computer RESETS! All you have told it to do is to jump to address zero and start executing the instructions from there, which is what occurs when you switch on. The other ABSOLUTE JUMP instructions can be summarised as:-

INSTRUCTION	MNEMONICS	HEX NOS.	DECIMAL NOS.
JUMP TO ADDRESS SPECIFIED BY (HL).	JP (HL)	E9	233
JUMP TO ADDRESS SPECIFIED BY (IX).	JP (IX)	DD E9	221 233
JUMP TO ADDRESS SPECIFIED BY (IY).	JP (IY)	FD E9	253 233
JUMP TO ADRRESS SPECIFIED BY xx xx.	JP ADDR	C3 xx xx	195 xx xx
JUMP, IF CARRY TO ADDRESS xx xx.	JP C,ADDR	DA xx xx	218 xx xx
JUMP, IF MINUS TO ADDRESS xx xx.	JP M,ADDR	FA xx xx	250 xx xx
JUMP, NO CARRY TO ADDRESS xx xx.	JP NC,ADDR	D2 xx xx	210 xx xx
JUMP, NOT ZERO TO ADDRESS xx xx.	JP NZ,ADDR	C2 xx xx	194 xx xx
JUMP, POSITIVE TO ADDRESS xx xx.	JP P,ADDR	F2 xx xx	242 xx xx
JUMP PARITY EVEN TO ADDRESS xx xx.	JP PE,ADDR	EA xx xx	234 xx xx
JUMP PARITY ODD TO ADDRESS xx xx.	JP PO,ADDR	E2 xx xx	226 xx xx
JUMP IF ZERO TO ADDRESS xx xx.	JP Z,ADDR	CA xx xx	202 xx xx

This is a convenient point to digress slightly to consider how the Z80 CPU in the Spectrum handles machine code instructions. The Z80 contains a 16 bit register called the PROGRAM COUNTER (see earlier), which is used to store the adrress OF THE NEXT INSTRUCTION TO BE EXECUTED. Thus, if the CPU encounters a 2 byte instruction, it adds 2 to the program counter (usually abbreviated to PC); with a 3 byte instruction it adds 3, and so on. However, with a normal JUMP instruction, on encountering the OP CODE C3 (Hex), the CPU knows that the following 2 bytes form a 16 bit address which specifies to where it must jump. As usual, the first byte after the C3 is the LOW order byte, the second is the HIGH order byte. How does the CPU "know" it needs to get the jump address from the next 2 bytes? - it is programmed to do just that. Remember, the OP CODE is the first byte of an instruction which specifies its type i.e. a JUMP, or an ADD etc.

Returning to the absolute jump i.e. JP xx xx, it can be seen that there are two main disadvantages with this instruction - these are:-

(1). We do not always need such a "long" jump - but it still needs 3 bytes to execute.

(2). The program cannot be easily relocated to a different area of memory, since we are specifying an ABSOLUTE address i.e. one which will only give the correct result in one area of memory.

Fortunately, there is an alternative instruction in the Z80 - if the JP xx xx or ABSOLUTE JUMP is considered as a "LONG" jump, the alternative "JUMP RELATIVE" instruction or JR d is the "short jump". The "d" is the relative displacement from the current address, and since it is only specified by one byte, we can ONLY have JUMP RELATIVE jumps of -128 (i.e. jumping backwards), to +127 (i.e. jumping forward).

The general form of the instruction is :-

JR d

- where d is the displacement from the current position.

The main advantage is obvious - the distance jumped is specified in only ONE byte and the whole instruction only takes 2 bytes. The value of the displacement "d" is added to the PC, thus the PC is always pointing to the next instruction specified by the jump relative. Similarly for a case where a condition is to be met i.e. JR cc,d e.g. JR Z,d. To illustrate this more clearly, consider the program:-

Memory Location		Mnemonic
32000		ADD A,B (1 BYTE)
32001		JR Z, 2 (2 BYTES)
32003		LD B,0 (2 BYTES)
32005	Next:-	LD HL,4000H (3 BYTES)

If the JR instruction at 32001 is ignored by the CPU (i.e. the zero flag is not set), the CPU executes thus:-

(1). Load the single byte instruction at 32000. Since only 1 byte, set the PC to 32001.
(2). Execute the instruction i.e. add contents of B register to A register.
(3). Load byte specified by PC at 32001 - byte is part of a 2 byte instruction, so add 2 to PC to make it 32003.
(4). Get next byte to complete the instruction and execute the instruction - the result here is to ignore the jump, since the zero flag is not set.
(5). Load byte specified by the PC at 32003. Byte is part of a 2 byte instruction, so add 2 to PC which is now 32005.
(6). Get next byte to complete the instruction and execute the instruction i.e. load the B register with zero.

NOTE:- At location 32001 the program encounters the Relative Jump instruction, since the zero flag is not set in the above example, the jump is ignored and the next instruction is executed i.e. LD B,0.

However, if the zero flag WAS set, the CPU would ADD TWO (the amount of the displacement for the jump) to the PC, which would now make it 32005 i.e. the next instruction to be executed would be the LD HL,4000H at 32005 - under these conditions the LD B,0 instruction would be OMITTED or jumped past. Thus, if the zero flag is SET, the LD B,0 is jumped over.

NOTE:- We can also have a NEGATIVE JUMP RELATIVE. As an exercise, consider what would the effect be on the above program of substituting a JR Z,-2 instruction for the JR Z,2? For the answer, see the bottom of page 4! (Hint:- Remember, the PC will be pointing to 32003 again, since the JR Z,-2 is still a 2 byte instruction).

Also, can you see how the displacement d would be represented as -2, a NEGATIVE number?

We can have a range of jump relative instructions for various flag conditions also, as for the absolute jumps. However, the range available is here limited to the following:-

Instruction.	Mnemonic	Hex Nos.	Decimal Nos.
JUMP RELATIVE, displacement.	JR, d	18 xx	24 xx
JUMP RELATIVE IF CARRY, displacement.	JR C, d	38 xx	56 xx
JUMP RELATIVE NO CARRY, displacement.	JR NC, d	30 xx	48 xx
JUMP RELATIVE IF ZERO, displacement.	JR Z, d	28 xx	40 xx
JUMP RELATIVE NOT ZERO, displacement.	JR NZ, d	20 xx	32 xx

Using "For...Next" Loops in Machine Code.

You are probably familiar with the usual form of a "For...Next" loop in Basic, i.e.:-

```
10: FOR I = 1 TO 6
20: LET C = C + 1
30: NEXT I
```

A similar case exists in machine code, but in a slightly different form. We could do this in machine code thus:-

```
        LD B,1          ; Set counter to 1.
        LD A,7          ; Maximum of counter + 1.
LOOP    INC C           ; Let C = C + 1.
        INC B           ; Increment counter.
        CP B            ; Is B = A?
        JR NZ, LOOP     ; If not loop again.
```

This would work, but it is a very bad piece of programming, since we are tying up 2 register pairs - one to increase, and one to hold the maximum; and the instruction which increments the counter does not set any flags on completion. A much better way would be to use a countdown! We know we have to do the loop 6 times, so why not set the B register to 6 and count down? This gives us:-

```
        LD B,6          ; Set counter.
LOOP    INC C           ; Let C = C + 1.
        DEC B           ; Decrease counter.
        JR NZ, LOOP     ; Loop if not finished.
```

As you can see, this is far more efficient.

NOTE:- The Z80 CPU has a special instruction which combines the last two lines above i.e.:-

```
DJNZ d
```

This stands for "decrement the B REGISTER, and jump if not zero". The d is the relative displacement as in the JR instruction. It is a TWO BYTE instruction, which saves one byte on the above coding.

Because of the existence of this special instruction, the B register is usually used as a counting register.

The obvious limitation of the "DJNZ" instruction is that it can only count up to 256. However, DJNZ instructions can be nested in a similar way as for the Basic For..Next loops:-

```
          LD B, 10H        ; B = 16 Decimal.
BIGLOOP   PUSH BC          ; Save value of B register on Stack.
          LD B,0           ; Set B = 256
LITLOOP   ......
                           ; Any required calculation etc.
          ......
          DJNZ LITLOOP     ; Done 256 times?
          POP BC           ; Get back value of B.
          DJNZ BIGLOOP     ; Do bigloop 16 times.
```

A useful exercise for you to try is to write down what would appear in each register after each instruction in the above program.

Waiting Loops.

There are times in machine language programs when things happen so fast that it is necessary to wait for a short time. A waiting loop can be set up using:-

```
       LD B, Count
WAIT   DJNZ WAIT
```

The instruction "DJNZ WAIT" will cause the CPU to jump back to the DJNZ instruction as many times as required to set the B register back to zero before continuing.

USE OF SUBROUTINES IN MACHINE LANGUAGE.

The use of subroutines involves the use of the CALL and RET instructions. The main point to remember is that when you CALL a subroutine, you MUST have a RET if you are to get back to the point you expect!

The action of a CALL instruction adds 3 bytes (the length of the instruction) to the PC, which is then PUSHED onto the stack to save it. Similarly, when the CALLed routine reaches a RET instruction, the contents of the last number PUSHED onto the stack is POPped back off into the PC, causing a return to the instruction immediately AFTER the original CALL.

As you can see, when you enter a CALLed subroutine, you MUST be very careful to ensure that any PUSHes in the routine equal the number of POPs, to ensure a correct return from the subroutine. Consider the program:-

Address	Mnemonic
32000	LD HL,4000H
32003	CALL 9000H
32006	LD DE,700H

The CPU will execute the program thus:-

(1). Load byte at 32000 - a 3 byte instruction, so add 3 to the PC which = 32003. Get next 2 bytes to complete the instruction then execute the instruction - load the HL register pair with 4000H.

(2). Load byte specified by the PC (32003), byte is part of a 3 byte instruction, so add 3 to the PC (= 32006). Get next 2 bytes and execute the instruction. Since it is a CALL, PUSH the contents of the PC (= 32006) onto the stack and go to the address specified in the CALL i.e. 9000H.

(3). On reaching the RET in the subroutine at 9000H, POP the PC of the stack (=32006) and jump there.

(4). Continue starting at LD DE,700H (at 32006).

Subroutines can also be called conditionally e.g. according to the status of a flag. The only flags involved here are the CARRY FLAG, the ZERO FLAG, the PARITY FLAG and the SIGN FLAG.

Care should be taken when in a subroutine to also not affect any flags or registers which are needed for the next comparisons. This is so that you don't branch off again to the wrong place on a following CALL, after returning from where you left off. Remember that the four flags above are set according to the last instruction which affected that particular flag. Thus, it is a good idea to have a CALL or RETurn instruction immediately AFTER the instruction which sets the flag e.g.:-

```
LD A, (Number)
CP 1
CALL Z, Routine 1
CP 2
CALL Z, Routine 2
```

This program allows you to jump to various routines depending on the value stored in the memory location "Number". NOTE:- It assumes that the subroutines do not change the value in the A register, otherwise on returning from one subroutine an incorrect branch may occur for the remaining ones, since the value for A from "Number" is not reloaded on returning from the subroutines. The value of A CAN be allowed to change in the subrotines IF the instruction LD A, (Number) is added after each CALL.

The solution to the JR -2 instruction is that it would cause a jump back to the JR Z, -2 instruction if the zero flag was set, causing an endless loop, since the JR instruction does not affect the flags.

The -2 number would be represented by having d = FE Hex.

"KOBRAHSOFT Z80 MACHINE CODE COURSE"

CHAPTER (9)

MOVING BLOCKS OF MEMORY AROUND.

We now come to the very useful and extremely powerful "BLOCK MOVE" instructions which we can use to move whole blocks of memory from one area to another - remember the use of LDIR earlier to move the WHOLE screen into place in a fraction of a second?

The simplest of the block instructions is:-

CPI

Here, the contents of the A register are compared with (HL) i.e. the contents of the location pointed to by HL. It also decrements the BC register pair, and checks to see if it is zero. It is thus used as a counter. If BC is not zero, the next (HL) is compared to A - the HL register having been automatically incremented - hence "compare and increment". The usual procedure is thus to load HL with the start of the block we wish to check, and BC with the number of bytes to be checked.
Let us assume the block length is less than 255 - the count can thus be stored in the C register only of the BC register pair. A suitable program would be:-

```
Search      CPI
            JR Z, Found
            INC C
            DEC C
            JR NZ,Search
Not Found   ........
            ........
Found       ........
```

We would obviously need a different routine if the length of the block was more than 255 bytes, but here we can use the INC and DEC instructions to check if C = 0. These two instructions only need one byte each, and as they both affect the zero flag, the net effect is to set the flag only if C was originally zero. The other benefit is that this method does not alter any of the other registers.
We might also wish to search a block of memory starting from the top rather than from the bottom - this is done using the instruction:-

CPD

- this reads "compare and decrement". Here HL is decremented, as also is BC as before.
More powerful still are the instructions:-

CPIR
CPDR

These read as "compare, increment and repeat", and "compare, decrement and repeat". The instructions each only take 2 bytes. They allow us to continue searching through a block of memory until either (a) a match is found i.e. the contents of A = the contents of (HL); or (b) the end of the block is reached. Thus, we need to include some code which will tell us which of these 2 possibilities has occurred.

The next instructions deal with moving blocks of memory around i.e.:-

LDI
LDIR
LDD
LDDR

LDI reads as "load and increase"; LDIR reads as "load increment and repeat"; LDD reads as "load and decrement"; LDDR reads as "load decrement and repeat".

Consider each instruction in more detail - the actual operations involved are:-

LDI:- (HL) --> (DE); increment HL and DE; decrement BC.

LDD:- (HL) --> (DE); decrement HL, DE and BC.

LDIR:- (HL) --> (DE); increment HL and DE; decrement BC - repeat until BC = 0.

LDDR:- (HL) --> (DE); decrement HL, DE and BC - repeat until BC = 0.

Points to note are:-

(1). These are the only instructions which will load from one memory location to another, without having to be loaded into a register first.
(2). In every case, THE HL REGISTER PAIR IS THE SOURCE ADDRESS; THE DE PAIR IS THE DEstination ADDRESS (easily remembered); BC CONTAINS THE COUNT.

NOTE:- Always be aware of the addresses involved when using these instructions, since it can happen that you can overwrite the information you are trying to move! It is important to remember this!

As an exercise, try writing a routine to transfer, say, 32 bytes from the ROM (i.e. start at address zero), to the first part of the screen display (starts at 4000H or 16384 decimal).

Also, try moving 256 bytes, then 2048 bytes. Note the unusual way in which the screen is arranged!

INSTRUCTIONS FOR BLOCK COMPARE AND MOVE GROUP.

Mnemonic	Bytes	Time Taken	C	Z	PV	S	N	H
LDI	2	16	-	-	£	-	0	0
LDD	2	16	-	-	£	-	0	0
LDIR	2	21/16	-	-	0	-	0	0
LDDR	2	21/16	-	-	0	-	0	0
CPI	2	16	-	£	£	£	1	£
CPD	2	16	-	£	£	£	1	£
CPIR	2	21/16	-	£	£	£	1	£
CPDR	2	21/16	-	£	£	£	1	£

(Effect on Flags: C, Z, PV, S, N, H)

NOTES:-

£ - indicates the flag is altered by the operation.

0 - indicates the flag is set to 0.

1 - indicates the flag is set to 1.

\- - indicates the flag is unaffected.

TIMING:- For repeat instructions, the times shown are for each cycle. The shorter time indicated is for the case of the instruction terminating - e.g. for CPIR, either BC = 0 or A = (HL).

Z80 INSTRUCTIONS THAT ARE LESS FREQUENTLY USED.

We will now discuss a few Z80 instructions which are usually only occasionally used, but which are nevertheless quite useful.

REGISTER EXCHANGE INSTRUCTIONS.

We briefly discussed earlier the fact that by exchanging registers, the CPU can store information in the alternate register set, which are more accessible than using memory locations. To recap, the following alternate registers are available:-

Normal Registers.	Alternate Registers.
A F	A' F'
B C	B' C'
D E	D' E'
H L	H' L'

The important point to remember is that, while you can store a value in an alternate register, YOU CANNOT USE THEM TO MANIPULATE THAT VALUE OR DO ANYTHING ELSE! Thus, they are usually used as a temporary storage place for numerical values contained in the normal register set. A singular case of register exchange is the instruction:-

```
EX AF,A'F'
```

This means "store the value in AF temporarily in A'F', and vice versa". This instruction is usually used as a temporary place to store the value in the A register (Accumulator).

A general register exchange instruction is:-

```
EXX
```

This means "exchange the values in BC and B'C'; DE and D'E'; and HL and H'L'". NOTE:- The AF register pair is NOT affected by this instruction - it has its own as shown above!

The obvious problem with the EXX instruction is that it acts on all the three registers - we cannot retain any single value. To do this, we would need a routine of the type:-

```
PUSH HL
EXX
POP HL
```

Here, the value in HL would be retained on the Stack.

A special single case of register exchange is the instruction:-

```
EX DE,HL
```

Here, the contents of the two registers are exchanged i.e. HL to DE and DE to HL. This can be a useful way of saving the value in HL, PROVIDED of course, the value in DE does not need to be saved!

BIT, SET AND RESET INSTRUCTIONS.

So far, all the instructions we have been dealing with have involved the manipulation of 8-bit or 16-bit numbers. The "BIT, SET and RESET" group allows us to manipulate single bits in the registers and/or the contents of memory locations. However, because they are rather complex and difficult to use, they are not commonly used.

Also, it tends to take even longer to set a single bit in a register or a memory location than it does to change or examine the entire 8 bits of that register or memory location, so speed (or rather, lack of it!) is another consideration.

However, there are times when we need to know whether a bit in the middle is set or not, or even to set a certain bit. Remember, however, that many of the bit setting or resetting operations can be performed using the logical operators as discussed earlier.

The "BIT, SET and RESET" group of instructions allows us to turn any bit "on" or "off" at will, or even just look at a specified bit to see what its status is. Consider the first set of instructions:-

```
SET n, r
SET n, (HL)
SET n, (IX+d)
SET n, (IY+d)
```

The "SET" instruction turns "on" (i.e. sets = 1) the bit numbered "n" (using the notation 0 - 7, where 0 is the lowest value bit, 7 is the highest value bit), in register "r" or in the specified memory location. NOTE:- The operation does NOT affect the flags.
The "RESET" group of instructions operate on exactly the same range of registers or memory locations, but instead of turning bits "on", it turns them "off"; i.e. = 0.
The "BIT" instructions should really read as "BIT?" in English, since the function of this instruction is to TEST the contents of the indicated bit.
No actual changes are made to the registers or memory locations, but the ZERO FLAG IS ALTERED ACCORDING TO THE STATUS OF THE BIT TESTED i.e.:-

If the Bit = 0 then the ZERO FLAG IS SET ON (=1).
If the Bit = 1 then the ZERO FLAG IS SET OFF (=0).

INTERRUPTS.

An interrupt is a signal sent to the microprocessor, which may occur at any time and will generally suspend the execution of the current program without the program knowing it!
THREE Interrupt mechanisms are provided on the Z80 i.e.:-

(1). The Bus Request (BUSRQ).
(2). The Non - Maskable Interrupt (NMI).
(3). The normal Interrupt (INT).

From a programming point of view, we will only look into the usual maskable interrupt (INT).
The DI (disable interrupt) instruction is used to reset (mask), while the EI (enable interrupt) instruction is used to set (unmask).
Generally, an ordinary interrupt will result in the current program counter being PUSHed onto the Stack, followed by a branch of execution to the zero page of the ROM by the RST (restart) instruction - see later. A RETI (return from interrupt) instruction is required to return from the interrupt in the correct manner.
In normal operation, the Spectrum has interrupts enabled (EI), and in fact the program is interrupted 50 times per second. This interrupt allows the keyboard to be scanned by the ROM's routine.
You may wish to disable interrupts in your programs, as this will speed execution. You can still read the keyboard, as long as you use your own routine to do so - for examples see later.
Make sure you enable interrupts when you return from your program e.g. to Basic, otherwise the system will be unable to read the keyboard!

RESTART INSTRUCTIONS.

This is rather a "leftover" from the 8080 processor implemented for compatibility. That is why you will be unlikely to use RESTART or RST instructions in your programs.
The RST performs the same actions as a CALL (see earlier), but allows a jump to only one of EIGHT addresses in the first 256 memory locations of the ROM. These are:-

00H
08H
10H
18H
20H
30H
38H

The advantage of the RST instruction is that frequently called subroutines can be called using only ONE byte! The RST instruction also takes much less time than a CALL instruction.
The obvious disadvantage of the RST instruction is that it can ONLY be used to call one of the above eight posible locations.
As all these locations are in the ROM, you cannot gain this advantage in your own programs. It IS possible however to make use of the ROM's subroutines if you know what they do, and thus use the RST instructions.

"KOBRAHSOFT Z80 MACHINE CODE COURSE"

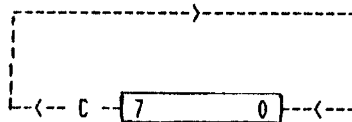
CHAPTER (10)

REGISTER ROTATE AND SHIFT INSTRUCTIONS.

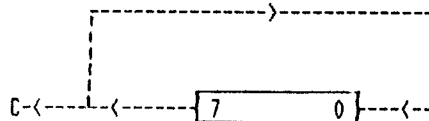
The bits in the various registers may be moved both left and right, using the appropriate instructions. The trick is to differentiate between the various shifts and rotations in order to know which one to use when. Also, we must consider the "carry" bit to be the 9th bit in the registers, i.e. the carry is bit number 8 if the bits are numbered 0 - 7 in the usual manner.

Some rotate instructions go right through the carry (as the 9th bit) so that the entire rotation goes through a cycle of 9 bits.

Consider, for example, the "RLA" instruction (for full meaning see later); the pattern of bit movement is:-



Other rotations involve only an 8-bit cycle, although the carry flag is changed according to the bit which has to go the "long way round". An example of this is the "RLCA" instruction i.e.:-



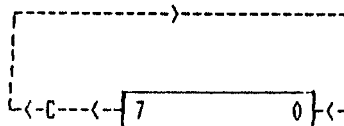
This means that in a left rotation as above, the contents of bit 0 are transferred to bit 1, bit 1 to bit 2, etc; but the contents of bit 7 are transferred to BOTH the carry bit AND to bit 0. Compare this with the "RLA" instruction above where bit 7 gets transferred to the carry bit and the carry bit gets transferred to bit 0.

LEFT ROTATIONS.

There are two main types of left rotations i.e.:-

(1). ROTATE LEFT REGISTERS.

This is a 9 bit cycle rotation as shown above for "RLA" i.e. :-



The two main types are:-

RLA - or "Rotate Left Accumulator".

RL r - or "Rotate Left Register r".

(2). ROTATE LEFT CIRCULAR.

Here, the "circular" means that the cycle is for only 8 bits as with the RLCA instruction illustrated above. The various options available are:-

RLCA - Rotate Left Circular 'A'.

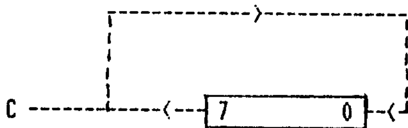
RLC r - Rotate Left Circular 'r'.

RLC (HL) - Rotate Left Circular (HL).

RLC (IX+d) - Rotate Left Circular (IX+d).

RLC (IY+d) - Rotate Left Circular (IY+d).

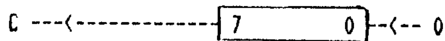
In all cases, we get:-



As well as these two left rotate instructions there is also a shift left instruction available, but this can only operate on register "A". This is:-

SLA - "Shift Left Accumulator"

or:-



The main difference here is that the contents of the carry bit are LOST, and bit 0 is filled with 0. This is effectively multiplying "A" by 2 as long as nothing is transferred to the accumulator.

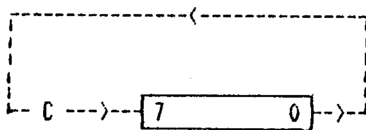
RIGHT ROTATIONS.

Here again we have the same two basic modes of rotations, but this time they rotate to the right. Exactly the same range of possible memory locations and rotations can be moved to the right as to the left i.e.:-

RRA - Rotate Right Accumulator.

RR r - Rotate Right Register.

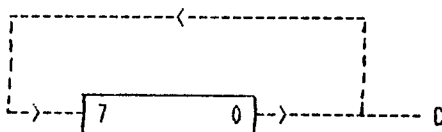
this is shown by:-



Again, the available options are:-

RRCA - Rotate Right Circular "A".
RRC r - Rotate Right Circular "r".
RRC (HL) - Rotate Right Circular (HL).
RRC (IX+d) - Rotate Right Circular (IX+d).
RRC (IY+d) - Rotate Right Circular (IY+d).

or:-



A similar shift right is available as for shift left i.e.:-

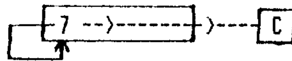
SRL r - Shift Right Logical Register "r".

i.e.:-

Here, this is division by 2, as long as we are using unsigned numbers i.e. 0 - 255. However, because in some applications we use the convention to indicate negative numbers by setting bit 7 to 1 (giving us a range of -128 to +127), there is an additional shift right instruction called:-

SRA r - Shift Right Arithmetic "r".

or:-



Again, this is also division by 2, but the sign bit is preserved.

IN AND OUT INSTRUCTIONS.

There are times when the CPU in your Spectrum needs to communicate with the outside world - to read the keyboard, or the tape recorder etc. This is achieved using the machine code IN and OUT instructions. The communication is performed by using PORTS - these are special locations used by the CPU, the most common of which is PORT 254 (FE Hex) - this is very often used for reading the keyboard etc. The number of ports available is usually limited to 256 for the Z80 CPU, due to hardware and other considerations. The keyboard is read, for example, using the instruction:-

IN A,(FE)

The 40 keys on the keyboard are arranged in a very special way so they can be represented by 8-bit bytes. In fact, the keyboard only returns information from FIVE keys at a time! It is the value of "A" in the IN instruction which determines which set of 5 keys will be read. The keyboard is divided into 4 rows, each comprising 2 blocks of 5 keys thus:-

BLOCK	KEYS										BLOCK
3 ----->	1	2	3	4	5	6	7	8	9	0	<--- 4
2 ----->	Q	W	E	R	T	Y	U	I	O	P	<--- 5
1 ----->	A	S	D	F	G	H	J	K	L	EN	<--- 6
0 ----->	SH	Z	X	C	V	B	N	M	.	SP	<--- 7

There are 8 blocks of letters, these can be correlated with the 8 bits of the "A" register i.e. ALL of the bits of "A" are set to "ON", except for one bit which specifies the block to be read.

Thus, to read the keys in the block "1 2 3 4 5", it is bit 3 of "A" which should be off i.e.:-

A = 1 1 1 1 0 1 1 1 = F7 Hex.

The contents of the keyboard are returned in "A", with the information coming into the lower bits of "A" i.e.:-

Key "1" ---> Bit 0 of "A"

Key "2" ---> Bit 1 of "A"

If block 4 was chosen instead, (i.e. A = EF Hex), then the information would come in as:-

Key "0" ---> Bit 0 of "A"

Key "9" ---> Bit 1 of "A"

You can think of the information coming into "A" from the outside edges first, so that both "0" and "1" would both go to bit "0" of register "A".
For some games applications you may wish to allow all of the top row to be read, and it is possible to read it all in one instruction (rather than the two instructions which would be required if we read one block at a time).

e.g. A = 1 1 1 0 0 1 1 1 = E7 Hex

Note that both bits "3" and "4" are "OFF". This tells us the CPU wants information from blocks 3 and 4. Of course the two lots of information get mixed and it is impossible to tell whether key "0" or key "1" was pressed, since both would set bit 0 of "A" i.e.:-

"1" or "0" ---> Bit 0 of A.
"2" or "9" ---> Bit 1 of A etc.

This is useful in movement games because it enables keys "5" and "8" to be used as the left and right direction arrows, even though they belong to different blocks on the keyboard.

Note that if you use the instruction:-

IN r, (C)

where register C specifies the port you want, then it is the contents of register B which define which keyboard block is being selected.
Another useful port is the cassette input/output port. This is still port FE as above. The major problem involved is the timing of the data going out and going in; this kind of problem requires a lot of experience with machine code.

The OUT instruction is also used to generate sound on the Spectrum, and to set the border colour.
Page 160 of your Spectrum manual discusses the Basic OUT instruction, and machine code programming of the OUT command is exactly the same i.e. bits 0,1, and 2 define the border colour, bit 3 sends a pulse out to the MIC and EAR sockets, while bit 4 sends a pulse to the internal loudspeaker.
To change the border colour, load A with the appropriate colour value and then execute the OUT (FE),A instruction. Note that this is only a TEMPORARY colour change. For a permanent change, you must perform the OUT, and also change the value of address 23624 (system variable BORDCR). This is because the ULA in the Spectrum gets its value for the border colour from this address. To stop the hardware messing with the border colour, you must disable interrupts (DI).

CREATING SOUND ON YOUR SPECTRUM.

Due to hardware limitations in the Spectrum, you can only produce sounds and noises using the OUT command if you have a 16K Spectrum, but not pure notes, you can with a 48K Spectrum.
To make a sound, you need to send a pulse to turn on the loudspeaker (and/or the MIC socket if it is to be amplified). Then a little while later, you need to send another pulse to turn it off. Then a little while later, on again; and so on.

In this way, sound is created. The total length of time between turning the loudspeaker on and the next time you turn it on again determines the frequency of the sound.
The length of time you leave the pulse ON, as opposed to the total time between pulses, can give you a small degree of control over the volume.

Note that you must use a value for A for on and off such that the border colour remains unchanged. Otherwise, you will get a banding pattern similar to the LOADing pattern.

"KOBRAHSOFT Z80 MACHINE CODE COURSE"

CHAPTER (11)

USEFUL FEATURES OF THE SPECTRUM FOR MACHINE CODE PROGRAMS.

Here we will consider in detail three useful features of the Spectrum which we may use in machine code programs.

THE KEYBOARD.

The keyboard is the main source of real-time input for the Spectrum. It can dynamically affect the processing of any program, either the operating system in the ROM, or the user's program in the RAM.

It is best considered as a two dimensional matrix with eight rows and five columns as shown below:-

SPECTRUM KEY INPUT TABLE

Input in A for FEH (Hex)	D4	D3	D2	D1	D0
FE	V	C	X	Z	CAP SHIFT
FD	G	F	D	S	A
FB	T	R	E	W	Q
F7	5	4	3	2	1
EF	6	7	8	9	0
DF	Y	U	I	O	P
BF	H	J	K	L	ENTER
7F	B	N	M	SYM SHIFT	BREAK SPACE
X	16	8	4	2	1

There are 4 main operations to test if a particular key has been pressed e.g. to check the "A" key:-

(1). Load the A register with the INPUT VALUE of the corresponding row:

```
LD A, FD Hex        ; Second Row
```

(2). Fetch from input port FE Hex - always used for keyboard:

```
IN A, (FE)
```

(3). Test Dx set to low for desired key:

```
AND 1               ; Test "A" key
```

(4). If zero flag set, then key has been pressed:

```
JR Z, Keyset        ; Normal (not pressed state always high - NZ)
```

Each of the 40 intersections represents a key of the keyboard. In their normal state (i.e. when they are NOT pressed), they are always in a high state i.e. the intersection is set as 1.

When a particular key is pressed, the intersection corresponding to that key will be reset to a low state i.e. 0.
Knowing the relationship between the keyboard and this inner matrix, we can derive a logical way of testing when a particular key is pressed, using machine code. This is summarised briefly on page 1. In more detail:-
In Basic, when we scan the keyboard, we need to provide an address for that particular half row of the keyboard where the desired key resides, before checking it using the IN function, as described on Page 160, Chapter 23 of the Spectrum manual.
Similarly, using machine code, we need to load into the A register (Accumulator) a value corresponding to the address of the half row of keys we want to test. The required value for each half row is listed in the leftmost column of the table on Page 1. e.g. For the "H to ENTER" half row we load A with BF Hex:-

```
LD A, BF Hex
```

The value in A will then be used to fetch the byte which contains the state of that particular half row of keys, and return to A when the INPUT instruction is used. e.g. the port used is the HE Hex port thus:-

```
IN A, (FE)
```

Since there are 5 keys per half row, we are only interested in the 5 low order bits of the byte returned in the A register.
If no key is pressed in that half row, the value of the low order 5 bits will be (2*2*2*2 + 2*2*2 + 2*2 + 2 + 1) i.e. 16+8+4+2+1 = 31. Thus. register A contains xxx11111 when no key is pressed.
If we want to test whether the rightmost bit is pressed, we check to see whether that bit is low. There are two ways to do this:-

(1). Use BIT test instructions, e.g. BIT 0,A etc. If the bit is low (not set), then the Zero Flag will be set (=1).
(2). Use Logical AND instructions e.g. AND 1 - if the bit is low (not set), then the result will be zero, and the Zero Flag will be set (=1).

The first method is easier because the particular bit we want to test is specified directly in the Bit Test instruction; but is has a shortcoming in that if we want to test TWO keys of that half row, we will need to use TWO Bit Test instructions, and possibly TWO relative jumps (JR's) e.g. to test Bit 0 and Bit 1, using the first method:-

```
            BIT 0, A            ; Test if Bit 0 of A is set or not.
            JR Z, NOPRESS       ; Jump if not pressed.
            BIT 1, A            ; Test if Bit 1 of A is set or not.
            JR z, NOPRESS       ; Jump if not pressed.
                  .
                  .             ; Do whatever if both pressed.
   NOPRESS        .
                  .             ; Do whatever if not pressed.
```

The second method of testing using logical AND instructions needs a little more logic. To test bit zero, we use "AND 1"; to test bit 1 we use "AND 2"; to test bit 2 we use "AND 4"; to test bit 3 we use "AND 8"; and so on. (Pardon the pun!).
To test TWO keys, we use "AND x", where x is the sum of the value we will use when testing each single key individually e.g. to test BOTH bit 0 and bit 1 of A are set:-

```
AND 3             ; Test both bit 0 and bit 1 is set.
CP  3             ; Test if BOTH are set.
JR NZ, NOTBOTH    ; Jump if both NOT pressed.
  .
  .
```

To test if EITHER bit 0 or bit 1 of A are set:-

```
AND 3             ; Test EITHER bit 0 and bit 1 is set.
JR Z, NOTONE      ; Jump if one is not pressed.
  .
  .
```

To check if you understand the method of testing the keys, write a routine in machine code to test if key "6" has been pressed; if it has, reset the computer. Hints:-

(1). Check the row address in the table on Page 1 to be loaded into A.
(2). Send it to the input port FE Hex.
(3). Test the bit that is set by the "6" key.
(4). If zero, jump to address zero to reset.
(5). If not zero, go back to (1) to repeat.

For the answer, seen the bottom of Page 4.

OUTPUT - THE VIDEO SCREEN DISPLAY.

The Video Screen display is the main source of output for the computer to communicate to the user.
The following machine code program will demonstrate the way the screen memory of the Spectrum is organised. Load it into a convenient address e.g. 32000 (7D00 Hex). This can either be done by POKING the numbers in with:-

```
10 FOR A = 32000 TO 32013: INPUT B: POKE A,B: NEXT A
```

RUN it and enter the numbers:- 33,0,64,54,255,17,1,64,1,1,0,237,176,201. This is the following program:-

Hex Nos	Decimal Nos	Mnemonics	Comments
21,00,40	33,0,64	LD HL,4000	; Load HL with start of dosplay file.
36,FF	54,255	LD (HL), FF	; Fill that screen location.
11,01,40	17,1,64	LD DE,4001	; Load DE with next byte in display.
01,01,00	1,1,0	LD BC,1	; Load BC with no. of bytes to move.
ED,B0	237,176	LDIR	; Move block of length BC from (HL) to (DE)
C9	201	RET	; End of program.

The other way is to load the KD1 disassembler, and enter the Hex numbers, using the Memory (M) facility.
RUN the program by typing RANDOMIZE USR 32000 - the single byte from 16384 will be moved to 16385.
Change line 4 to read LD BC,20 Hex or 01,1F,00 Hex. Again RUN it - you may be surprised to see which are the first 32 bytes of the screen display! Note how a very thin line has been drawn across the top of the screen. The first 32 bytes of the screen memory relate to the first byte of each of the first 32 characters.
Next, change line 4 to read LD BC,FF Hex or 01,FF,00 Hex. and RUN it. Again you may be surprised! The 33rd byte is NOT on the 2nd row of the screen - it is the first byte of the 32nd character! And so on up to the 256th character.
Where do you think the next byte would go? Change line 4 to LD BC,2047 Decimal or 01,FF,07 Hex and RUN it. The top third only of the screen is filled.
Experiment with this program, using different values for BC up to LD BC,6143 Decimal or 01,FF,17 Hex. In this way you can see how the display is organised. The screen memory is divided into THREE separate parts i.e. :-

(1). Memory 4000 Hex - 47FF Hex = first 8 lines
(2). Memory 4800 Hex - 4FFF Hex = second 8 lines.
(3). Memory 5000 Hex - 57FF Hex = third 8 lines.

Not only that, but you will recall that each character of the Spectrum is composed of EIGHT 8 bit bytes which makes 64 dots. e.g. for the character "!", we have:-

Decimal No	Binary No	Hex No
0	00000000	0
16	00010000	10
16	00010000	10
16	00010000	10
16	00010000	10
0	00000000	0
16	00010000	10
0	00000000	0

The organisation of the Spectrum screen display memory is such that the first 256 bytes from 4000 Hex to 40FF Hex correspond to the first byte of each of the 256 8 byte characters of the first 8 lines.

Then the next 256 bytes from 4100 Hex to 41FF Hex correspond to the SECOND byte of each of the 256 8 byte characters of the first 8 lines and so on.

Thus, the memory location of the 8 bytes corresponding to the first character of the screen is:-

1st byte:- 4000 Hex
2nd byte:- 4100 Hex
3rd byte:- 4200 Hex
4th byte:- 4300 Hex
5th byte:- 4400 Hex
6th byte:- 4500 Hex
7th byte:- 4600 Hex
8th byte:- 4700 Hex

Unusual, but that's the Spectrum! The full screen display organisation is shown below:-

MEMORY IN HEX	ATTRIBUTE IN HEX	LINE	MEMORY IN HEX	ATTRIBUTE IN HEX
4000	5800	0	401F	581F
4020	5820	1	403F	583F
4040	5840	2	405F	585F
4060	5860	3	407F	587F
4080	5880	4	409F	589F
40A0	58A0	5	40BF	58BF
40C0	58C0	6	40DF	58DF
40E0	58E0	7	40FF	58FF
4800	5900	8	481F	591F
4820	5920	9	483F	593F
4840	5940	10	485F	595F
4860	5960	11	487F	597F
4880	5980	12	489F	599F
48A0	59A0	13	48BF	59BF
48C0	59C0	14	48DF	59DF
48E0	59E0	15	48FF	59FF
5000	5A00	16	501F	5A1F
5020	5A20	17	503F	5A3F
5040	5A40	18	505F	5A5F
5060	5A60	19	507F	5A7F
5080	5A80	20	509F	5A9F
50A0	5AA0	21	50BF	5ABF
50C0	5AC0	22	50DF	5ADF
50E0	5AE0	23	50FF	5AFF

The screen complexities are, however, well worth understanding since, using machine code, we can also access the bottom two lines of the screen, which we never could using Basic!

SOLUTION TO TRAP "6" KEY ROUTINE.

Referring to the table on Page 1, we can see that the required row address is FD Hex. Also, to test the bit, we must use the AND 16 instruction. The full routine is thus:-

```
Hex Nos     Decimal Nos    Mnemonics           Comments
-------     -----------    ---------           --------
3E,FD       62,253         AGAIN LD A,FD Hex   ; Input value of row = FD Hex
DB,FE       219,254              IN A,(FE)     ; Fetch from port FE Hex.
E6,10       230,16               AND 10 Hex    ; Trap "6" key.
CA,0C,7D    202,12,125           JP Z,RESET    ; If zero, reset.
C3,00,7D    195,0,125            JP AGAIN      ; If not, try again.
C3,00,00    195,0,0        RESET JP 0000       ; Jump to address 0 - RESET.
```

Try it - enter the numbers 62,253,219,254,230,16,202,12,125,195,0,125,195,0,0 . Start at 32000, RUN - pressing "6" should reset the computer.

"KOBRAHSOFT Z80 MACHINE CODE COURSE"

CHAPTER (12)

USEFUL FEATURES OF THE SPECTRUM FOR MACHINE CODE PROGRAMS (contd.)

In this, the final part of our Machine Code Course, we will consider two more important features of the Spectrum for use in machine code programming i.e. The Video Attributes Display and Sound Output. We will also show you how to construct a simple machine code program which will also be very useful - a Header Reader program.

THE VIDEO DISPLAY ATTRIBUTES.

The word "attribute" can be seen as meaning "colours"; whatever value lies in a certain attribute area will give that area a certain PAPER, INK, FLASH or BRIGHT colour.
The attribute memory area is easier to understand than the Spectrum display (screen) layout, since it bears a simple one-to-one relationship with the characters displayed on the screen.
The attribute file occupies the memory area from 5800 Hex (22528 Decimal) to 5AFF Hex (23295 Decimal). It is thus 768 bytes long, which corresponds to 24 lines of 32 characters in each line i.e. there is ONE attribute byte for each character position.

Thus, 5800 Hex is the attribute of the first character of the first line, 5801 Hex the second character, 5802 Hex the third and so on. 581F Hex corresponds to the 32nd character of the first line. Similarly, 5820 Hex holds the attribute of the first character of the SECOND line, 5840 Hex the first character of the THIRD line, and 5AE0 Hex the first character of the LAST line of the screen.

NOTE:- For each character position on the screen, each corresponding attribute byte is made up as follows:-

Attribute Byte:- 8 bits i.e. 7 6 5 4 3 2 1 0

Bits 0,1, and 2:- These determine the INK colour of the character i.e. colours 0 to 7 (black to white). The usual values for the bits apply i.e.:- Bit 0 if set represents 1, Bit 1 represents 2, Bit 2 represents 4. Thus, if we have:-

Bit:-	2	1	0	
State (Set/Reset):-	0	1	0	
Value:-	0	2	0	Total = 2 = colour is RED.

Bits 3,4, and 5:- These determine the PAPER colour of the character 0 to 7.

Bit 6:- This makes the character BRIGHT if SET (=1); NORMAL if RESET (=0).

Bit 7:- This makes the character FLASH if SET, NOT FLASH if RESET.

As an exercise, how would make the first character of the second line of the screen have PAPER 6, INK 3, and be BRIGHT but NOT FLASHING? For the answer, see later.
We will now write a subroutine which will convert a given screen address to its corresponding attribute address - this means finding which area of the screen the character belongs to, then adding this to 5800 Hex. This can be done thus:-

```
(1) LD HL,4529H - Load the address in HL.
(2) LD A,H      - Put the High Order Byte in A.
(3) AND 18H     - Trap bits 3 and 4; gives the screen area.
(4) SRA A       - Shift Right Accumulator 3 times i.e. divide by 8. The result will
    SRA A         be either 0, 1, or 2 depending on the value in H.
    SRA A
(5) ADD A,58H   - Transform to attribute address.
(6) LD H,A      - Store in H register. H = 58H, 59H or 5AH.
```

We will illustrate this with an example. Consider the address 4800H. Step (1) puts this in HL i.e. H contains 48H, L contains zero. Step (2) puts 48H in A. Consider Step (3):-

```
   Value in A:-  01001000  =  48H.
AND with 18H:-  00011000  =  18H.
                ------------------
                00001000  =  8H.
```

Performing Step (4) three times then gives the equivalent of dividing by 8 i.e. here the result will give 1 in the A register. Step (5) then adds 58H to this to give here 59H; this is stored in the H register by Step (6). The result gives 5900H in HL. NOTE:- The L register stays the same. The answer is thus 5900H - correct. For verification - check the screen display table on Page 4 of Chapter 11.

Answer to the Attribute Exercise.

The first character of the second line of the screen is represented by the attribute address 5820H = 22560 Decimal. To make it contain PAPER 6, INK 3, BRIGHT 1, we must have:-

Function:-		FLASH	BRIGHT	PAPER	INK	
Address 22560:-	Bit No.	7	6	5 4 3	2 1 0	
Answer:-		0	1	1 1 0	0 1 1	= 73H = 115 Dec.
		FLASH 0	BRIGHT 1	PAPER 6	INK 3	

The correct answer is thus obtained by POKEing 22560 with 115. Try it, type POKE 22560,115 (ENTER). Any character on the screen at this position will be BRIGHT, and have PAPER 6, INK 3.

SOUND OUTPUT ON THE SPECTRUM.

There are TWO ways of generating sound using machine code on the Spectrum i.e.:-

(1). Sending signals to the cassette output port 254 for a certain duration of time using the OUT 254 instruction i.e. OUT (254),A.

(2). Using the BEEP routine in your Spectrum's ROM, we can set HL and DE registers to contain certain values i.e. DE contains the duration of the sound in seconds X frequency; HL contains the frequency. Then we execute the routine with the instruction CALL 03B5H.

Method (1) has the advantage of being free from any ROM calls, which can change from one model of the Spectrum to another - bear this in mind when considering using ROM calls in your machine code programs. The snag is that because the ULA is constantly accessing the video display, if your program resides in the first 16K of your Spectrum's memory it will be frequently interrupted by this process. If the program generates sound, it will occur in bursts of unpredictable duration. This means the sound will not be pure. One solution is to move that part of the program which generates sound to higher memory if you have a 48K Spectrum. Another problem is that since we send values to output port 254, it will also affect the border colour and MIC and LOUDSPEAKER, depending on the value sent. (See Page 4 of Chapter 10).
Method (2) does not suffer from these problems, and is consequently the best to use.
As an example, for the note "Middle C" to be produced for one second, DE must hold 105H, HL must hold 66AH. The following program will produce this i.e.:-

Mnemonics	Hex Numbers	Decimal Numbers
LD HL,66AH	21,6A,6	33,106,6
LD DE,105H	11,5,1	17,5,1
CALL 03B5H	CD,B5,3	205,181,3
RET	RET	201

To try this, put the code at a suitable address e.g. 40000, with the following program:-

```
10: FOR A=40000 TO 40009: INPUT B: POKE A,B: NEXT A
```

RUN it, and enter the above numbers i.e. 33,106,6,17,5,1,205,181,3,201. Activate the routine with RANDOMIZE USR 40000. You will hear the note Middle C for 1 second! Experiment with different values for HL and DE to get any different sounds you want.

AN EXAMPLE MACHINE CODE PROGRAM - WRITING A HEADER READER PROGRAM.

We will now consider the writing of a Header Reader program in machine code, which will read the Headers of any program PLAYed in, and list their composition. What is a Header? When you load a program on your Spectrum, you usually see at first a burst of RED/CYAN THICK STRIPES - this is called a LEADER (L), and is around 5 sec. in length for a "HEADER", but only 2 sec. for a "CODE BLOCK". After, comes a burst of BLUE/YELLOW NARROWER STRIPES - these are BYTES (B). Thus, for a typical Basic program you get:-

```
          HEADER                    CODE BLOCK
          ------                    ----------
          L----B                      L----B
          ^    ^                      ^    ^

   5 secs   V.short            2 secs   Any length
 RED/CYAN   BLUE/YELLOW      RED/CYAN   BLUE/YELLOW
THICK BANDS THINNER BANDS  THICK BANDS  THINNER BANDS
```

The HEADER must always come first, since it tells the Spectrum where the following code must go in memory. The burst of BYTES for a Header is always very short, since it always contains only 17 bytes. These 17 bytes give the following information:-

BYTE NUMBER	INFORMATION
1	This gives the TYPE of program i.e. 0 for BASIC, 1 for a NUMERIC ARRAY, 2 for a STRING ARRAY, 3 for MACHINE CODE.
2 - 11	These 10 bytes give the PROGRAM NAME, in Spectrum character codes.
12,13	These 2 bytes give, for a block of code, the CODE LENGTH; or, for a Basic program, the length of the program area PLUS its variables.
14,15	These 2 bytes give, for a block of code, the Start Address of the block in memory; or, for a Basic program, the AUTO-RUN line number.
16,17	These 2 bytes give, for a block of code, a repeat of the CODE LENGTH; or, for a Basic program, the length of the program area MINUS its variables.

The information in these 17 bytes can be obtained by the Header Reader program we will now write. The first problem - where to load the Header? This can be any convenient address, we will use 32000. We will be loading the Header using machine code - this can be placed in a REM statement at the start of the program. Type in the following Basic program, it will be explained fully later:-

```
1: REM hhhhhhhhhhhhhhhhhhhhhhhhhhhhhhh
5: CLEAR 31999: CLS
10: LET A=32000: DEF FN A(X)=PEEK (A+X)+256*PEEK (A+X+1)
15: PRINT AT 0,5;"HEADER READER PROGRAM": PRINT AT 1,5;"=====================":PRINT: PRINT
16: LET B=PEEK 23635+256*PEEK 23636
17: RANDOMIZE USR B: CLS
18: PRINT AT 0,5;"HEADER READER PROGRAM": PRINT AT 1,5;"=====================": PRINT: PRINT
20: PRINT "Filename: ";
25: FOR N=A+1 TO A+10: PRINT CHR$ PEEK N;: NEXT N
26: LET R=PEEK A: IF R=0 THEN GOSUB 35: GOTO 15
27: IF R=1 THEN GOSUB 60: GOTO 15
28: IF R=2 THEN GOSUB 65: GOTO 15
30: IF R=3 THEN GOSUB 80: GOTO 15
35: PRINT ''"Basic Program"
40: PRINT ''"Total Length: ";FN A(11);" Bytes."
45: PRINT ''"Program Length: ";FN A(15);" Bytes."
50: IF FN A(13)>9999 THEN PRINT ''"Loading Only":RETURN
55: PRINT ''"Auto-Runs from Line: ";FN A(13):RETURN
60: PRINT ''"Numeric Array": LET A$="": GOTO 70
65: LET A$="$": PRINT ''"Character Array"
70: PRINT ''"Array Length: ";FN A(11);" Bytes."
75: LET X=PEEK (A+14): PRINT ''"Array Name: ";CHR$ (64+32*(X/32-INT (X/32)));A$: RETURN
80: IF FN A(13)=16384 AND FN A(11)=6912 THEN PRINT ''"Screen$ ": RETURN
85: PRINT ''"Machine Code"''"Start Address: ";FN A(13) ''"Code Length: ";FN A(11);" Bytes.": RETURN
```

We now need to POKE the machine code to load the Header into the REM statement in Line 1. Find the start of the Basic program area by typing PRINT PEEK 23635+256*PEEK 23636. This will give a value of either 23755 or 23813 according to whether a microdrive is fitted. The machine code to load the Header will be:-

Mnemonics	Hex Nos.	Decimal Nos.	Remarks
LD HL,(5C3D)	2A,3D,5C	42,61,92	Load HL from ERR-SP; gives RESET with BREAK.
LD (HL),0	36,00	54,0	Load first byte with 0.
INC HL	23	35	Increment to next byte.
LD (HL),0	36,00	54,0	Load second byte with 0.
XOR A	AF	175	Signal "Loading Header".
SCF	37	55	Set Carry Flag - signal "Load".
LD IX,7D00	DD,21,00,7D	221,33,0,125	Load start address = 32000.
LD DE,0011	11,11,00	17,17,0	Length to load = 17 bytes.
CALL 0556	CD,56,05	205,86,5	Call the ROM load routine.
RET	C9	201	Return to Basic.

This is POKed in as follows - type in:-

```
90: FOR A=23760 (or 23818) TO 23780 (or 23838): INPUT B: POKE A,B: NEXT A
```

RUN 90, and enter the numbers 42,61,92,54,0,35,54,0,175,55,221,33,0,125,17,17,0,205,86,5,201. The start is 5 bytes past the start of the Basic program area, since the first 2 bytes are the Line No.; the next 2 are the line length; and the next is the REM character. NOTE:- LISTing the program will now give the message "Invalid Colour" because of the weird character tokens in the REM line. An explanation of the Basic is:-

Line 1:- Contains the REM which contains the machine code.
Line 5:- Sets the stack below RAMTOP; clears screen.
Line 10:- Sets A to 32000; defines a general function.
Line 15:- Prints the Heading.
Line 16:- Checks the start of the Basic program area.
Line 17:- Call the machine code in the REM to load the Header; clear screen after.
Line 18:- Reprint Heading.
Lines 20 - 85:- Print the various Headings read from the Header.

Delete Line 90. Next, POKE Line 1 to zero with POKE 23756 (or 23814), 0. It cannot then be edited out! Save the program to auto-run with SAVE"HReader" LINE 0.

That brings us to the end of our "KOBRAHSOFT MACHINE CODE COURSE". We hope you enjoyed it and will now go on to write your own machine code programs. Remember, with machine code, the only limit is your imagination!