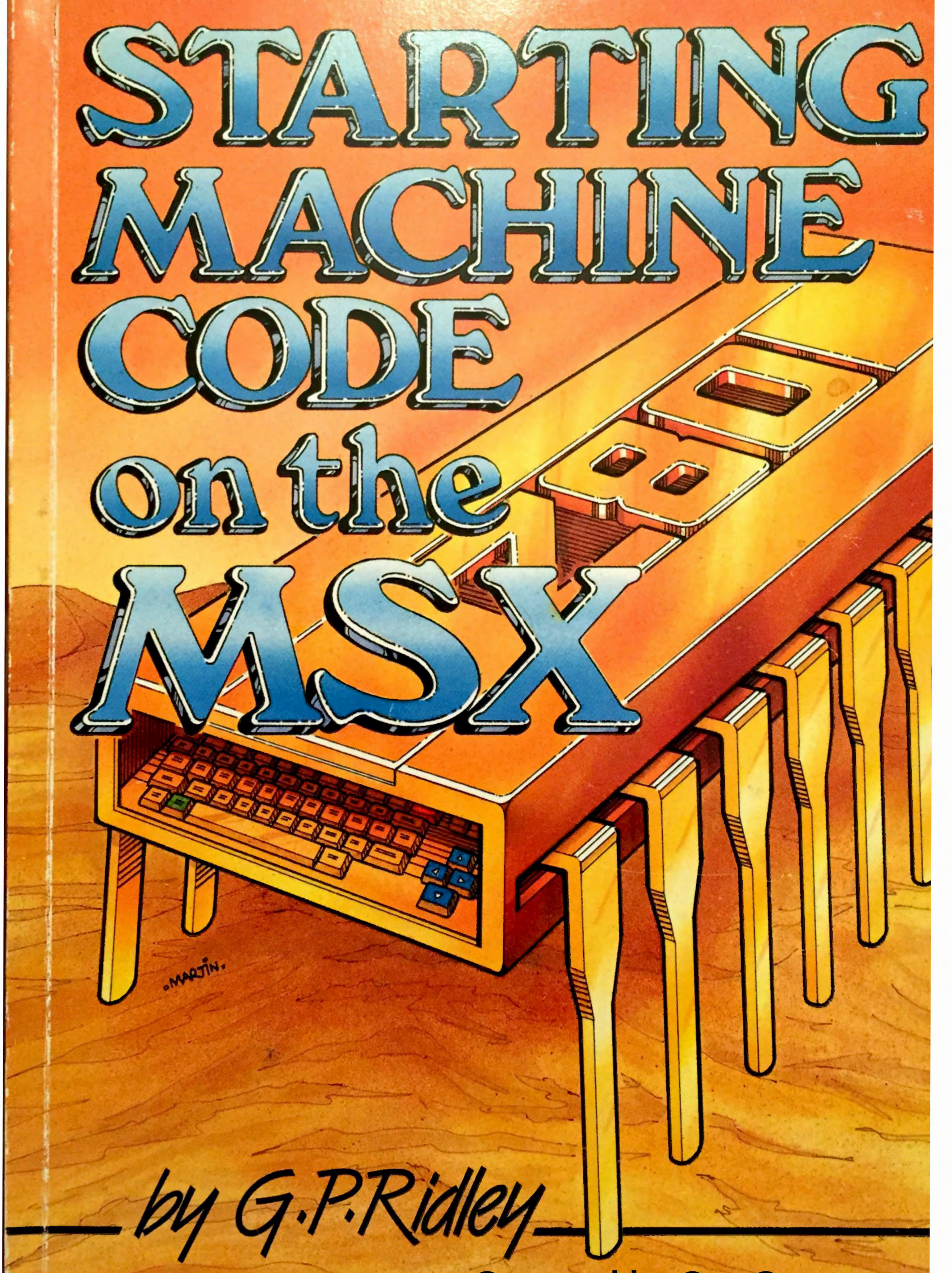
STARTING MACHINE CODE on the MSX

by G.P. Ridley



nR 83
39=

STARTING

MACHINE CODE

ON THE

MSX MICROCOMPUTER

ISBN NO 07457-0132-9

The Author wishes to thank Mike Shaw for his contributions to the text and programs.

Published by:- Kuma Computers Ltd.,
12 Horseshoe Park,
Pangbourne,
Berks RG8 7JW

Telex 849462 Tel 07357 4335

Contents

Introduction

This book has been written as an introduction to writing Machine Code programs and routines using Assembler language on the MSX range of home computers.

Not so many years ago machine code was the programmers first language, but with the popularity of the home computer Basic has become the common language which most micro users cut their teeth on and machine code remains a somewhat grey area which most of us see in program listings as a series of numbers within DATA statements POKEd into high memory locations and then called by the USR command, and are left without a clue as to what is happening.

Machine code programs, operate far quicker than those written in Basic and that is one reason for a Basic program to contain a machine code routine in order to achieve greater speed, or it could be used to modify Basic to do tasks it cannot normally do.

This book hopefully will make machine code clearer and more understandable to the average user, one will not need a degree to grasp what is happening, and computer jargon will be kept to minimum levels.

Good Luck.

1

Machine Code from Basic

The Basic language is generally the simplest way of writing programs, it is easy to follow and debugging a faulty program is usually made quite easy with the editing facilities for altering lines in a program, so why use machine code?

The main reason must be speed of execution, not purely based on games programs such as space invaders or the like which would not be worth playing if they were written in Basic, but more serious applications which will be shown in the book. In order to grasp some idea of the speed of a program written in assembler we will compare the execution time with a similar program written in Basic.:-

```
10 SCREEN0:KEYOFF
20 WIDTH40:CLS
30 TIME=0
40 FOR X = 0 TO 959
50 PRINT"B";
60 NEXT
70 PRINT TIME
```

Now entering the function key 'F5' or by entering 'RUN' followed by the 'RETURN' key one will see that the MSX took 151 time cycles (or 3.02 seconds if line 70 was altered to TIME/50) to fill the screen.

Another method for displaying characters is to VPOKE directly to a specific location on the screen. In screen 0 mode the top left position of the 40 column screen is 0000 hex, let us alter the above program so that instead of printing one character after another using the Basic PRINT statement we shall print the characters directly to the screen area of memory using the VPOKE statement.

Add line 15:-

```
15 Z=&H0
```

And alter the following lines to read:-

```
50 VPOKEZ+X,66
70 LOCATE,23:PRINT TIME
```

Once again Run the program.

The time was 225, so directly VPokeing to the screen is no quicker. Note that the cursor remains in the same position whilst a VPOKE statement is carried out, therefore line 70 repositioned the cursor to screen line 23 by the LOCATE statement. If you find that the 'Ok' is displayed slightly off-screen then enter in direct mode 'WIDTH37' and the 'RETURN' key to clear the screen and return to 37 column display which is the condition the MSX wakes up in when first switched on.

Now if that program, although not the most interesting in the world but quite effective as an example, is re-written in machine code and called by the A=USR(0) function from Basic the dramatic increase in speed will be instantly obvious.

Program 1 Direct Screen Addressing from Basic

Assembly language instructions used:-

```
LD HL,nnnn    LD BC,nnnn    LD A,nn    CALL nnnn    RET
```

These instructions are detailed in chapter 2.

Enter 'NEW' and 'RETURN' and input the program:-

```
10 CLEAR 200,&H9FFF
20 FOR X = &HA000 TO &HA00B
30 READ A:POKE X,A:NEXT
40 DATA 62,66,33,0,0,1,192,3,205,86,0,201
```

Enter 'F5' to run the program.
The screen will instantly display the 'Ok' message and one could be excused in thinking that not much has just happened. But happen it has in that now a machine code routine has been placed in memory, starting at location A000 hex (40960 decimal), which will print the entire screen with the letter 'B' in a fraction of the time taken previously using normal Basic PRINT or VPOKE statements.

Enter 'NEW' and 'RETURN' and this program will fill the screen:-

```
10 DEF USR=&HA000:SCREEN0
20 TIME=0
30 A=USR(0)
40 LOCATE,23:PRINT TIME
```

RUN the program.
Its speed is amazingly fast and time was printed as 1 or 2.

As will be seen in the next chapter Assembly language is made up of several registers which we load with addresses and values, you can also check the codes in the Appendix. If we disassembled the DATA which was placed at A000 to A00B it would look like this:-

```
A000   3E 42          LD A,42H
A002   21 00 00       LD HL,0000H
A005   01 C0 03       LD BC,03C0H
A008   CD 56 00       CALL 0056H
A00B   C9             RET
```

We POKEd the DATA into memory starting at address A000 hex which if we convert to decimal gives us 40960, use the conversion chart in the Appendix if you aren't sure. The first two items in the DATA line were 62,66 decimal. 62 converts to 3E hex which means we want to load the A register with the value of the next byte which in this case was 66 (42 hex) in line 1 on the previous page.

The next three bytes were 33,0,0 which convert to 21,00,00 hex. 21 hex signifies Load the register pair HL with the following two bytes in reverse order, low address first, in this instance we want to load HL with the address of the top left corner of Screen 0 (Video RAM) which is 0000 hex, therefore in this particular case reverse order doesn't show us much as the address is zero but the next line will prove the point.

These were followed by 1,192,3 and the number 1 signifies Load BC with the following two bytes in low byte first, high byte second. The figure we want to load into BC is the amount of bytes we wish to print to the screen. Screen 0 has a maximum size capacity of 960 locations, 24 lines by 40 columns, therefore 960 decimal equals 03C0 hex, which in reverse order becomes C0 03 (line 3).

Next came 205,86,0. 205 converts to CD hex which translates to CALL the address of the next two bytes which were in reverse order again. The address of the routine we wish to call is 0056 hex therefore if we reverse them and convert to decimal these become 86,0.

<u>NOTE</u> The ROM section of memory contains many routines which can be called upon to perform different functions, location 0056 hex contains the instruction to jump to a routine which fills VRAM area with the character contained in the A register. However before calling the routine one must ensure that HL contains the start address, BC the number of bytes to fill and A the data. And this our program has already done in lines 1 to 3.

The final number in the DATA line was 201 which converts to C9 hex, this command is RET for return, just as one would use after a GOSUB routine in Basic. Remember that we went to this routine by the USR(0) statement which is a Call instruction just like the Basic GOSUB and to quit the routine we enter the RET command to return.

The USR statement can contain within the brackets an 'argument' such as an integer, string, single or double precision variable to pass on for the machine code program to use. This is explained in chapter 4, but for this example no data was required to be accessed by our routine so a simple call was made with a dummy argument (0).

Now that routine although it executed in a fraction of the time it took using Basic was not quick to program, and a lot of thought would go into producing a simple output such as that. It is also more complicated translating decimal values back to hex and then translating them into assembly language Mnemonics and operands.

In later examples of machine code we will use an Assembler, Editor called 'ZEN - Z80 Assembly Language Programming System for the MSX Micro-Computer' which includes a disassembler. An Assembler/Editor will do most of the dirty work for you and produce a printout such as we have just seen, furthermore entering assembly language is made childs-play, well almost, as they allow one to enter opcodes and operands such as:- LD BC,03C0H directly. After entering the listing one selects the assemble option and the assembler will then translate all the instructions into machine code automatically and output a version known as Object code. This small piece of jargon simply means assembled machine code ready to record on tape for future loading.

It is virtually impossible to write machine code programs of any size without an assembler, it will pick up any false statements just like Basic does with the Syntax errors and it will allow one to run the programs and use breakpoints to stop the running at certain points so that one may check on the state of the registers etc. This is most important as the programs run so fast it would be difficult to make these checks without the facility.

Program 2 Storing Screens

New Assembly language instructions used:-
LD DE,nnnn

Two other routines within ROM allow the screen area to be copied into other parts of memory for storage and recalled when required. One may have a program which is menu driven in which options the user can make are listed on screen. That complete screen display could be stored somewhere in RAM and when needed a A=USRn(0) instruction will immediately transpose that block of memory back to the screen in a flash.

Enter 'NEW' and 'RETURN'

```
10 CLEAR200,&HDFFF
20 DEF USR0=&HF000:DEF USR1=&HF010
30 FOR X = &HF000 TO &HF00C
40 READ A:POKE X,A:NEXT
50 DATA 33,0,0,17,0,224,1,192,3,195,89,0,201
60 FOR X = &HF010 TO &HF01C
70 READ A:POKE X,A:NEXT
80 DATA 33,0,224,17,0,0,1,192,3,195,92,0,201
```

Now enter the 'F5' key or 'RUN' and 'RETURN'
Once again the 'Ok' message was displayed almost immediately, and we now have this screen move routine in memory.

One does not need to write a separate program to demonstrate this routine, providing there is a fair amount of text presently on the screen, if there isn't put something on the screen, anything.

Enter in direct mode (without a line number) A=USR0(0) and 'RETURN'. The 'Ok' will be displayed instantly and the total displayed area has been copied into memory locations E000 to E3BF hex. It has not disappeared off the screen it has been duplicated into the other area. If one was running a program the screen could now be cleared and the program continue until one needed to bring back the previous display.

Now clear the screen by entering the 'SHIFT' and 'HOME' keys and to prove the point enter some characters onto the screen, it does not matter if one gets 'Syntax error' printed just get something on the screen.

Enter in direct mode A=USR1(0) and 'ENTER'
The screen will instantly change back to the previous display which was saved when we entered A=USR0(0)

One could save more than one screen, providing they were moved to separate areas of memory, the Screen 0 text screen can contain up to 960 bytes so one will need to adjust the program for different storage areas. Here is the assembled listing, remember it was in 2 sections the first stores a screen:-

```
F000   21 00 00       LD HL,0000
F003   11 00 E0       LD DE,E000
F006   01 C0 03       LD BC,03C0
F009   CD 59 00       CALL 0059
F00C   C9             RET
```

and the second section recalls it to the display:-

```
F010   21 00 E0       LD HL,E000
F013   11 00 00       LD DE,0000
F016   01 C0 03       LD BC,03C0
F019   CD 5C 00       CALL 005C
F01C   C9             RET
```

The ROM routines which control the copying are at 0059 and 005C hex and as with the previous example certain registers need loading with data before they are called. Whether storing or recalling a screen of information registers HL require the source address. When storing a screen in the Screen 0 mode we know that the source will be address 0000, so in line 1 HL is loaded with 0000. DE represents the destination address and is loaded with the start address of where in RAM we wish it to be stored from, so in line 2 DE is loaded with E000 hex. Registers BC always contains the amount of bytes to move and is loaded with 03C0 hex (960 decimal) in line 3, if one wished to only store the top half of a screen, say lines 0 to 11, BC could be loaded 480 (01E0 hex). Line 4 is the call to the ROM routine which carries out the copying and this is followed by RET in line 5 which returns us to our Basic program.

The second section for recalling a stored screen works in a similar fashion with only alterations to HL, which contains the source address, and is loaded with the start of the storage area E000, DE for the destination which in Screen 0 will be the 0000, and finally the call to execute the copying back to screen which is at 005C. The amount of bytes to transfer, in BC, remains the same at 03C0.

As we now have the facility to store and recall one screen display it is straightforward to modify the program to cater for four screens. One only needs to alter various items in our basic program which wrote this machine code routine into memory.

List the program and alter the following lines to read thus:-

```
20 DEF USR2=&HF020:DEF USR3=&HF030
30 FOR X = &HF020 TO &HF02C
60 FOR X = &HF030 TO &HF03C
```

And in line 50 alter the sixth number from 224 to 228
and line 80 alter the third number from 224 to 228

<u>NOTE</u> Care must be taken when modifying an existing line on screen. In lines 50 and 80, which exceed one screen line, ensure that you do not press 'RETURN' until you have moved the cursor to the end of the complete program line otherwise the MSX will forget the characters shown on the next line and shorten the line so producing an error message. List the program before running it.

After running the altered program one should have the facility to store another screen display in memory, only this time we have written the copying routine at F020 and the recall routine at F030 hex, and the storing of this second screen commences at E400 hex. Storing a second screen is achieved by entering:-

```
A=USR2(0)
```

and to recall to the display:-

```
A=USR3(0)
```

If one requires three screens to be stored the following alterations should be made:-

```
20 DEF USR4=&HF040:DEF USR5=&HF050
30 FOR X = &HF040 TO &HF04C
60 FOR X = &HF050 TO &HF05C
```

And in line 50 alter the sixth number from 228 to 232
and line 80 alter the third number from 228 to 232

Check the listing and run.

And these are the alterations for the fourth screen:-

```
20 DEF USR6=&HF060:DEF USR7=&HF070
30 FOR X = &HF060 TO &HF06C
60 FOR X = &HF070 TO &HF07C
```

And in line 50 alter the sixth number from 232 to 236
and line 80 alter the third number from 232 to 236

Once again list and run the program. The routines are accessed by:-

```
A=USR0(0) stores-   A=USR1(0) recalls
A=USR2(0)   "   "   A=USR3(0)   "   "
A=USR4(0)   "   "   A=USR5(0)   "   "
A=USR6(0)   "   "   A=USR7(0)   "   "
```

2

Z80 Instructions

In this Chapter, we're going to take a broad look at the way the Z80 chip interprets the machine code numbers, the Z80 Registers and the way they are generally used, and then at the different types of Assembler instruction. You'll find a complete list of these mnemonic instructions in the Appendices - listed alphabetically and numerically by the first byte of their instruction code. There are several books available which explain each Z80 instruction in greater depth, rather like an encyclopaedia and almost as large, but these are general references and do not show examples for specific micros like the MSX range. However if one requires more detailed information regarding the Z80 instruction set then the purchase should prove worthwhile.

BASIC has well over 200 instructions - taking into account all the subtle variations like 'IF-THEN GOSUB' and 'IF THEN PRINT'. Z80 machine code has nearly 700 - but don't panic, many of them are simply variations on a theme.

The difference, as you will have already appreciated, is that one BASIC instruction calls up a host of machine code instructions within the interpreter. When you write in machine code you have to generate those instructions yourself - although you can, of course, call up useful routines resident in the ROM section of memory (as indeed some of the demonstration programs in this book do).

It is possible to write programs without having a full knowledge of the entire instruction set - indeed many people do quite happily and successfully, adding to their knowledge as they gain experience. The same is true to some extent when programming in BASIC.

For example - how would you do a count of 1 to 1000 in BASIC? Probably:-

```
10 FOR I=1 TO 1000
20 NEXT
30 PRINT "ALL DONE"
```

Fine, but supposing you didn't know about FOR-NEXT loops? You'd probably tackle it this way:-

```
10 A=0
20 A=A+1
30 IF A<1000 THEN 20
40 PRINT "ALL DONE"
```

But supposing you didn't know about IF-THEN constructions either. You'd really have to put your thinking cap on:-

```
10 A=0
20 A=A+1
30 B=-1*(A<1000)-2*(A=1000)
40 ON B GOTO 20,50
50 PRINT "ALL DONE"
```

As you can see, the programs become longer - and take longer to run - when the most suitable commands are not used. Knowing all the commands at your disposal helps you to make your programs shorter and/or faster running...and your life easier. Usually machine code programs run fast enough even when written the 'long way round', but when a very large number of repetitive actions are involved, such as in a Chess Game program, even a few microseconds knocked out of a loop can result in a considerable time saving when the program is running.

Having said that, the programs in this book have been written to demonstrate principles, and are not necessarily the fastest or shortest way of achieving the desired result.

What do all the numbers mean?

Machine coding, as you know, is all about numbers. A number can mean one of two things to the Z80 central processing unit in your computer. It can mean an instruction or part of an instruction to do something. Or it can mean a piece of information to be worked on or used in some way. Fortunately, the Z80 knows exactly which of these the number represents (in a correctly written program), and acts accordingly.

Take an instruction to load Register A with the value '7' (we'll be discussing the Registers in more detail later). In Assembly language mnemonics this instruction is written LD A,7 . In machine code language, the instruction is represented by the two hex numbers '3E 07'. When the Z80 sees the first of these it says "3E means I must load the next number along into Register A". It takes up the 7, puts it into Register A, then looks to the number after the 7 for the next instruction. So it wouldn't be confused if it saw, for example, the two hex numbers '3E 3E' - this time it would load 3E hex (62 decimal) into its Register A, then look to the number after the second 3E for its next instruction.

Note that each single byte of information can have a value from 0 to FF hex (0 to 255 decimal). Let us take a look at that in more detail.

A byte consists of 8 bits, each bit being a binary 0 or 1. So the binary number 11001001 can be represented thus:-

```
           Bit No:  7 6 5 4 3 2 1 0
     Binary Value:  1 1 0 0 1 0 0 1
```

Wherever a '1' appears in the binary representation, raise 2 to the power of the corresponding Bit Number, add the results together, and you have the decimal value of the Binary number. Thus, using the above example:-

```
2 to the power 7   = 128
2 to the power 6   =  64
2 to the power 3   =   8
2 to the power 0   =   1 (any no. to the power 0 = 1)
                     ---
                     201
```

So the binary number 11001001 is 201 in decimal.

To convert a binary number to Hex, split the eight digits into two groups of four (called 'nibbles'). Thus:-

```
   Nibble 'bit' no.:   3 2 1 0   3 2 1 0
       Binary value:   1 1 0 0   1 0 0 1

       Left side:  2^3 = 8   Right side:  2^3 = 8
                   2^2 = 4                2^0 = 1
                         --                     --
                         12                     9
```

Remembering that decimal 12 = C in hex, the hex value of binary 11001001 is C9.

How the Z80 handles 2-Byte numbers

Many instructions to the Z80 tell it to operate not on one byte - as in our 'LD A,7' - but on two bytes. For example, an Assembly instruction might be 'LD HL,49AFH' (the 'H' at the end tells the Assembler that 49AF is a hex number). Two-byte numbers increase the decimal values that can be represented from 0-255 to 0-65535 (0-FFFF hex) - which is absolutely vital for addressing or pointing to the memory locations in your computer.

In the instruction LD HL,49AFH, we want the High byte, 49 (hex) to go into the H Register, and the Low byte AF (hex) to go into the L Register. The machine code instruction for loading H and L Registers with 'direct' data is 21 hex. When the Z80 sees 21 hex as an instruction, it takes the NEXT number and loads it into the L Register. That's right - the L Register. Then it takes the following number and loads it into the H Register. So the machine code for LD HL,49AFH looks like this:-

```
21 AF 49 (hex)
```

Note how, in actual machine code, the order of the two information bytes is reversed. Now you know why.

When using an Assembler, you don't have to worry about this point - the Assembler sorts it out for you. But if you are entering machine code by hand, as was shown in chapter 1, forget the order of the two information bytes at your peril.

Needless to say, when loading any Register pair with data (we'll discuss Register pairs later on), the Low byte always appears in the machine code listing before the High byte. In Assembly language remember, you write the number in the normal way, and let the Assembler put things in the correct order.

Inside the Z80 chip

The elements that go to make up a Z80 chip include an Arithmetic-Logic-Unit, which performs all the (simple) arithmetical and logical functions, a 'control box' which makes sure data is passed in, decoded and acted on in the correct order, and a number of 8-bit (one byte) and 16 bit (two-byte) Registers. Just to confuse you, pairs of the one-byte Registers can also be used as two-byte Registers.

The Program Counter

Let us look first at the Program Counter (PC) two-byte Register. This holds the address of the NEXT instruction. It is automatically up-dated every time a new instruction is executed. However, the address it holds can be changed by, for example, a CALL instruction (like GOSUB in BASIC).

In this case, the address in the Program Counter is put aside - on the STACK - and the address CALLed is put in the Program Counter in its place. When the CALLed routine is done it meets a RET (RETURN) command, which takes the two-byte number ON THE TOP OF THE STACK and puts it back into the Program Counter. Execution then continues from that address. If you use the Stack (and you will use it), it is important to remember that the next instruction address after a RETurn is taken from the top of the STACK. Many a program has gone wild because a number has been unwittingly left on the stack: on the other hand, the fact that you know that the address of the next (apparent) instruction is on the Stack can be useful when, for example, transferring data to a subroutine.

A number of other instructions also affect the PC Register - jump instructions (JP or JR) for example. But for most instructions, the length of the instruction (including any information data elements) is added to the PC by the chip's control system, so that it knows where to look for the next instruction.

The Stack Pointer

Another two-byte Register, the Stack Pointer (SP), keeps track of the top of the Stack - since many instructions enable you, as well as the Z80, to use the Stack. The Stack area is within the RAM of your computer - and an address is set up by the ROM routines when you switch on.

You can if you wish set up your own address for the Stack but you must remember that the Stack runs BACKWARDS in memory, and it uses a last-in, first-out system. Think of it as a pile of plates, you can put plates on top or take them off the top, but you can't touch the plates anywhere else in the pile.

The other point about the Stack is that it ALWAYS accepts or delivers two-bytes of data. So, if we put 11A0H, 22B0H and 33C0H on the Stack in that order, and the Stack Pointer is loaded with F090 it will look like this:-

Address	Contents
F08B	C0
F08C	33
F08D	B0
F08E	22
F08F	A0
F090	11

The Stack Pointer in the Z80 will be pointing to the last (low) byte of the 33C0H data. If another piece of two-byte data - say 4567H - is put on the Stack, the Stack Pointer is DECREASED by one (decremented), the first (high) byte 45 hex is put into the address now pointed to by the Stack Pointer (F08A), the Stack Pointer address is DECREMENTED again, and then the low byte of the data, 67 hex, is put on the Stack (at F089).

When taking data off the Stack, the system works in reverse. In our example, first the Low order byte (67 hex) is removed, the Stack

Pointer is INCREMENTED, the high order byte (45 hex) is removed and the Stack Pointer INCREMENTED again. So now the Stack Pointer is once again pointing to the low order byte of the 33C0 hex data.

The 8-Bit Registers

There are two sets of 8-bit Registers:-

A, F, B, C, D, E, H, L
and A',F',B',C',D',E',H',L'

(Notice the F and F' Registers have been put next to the respective A Registers - that's because they are usually associated with the A Registers, and they have a function all of their own).

Only one set of these Registers can be used at a time. Why have two sets? So that you can 'stop' in the middle of one operation, switch to the alternate set, carry out an intermediate operation, then switch back and continue with the original operation. There are several ways of passing data between one set and the other.

Registers B and C, Registers D and E, and Registers H and L are also used as Register pairs to hold two-byte data. In a few commands, Registers A and F are also treated as a pair.

The A Register

The A Register is the Accumulator. It's where Almost All of the Action takes place. It is like Grand Central Station and in any program of consequence, it is kept extremely busy. Practically all comparisons, single-byte adding and subtracting instructions, and many special 'transfer' and 'load' instructions demand use of the A Register. God bless its cotton socks.

The B and C Registers

Several commands use the B Register or the B and C Registers together as a Byte Counter. (BC = Byte Counter - easy to remember).

Take for example the DJNZ Assembly command, which must always be followed by a Label. This instruction says 'Decrement whatever value is held in Register B by 1, and if it is NOT zero as a result, jump to the address denoted by the Label'. It's like a FOR-NEXT loop in BASIC, with the number of repetitions required being held in Register B. When B reaches zero, processing continues with the next instruction. (Note the mnemonic DJNZ = Decrement and Jump on Non Zero).

Similar commands (e.g. 'LDIR') use Registers B and C as a pair - permitting for example the transfer of large or small chunks of data from one area in the computer to another extremely quickly. The number of bytes to be transferred in this way is held in the Register pair BC.

Apart from these special uses, these two Registers can be used together or independently for your own requirements.

The D and E Registers

These too can be used independently, but are used together by some Z80 instructions to define a DEstination address. For example, the DEstination address of a block transfer of data (the 'LDIR' command again) is taken from Register pair DE: you have to put the address there, of course.

The H and L Registers

These Registers are used as a pair for quite a number of Z80 instructions. In the 'LDIR' command, for example, the start address of data to be transferred is taken from the contents of HL Registers - so don't forget to put it there. You'll find that there are quite a few commands which allow you to use the HL Registers to 'point' to data areas.

The F or 'Flag' Register

This is a very important Register indeed. Unlike the other 8-bit Registers, you cannot load data into it in the normal way. Its purpose is to hold Flag results of any logic and arithmetic operation undertaken, and for some other instructions, to 'flag' a status. The important point is that some of the Flags can be 'tested' to provide, for example, conditional jumps, calls or returns.

NOTE THAT WHILE MOST OF THE INSTRUCTIONS AFFECT SOME OR ALL OF THE FLAGS, FLAGS REMAIN IN A CURRENT STATE UNTIL AFFECTED BY A SUBSEQUENT INSTRUCTION. This means the state of a Flag can be tested several instructions after the instruction that affected it - but do be sure that the intermediate instructions do not affect the Flag in question. This feature can help to reduce the amount of coding needed. For example, all but two of the 'load' instructions do not affect the Flags at all. So if one of two subroutines are to be called, depending on the status of a particular Flag, and if both subroutines require the same 'load' at their start, then the 'load' can be done before the conditional test is made.

Certain bits of the Flag Register are allocated to specific functions, as follows:-

Bit Number:	7	6	5	4	3	2	1	0
Function:	S	Z	-	H	-	P/V	N	C
Testable:	*	*				*		*

The 'Testable' line indicates which of the Flags you can test in one way or another using the instructions available. Now we'll look at the functions of each one-bit flag.

The S or Sign Flag

This Flag 'repeats' the value of the most significant bit in the result of an arithmetic or logic operation, including 'shifts'. When a byte is transferred into the A Register, it 'repeats' the value of the most significant bit of that byte.

In many instances, bit 7 (the most significant) is used to indicate a particular condition. In 'two's complement' notation, for example (a brief discussion of which is given later in this chapter), bit 7 represents the SIGN of the number. This means the binary numbers are only 7 bits long, but represent from -128 to +128. In this instance, Bit 7 is 'SET' (equal to a '1') if the number is NEGATIVE and 'RESET' (equal to '0') if the number is POSITIVE. Bit 7 of a data byte can also play a role when a program is 'communicating' with input/output devices, such as a Printer. The S Flag enables Bit 7 of such a byte to be tested.

A number of Assembly commands allow the S Flag to be tested, by adding a 'P' (is it Positive?), or an 'M' (is it NEGATIVE?). The command JP (Jump), for example, can be turned into a CONDITIONAL jump by the addition of P - ' JP P,Label'. This tests the S Flag, and if it IS positive (i.e., equal to zero) as a result of some previous action, then the jump will occur. Otherwise processing continues with the next instruction.

The Z or Zero Flag

This Flag is used to indicate whether or not the result of an arithmetic operation is zero, or whether or not a 'comparison' test succeeds.

When a result is Zero or a comparison test succeeds, the Z Flag is set to a '1'. Otherwise, it is reset to a '0'.

The Z Flag can be tested by adding 'Z' (is it Zero?) or 'NZ' (is it Non-Zero?) to certain Assembly commands. For example, 'RET Z' (RETurn on Zero) provides a conditional return from a subroutine: if a previous operation has left the Z Flag set to '1', a RETurn will be made. Otherwise processing will continue with the next instruction. (As you can see, you don't have to worry too much about the actual value of the Z Flag bit - the Z80 looks at it and acts accordingly on your behalf).

The H or Half-Carry Flag

This Flag is used by the computer during Binary Coded Decimal arithmetic operations, to indicate whether or not there's been a carry from bit 3 to bit 4. It cannot be used in any conditional tests.

The P/V or Parity Overflow Flag

This Flag has three functions. Some instructions set or reset it according to whether the byte of a result has an even number of '1's (Parity Even = Flag set to "1"), or an odd number (Parity Odd = Flag reset to "0").

The second use of the P/V Flag is to indicate, during Binary Coded Decimal operations, whether or not Bit 7 (the 'Sign' Bit) has been affected by an overflow from Bit 6, thus accidentally changing the sign of the result.

Finally, during block transfer instructions, such as 'LDIR', this Flag is used to detect whether the counter has reached zero.

The Flag can be tested by adding 'PO' (is the Parity Odd?) or 'PE' (is the Parity Even?) to commands used to transfer program execution. For example, a CALL command can be turned into a conditional CALL if the Parity Flag is indicating 'odd', by writing 'CALL PO,Label' instead of the unconditional command 'CALL Label'.

The N or Subtract Flag

This Flag is used by the Z80 during its own Binary Coded Decimal calculations, and cannot be tested.

The C or Carry Flag

This Flag plays a dual role. First, it is used to indicate whether or not an addition or subtraction has resulted in a 'borrow'. If a borrow has occured, the Flag is set to "1". Otherwise it is reset to "0". Since comparison commands (e.g. CP B - which compares the contents of Register B with the contents of Register A) are achieved by subtracting the selected Register from Register A (and discarding the result), the Carry Flag can indicate whether the selected Register has a value greater than that in Register A (which produces a Carry), or has a value equal to or less than that in Register A (which produces a No Carry). Very useful.

The second use of the Carry Flag is in many of the rotate and shift instructions - which move data along the byte one way or the other in a particular manner. For these instructions, the Carry Flag is used as a 'ninth' Bit. For example, the RRA Assembly command (Rotate Right the Accumulator - Register A), moves Bit 0 of Register A into the Carry Flag, moves whatever was in the Carry Flag into Bit 7 of Register A, moves what was in Bit 7 to Bit 6 - and so on. Thus, this particular command effectively rotates the information held by the bits round one and includes the Carry Flag in the process.

With logical commands AND, OR, XOR, the Carry Flag is always set to '0' (No Carry). AND A and OR A will leave Register A intact, since the Register is being ANDed or ORed with itself, whilst XOR A not only clears the Carry Flag but also clears Register A, as there can be no 'exclusive' bits if it is being XORed with itself.

The Flag can be tested to produce conditional commands by the addition of 'C' (Carry) or 'NC' (No Carry) to the command. Thus a CALL command can be turned into a CALL if the Carry Flag is set, by writing 'CALL C,Label' instead of 'CALL Label.

How the Commands affect the Flags

The following Table shows how the Flags are affected by various types of Command. Commands not listed - e.g. 'PUSH' and most 'LD' commands - do not affect the Flags at all. Please note that, where unnecessary, the 'Register' element of the Command has not been included in the Table: thus the OR command could be OR A, OR B, OR C and so on - all having the same effect on the Flags. Only those Flags that can be tested have been included.

	FLAGS			
COMMAND	C	Z	P/V	S
ADD A,ADC,SUB,SBC, CP,NEG	?	?	?V	?
AND,OR,XOR	0	?	?P	?
INC,DEC	-	?	?V	?
ADD RR,CCF	?	-	-	-
RLA,RLCA,RRA,RRCA	?	-	-	-
RL,RLC,RR,RRC, SLA,SRA,SRL,DAA	?	?	?P	?
SCF	1	-	-	-
IN	-	?	?P	?
INI,IND,OUTI,OUTD	-	?		
INIR,INDR,OTIR,OTDR	-	1		
LDI,LDD	-		?	
LDIR,LDDR	-		0	
CPI,CPIR,CPD,CPDR	-	?	?	?
LD A,I; LD A,R;	-	?	IFF	?
BIT	-	?		

KEY:

? = Depends on the result of the operation.
?P = Depends on the Parity of result
?V = Depends on overflow in result
0 = Flag reset to zero
1 = Flag set to 1
- = Flag unaffected: previous state retained
IFF= Contents of interrupt flip-flop

Where there are blanks, the Flags contain irrelevant information.

To summarise the conditional tests available for JumP, CALL,Jum
Relative and RETurn commands:

Z = If result is Zero, act.
NZ = If the result is Not Zero, act.
C = If there's a Carry, act.
NC = If there's No Carry, act.
PO = If Parity is Odd, act.
PE = If Parity is even, act.
P = If the Sign Flag is 'positive (S=0), act.
M = If the Sign Flag shows a minus (S=1), act.

The Index Registers IX and IY

We now come to two very valuable 16-bit Registers in the Z80, the 'Index' Registers. Unlike Registers A to F, there is no 'second set' of Index Registers: their contents are accessible to both of the A to F Register sets.

The 'load' instruction commands related to these Registers can (indeed must, even if it's 0) include a displacement value. This enables, for example, data tables to be very easily set up, using the Register IX or IY to point to a 'base' address, and the displacement to point to the particular place required in the table.

An example will help to explain this. Supposing we decide to have a

Table of information that contains a number of names, addresses and telephone numbers. We allocate, say, 20 bytes to cover the name data, 60 bytes to cover the address data, 12 bytes to cover the telephone number data.

Our Table will then consist of a series of chunks, each 92 bytes long (20+60+12). We know that the telephone data for any name begins at the 80th byte from the start of the name. If we 'point' the IX Register to the start of the name in the Table, we know that the Telephone data will start at IX+80. This saves counting out the bytes to get to the correct address. A typical program might look like this:-

```
        LD B,11
        LD IX,NAME3
        LD DE,BUFFER
GETTEL:LD A,(IX+80)
        LD (DE),A
        INC IX
        INC DE
        DJNZ GETTEL
        Next operation
```

The first instruction sets up Register B as a counter.

The second instruction loads up the IX Register with the 2-byte address we require - that for NAME3.

The next instruction loads up Registers DE to point to a BUFFER area, where we want to hold the Telephone number - possibly for printing out.

We then come to the start of a little loop which will collect the bytes of data from the Table. We collect one byte, then increment the value in the IX Registers, increment the value in the DE Registers (i.e move both to point to the next address along), then collect another byte and so on until our 'counter', B reaches zero.

Note that LD A,(IX+80) means load Register A with the data byte to be found at the address pointed to by IX+80. Similarly, LD (DE),A means load the data byte in A into the address held in the Register pair DE.

The IY Register can, of course, be used in a similar way. As well *as 'loads', the Index Registers can be used for* ADD, INC, RLC, BIT *and SET commands - INC (IX+80), for example, means* go to the address *pointed to by IX+80,* and whatever byte is stored there, add one to *it.*

How big can the displacement value be? Glad you asked - because the displacement value is treated as a signed number. That means it can be 7 bits long, with the Most Significant Bit representing the sign of the value. So, to answer your question, the displacement value can be anything from -128 to +127, '0' being treated as a positive value.

The I and R Registers

Two more 8-bit Rregisters exist in the Z80 which can be accessed by commands. These are 'I', which stands for the Interrupt-Page Register, and 'R', which is the Memory-Refresh Register.

The I register is used in a special interrupt mode of operation to which the Z80 can be set (by command), and it stores the high-byte of an address that will be called in the event of an 'interrupt' process. The low-byte is generated by the device generating the 'interrupt'.

Let us briefly examine the concept of an interrupt. When you write a program, providing all is well, it will run the way you want it to, branching to subroutines and returning to the main program as scheduled. However, some input/output devices demand attention even while your program is running quite happily. The 'Video Display Processor' (VDP) in your MSX is one of these 'devices'.

An interrupt signal is sent by the device to the Z80. It says 'Hang on, I need attention'. Your 'main' program stops while the interrupt request is attended to - in the case of the VDP it is to 'refresh' the screen display - and then control is passed back to the main program, to continue where it left off (see footnote).

The programmer can call on the interrupt process himself, and indeed, you'll find a 'hook' at address FD9A and FD9F which is accessed 50 times a second. A Hook is 5 bytes in RAM which are initialised to Returns (they contain code C9 for RETurn) and the user can utilise these 5 bytes to do a CALL nnnn to his own interrupt routine and return to the main program.

There are three interrupt modes, called up by the commands IM 0, IM 1, and IM 2. In Interrupt Mode 0 - which is the mode your machine is in when you switch on - the external device must provide the instructions for what it wants the Z80 to do when it makes an interrupt request.

In Interrupt Mode 1 (which is the mode the ROM places the Z80 within microseconds of you switching on), when an interrupt request occurs an automatic jump is made by the Z80 to memory address 38 hex. The current location of any program running at the time is, of course, temporarily stored so that after the interrupt routine is complete, a return can be made to the original program. This interrupt mode always calls to address 38 hex. On the MSX, 38 hex provides a jump to the Hardware Interrupt routines at address 0C3C hex.

Footnote

Users who require further information on the VDP should refer to the publication 'Behind the Screens of MSX Home Computers' by Mike Shaw, which examines in detail the operation of the VDP and the way it is used in the MSX.

The third mode operates in a similar manner, except that it starts by going to one of 128 addresses (instead of one), as supplied by the calling device in conjunction with the contents of the I Register. Note that bit 0 of the address byte from the calling device is always zero.

The address pointed to, plus the next address, provide the 2-byte address of the interrupt handling routine, to which control is then passed.

In some programs it may be necessary to ensure that an interrupt does not occur during a specific process: a Dissable Interrupt command (DI) lets you do this - but for heaven's sake remember to Enable Interrupts (EI) again when that part of your program is complete.

Finally, the Refresh 'R' Register: this is provided to refresh dynamic memories automatically. You can use this as a kind of 'software clock', but since its values run only from 0 to 255 decimal, it's not exactly the most useful Register available.

There are a number of ways to classify the many Assembly commands you have at your disposal. We are going to herd them together under five headings to cover instructions which:

1. Transfer data from one place to another
2. Manipulate and test the data in some way
3. Re-route program running sequence
4. Handle input/output devices
5. System controls

Before we go into the commands, it may be useful to spend a few brief moments looking at the way a command is carried out by the Z80.

Every instruction is executed in three phases. In Phase 1, the instruction is fetched from the correct place in the program. The Program Counter tells the Z80 where to look (we dealt with this earlier). The first - perhaps only - byte of the instruction is then placed in a Register the Z80 keeps all to itself (called, believe it or not, the Instruction Register). In Phase 2, the instruction is decoded by the Z80 - that is, it sets up the cycle of operations for the third phase, which is to actually execute the instruction.

Each phase operates within finite steps, called clock cycles or T-States. The cycles themselves operate in 'machine cycles' - called 'M Cycles'. The shortest machine cycle lasts three clock cycles. Now as each cycle means a discrete unit of time, the more cycles an instruction needs for its fetching, decoding and execution, the longer it takes to execute. Pretty obvious really.

The point of all this is, generally speaking the more bytes there are to an instruction, the longer it takes to execute. However, the 'complexity' of the instruction also plays a part, so some instructions take longer than others of the same byte length. For example, the one-byte instruction to Decrement Register pair BC - DEC BC - takes 1 machine cycle, 6 T-States, while DEC A, also a one byte instruction, takes 1 machine cycle, 4 T-States. DEC A is faster by 2 T-States - or one miserable microsecond if the clock is 'running' at 2 MHz. or even less at 3.58 MHz on the MSX.

For the newcomer to machine coding, this discussion on machine cycles and T-States should be quite enough to cope with: it is beyond the scope of this book to discuss the actual speed of every instruction, since that becomes important only when one has gained experience. As mentioned before, most machine code programs run quite fast enough without any fine pruning.

The 'Brackets' Convention

Before we finally get down to the commands, there is one 'convention' you must be perfectly clear about - and that is the use of 'brackets' within a command.

An address can be referred to in two ways. If we want the address itself, it is written in the normal way - 1234H, for example. If we wish to refer to the CONTENTS of the address, then the address is placed in brackets.

Thus the command 'LD HL,1234H' means 'load Registers HL with the address 1234 hex'. You will recall from an earlier discussion that the Low byte goes into Register L (34 hex), and the High byte goes into Register H (12 hex).

The command 'LD HL,(1234H)', on the other hand, means 'go to address 1234 hex, and whatever byte you find there, put it in Register L. Then go to the next address - 1235 hex - and put the byte you find there into Register H'. (Look back a few pages to refresh your memory on how the Z80 requires addresses to be stored). So if addresses 1234H and 1235H hold bytes 89 hex and 67 hex respectively, then HL will be left holding the value 6789 hex after this command.

Similarly, take the command 'LD A,(HL)'. This means 'go to the address pointed to by Registers HL, and put the byte you find there into Register A'. If the HL Registers had been 'set up' to hold 1234H, then whatever byte is at that address (in our example above, it was 89 hex) is loaded into Register A. If HL Registers had been 'set up' to hold 6789H, as in the second example above, then whatever byte is at the address 6789H gets loaded into Register A.

Note that the command 'LD A,HL' cannot exist, since you will be trying to load two bytes of data into a one-byte store. Even an MSX computer can't do that.

1. Data transfer commands

In this section, we will be looking at all the different ways you can shift one or more bytes of data from one place in memory to another - and that includes shifting data around the Registers themselves. For convenience, it also includes the 'creation' of new data - that is, loading a Register with a specific value rather than a value to be found elsewhere in RAM. What we won't include in this section are the commands which read or write to input or output devices.

You may think this an obvious point to make, but we'll make it nonetheless: data remains in an address or Register until it is 'overwritten'. Thus, if we say 'Load Register A from Register B (LD A,B) then both Registers A and B will be holding the data that was in B, and the data that was in A will be lost.

The 8-Bit Load Group

All 8-bit transfers are achieved by a straightforward load instruction which takes the following format:-

LD destination,source

Thus a typical example might be LD B,D - which means load the contents of Register D into Register B.

The following table shows the 8-bit load commands available:-

	Source of the load													
Load Dest.	A	B	C	D	E	H	L	(HL)	(BC)	(DE)	(IX+d)	(IY+d)	(nn)	n
A	x	x	x	x	x	x	x	x	x	x	x	x	x	x
B	x	x	x	x	x	x	x	x			x	x		x
C	x	x	x	x	x	x	x	x			x	x		x
D	x	x	x	x	x	x	x	x			x	x		x
E	x	x	x	x	x	x	x	x			x	x		x
H	x	x	x	x	x	x	x	x			x	x		x
L	x	x	x	x	x	x	x	x			x	x		x
(HL)	x	x	x	x	x	x	x							x
(BC)	x													
(DE)	x													
(IX+d)	x	x	x	x	x	x	x							x
(IY+d)	x	x	x	x	x	x	x							x
(nn)	x													

The Registers down the left hand side represent the DESTINATIONS of a load, and the Registers across the top represent the SOURCE of a load, in the command format 'LD destination,source'. The x's denote where a command is available.

So reading across the top line, you can have as valid commands: LD A,A; LD A,B; LD A,C; and so on. Notice that no command is available to load Register D from the address pointed to by Register pair BC (i.e. there's no LD D,(BC) command). Sad - but no problem.

In the Table, 'nn' means a two-byte number, which could represent an address. You'll notice that only Register A can be loaded from the contents of a specific address (top line - LD A,(nn)). Also, at the end of the Table, you'll see only Register A can be loaded into a specified address. Let's discuss the ramifications of this.

If you want to load a specific address with a data byte, you can either do it by first placing the data byte in Register A (if it isn't already there), then do a 'LD (nn),A' command (nn being the required address). Or - take a look at the horizontal line for '(HL)'. If HL is loaded with the desired address - i.e. LD HL,nn (we'll come to that command later on), then data from any of the Registers A,B,C,D,E and yes, even H and L can be loaded into the desired address - using the LD (HL), 'register' command.

If you study the Table, you'll see that the same applies 'in reverse' - that is, you can load any of the Registers (including H and L.) from the address pointed to by the HL Registers (vertical column (HL)). Thus, you can write LD C,(HL) - meaning load Register C with the contents of the address pointed to by HL. Easy isn't it, when you know how.

Now let's look at another aspect of this Table - that 'n' column on the right hand side. As you've probably already guessed, 'n' stands for a data byte - any value from 0 to FF hex or 255 decimal. Notice, now, how you can load a specific byte of data into the address pointed to by HL - the LD (HL),n command.

You may wonder, looking at the table, how you can load for example the contents of Register D into an address pointed to by Register pair BC - that is, how do you cope without a command LD (BC),D. Well, good Register management, in the first place. But that isn't always feasible. So you'll have to transfer the data in D to A (having first 'saved' A somewhere, if you want to keep it), using LD A, D; then simply use LD (BC),A.

Four commands missing from the Table which were discussed earlier but will not be required for a while are:-

LD A,I (load A from the Interrupt Register)
LD A,R (load A from the Refresh Register)
LD I,A (Load Interrupt Register from A)
LD R,A (load Refresh Register from A)

<u>The 16-bit Load group</u>

The basic format for 16-bit (two-byte) data loads is essentially the same as that for 8-bit loads, namely:-

LD destination, source

There are however some important exceptions, which we will come to in a moment. Since we are talking about two-byte loads, either the source or the destination must, of course, be a Register pair.

The following Table shows the commands available within the format 'LD destination,source':-

	Source of the load							
	BC	DE	HL	SP	IX	IY	nn	(nn)
Load Dest.								
BC							x	x
DE							x	x
HL							x	x
SP			x		x	x	x	x
IX							x	x
IY							x	x
(nn)	x	x	x	x	x	x		

Doesn't look a very busy Table, does it? It would appear that you can't - as an example - directly load Register pair BC from the contents of, say, Register DE. Appearances are correct: there is no LD BC,DE command. But as we shall see, this isn't really a problem.

In the Table, 'nn' of course represents two bytes of data - which could be an address, or simply a number for some arithmetical operation - while '(nn)' represents the CONTENTS of address 'nn'.

Probably the most important things to notice about this table are the absence of the A Register in a pairing, and the fact that the Stack Pointer Register, SP, can be loaded from the contents of Register pair HL, or the two-byte Registers IX or IY, or with an immediate address -'nn', or from the contents of a specific address - '(nn)'. So there are several ways to set up the Stack Pointer - or even to change it during a program (as long as you know what you're doing).

The reverse isn't true, however: as far as load - LD - commands are concerned, the SP address can only be loaded into '(nn)' - to save its value.

Now, what about the other ways we have to transfer two bytes of data, and what about the poor old A Register? What the Table could have shown is an extra column and an extra row headed (SP) - that is, for example, a LD (SP),BC command, or a LD BC,(SP) command. These functions are possible - but they are not invoked by this type of command.

Let's see what LD (SP),BC means. '(SP)' means the contents of the address 'named' in the Stack Pointer Register. That's the top of the Stack. So 'LD (SP),BC' means - 'put the contents of Register pair BC onto the Stack'. Similarly, 'LD BC,(SP)' means - 'load Register pair BC from the contents at the top of the Stack'. In both instances, the address held in the Stack Pointer Register is 'updated' after the transfer of each byte (see the earlier discussion on the Stack Pointer).

There is a command all of its own to put the contents of a Register pair on the Stack, and another command to take two bytes off. The commands are PUSH and POP, respectively.

These are the Register pairs and two-byte Registers you can PUSH and POP:-

AF,BC,DE,HL,IX,IY

Thus, to store the contents of Register pair DE on the Stack, you can write PUSH DE. And to get the data at the top of the Stack into DE, you can write POP DE.

You noticed, didn't you - Register pair AF can be PUSHed and POPed to and from the Stack. That's so you can conveniently put aside what may be important data in both or either the A Register and the Flag Register.

Now, what about that poser we set earlier - loading BC from DE, for example. How do we do that? There are two ways. One, you can PUSH DE, then POP BC - that puts DE's data on the Stack, then reads it off into BC. Method two - use the two single-byte load commands, LD B,D; LD C,E. Both methods work, both methods are exactly two instruction bytes long, both methods are used quite extensively. But, the PUSH and POP method makes the Z80 look 'beyond' itself and into RAM area to execute the commands - whereas the LD Register,Register method doesn't. So the LD Register,Register method is faster (by 16 T-States, as it happens). If you want to put the two byte data that's in one of the Index Registers IX or IY into a Register pair, then you have no option but to go via the Stack. Notice, though, you do not specify the 'displacement' with the Registers: it's PUSH IX, not PUSH IX+d.

There are some more commands that enable you to shift two bytes of data from one place to another. They are called 'Exchanges'. Here they are:-

```
EX (SP),HL
EX (SP),IX
EX (SP),IY
EX DE,HL
EX AF,AF'
EXX
```

An Exchange is different from a load, in that the contents of both places designated are 'swapped'. Thus, the first three commands swap the contents at the top of the Stack with the respective Register named - HL,IX or IY.

For example, when a subroutine is called (through a CALL command) the address of the next instruction after the CALL is put on the Stack. That's the address that will be put back into the Program Counter when a RETurn is made from the subroutine. But supposing we choose to put after the CALL command not the next instruction, but an item or items of data that we wish to pass into the subroutine. In the subroutine, we do an EX (SP),HL command. So now what was in HL is on the top of the Stack, and what was on the top of the Stack - the address of where our data is - is in HL. We can pick up the data now by doing, for example, a 'LD A,(HL)' command. Now - and this is important - we increment HL so that it points (or 'bumps') over the information byte(s) to the address of the next instruction, and then do another EX (SP),HL. The correct address for the next instruction when we RETurn is now in the right place ready to be picked up by the Program Counter, and we've passed data into the subroutine for processing. That's by no means the only way to pass data into a subroutine, but it is a useful way.

The EX DE,HL command is invaluable when doing arithmetical operations, or when you want to exchange a DEstination address in DE and a source address in HL.

The EXX command exchanges the contents of the three Register pairs BC,DE and HL with their counterparts in the second Register set - BC', DE' and HL'. But not, you'll notice, the AF Registers - they have their own command EX AF,AF'. The information contained in the second Register set is not worked on, merely 'held in abeyance', so you have another way of temporarily holding onto data without setting up storage addresses or using the Stack. However, you'll find in some computers, the second set is used quite extensively to handle interrupt routines and so on, so if you unwittingly wipe out or leave 'strange' data in the second set, you could have some peculiar things happening.

The Block Transfer Group

We now come to the commands which enable any number of data bytes to be transferred from one place in RAM memory to another. These commands and their functions are:-

LDI - Load (DE) from (HL)
Increment DE and HL
Decrement BC

LDIR - Load (DE) from (HL)
Increment DE and HL
Decrement BC
Repeat until BC = 0

LDD - Load (DE) from (HL)
Decrement DE and HL
Decrement BC

LDDR - Load (DE) from (HL)
Decrement DE and HL
Decrement BC
Repeat until BC = 0

All of these commands transfer the data byte found at the address pointed to by the Register pair HL, to the address pointed to by the Register pair DE. After each data transfer the value held in Register pair BC is decremented. (Obviously, these three Register pairs must therefore be 'primed' before the block transfer command is invoked).

In the case of the LDI and LDIR commands, DE and HL are incremented after each transfer, while for the LDD and LDDR commands they are decremented after each transfer. Thus HL and DE are always left pointing to the correct addresses for the next data byte transfer.

With the LDIR and LDDR commands, the transfer of data continues until BC becomes zero, at which point processing continues with the next command.

With the LDI and LDD commands, processing continues with the next command after each transfer: this enables other actions to be taken before the next transfer of data - though you must remember not to 'upset' the values in the DE,HL, or BC Registers (unless that is all part of your cunning program). The LDI and LDD commands set the P/V Flag to zero if they decrement BC to zero. The following program will transfer only those data bytes that have their most significant bit (Bit 7) 'set' - that is, equal to '1': the program assumes that DE and HL have been set up with the Destination and Source 'start' addresses, and that BC is set to count the maximum number of bytes to be examined, and transferred if Bit 7 is equal to '1'.

```
NEXT:LD  A,(HL)    ;Get 'next' byte
     BIT 7,A       ;Test top bit
     JR NZ,MOVE    ;Byte wanted - shift it
     INC   HL      ;Byte unwanted - increment HL
     DEC   BC      ; and decrement the counter BC
TEST:LD  A,B       ;Check if BC is zero
     OR    C       ;by ORing B with C
     JR NZ,NEXT    ;Do it again if BC not zero
     JR    DONE    ;BC is zero - so finish
MOVE:LDI           ;Move the byte
     JP PE,NEXT    ;Do again if BC not zero
DONE:Your next command...
```

Instead of the 'JP PE,NEXT' command after the LDI, one could do a relative jump back to the 'TEST' point - JR TEST - which checks if BC has reached zero after being decremented. But we wanted to demonstrate the use of the JP PE command. Note, incidentally, one cannot do a Relative Jump (JR Label) when testing for parity. But more about this, and the other commands 'BIT 7,A',INC and DEC later.

You may ask why do we need both LDIR and LDDR commands. It is so that we never 'overwrite' data we want to shift.

Suppose for example we want to shift a data block of 1001H bytes from 8000H to 8500H. If we use the LDIR command with HL pointing to 8000H and DE pointing to 8500H, the first byte will be transferred from 8000 to 8500H - overwriting data within the block of 1001H bytes we're going to transfer.

In this instance, we would use the LDDR command - and set the HL Register to point to the END of the block we wish to shift (i.e. 9000H), and DE to the END of the destination area (i.e. 9500H). So now, by the time DE has been decremented to 9000H, we've already shifted the data from there, so it's o.k. to overwrite it.

2. Data manipulation & test commands

The 8-Bit Arithmetic and Logic Group

The simplest arithmetical operation that can be done on a single byte is to add one to it (INC) or deduct one from it (DEC). These operations can be performed on the following Registers and addresses pointed to by Registers:-

A, B, C, D, E, H, L, (HL), (IX+d), (IY+d)

The Z, P/V and S Flags are affected as a result of the operation.

The rest of the operations in this section ALL operate on Register A: the OTHER data byte source - even if that is Register A as well, must be specified. The following sources can be used for the 'other' data byte:-

A, B, C, D, H, L, (HL), (IX+d), (IY+d), n

The 'n' of course represents a specific value.

The commands available are:-

ADD A; ADC A; SUB; SBC; AND; OR; XOR; CP

We will examine each command:-

ADD A (examples - ADD A,B; ADD A,(HL); ADD A,2)
Note the A Register must be specified. This command simply adds the specified data byte to that in Register A. Thus ADD A,(HL) means add the contents of the address pointed to by HL to the contents of the A Register, leaving the result in the A Register. If the result exceeds FF hex (255 decimal), the Carry Flag is set, and A holds the result minus 256. Thus, with FF hex in Register A 'ADD A,2' would result in A holding '1', and the Carry Flag set to '1'.

The Z, P/V and S Flags are also affected according to the result of the ADD operation.

ADC A (examples - ADC A,B; ADC A,(HL); ADC A,2)
This is exactly the same as the ADD command, except that the contents of the Carry Register before the operation commences are also added to Register A. Thus if the Carry Flag is set and Register A holds 21 hex, 'ADC A,2' results in A holding 24 hex, and, because the operation did not require a 'carry', the Carry Fla would be reset to zero.

SUB (examples - SUB B; SUB,(HL); SUB 2)
Note that Register A is not specified (unless one wants to SUB i.e. subtract the contents of A from A). This command subtracts t specified data from Register A, and leaves the result in Register As with 'ADD', the Flags are affected according to the result.

SBC (examples - SBC B; SBC (HL); SBC 2)

Similar to the SUB command, except that the contents of the Carry Flag are also subtracted from Register A.

AND (examples - AND A; AND (HL); AND 0FH)

This performs a logic AND function between the A Register and the specified data byte, leaving the result in Register A.

'ANDing' means 'compare the two bytes, bit by bit. If both bits are a 1, then the corresponding bit of the result will be a '1'. Otherwise it's '0' '.

Thus, with 0A7H in Register A, 'AND 0FH' produces:-

10100111 (A7 hex, 167 decimal)
00001111 (0F hex, 15 decimal)
Result = 00000111 (7)

This technique is often used to provide a 'mask' - that is, to eliminate parts of a byte that are not wanted. The 'masking' data - in the above example '0FH' - covers that part of the data byte we want to keep.

ANDing always resets the Carry Flag to zero. Thus AND A will reset the Carry Flag to zero, and leave Register A as it was before the operation: this command can therefore be used to clear the Carry Flag without upsetting Register A.

OR (examples - OR A, OR (HL), OR 80H)

This performs a logic OR function on the A Register, leaving the result in the A Register.

'ORing' means 'test the two data bytes, bit by bit. If either or both bits are a '1', then the corresponding bit in the result will be a '1'. Otherwise it's a '0' '

Thus with 1B hex in Register A, OR 80H produces:-

```
         00011011  (1BH, 27 decimal)
         10000000  (80H, 128 decimal)
Result = 10011011  (9BH, 155 decimal)
```

This can be a useful way to add in bits to a byte: if A for example holds a value between 0 and 9, OR 30H will leave in A the ASCII code for that number.

OR always clears the Carry Flag, and affects the other Flags according to the result. Thus, OR A leaves Register A unchanged, but clears the Carry Flag.

XOR (examples - XOR A, XOR (HL), XOR 0FH)
This performs a logic XOR function on the A Register, leaving the result in the A Register.

'XORing' means 'compare the two data bytes bit by bit. If one is a '1' and the other is a '0', then the corresponding bit of the result will be set to a '1'. Otherwise it will be '0' '. Thus if Register A holds 14H, then XOR 17H produces:-

```
         00010100  (14H, 20 decimal)
         00010111  (17H, 23 decimal)
Result = 00000011  (3)
```

XOR always resets the Carry Flag, and affects the other Flags according to the result. XOR A must always result in Register A becoming zero - thus this is a useful command to clear Register A and the Carry Flag to zero: the Zero Flag will be set to '1' - meaning the value of Register A is zero.

CP (examples - CP B, CP (HL), CP 9)
This subtracts the specified data byte from the value held in Register A - AND DISCARDS THE RESULT: thus, only the Flags are affected by the command.

If the Test byte is greater than that in Register A, then the Carry Flag will be set.

If the test byte is the same as that in Register A, then the Zero Flag will be set.

If the test byte is equal to or less than that in Register A, then the Carry Flag is reset.

The Sign Flag and the P/V Flags will be set or reset according to the value in Register A.

The 16-Bit Arithmetic & Logic Group

As with the 8-bit Group, the simplest commands in this Group are INC and DEC. These commands can be used to increment or decrement Register pairs:-

BC, DE, HL

and the 16-bit Registers:-

SP, IX, IY

Note however that, unlike the 8-bit INC and DEC, for the 16-bit versions, the Flags are completely unaffected.

The following Table shows the ADD, ADC and SBC commands available (indicated by the x's):-

	This pair	BC	DE	HL	SP	IX	IY
		with					
ADD	HL	x	x	x	x		
ADD	IX	x	x		x	x	
ADD	IY	x	x		x		x
ADC	HL	x	x	x	x		
SBC	HL	x	x	x	x		

Note that the SUB command is not available - the Carry Flag is always involved on a subtract operation. If you don't want the Carry Flag involved - in case it may be set to '1', use an OR A command first to clear it.

The ADD, ADC and SBC functions are the same as those for the 8-bit commands except, of course, here they are operating on 16-bits.

The 8-Bit Shifts and Rotates

These commands operate on a specified byte of information, shifting or rotating its contents 'to the left' or 'to the right'.

The byte operated on can be in:-

A, B, C, D, E, H, L, (HL), (IX+d), (IY+d)

The commands available are as follows:-

RLC (Examples - RLC B; RLC (HL))
This moves the contents of bit 0 to bit 1, bit 1 to bit 2 and so on. Bit 7 is moved into the Carry Flag AND into bit 0. The data is thus ROTATED Left, with the Carry Flag reflecting Bit 7. Note, for Register A the command can be written RLC A or RLCA: RLCA is a different command, requiring one less instruction byte.

RRC (examples - RRC B; RRC (HL))
This moves the contents of bit 7 to bit 6, bit 6 to bit 5 and so on. The contents of bit 0 are moved into the Carry Flag AND bit 7. The data is thus ROTATED Right, with the Carry Flag reflecting bit 0. Note for Register A, the command can be written RRC A or RRCA: RRCA is the shorter, faster version of the two.

RL (examples - RL B; RL (HL))
This moves the contents of bit 0 to bit 1, bit 1 to bit 2 and so on. Bit 7 is moved into the Carry Flag, and the Carry Flag contents are moved into bit 0. Thus nine bits are involved in a ROTATE Left. Note that for the A Register this command can be written RLA instead of RL A, RLA being a shorter, faster command.

RR (examples RR B; RR (HL))
This moves the contents of the Carry Flag into bit 7, bit 7 into bit 6 and so on. Bit 0 is moved into the Carry Flag. Thus nine bits are involved in a ROTATE Right. For the A Register, the command can be written RRA instead of RR A, RRA being the shorter and faster of the two commands.

SLA (examples - SLA B; SLA (HL))
This moves bit 0 into bit 1, bit 1 into bit 2, and so on. Bit 7 is moved into the Carry Flag. A '0' is placed in bit 0. Thus the data is SHIFTED left.

SRA (examples - SRA B; SRA (HL))
This moves bit 7 into bit 6, bit 6 into bit 5 and so on. Bit 0 is moved into the Carry Flag. Bit 7 is 'refilled' with its original value (this is for 'signed' arithmetic' operations, to preserve the sign bit 7). Thus the data is SHIFTED right, arithmetically.

SRL (examples - SRL B; SRL (HL))
This moves bit 7 to bit 6, bit 6 to bit 5 and so on. Bit 0 is moved into the Carry Flag, and a '0' is placed in bit 7. Thus the data is SHIFTED right.

Decimal Arithmetic Rotates

We now come to two very special rotate functions, used when handling Binary Coded Decimal Arithmetic. Both commands operate between Register A, and the data byte in the address pointed to by the Register pair HL (i.e. '(HL)'). They are:-

RLD

This command puts the bottom nibble (lower four bytes) of the A Register into the bottom nibble of (HL), the bottom nibble of (HL) into the top nibble of (HL), and the top nibble of (HL) into the lower nibble of Register A. The nibbles are thus rotated. The top nibble of Register A is unaffected by the operation.

RRD

This does the same as RLD, but in the other direction. Thus, the bottom nibble of Register A is moved to the top nibble of (HL), the top nibble of (HL) is moved to the bottom nibble of (HL) and the bottom nibble of (HL) is moved to the bottom nibble of Register A. The top nibble of Register A is unaffected by the operation.

BIT MANIPULATION

Quite often, one wants to test a specific bit in a data byte, to see whether it's a '1' or a '0'. Equally it can be very useful to be able to set a specific bit to a '1', or reset it to '0'. The Z80 allows you to do this.

The three basic command words available are:-

BIT b,l: Test bit 'b' at location 'l'
SET b,l: Set bit 'b' at location 'l' to a '1'
RES b,l: Reset bit 'b' at location 'l' to a '0'

The bit 'b' can, of course, be any bit from 0 to 7. (Remember that bit 7 is the most significant, and bit 0 is the least significant).

The location 'l' can be any of the following:-

A, B, C, D, E, H, L, (HL), (IX+d), (IY+d)

Thus there are three basic commands, each of which can operate on one of eight bits in ten different locations - a total of 240 commands in all. Typical examples of the three basic commands are now given.

BIT 3,B

This tests whether bit 3 of Register B is a '0' or a '1'. If it is a '0', the Zero Flag is set to a '1' so that a subsequent test for Zero would succeed. Thus, in this program segment:-

```
BIT 3,B
JP Z,WASZERO
```

a Jump will be made to the program segment labelled 'WASZERO' if BIT 3 of Register B is '0'. Otherwise, processing continues with the next command.

Note that whilst the Zero Flag is specifically set or reset by BIT commands, the Sign Flag 'S' and the Parity/Overflow Flag 'P/V' may or may not be affected - the information they contain is irrelevant and untestable. The Carry Flag is unaffected by the operation - it will contain a previously held value.

SET 7,(HL)
This command makes bit 7 of the data byte at the address pointed to by the HL Register pair equal to a '1'.

RES 5,(IX+3)
This command operates on the data byte at the address pointed to by the IX Register PLUS 3, resetting its bit 5 to a '0'. Thus if the IX Register holds '8000H', then the data byte at address '8003H' will have its bit 5 turned into a '0'.

These bit manipulation functions can prove invaluable in some types of program. To give just one broad example, in an Adventure game one data byte may be used to indicate the possible exits from a given location - a '0' meaning 'no exit', and a '1' meaning 'exit possible'. Bit 7 could represent North, bit 6 East and so on, with four bits 'left over' to represent say 'up', 'down' and two other possible ways out. Checking whether or not an exit is possible is then simply a matter of testing the appropriate bit: changing the status of an exit is simply a matter of 'SETting or RESetting it.

SPECIAL A and F REGISTER MANIPULATIONS

There are five instructions which operate specifically on Register A or on the Carry Flag in Register F. These are as follows:-

DAA

This is a very special command for use when performing Binary Coded Decimal arithmetic (BCD). In BCD, a four-bit nibble is used to store one decimal digit: thus one byte can store two decimal digits (this is referred to as 'packed BCD'). The values '11' to '15' decimal can all be represented within one nibble: however, for BCD we only want one decimal digit per nibble, and so the binary representations of '11' to '15' decimal are meaningless and not wanted.

Let us look at two examples. First, we will add '22' decimal to '43' decimal. The program to do this in Binary Coded decimal could be:-

```
LD  A,22H;22H = 0010 0010 binary,'22' in BCD
ADD A,43H;43H = 0100 0011 binary,'43' in BCD
```

As you can see, adding the binary values would yield 0110 0101 - which in BCD is '65'. Just what we wanted, so there's no problem. Now let us look at what happens if we add '26' decimal to '17' decimal. Using the program segment as before, the binary representation for this would be:-

```
0010 0110   (26H)
0001 0111   (17H)
```

and if we add these, we get

```
0011 1101   (3DH)
```

Here, the 'D' is meaningless as a decimal number. And that, patient reader, is where the DAA command comes in. Added after the 'ADD A' instruction in the program above, it Decimal Adjusts any result in the A Register. Thus, in the first example, the 'DAA' command would do nothing, for all is fine and dandy. But in the second example, it would see that things have gone wrong with the lower nibble, sort out exactly what had gone wrong (depending on whether we'd been adding or subtracting), and adjust the result accordingly. In the second example, it would leave Register A holding 0100 0011 - '43H' or 43 in BCD - which is correct. In this specific instance it achieves this result by adding a further 6 to the lower nibble, but don't worry about that. Sufficient to know that it makes the correct adjustment.

What you should know, however, is that to sort things out the DAA command makes use of the Flags - so after a DAA command, all the Flags are affected in some way.

CPL

This command 'complements' whatever value is held in the A Register: that is, every '0' becomes a '1', and every '1' becomes a '0'. Thus, if the A Register held the binary value '00101100', after a CPL command it would hold the binary value '11010011'.

This is called the 'one's complement' of the number, and is a way of representing positive and negative values. For example, a '5' in binary is represented by '00000101'. On the other hand '-5' can be represented by the 'one's complement', namely '11111010'. Notice that bit 7 is now '1' - representing a minus value. (See also the discussion on Flags).

The 'testable Flags are not affected.

NEG

In this command, the contents of Register A are subtracted from zero, and the resulting value is stored back in Register A. This is called the 'two's complement' of the number.

In two's complement representation, positive values are represented just as in 'one's complement' - i.e. in the usual signed binary way, with bit 7 showing the sign (0=positive,1=negative). Negative numbers however are represented as the 'one's complement' value PLUS one. Thus the two's complement of '-5' is '11111011'.

Why go to all this bother? Two's complement makes signed arithmetic easier for the computer to handle. Consider the sum '3 minus 5'.

00000011 (+3)
11111011 (-5)

Adding these (since we are representing the 'minus' as -5 in two's complement), we get:-

11111110

Here, bit 7 tells us the answer is negative. Taking the two's complement of 1111110, therefore, we get 00000010 (two's complement, remember, is the one's complement of 1111110, which is 0000001, plus 1). Thus, the value is '2', and the Sign is negative. Answer, -2. Just what the doctor ordered.

The Z80 command NEG, then, obtains the two's complement of a value in Register A and leaves it in Register A, thus saving the bother of doing a one's complement (CPL) and adding 1 (ADD A,1). This is a very scant description of the principles behind one's and two's complement arithmetic, but it should be enough to give the newcomer to machine coding an idea of what it's all about.

Note that all the Flags may be affected by NEG command.

CCF

This command 'complements' the Carry Flag in the F Register. If the Carry Flag is '0', then CCF makes it a '1'. If the Carry Flag is '1', CCF makes it '0'.

SCF

This command makes the Carry Flag equal to a '1' (i.e. 'Set Carry Flag').

There isn't a command to 'reset' the Carry Flag - that is, to clear it. However, as mentioned before, AND A and OR A will do this, without affecting anything else. XOR A clears the Carry Flag as well, but also clears Register A - makes it '0' - and consequently also sets the Zero Flag and possibly affects the Sign Flag (which reflects bit 7, remember). Observant readers might see that an alternative way to clear the Carry Flag would be to set it first (SCF), then complement it (CCF) - but this takes two bytes of instruction code, whereas OR A takes one. So it's not much good as an alternative. But well spotted anyway.

BLOCK COMPARISONS

The last 'manipulation and test' commands to be examined are the 'block comparisons'. In many respects these are similar to the 'block transfer' commands discussed earlier. They enable a whole chunk of data to be 'searched' to find a byte that is the same as that in Register A. Like the block transfer commands, they need you to set up the Registers first: HL with the start address of the area to be searched, BC with the number of bytes to be searched, and A with the data byte we're looking for. The commands are:-

CPI	Increment HL Decrement BC
CPD	Decrement HL Decrement BC
CPIR	Increment HL Decrement BC Continue until BC=0 or A=(HL)
CPDR	Decrement HL Decrement BC Continue until BC=0 or A=(HL)

As with the block transfers, the CPI and CPD commands enable other operations to be undertaken within the 'search loop'. When a match is found, the Zero Flag is set. When BC reaches zero, the P/V Flag becomes 0 (Reset).

The CPIR and CPDR commands whiz through the block to be searched until BC reaches zero, or a match is found.

When a match is found, of course, Register pair HL will be pointing to the matching byte in the data block.

3. Re-routing program running sequence

We now come to the commands which let you change the 'batting order' of your program instructions - the commands which emulate the 'GOTO's' and 'GOSUB'S' in BASIC, and of course 'RETURN'. In machine coding, however, you have more scope.

Jumps and Relative Jumps

The BASIC 'GOTO' instruction can be emulated by a JumP (JP) or a Relative Jump (JR). A straight Jump is like a straight GOTO. The format is:-

```
JP Label   or   JP address
```

'Label' of course representing the label you have given at a particular point in your Assembly Language program, or which has been defined by an EQUate.

Jumps can also be conditional - that is, any of the Flags can be tested, and the Jump made if the test succeeds. The format for this is:-

```
JP cc,Label   or   JP cc,address
```

where cc represents any of the Flag conditions that can be tested (e.g. NZ,Z,NC,C,PO,PE,P,M - see the section on 'Flags'). Thus a typical instruction might be JP NZ,ENDGAME, which means 'if the Zero flag is not set (non zero condition) - as a result of a previous operation - then continue processing from the address labelled ENDGAME'.

Relative jumps need a little explaining. Their instruction codes are shorter than straight jumps. The address they provide a jump to is relative to the current address, and is given by a displacement value: consequently the actual address doesn't figure in the instruction code itself. If none of the addresses within the routine itself are 'mentioned' directly, the routine can be located anywhere in memory. It is thus called a 'rellocatable' routine. Many programmers write small subroutines (to do specific functions) in a rellocatable form, so that they can add the routines to any major program they are preparing. All they need then is the 'start' point of the routine - which is done by a label.

The format for a relative jump is:-

```
JR Label   or  JR sc,Label
```

where 'sc' represents a conditional test. Unlike Jumps, which can test any of the Flags, only the Zero and Carry Flags can be tested for a conditional relative jump - i.e. Z, NZ, C or NC. So you cannot write, for example 'JR M,LABEL'.

The relative jump can be made forwards or backwards. The displacement value is in two's complement, and is added to the Program Counter plus 2. If you work it out, you'll find that relative jumps can be made to addresses within -126 and +129 bytes of the address of the first byte of the 'JR' instruction: fortunately, the Assembler calculates the displacement value for you when generating the machine code.

Special Jumps

There are four more kinds of jump you can do in machine coding. Three of these enable you to jump to an address specified in the Registers. They are:-

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

and they're extremely useful when using 'jump tables'. One could for example have a data table of items, each item being three bytes long. The first byte of each item would be the 'menu selector'. The next two bytes would be the address (in the order Low byte, High byte, remember) of the 'action' routine for that menu item. The 'menu selectors' through the table are searched (jumping over the next two bytes of the item where no match is found) until a match is found.

With HL pointing to the matching byte, it is then a simple matter to: INC HL (so it points to the Low Byte of the action address); LD E,(HL) - pick up the low byte in E; INC HL - point to the High byte of the action address); LD D,(HL) - pick it up; EX DE,HL - put the address into HL; JP (HL) - and go.

This procedure is just one of the many, many ways in which one can pick up the address of a required routine. It's also a fairly crude way, but it demonstrates a point.

The fourth kind of jump emulates to some extent the 'FOR-NEXT' loop in BASIC. It is a type of Relative Jump, and has the format:-

```
        DJNZ Label
```

For this instruction Register B is used as a counter, so you must set it up with a value equal to the number of times you want the operation done. At the beginning of the 'loop', you have a Label. When the DJNZ command is met, Register B is decremented and, if it is not zero as a result, a jump is made to the Label address. It is a Relative Jump, so the Label address must be within -126 and +129 bytes of the DJNZ instruction's address (the Assembler calculates the displacement for you).

You can jump out of the loop at any time - if a subsidiary test succeeds, perhaps. Register B will then be holding the number of operations left to do when the test succeeded - which may be useful information.

Calls

A 'CALL' command is just like 'GOSUB' in BASIC. Like the JP jump command, it can be unconditional:-

```
CALL Label    or    CALL address
```

or conditional:-

```
CALL cc,Label    or    CALL cc,address
```

the 'cc' representing one of the Flag tests, just as for the conditional Jump command.

When a CALL command is met, the Program Counter address for the next command is put on the Stack, ready for when a RETurn is made - we discussed this when reviewing the Registers of the Z80. You must therefore ensure that the Stack still has the RETurn address 'on top' when the RETurn is made (it's utter disaster if you don't).

Restore

There is another kind of special Call command, called RST - which stands for ReSTore. The format is:-

```
RST a
```

where 'a' stands for one of the following:-

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

When the RST command is encountered, the Program Counter address is put on the Stack (just as in a CALL command), and a jump is made to the specified address. The point about this instruction is that it is only one byte long, and provides an extremely fast jump.

You'll notice though that all the addresses concerned lie within the ROM area. So, for example, RST 00H gives you a cold start - like pressing 'reset', if your MSX has one. The other addresses provide jumps to specific routines used by MSX Basic, getting the next character in a Basic line of text, for slot management, for outputting to a currently operative device, and so on.

Returns

These RETurn control from a subroutine, just like 'RETURN' in BASIC. The format is:-

```
RET     or     RET cc
```

where 'cc' is one of the Flag tests, as for the jump (JP) and CALL commands.

There are two special Return commands. The first is RETI (return from an interrupt), which must always be preceded by an EI (Enable Interrupt) command. The second is RETN, which provides a return from a non-maskable interrupt, and resets the Z80's interrupt Flag to the condition it held before the non-maskable interrupt was made.

4. Input/Output commands

There are a number of commands available for inputs from or outputs to peripheral devices. In many ways most of these are like the block transfer commands, in that they enable blocks of data to be transmitted either automatically or within a 'loop' performing other functions. These particular commands are:-

Input commands	Output commands
INI	OUTI
INIR	OTIR
IND	OUTD
INDR	OTDR

For the input commands, the peripheral device addressed by Register C is 'read', and the information is loaded into the address pointed to by Register pair HL. Then Register B is decremented, and Register pair HL incremented (INI, INIR) or decremented (IND, INDR).

For the Output commands, the procedure is reversed - that is, the contents of the address pointed to by HL is output to the peripheral device addressed by Register C, B being decremented and HL incremented or decremented after each transfer.

For the input or output commands ending with 'R', the procedure continues apace until B = 0.

Four other input and output commands are available. These are:-

Input commands	Output commands
IN A,(p)	OUT (p),A
IN r,(C)	OUT (C),r

IN A,(p) loads Register A with a byte of data read from the peripheral Port 'p'. Similarly, OUT (p),A outputs the data byte in A to the port 'p'.

IN r,(C) and OUT (C),r do the same kind of thing, except the port device is addressed by the C Register, and the specified Register 'r' can be any of:-

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

5. System controls

These commands are used for controlling the Z80 'system':-

<u>NOP</u>

This means, quite simply, No OPeration. That is, do nothing. Carry on with the next command you find. It's useful when writing programs in Assembly language, to provide a suitable spot for a 'Breakpoint'. Since it takes time to 'execute', it can also be used to provide a very short (a very, very short) delay.

<u>HALT</u>

This shuts down the operation of the Z80 completely, until an interrupt is received, or a 'reset' performed.

<u>DI,EI</u>

These Disable or Enable the Interrupt procedures. Interrupts are discussed in the section on the Z80 Registers.

<u>IM 0,1 or 2</u>

The IM commands set the Z80 in a particular Interrupt Mode. See the discussion on Interrupts in the section on Z80 Registers.

NON Z80 COMMANDS (Pseudo Ops)

If using an Assembler, you'll find other commands are available which are essential for writing in Assembly Language. These are used by the Assembler to tell it what to do - reserve data space, assemble at a specific address and so on. They do not 'translate' into Z80 instruction codes, and will not normally appear in a dissassembled listing. Please refer to the manual for your Assembler for details of these commands.

3
Using ZEN Assembler

This chapter will deal with getting started on writing your own machine code programs using an Assembler/editor program such as ZEN which is widely available for all MSX home computers.

Any differences on entering programs between ZEN and other assemblers should be minimal as the principles are the same. If you already know the methods of entering lines into an assembler then some of this chapter obviously could be skipped, as we will start from loading the assembler and describe some of the errors which can too easily be made by first time users. The first program we will enter simply prints the alphabet along one screen line, which is not very exciting, but it is nice and short and will demonstrate how lines are entered.

The ROM section of memory (addresses between 0000 and 8000 hex) within the MSX not only contains the Basic Interpreter but routines for carrying out tasks for what are simply termed as housekeeping jobs, as were used in chapter 1. These routines take care of tasks such as printing a character on screen, printing a new line, accessing the clock, using the PSG chip, reading a program from tape, verifying and saving of programs etc., and obviously they are made full use of when running any program, Basic or machine code as it is far simpler to form a message to be printed from within your program and then simply call the ROM routine to get that message printed on the screen than writing a routine in your program to do the same job.

ZEN loads into RAM at A000 hex this is why one is instructed to enter CLEAR200,&H9FFF before BLOAD"ZEN",R is entered.

On completion the screen will display:-

ZEN >

Enter exactly, spaces included, all entries under the TO ENTER column (NOT THE DISPLAYED COLUMN) followed by the 'RETURN' key at the end of each line. There is an error in the program which has been entered deliberately and we will alter it later. Remember any calls or jumps to addresses between 0000H and 8000H are to routines within the ROM section, and their functions will be described.

DISPLAYED	TO ENTER
ZEN >	E
1	LOOP:EQU 0A003H
2	CALL 0849H
3	LD A,"A"
4	NEXT:CALL00A2H
5	INC A
6	CP "Z"+1
7	JR NZ,NEXT
8	LD A,0DH
9	CALL 00A2H
10	LD A,0AH
11	CALL 00A2H
12	JP LOOP
13	END
14	.
ZEN >	

At the end of a program one must enter 'END' on a separate line, and to cease entering and move back to command level a full stop must be entered on a separate line too.

Now we will analyse what has been entered.

Line 1 of the program was an equate line and this simply tells the assembler that the Label 'LOOP' equates to A003H which is the address we wish to jump to at the end of the program as one can see in line 9 we have entered JP LOOP, we don't need to specify an address to jump to as the assembler has noted which address LOOP equals. One reason for these equates is that if we wished to alter the address at some future stage we would not need to list the whole program and alter each line which contained this address, all that is required is to change the first line to the different address and the assembler will do the work for us. This address is the warm start entry point to the ZEN Assembler, when this short program finishes running we need to tell the computer where to jump to and the mainloop of ZEN seems to be as good a place at this stage, we don't want the program running off wildly into memory.

NOTE Whenever a hex number begins with a letter (A-F), as in this case, it must be prefixed with a zero as is shown in line 1 otherwise the Assembler could confuse it for a label which always start with an alpha letter. Secondly a colon must be entered between the label and the letters EQU.

Line 2 calls a ROM routine at address 0849H which simply clears the screen and returns to our program. This is similar to a GOSUB in basic but in this case the subroutine is already in ROM and all our program needs to do is call it.

Line 3 loads the A register with the value of the letter 'A'. ZEN is quite versatile in that it allows entries within quotes and it simply converts this to the Hex equivalent value of the letter, in fact this line would have the same meaning if we entered LD A,41H which is how it would be assembled and loaded into memory by ZEN anyway. 41Hex is the hexadecimal ASCII value for the letter 'A', or we could have entered LD A,65 which is the decimal ASCII value of the letter 'A' and so omitting the suffix H which signifies to ZEN that the value is decimal and ZEN must convert it to Hex.

Line 4 contains the label NEXT as we will jump back here to continue printing letters. It is followed after the colon by CALL to 00A2H which once again is a subroutine in ROM which prints the ASCII value currently stored in register A, and returns to our program.

Line 5 increments register A so the first time round after printing A on the screen we want it to increase its value by 1, so it will increase from 41H to 42H, the letter 'B'.

Line 6 compares the value of register A to see if it has reached Z + 1, and if it hasn't line 7 tests and jumps back to NEXT to do it all again. Once again it is easier to enter line 6 as "Z"+1 but when it is assembled this will be automatically altered to the ASCII Hex value of Z plus 1 making 5B hex.

Line 7 is the relative jump and here one can see the advantage of giving lines a label for one does not need to calculate the number of bytes to jump back as the assembler does it for us. Furthermore one could add extra lines between 4 and 7 which will obviously alter the amount of bytes to jump back over without the need to adjust anything else as the assembler will adjust the relative jump automatically providing the jump does not exceed -126 or +129.

Line 8 is only reached when register A equals Z+1, when the alphabet is completed, then line 8 loads register A with the ASCII code for a carriage return, which returns the cursor to the left most position on the line, and line 9 calls 00A2H again to print it.

> NOTE ASCII codes below 20 hex are control characters, for positioning the cursor etc., and can be used with a call to 00A2 as was done with the alphabet.

Line 10 loads A with the ASCII code for a line feed (0A) as not only do we require the cursor to return to the left of the screen we also want it to move down to the next line, so a further call is made to 00A2 in line 11 to carry out the task.

Line 12 puts us back under the control of ZEN when the program finishes with a jump to Loop (A003H).

The next task is to find out if we have entered the program correctly, some bright sparks may have noticed some errors already, as one will get errors when entering and it is better to discover some of the more common types of error messages at this early stage. Enter 'A' and 'RETURN', this tells ZEN we wish to assemble the program.
The screen will prompt for an 'OPTION' which will determine if we wish to assemble to a printer by entering 'P', or 'E' for an external device, or by entering 'V' for video to print on screen the assembled version, or if we just enter the 'RETURN' key on its own it will be assembled internally only stopping at a line which contains any errors, which is the fastest option. So after the 'OPTION' prompt enter 'RETURN'.

The screen will display:-

```
ORG!
   2 START:CALL 0849H
ZEN>
```

which simply means we did not enter the origin of the program, which is where in memory we want it to reside. This is obviously a major omission as the assembler must know where to place the program.
Enter 'T' followed by 'RETURN' and the first line of the program will be displayed. 'T' is the target line you wish to be displayed, entering 'T4' would display line 4, whereas just entering 'T' on its own moves up to the first line.

Entering 'E', as we did to begin entering the program, will let us enter extra program lines from the current line, which after entering 'T' will be line 1, and as we enter these extra lines all the lines already in the program will simply shift up a line, the existing line 1 will remain intact but will now become line 2 etc. We should also enter a line to determine where we wish the program to load into memory once it is assembled, this does not need to be the same address as the ORG address, but to keep this program as simple as we can we will load in the same place.

```
                  TO
DISPLAYED        ENTER

ZEN >             E
   1              ORG 0E000H
   2              LOAD 0E000H
   3              .
ZEN >
```

Note the full stop to bring us back into command level.

Entering 'T' and 'RETURN' will display line 1:-

```
   1 ORG 0E000H
ZEN >
```

Now entering 'P16' and 'RETURN' will list the program from line 1 through to the end of the program which is always displayed as 'EOF'. If one entered 'P8' only the first 8 lines would be listed, so if the whole program is to be listed ensure you enter 'P' followed by a value equal to, or larger than, the last line number. Notice that the original lines in memory have been moved up 2 lines.

Once again enter 'A' and 'RETURN' followed by 'RETURN' in response to 'OPTION' prompt to see if our program is correct and will assemble. If one entered the program as shown it should stop and display:-

```
HUH?
   6 NEXT:CALL00A2H
ZEN >
```

Faced with this error one must look at the line and discover the mistake because the prompt 'HUH?' does not tell us much, only this will happen many times when writing your own programs. The line looks O.K. but the fault lies in the basic fact that we did not

enter a space between CALL and the address.
Enter 'N' and 'RETURN' and the line will be displayed with the cursor to the right of the line of characters:-

```
   6 NEXT:CALL00A2H
```

Simply delete the characters from the right, by using the 'BS' backspace key as the cursor keys are inoperative under ZEN, until the cursor is over the first zero after CALL and enter a space followed by 00A2H and 'RETURN'.
The line should now look like this:-

```
   6 NEXT:CALL 00A2H
```

Entering 'A' followed by 'RETURN' twice should result in no error message this time and the 'ZEN' prompt should be displayed almost immediately on the next line, which tells us that it assembled O.K. and is loaded into memory.
Enter 'GE000H' followed by 'RETURN' and the screen will display:-

```
BKPT>
```

this is asking us to enter a breakpoint in the program, for if one is testing certain parts of a lengthy program it can be halted at a specified address in memory, and control will pass back to ZEN. This can be very useful as machine code programs run so quickly that it is very hard to keep track of them.
In this case we do not want to enter a breakpoint, so in response to the 'BKPT' prompt enter the 'RETURN' key.

The screen should clear and this display should appear:-

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
ZEN>
```

Don't expect too much from your first machine code program, this was only to demonstrate the principles in entering code, but now we have lost all the bugs it seems a good time to assemble the program onto the screen to see what has happened. Enter 'A' and 'RETURN' and this time when prompted for 'OPTION' enter 'v' and 'RETURN' and the result should be as follows:-

```
PAGE    1


                          ORG   0E000H
                          LOAD  0E000H
                    LOOP: EQU   0A003H
E000  CD4908              CALL  0849H
E003  3E41                LD    A,"A"
E005  CDA200        NEXT: CALL  00A2H
E008  3C                  INC   A
E009  FE5B                CP    "Z"+1
E00B  20F8                JR    NZ,NEXT
E00D  3E0D                LD    A,0DH
E00F  CDA200              CALL  00A2H
E012  3E0A                LD    A,0AH
E014  CDA200              CALL  00A2H
E017  C303A0              JP    LOOP
                          END

ZEN>
```

In the above program, due to its simplicity, we did not document the functions of any lines but in a longer program it will be essential to describe certain parts of the programs. Comments can be included in any line by simply entering a semi-colon followed by the comment. To add a comment to line 3 enter 'T3' and 'RETURN' followed by 'N' and 'RETURN' and line 3 should be displayed with the cursor to the right of the characters:-

```
3 LOOP:EQU 0A000H
```

add the following:-

```
                        ;JUMP ON END        and'RETURN'
```

This line when listed will now show the comments after the semi-colon which will remind one at a future date what the line was achieving. Unlike Basic ZEN only allows entry on a single screen line therefore if one required additional space for comments a line may be entered with no instructions just a semi-colon followed by the comments, these will be used on subsequent listings for clarity. If one has a printer the assembled listing to 'P' for printer will show the comment fields after an instruction formatted to the right of the paper, but they will not be displayed on screen when assembling to the 'V' for video option due to the limitations of the 37 column screen, unless they are entered on separate lines.

Alterations and Additions

If one followed and understood the instructions and how they worked try the following:-
Alter the program to print the alphabet from Z down to A.
Change line 5 to LD A,"Z"
line 7 to DEC A
line 8 to CP "A"-1
This will initially load register A with letter Z and instead of incrementing in line 7 it will decrement, so the first time round the value in register A will reduce to the letter Y and so on. Line 8 checks if has reached A-1 and if not loops back to print again.

SCREEN MESSAGES

One will almost certainly require messages and inputs to be printed on screen, and as this test program is short it is ideal for modifying quite simply. The first line after the Clear screen call is line 5, so enter 'T5' and 'RETURN' and line 5 will get displayed:-

```
    5 LD A,"Z"
ZEN>
```

Entering 'E' and 'RETURN' will now enable one to add lines to the program, and move the existing lines up in memory.

```
               TO
DISPLAYED      ENTER

   5           LD HL,MESG1
   6           CALL 6678H
   7           .
ZEN >
```

These new instructions are thus:-
LD HL,MESG1 loads register pair HL with the address in memory of the start of a screen message which will have the label MESG1 assigned to it. CALL 6678H is a ROM routine which prints, at the cursors current position on screen, the message which starts at the address stored in HL. The message, as you will see below, also contains any control characters to move the cursor which can be entered before or after the quotes containing the string. Furthermore the message must terminate with the NOP code (0) which is used as the 'End of String' marker.

The next job is to enter MESG1 into our program. List the program on screen to discover the last line number, as it is here we will place the string of characters in our message. END should appear as line 17, so enter 'T17' and 'RETURN' followed by 'E' and 'RETURN'

```
               TO
DISPLAYED      ENTER

   17          MESG1:DB"TEST",0DH,0AH,0
   18          .
ZEN>
```

If your message was longer than can fit onto one line then finish the first line of the message by adding the closing quotes and continue the message on the following line making sure it commences with 'DB"' and only enter ',0' at the end of message.

In order to run the program it must be assembled again, making sure no bugs have crept in. When assembling to the screen it will be seen that long messages are not printed in full to the right of the screen, however the bytes representing that message are entered into memory as will be seen on the left of the screen. If the MESG1 line was entered as shown above the display would actually cut off after the comma following '0DH'

Running the program can be entered as 'GE000H'. It will be seen that the screen clears and 'TEST' gets printed on the first line, and the alphabet gets printed, in reverse order, on the following line. One could have entered additional codes for line feeds '0AH' to print further down the screen.

Ensure your program lists as below, as we shall alter it further.

```
    1 ORG 0E000H
    2 LOAD 0E000H
    3 LOOP:EQU 0A000H;JUMP ON END
    4 CALL 0849H
    5 LD HL,MESG1
    6 CALL 6678H
    7 LD A,"Z"
    8 NEXT:CALL 00A2H
    9 DEC A
   10 CP "A"-1
   11 JR NZ,NEXT
   12 LD A,0DH
   13 CALL 00A2H
   14 LD A,0AH
   15 CALL 00A2H
   16 JP LOOP
   17 MESG1:DB"TEST",0DH,0AH,0
   18 END
EOF
```

USER INPUTS 1

We will assume that we wish the user to input a number from 1 to 9 in order for the alphabet to be printed several times. A routine exists within the ROM that will stop the program and wait for a key to be pressed before continuing and can be utilised quite simply.

Alter line 17 by entering 'T17' and 'RETURN' followed by 'N' and 'RETURN' to alter MESG1. With the cursor to the right of the line delete back with the Backspace key to the start of the string and alter the line to the following:-

```
17 MESG1:DB"INPUT 1to9",0AH,0DH,0
```

We also need to change the program to accept an input from the keyboard between 1 and 9. Enter 'T7' and 'RETURN' followed by 'E' and RETURN'.

DISPLAYED	TO ENTER
7	TIMES:CALL 009FH
8	CP 31H
9	JR C,TIMES
10	CP 3AH
11	JR NC,TIMES
12	SUB 30H
13	LD B,A
14	.
ZEN>	

A label must be added to the current line 14 as it will be required to loop back. Alter it to read:-

```
14 START:LD A,"Z"
```

Line 7 (labelled TIMES) now calls a routine within ROM (009FH) which halts the program and waits for a key to be pressed. Once a key is pressed the subroutine returns to our program with the ASCII value of the key stored in register A.

As we only require keys 1 to 9 to be accepted the contents of register A must be checked, and line 8 checks that the key pressed was equal to or greater than 31H, which is the ASCII code for the number 1 (check with the ASCII code table). It simply subtracts (temporarily) 31H from the A register and if it contained a lower ASCII code than 31H the carry flag will be set, hence line 9 is a relative jump back to line 7, for the processor to wait for another key to be pressed, if there was such a carry.

Subsequently the program must now check for a higher key than 9. Line 10 compares for 3AH, which in the ASCII table will be seen to equal the colon ':' which is one higher than 9. Line 11 is a relative jump back to line 7 if after subtracting 3AH from register A the carry flag is not set then the key pressed must have been equal to or higher than 3AH, which means the key was higher in the ASCII table than 9 and we must jump back and wait for another key.

Assuming that a correct key was entered we now know register A contains a number between 31H and 39H and we must convert this to between 1 and 9, and line 12 does exactly that it subtracts 30H from register A leaving it with a value 1 to 9.

Line 13 loads register B with the contents of register A, as B is to be the counter for the amount of times we will print the alphabet. There is one extra line. Enter 'T23' 'RETURN' and 'E' 'RETURN'

```
             TO
DISPLAYED    ENTER

   23        DJNZ START
   24
ZEN>
```

This command was discussed in the 'Special jumps' section in chapter 2 and is a unique Z80 instruction for the B register which decrements B and executes a relative jump back to wherever you nominate, to carry out the instructions in the loop again until B decreases to zero, similar to a FOR..NEXT loop in Basic. In this case it jumps back to line 14 which is labelled START.

One will have to assemble the program before it is capable of being run. If errors occur during assembly refer back to the specified line and check it in this chapter. To run enter 'GE000H' 'RETURN' and for BKPT enter 'RETURN'.

The assembled listing:-

```
PAGE    1

  1                               ORG   0E000H
  2                               LOAD  0E000H
  3                     LOOP:     EQU   0A000H              ;JUMP ON END
  4  E000  CD4908                 CALL  0849H
  5  E003  2130E0                 LD    HL,MESG1
  6  E006  CD7866                 CALL  6678H
  7  E009  CD9F00       TIMES:    CALL  009FH
  8  E00C  FE31                   CP    31H
  9  E00E  38F9                   JR    C,TIMES
 10  E010  FE3A                   CP    3AH
 11  E012  30F5                   JR    NC,TIMES
 12  E014  D630                   SUB   30H
 13  E016  47                     LD    B,A
 14  E017  3E5A         START:    LD    A,"Z"
 15  E019  CDA200       NEXT:     CALL  00A2H
 16  E01C  3D                     DEC   A
 17  E01D  FE40                   CP    "A"-1
 18  E01F  20F8                   JR    NZ,NEXT
 19  E021  3E0D                   LD    A,0DH
 20  E023  CDA200                 CALL  00A2H
 21  E026  3E0A                   LD    A,0AH
 22  E028  CDA200                 CALL  00A2H
 23  E02B  10EA                   DJNZ  START
 24  E02D  C300A0                 JP    LOOP
 25  E030  494E5055     MESG1:    DB    "INPUT 1to9",0AH,0DH,0
 25  E034  54203174
 25  E038  6F390A0D
 25  E03C  00
 26    .                          END
```

USER INPUTS 2

This section deals with user inputs of unspecified length, as against single key inputs, entering a string from the keyboard to be printed a number of times.

In this example all addresses have been labelled, as one would when writing a longer program, and entering should be good practise at getting it right. Enter 'K' and 'RETURN' to kill the existing program followed by 'E' and 'RETURN'.

```
                TO
DISPLAYED     ENTER

    1         ORG 0E000H
    2         LOAD 0E000H
    3         LOOP:EQU 0A003H
    4         ;ROM ROUTINES
    5         PTMESG:EQU 6678H
    6         PINLIN:EQU 00AEH
    7         CLS:EQU 0849H
    8         INPBUF:EQU 0F55EH
    9         CHGET:EQU 009FH
   10         CHPUT:EQU 00A2H
   11         ;CONTROL CODES
   12         BL:EQU 7
   13         CR:EQU 0DH
   14         NEWLNE:EQU 0AH
   15         ;
   16         CALL CLS
   17         LD HL,MSG1
   18         CALL PTMESG
   19         CALL BELL
   20         CALL PINLIN
   21         CALL CRLF
   22         LD HL,MSG2
   23         CALL BELL
```

```
CALL PTMESG
TIMES:CALL CHGET
CP 31H
JR C,TIMES
CP 3AH
JR NC,TIMES
SUB 30H
LD B,A
AGAIN:CALL CRLF
LD HL,INPBUF
NEXTCH:LD A,(HL)
CP 0
JR Z,FINI
CALL OUTPUT
INC HL
JR NEXTCH
FINI:DJNZ AGAIN
CALL CRLF
JP LOOP
;
;OUTPUT ROUTINES
BELL:LD A,BL
JR OUTPUT
CRLF:LD A,NEWLNE
CALL OUTPUT
LD A,CR
OUTPUT:CALL CHPUT
RET
;
;MESSAGES
MSG1:DB"ENTER A "
DB"STRING",0DH,0AH,0
MSG2:DB"INPUT 1to9",0DH,0AH,0
END
.
ZEN
```

Line 16 commences the program with a call to the clear scre routine, (CLS) assigned an address in the equates, at 0849H. T message to enter a string is loaded into HL and printed by a call 6678H (PTMESG).

Line 19 calls BELL which is entered in the output routines in lin 45 where register A is loaded with the desired character, in this case BL (7), and the program jumps to OUTPUT (line 50) where a call to CHPUT (00A2H) outputs the contents of reg A afterwhich a return is made back to the next line (20). This line calls the ROM routine PINLIN (00AEH) which allows input from the keyboard until the 'RETURN' key is pressed and stores the string in the input buffer (INPBUF) at 0F55EH. Line 21 calls the carriage return and line feed subroutine (line 47) where once again reg A is loaded with the ASCII value of the control character, first with NEWLNE (0AH), and is outputted by a call to OUTPUT in line 50. Line 51 returns to the line after the last call (line 49) where A is loaded with the CR code (0DH) and this time the program runs into, it does not call, line 50 to output the character in A once again. This time line 51 will return to the program line after the original call (line 22) whereupon the second message is loaded into HL and is followed by a call to BELL and the message is printed in line 24.

Lines 25 to 31 get a number from 1 to 9 as the previous program. Line 33 loads the start of the input buffer, where the string is stored, into HL and line 34 loads the first character of the string into reg A and it gets printed in line 37 which calls OUTPUT. When a string is stored in the input buffer the byte after the last byte of the string is loaded with a zero, therefore in line 35 we compare the contents of reg A for zero and if the test is positive a relative jump to FINI (line 40) is carried out in line 36. Line 38 increments HL to move it up to the next character in the input buffer and line 39 jumps back to NEXTCH (line 34) to load the character into reg A once again and compare it with zero.

Line 40 decrements reg B, which was set up as a counter, and loops back to line 32 (AGAIN) to print the string once more. Line 41 performs another line feed and carriage return before the program jumps back to LOOP (0A003H) the warm start address of ZEN. The main difference with the ZEN addresses 0A000 and 0A003 is that on completion a jump to 0A003 will maintain the condition of the registers allowing one to enter 'X' and 'RETURN' to examine the user registers. Very useful when programs are playing up.

The following assembled listing is reproduced using the 'P' option for printer, the main differences between this and the 'V' video option is that line numbers are included in the printout and the any comment fields are shown in full due to the additional columns being available. To run the program enter GE000H and 'RETURN' twice, afterwhich the screen will clear and the 'ENTER A STRING' message will be printed. After one has entered a string of characters the 'INPUT 1to9' message will be shown and on entering a value between 1 and 9 the string will be printed.

```
PAGE    1

  1                                  ORG   0E000H
  2                                  LOAD  0E000H
  3                     LOOP:        EQU   0A003H
  4                     ;ROM ROUTINES
  5                     PTMESG:      EQU   6678H
  6                     PINLIN:      EQU   00AEH
  7                     CLS:         EQU   0849H
  8                     INPBUF:      EQU   0F55EH
  9                     CHGET:       EQU   009FH
 10                     CHPUT:       EQU   00A2H
 11                     ;CONTROL CODES
 12                     BL:          EQU   7
 13                     CR:          EQU   0DH
 14                     NEWLNE:      EQU   0AH
 15                     ;
 16 E000  CD4908                     CALL  CLS
 17 E003  2151E0                     LD    HL,MSG1
 18 E006  CD7866                     CALL  PTMESG
 19 E009  CD42E0                     CALL  BELL
 20 E00C  CDAE00                     CALL  PINLIN
 21 E00F  CD46E0                     CALL  CRLF
 22 E012  2162E0                     LD    HL,MSG2
 23 E015  CD42E0                     CALL  BELL
 24 E018  CD7866                     CALL  PTMESG
 25 E01B  CD9F00        TIMES:       CALL  CHGET
 26 E01E  FE31                       CP    31H
 27 E020  38F9                       JR    C,TIMES
 28 E022  FE3A                       CP    3AH
 29 E024  30F5                       JR    NC,TIMES
```

```
E026 D630                  SUB  30H
E028 47                    LD   B,A
E029 CD46E0    AGAIN:      CALL CRLF
E02C 215EF5                LD   HL,INPBUF
E02F 7E        NEXTCH:     LD   A,(HL)
E030 FE00                  CP   0
E032 2806                  JR   Z,FINI
E034 CD4DE0                CALL OUTPUT
E037 23                    INC  HL
E038 18F5                  JR   NEXTCH
E03A 10ED      FINI:       DJNZ AGAIN
E03C CD46E0                CALL CRLF
E03F C303A0                JP   LOOP
               ;
               ;OUTPUT ROUTINES
E042 3E07      BELL:       LD   A,BL
E044 1807                  JR   OUTPUT
E046 3E0A      CRLF:       LD   A,NEWLNE
E048 CD4DE0                CALL OUTPUT
E04B 3E0D                  LD   A,CR
E04D CDA200    OUTPUT:     CALL CHPUT
E050 C9                    RET
               ;
               ;MESSAGES
E051 454E5445  MSG1:       DB   "ENTER A "
E055 52204120
E059 53545249              DB   "STRING",0DH,0AH,0
E05D 4E470D0A
E061 00
E062 494E5055  MSG2:       DB   "INPUT 1to9",0DH,0AH,0
E066 54203174
E06A 6F390D0A
E06E 00
                           END
```

SAVING PROGRAMS

Although one probably won't need to save this program on tape it is a good idea to use this small program to practise getting it right, it is not so straightforward as saving a basic program, so making mistakes now will be less costly than when your own machine code masterpiece is at stake.

ZEN has 2 methods of saving machine code programs. The first is to save the source file as an ASCII text file. ASCII text files (or programs) are made up of the pure text which has been entered from the keyboard. One will require this option for saving unfinished programs, which obviously cannot be assembled in that state, for future loading using ZEN which would be achieved by entering 'R' and 'RETURN' after the ZEN prompt.

Entering 'H' and 'RETURN' will now display the start and end of the source file and the top of memory. At this stage the last program should display, if one hasn't added extra spaces or comments:-

```
C000 C2A7 F37F
```

If one enters 'QC000H' and 'RETURN' the text entered will be shown in memory byte by byte. To save an ASCII text file using ZEN enter 'W' and 'RETURN' and one will be prompted for a file name, afterwhich it will be saved on tape as normal. Afterwhich one should verify the saved file.

The second method is for saving the object file as a binary file. Binary files are the assembled program, and what gets saved is the pure machine code file, without comments, ready to run. In the last program it could be saved and then run directly from the loading, without ZEN being present, by simply BLOAD although one would need to alter line 42 from JP LOOP to RET as we would not require a jump to A003 if ZEN was not loaded.

To test that one is conversant in saving a binary file carry out the following:-

Alter what should be line 42 to read:-

```
   42 RET
```

One will need to assemble the program once again but if the above entry is correct that will take no time at all only this time assemble to the screen by entering 'V' and 'RETURN' as we MUST know the end address of the file. After altering line 42 the program will be 2 bytes shorter making the end of program E06CH.

Place a fresh tape in the cassette and enter 'WB' which stands for write binary. One will be prompted for the START address so enter 'E000H' and 'RETURN', it is important to enter the suffix 'H' otherwise ZEN will believe it is a decimal number which it is not.

Next prompt is for the STOP address so enter 'E06CH' and RETURN' which is the last byte of the program.

The next prompt is for the EXEC address which is where the program should run from. In this case we want to run from the same address as it loaded from so enter once again 'E000H', 'RETURN'. EXEC is added because a program does not always execute from its start address in memory. It may be that a program is written and then has some screen graphics titles added to the end of it but which one wants to run first, so the execution address could well be different to that of the loading one.

This is followed by the LOAD prompt for the address at which it should load into, and again enter 'E000H'.

The final prompt is for a file name, we could simply call this 'TEST' and all that remains is to set the cassette to record mode.

Once the file has been saved switch off the computer, wait a few seconds, (never switch off and on quickly) and turn it back on and load the test program by entering:-

```
BLOAD"TEST",R
```

which after a few seconds, will automatically run if you saved it correctly, and when finished will jump into the Basic mainloop and display the 'Ok' message.

Please understand that this was an exercise to correctly save and subsequently load and run machine code programs, which normally should be far more exciting than the Test program, and it is far less costly in programming hours to get it right at this stage than get it wrong and lose many hours work.

CRASHES

When testing programs in ZEN it is quite feasible that there may be something wrong with your program and it may crash, fall out of Zen's control, into Basic or even re-initialise and display the switch-on MSX screen message. One should be able to jump back into ZEN by entering:-

```
DEF USR=&HA000
A=USR(0)
```

and hopefully ZEN, and your program, will still be in memory and debugging can continue. This may also happen when accessing Basic routines from a machine code program for if an error occurs the error trap routine within Basic could pick up the error, display an error message, and dump you into Basic's mainloop with the 'Ok' message. To simplify the restoration one could enter the above 2 lines with line numbers, making it a 2 line Basic program, and if the crash was not too severe entering the 'F5' key for Run should restore control to ZENs mainloop.

4

MSX Routines

This chapter demonstrates more routines which are provided in the ROM of the MSX and how to access them.

TABLE CONSTRUCTION

The following program uses keyboard input to produce notes in the range C to B in any of the 8 octaves, which gives it some appeal, but its main purpose is to demonstrate one method of accessing tables. The keys which will produce sounds are as follows:-

```
  R T   U I O
 D F G H J K L
```

The lower row are used for notes C to B whilst the keys on the top row signify the sharp keys (C+ etc). Pressing the '£' key exits the program. The Octave is set to 4 when the program runs but can be altered by entering keys 1 to 8 while it is running.

The current note and octave are displayed on the screen. The program uses the Basic PLAY routine at 73E5H, therefore the string to be played must look as it would in a Basic program line with quotes (") surrounding it and, like the previously used print strings, must terminate with a zero value byte otherwise an error will occur and the program will drop into Basic and display an error statement.

One should now be familiar with entering programs so only the assembled listing is shown, however the various ROM routines which are utilised are described after the listing.

```
                        ORG   0E000H
                        LOAD  0E000H
                QUIT:   EQU   0A003H           ;ZEN MAINLOOP
                CHGET:  EQU   009FH            ;WAIT FOR KEY
                CLS:    EQU   00C3H            ;CLEAR SCREEN
                POSIT:  EQU   00C6H            ;CURSOR SET UP
                PTMESG: EQU   6678H            ;PRINT MESSAGE
                CLIKSW: EQU   0F3DBH           ;KEY CLICK SW
                CHPUT:  EQU   00A2H            ;OUTPUT CHARACT.
                ERAFNK: EQU   00CCH            ;ERASE FUNC KEY
                CHSNS:  EQU   009CH            ;KEY SCAN
                ;
E000 CDCC00     START:  CALL  ERAFNK           ;FUNC KEYS OFF
E003 AF                 XOR   A                ;ZERO A
E004 32DBF3             LD    (CLIKSW),A       ;TURN OFF CLICK
E007 CDC300             CALL  CLS              ;CLEAR SCREEN
E00A 2608               LD    H,8              ;SET CURSOR COLUMN
E00C 2E02               LD    L,2              ;SET CURSOR LINE
E00E CDC600             CALL  POSIT            ;POSITION CURSOR
E011 21CCE0             LD    HL,MESG1
E014 CD7866             CALL  PTMESG
E017 260C               LD    H,12
E019 2E0A               LD    L,10
E01B CDC600             CALL  POSIT
E01E 21E0E0             LD    HL,MESG2
E021 CD7866             CALL  PTMESG
E024 260E               LD    H,14
E026 2E0C               LD    L,12
E028 CDC600             CALL  POSIT
E02B 21E9E0             LD    HL,MESG3
E02E CD7866             CALL  PTMESG
E031 C34CE0             JP    PTOCT            ;PRINT OCTAVE VALUE
E034 CD9C00     INPUT:  CALL  CHSNS            ;IS KEY DOWN
E037 28FB               JR    Z,INPUT          ;NO LOOP BACK
E039 CD9F00             CALL  CHGET            ;GET KEY IN REG A
E03C FE23               CP    "£"              ;IS IT £ KEY
E03E CA03A0             JP    Z,QUIT           ;YES FINISH
```

```
E041 FE31                 CP    31H             ;TEST FOR 1
E043 38EF                 JR    C,INPUT         ;IF LESS GET NEXT
E045 FE39                 CP    39H             ;TEST FOR 9
E047 3012                 JR    NC,SAMOCT       ;STILL SAME OCTAVE
E049 3295E0               LD    (OCTVE+1),A     ;DISPLAY OCTAVE
E04C 2615       PTOCT:    LD    H,21            ;POSITION CURSOR
E04E 2E0A                 LD    L,10            ;TO PRINT OCTAVE No.
E050 CDC600               CALL  POSIT
E053 3A95E0               LD    A,(OCTVE+1)     ;NEW OCTAVE
E056 CDA200               CALL  CHPUT           ;PRINT IT
E059 18D9                 JR    INPUT           ;GET NEXT KEY
E05B CD9000     SAMOCT:   CALL  0090H           ;NO QUEUES
E05E 47                   LD    B,A             ;SAVE KEY IN B
E05F 21A7E0               LD    HL,TABLE
E062 7E         COMPR:    LD    A,(HL)          ;TABLE IN A
E063 FE0F                 CP    0FH             ;END OF TABLE?
E065 28CD                 JR    Z,INPUT         ;YES WRONG KEY
E067 23                   INC   HL
E068 B8                   CP    B               ;COMPARE KEY/TABLE
E069 2804                 JR    Z,FOUND         ;GO PLAY
E06B 23                   INC   HL              ;NOT FOUND. BUMP OVER
E06C 23                   INC   HL              ;NOTE STRING AND
E06D 18F3                 JR    COMPR           ;TEST NEXT IN TABLE
                ;
E06F 7E         FOUND:    LD    A,(HL)          ;NOTE TO PLAY
E070 32A3E0               LD    (NOTE),A
E073 23                   INC   HL
E074 7E                   LD    A,(HL)          ;SECOND PART OF NOTE
E075 32A4E0               LD    (NOTE+1),A
E078 2193E0               LD    HL,STRING       ;HL=PLAY STRING
E07B CDE573               CALL  73E5H           ;BASIC PLAY ROUTINE
E07E 2615                 LD    H,21            ;POSITION CURSOR TO
E080 2E0C                 LD    L,12            ;RIGHT OF NOTE:-
E082 CDC600               CALL  POSIT
E085 3AA3E0               LD    A,(NOTE)        ;PRINT CURRENT
E088 CDA200               CALL  CHPUT           ;NOTE, AND
E08B 3AA4E0               LD    A,(NOTE+1)      ;PRINT + CHARACTER
```

```
 75 E08E CDA200                   CALL CHPUT          ;OR SPACE
 76 E091 18A1                     JR   INPUT          ;GET NEXT KEY
 77                      ;
 78 E093 22              STRING:  DB   22H            ;START WITH QUOTES
 79 E094 4F34            OCTVE:   DB   "O4"           ;OCTAVE 4
 80 E096 543630          TEMPO:   DB   "T60"          ;TEMPO 60
 81 E099 4C38            DURAT:   DB   "L8"           ;DURATION 8
 82 E09B 5330            ENVPAT:  DB   "S0"           ;ENV WAVEFORM S0
 83 E09D 4D313030        ENVPER:  DB   "M10000"       ;PERIOD M10000
 83 E0A1 3030
 84                      NOTE:    DS   2              ;NOTE STORAGE
 85 E0A5 22                       DB   22H            ;PLAY END QUOTES
 86 E0A6 00                       DB   0              ;END STRING WITH 0
 87                      ;
 88 E0A7 444320          TABLE:   DB   "D","C "
 89 E0AA 524323                   DB   "R","C+"
 90 E0AD 464420                   DB   "F","D "
 91 E0B0 544423                   DB   "T","D+"
 92 E0B3 474520                   DB   "G","E "
 93 E0B6 484620                   DB   "H","F "
 94 E0B9 554623                   DB   "U","F+"
 95 E0BC 4A4720                   DB   "J","G "
 96 E0BF 494723                   DB   "I","G+"
 97 E0C2 4B4120                   DB   "K","A "
 98 E0C5 4F4123                   DB   "O","A+"
 99 E0C8 4C4220                   DB   "L","B "
100 E0CB 0F                       DB   0FH              ;END OF TABLE MARKER
101                      ;
102 E0CC 43555252        MESG1:   DB   "CURRENT NOTE "
102 E0D0 454E5420
102 E0D4 4E4F5445
102 E0D8 20
103 E0D9 53544154                 DB   "STATUS",0
103 E0DD 555300
104 E0E0 4F435441        MESG2:   DB   "OCTAVE:-",0
104 E0E4 56453A2D
104 E0E8 00
```

```
105 E0E9 4E4F5445 MESG3:      DB    "NOTE:-",0
105 E0ED 3A2D00
106                           END
```

Where practical the names, or labels, assigned to the ROM routines are as used in the MSX specification and should be compatible with other publications on MSX. They have a maximum length of 6 characters and are usually an abbreviation of the function - CHGET is assigned to the routine which gets a character from the keyboard, CHaracter GET.

Analysis

Line 13-CALL ERAFNK (00CCH) turns off the function key display. The sister routine is CALL DSPFNK (00CFH) which turns it back on.

Line 15-CLIKSW (F3DBH) is the switch for the key press click. Here we zeroed A with XOR A and loaded zero into F3DB which turns it off, any other value switches it back on.

Line 16-CALL CLS (00C3H) clears the screen but only if register A has been cleared by XOR A. 00C3H contains a jump to the actual CLS routine at 0848H, if you wish to clear the screen but aren't sure of the contents of A, a CALL 0849H will achieve the same goal by skipping the test on the flag. Used in the previous chapter.

Line 19-CALL POSIT (00C6H) positions the cursor depending on the value of HL. In lines 17/18 the column was entered into H and the line into L.

Line 21-CALL PTMESG (6678H) as used previously, prints a message with the start address in HL and must terminate with a zero. MESG1 is shown in line 102.

Lines 22-31 The same as above. To aid readability H and L are loaded on separate lines and decimal values have been used, not hex.

Lines 27/28 could have been entered using only one line by converting the column and line values to hex (14 and 12 become 0E and 0C) so entering:- LD HL,0E0CH would make the program shorter.

Line 33-CALL CHSNS (009CH) checks the keyboard buffer, where a pressed key is stored, and returns with Z flag reset if there was. It does not return with the character. If there was no key line 34 is a relative jump back to line 33 to do it again. The program will not pass these 2 lines until a key is entered.

Line 35-CALL CHGET (009FH) waits until a key is entered and returns with the ASCII value in register A. In fact we could have dispenced with lines 33/34 as this routine waits for a key but it also displays the cursor if it needs to do any waiting, and in my opinion spoilt the display, therefore using the CHSNS routine first means that the program does not reach here until a key is in the buffer and this routine picks up the key and does not need to wait thereby the cursor is not displayed.

Line 36-checks if it was the '£' key and if so line 37 quits the program. This line jumps back to ZEN but if one saved this as a binary file and ran it without ZEN this instruction would be altered to RET Z.

Lines 38/41- check for input of keys 1 to 8 to change the octave. Similar to the last chapter except that if the key is higher than an 8 the program jumps to the SAMOCT label in line 49 to check on a note to be played.

Line 42- is reached if the key was between 1 and 8 and loads the value into OCTVE+1, where the octave is stored.

Lines 43/45 position the cursor next to the octave message on screen with a call to POSIT.

Line 47-CALL CHPUT (00A2H) outputs the character in the A register, which was loaded in line 46, at the current cursor position already specified in lines 43/45 and line 48 jumps back to INPUT for the next key. One could alter line 47 from CALL CHPUT to RST 18H which outputs register A to the current device, be it printer, screen or whatever.

Line 49-CALL 0090H (GICINI) initialises the Programmable Sound Generator (PSG) and has been used to eliminate a queue of notes being stored and so not continuing to play for minutes after the key was released. Try deleting this line for different results.

Line 51-LD HL,TABLE loads HL with the start address of the table of notes in line 88. The key entered has been stored in register B in line 50 and the first entry of the table is loaded into A in line 52. Line 53 compares A with 0FH as this defines the end of the table and if the key was not found it is assumed that an alien key, such as Z or X, was pressed and therefore nothing should be played and a jump back to INPUT is made for another key.

Line 55 increments HL, moves it up a byte, and line 56 compares the key pressed (in B) with the table (in A) and if the two match a relative jump is made to FOUND to play the note (first time round it would be comparing with the first key in the table, key D). If the key did not match then HL must be moved up past the following bytes and must now point to the key 'R' in line 89, remembering it has been incremented once, so lines 58/59 incement HL twice more and line 60 jumps back to compare the next entry in the table. This comparing continues until HL is looking at the first byte in line 100 (0FH) as this is where lines 53/54 checked for the end of the table, which 0FH is, and aborted back to INPUT.

Line 62-FOUND is reached when the key pressed matched a key in the table and a note must be played. Remember that HL is pointing to the table and has previously (line 55) been incremented. Assume the 'D' key was entered and HL will now be pointing to the byte after 'D' in line 88 of the table. This is the letter of the note to be played and therefore gets loaded into register A (line 62). Line 63 loads this note into position in the string, labelled NOTE in line 84.

In fact 2 bytes have been reserved for this note in line 84 by entering it as 'DS 2'. The first is the letter of the note while the second byte is used to store the '+' sign. The first note in the table (line 88) is not followed by '+' and therefore a space is entered after the note - spaces are allowed in PLAY strings, although no action is taken - which will subsequently overwrite the previously stored note along with the '+' sign if it contained it. To load in the second part of the note line 64 increments HL, register A is either loaded with a space or '+' sign and line 66 stores it into NOTE+1 which is the second byte of the storage.

Line 68-CALL 73E5H calls the Basic routine for PLAY which requires the start of the string to be played to be in HL, which line 67 achieves.

Lines 69/71- position the cursor adjacent to the NOTE:- message on screen in order to display the note being played.

Lines 72/75- load the note into A and print it with CALL CHPUT as used for the octave print in lines 43/47. As the note is always 2 characters long register A is subsequently loaded with the second byte of the note storage and similarly gets displayed in line 75. The cursor does not require positioning for the second byte as it will have been automatically moved along one screen position after the previous CHPUT in line 73. This is then followed by JP INPUT to get the next key.

Line 78- STRING is where the whole of the PLAY string is stored. Line 78 contains the ASCII value for '"' which must open and close a play string. As we are accessing a Basic function it must appear syntactically correct else an error will be instituted and our machine code program will crash back into Basic, not a pleasant thought. Line 79 stores the octave and commences with the character O, not zero, and is followed by the starting value 4. This second byte obviously gets altered if one presses keys 1 to 8 whilst the program is running and so changes the octave.

Lines 80/83- set the remainder of the string for Tempo (T60), Duration (L8), Envelope waveform (S0 zero this time, not O) and Envelope Period (M10000). These values remain constant, the only alterations via the keyboard are to the note and octave, although one can change them and re-assemble the program which takes seconds, for different results. One could also alter the program to accept the cursor keys for instance to alter tempo, duration or waveform.

Line 84- contains the note storage which is blank when the program first runs. Line 85 is the ASCII value for the closing '"' sign and line 86 contains a byte with a zero value which must be entered to signify the end of a string, just like the print strings.

Lines 88/100 contain the table of keys followed by there respective notes and the line 100 contains the end of table marker 0FH.

Lines 102/105- are the print strings which one should be familiar with by now, taking note of the trailing zero byte after each.

To save as a binary file use the same procedure as in the last chapter. Alter line 37 to RET Z and note the last byte of the program when re-assembling to video and enter it when prompted as the STOP address.

This program was written to run on screen 0 but is quite possible to run on screen 1 except the display will be slightly moved to the right, it will not upset the POSIT routine which positions the cursor. One could enter a line at the start of the code to initialise the screen:-

CALL 006FH
(INIT32) will initialise to screen 1.

CALL 006CH
(INITXT) initialises screen 0.

HOOKS

The MSX allocates memory locations FD9A to FFC9 to what is known as a Hook area. There are 112 hooks each containing 5 bytes. Several routines within ROM make a call to these hooks to find if they contain additional instructions for tasks that should be performed. Normally they all contain the value C9 hex, which is the code for the RETurn instruction. Quite simply the routine in ROM calls the hook, finds it must return and do nothing, and carries on from where it left off. As sophisticated software becomes available for MSX these hooks will be used to hook up to disc drives and other peripheral devices in order to expand the system without the need to change the ROM. In order to write to a hook one must obviously know from which ROM routine it is called, so indiscriminant use could cause all sorts of problems, the rule should be, if you aren't sure leave it.

The following program writes instructions into one such hook, at FD9F, labelled HTIMI. It is called from the timer interrupt handler routine, which means it is accessed 50 times a second whatever task the MSX is performing, excluding reading or writing to tape. This obviously lends itself to be used as a built in timer as one knows how many times a second it will be encountered.

It has been used in the next program to slow down the movement of sprites, as without the delay they would move too quickly for the eye to see. Yes we could have written a machine code routine to create a delay, but that would not have demonstrated the use of a hook.

SPRITES

The Video Display Processor (VDP) used in the MSX is extremely powerful and at first may appear rather complex to the average user, but then again so did Basic once upon a time. Your awareness of its capabilities will probably depend on the amount of information given in the manual supplied for your particular machine. In order to get the best from the VDP in the MSX one should at least be conversant with the Basic VDP system variable commands and how to access the various registers. This, unfortunately, cannot be described in one chapter and is beyond the scope of this introduction to machine code. For a fuller knowledge of its workings one would be well advised to obtain a book specifically written on the VDP of the MSX, one such book is titled 'Behind the Screens of the MSX' by Mike Shaw and should answer most if not all of ones questions.

The main task of this next program is to set up 2 sprite patterns and move one across the screen until it collides with the other whereupon it will move up to the top of the screen. This program is only a demonstration of how to get sprites moving and detecting collisions, but with the machine coding practise you should have now, it could prove a good core program to fire-up the grey matter in order to get a complete display moving. One could try testing for the cursor keys being pressed and move the sprites accordingly, altering the shape and colours of the sprites etc.

The explanations follow the assembled listing.

```
                          ORG  0E000H
                          LOAD 0E000H
              CHSNS:      EQU  009CH
              HTIMI:      EQU  0FD9FH
              ERAFNK:     EQU  00CCH
              WRTVDP:     EQU  0047H
              RDVRM:      EQU  004AH
              WRTVRM:     EQU  004DH
              INIT32:     EQU  006FH
              RG1SAV:     EQU  0F3E0H
              STATFL:     EQU  0F3E7H
              ATTR1:      EQU  1B00H
              ATTR2:      EQU  1B04H
              ;
              ;WRITE CODE TO HOOK (HTIMI)
E000 219EE0               LD   HL,CODE
E003 119FFD               LD   DE,HTIMI
E006 010300               LD   BC,3
E009 EDB0                 LDIR
              ;
E00B CDCC00               CALL ERAFNK           ;TURN OFF FUNC KEYS
E00E CD6F00               CALL INIT32           ;SCREEN 1
              ;
              ;ALTER SPRITE PATTERN
              ;GENERATOR BASE ADDRESS TO 0000
E011 AF                   XOR  A
E012 47                   LD   B,A
E013 0E06                 LD   C,6
E015 CD4700               CALL WRTVDP
              ;
              ;ALTER BIT 0 OF VDP REG 1
              ;TO 1.  TO INCREASE MAGNITUDE
E018 3AE0F3               LD   A,(RG1SAV)
E01B F601                 OR   1
E01D 47                   LD   B,A
E01E 0E01                 LD   C,1
E020 CD4700               CALL WRTVDP
```

```
                 ;
                 ;SET UP SPRITE 1
E023 3E8C                 LD   A,140        ;VERTICAL POS
E025 21001B               LD   HL,ATTR1
E028 CD4D00               CALL WRTVRM
E02B 3EC8                 LD   A,200        ;HORIZ POS
E02D 21011B               LD   HL,ATTR1+1
E030 CD4D00               CALL WRTVRM
E033 3E41                 LD   A,65         ;CHARACTER 65=A
E035 21021B               LD   HL,ATTR1+2
E038 CD4D00               CALL WRTVRM
E03B 3E01                 LD   A,1          ;COLOUR BLACK
E03D 21031B               LD   HL,ATTR1+3
E040 CD4D00               CALL WRTVRM
                 ;
                 ;SET UP SPRITE 2
E043 3E8C                 LD   A,140        ;VERTICAL POS
E045 21041B               LD   HL,ATTR2
E048 CD4D00               CALL WRTVRM
E04B 3E1E                 LD   A,30         ;HORIZ POS
E04D 21051B               LD   HL,ATTR2+1
E050 CD4D00               CALL WRTVRM
E053 3E42                 LD   A,66         ;CHARACTER 66=B
E055 21061B               LD   HL,ATTR2+2
E058 CD4D00               CALL WRTVRM
E05B 3E0F                 LD   A,15         ;COLOUR WHITE
E05D 21071B               LD   HL,ATTR2+3
E060 CD4D00               CALL WRTVRM
                 ;
E063 AF                   XOR  A
E064 3298E0               LD   (COUNT),A    ;ZERO COUNTER
                 ;DELAY COUNTER CHECK
E067 CD9C00      CKMOVE:  CALL CHSNS
E06A 2024                 JR   NZ,QUIT
E06C 3A98E0               LD   A,(COUNT)
E06F FE01                 CP   1
E071 38F4                 JR   C,CKMOVE     ;IF LESS DONT MOVE
```

```
                   ;
                   
                   ;MOVE SPRITE 2
E073 21051B                 LD     HL,ATTR2+1          ;VERT POS
E076 CD4A00                 CALL   RDVRM               ;PUT INTO A
E079 3C                     INC    A                   ;MOVE 1 PIXEL RGHT
E07A CD4D00                 CALL   WRTVRM              ;NEW POS OF SPRT 2
E07D AF                     XOR    A
E07E 3298E0                 LD     (COUNT),A           ;ZERO COUNTER
                   ;CHECK FOR COLLISION
E081 3AE7F3                 LD     A,(STATFL)
E084 CB6F                   BIT    5,A                 ;TEST COLLISION BIT
E086 28DF                   JR     Z,CKMOVE            ;IF ZERO KEEP MOVING
                   ;
                   ;COLLISION OCCURED
E088 21041B                 LD     HL,ATTR2            ;HORIZ POS
E08B 3E28                   LD     A,40
E08D CD4D00                 CALL   WRTVRM              ;MOVE IT UP
                   ;
                   ;PROG END, SO REPLACE RET
                   ;INTO HOOK (HTIMI)
E090 3EC9          QUIT:    LD     A,0C9H
E092 329FFD                 LD     (HTIMI),A
E095 C303A0                 JP     0A003H
                   ;
E098 00            COUNT:   DB     0
                   ;
                   ;INCREMENT COUNT 50 TIMES A SEC
E099 2198E0        INCCNT:  LD     HL,COUNT
E09C 34                     INC    (HL)
E09D C9                     RET
                   ;
E09E C399E0        CODE:    JP     INCCNT
                   ;
                            END
```

Lines 16/19 load code into the hook labelled HTIMI at FD9F. The LDIR instruction has been used which, as you should know by now, loads code from the address pointed to by HL into that pointed to by DE. The amount to transfer is held in BC, in this case 3 bytes which are shown in line 106 which tell the hook to jump to INCCNT which increments the counter. This has the effect of slowing the movement and can be speeded up or slowed as will be seen.

Line 21 is the erase function key routine and this is followed by INIT32 which switches the display to screen 1, as you cannot have sprites on screen 0.

Lines 26/29 Write to the VDP register whose number (0 to 7) must be in register C and the data to be loaded into the VDP held in register B. Here we are loading VDP register 6(C) with 0(B). This in effect is altering the base address of sprite pattern table to that of the character generator base address. So we now have the full ASCII character set stored as sprites.

Line 33 Loads the value of RG1SAV (F3E0) which stores the current value of register 1 of the VDP. The only bit of VDP 1 we are interested in is bit 0 which controls the magnification for sprites. Zero is the normal size whilst altering it to 1 magnifies the sprites, so in line 34 we OR 1 which will not effect the remainder of the bits but will turn bit 0 to a 1, putting it in magnify mode. And once again we must load the contents of A into register B, select the VDP register in C and call WRTVDP, which will write to the VDP register 1.

Now if that appeared slightly beyond you don't panic, as practise makes perfect, just enter the code and alter it later, it can only get easier and you will pick it up.

Lines 40/51 set up sprite 1. The sprite attribute table in screen 1 commences at address 1B00H and contains 4 bytes for each sprite. Therefore the attributes for sprite 2 will commence at 1B04H. These were given equates in lines 12/13, ATTR1 and ATTR2. The first byte holds the vertical pixel position of the sprite and line 40 loads register A with 140. The screen pixels are from 0 (top) to 191 (bottom). Line 41 loads HL with the address of ATTR1 (1B00H) and line 42 loads register A into VRAM at 1B00H by CALL WRTVRM.

The process continues with loading the horizontal position into register A and storing it in the second byte of the attribute for sprite 1 at 1B01H by loading HL with ATTR1+1.

> NOTE Line 47 could be entered as INC HL as when the program returns from the WRTVRM routine register HL is not changed in any way, therefore it still points to the previous location, and one could simply increment it. But it was shown in this form for clarity and not program elegance.

The third byte of the attribute holds the sprite character number. But we have not defined our own sprite we have shifted the sprite pattern table so it is looking at the ASCII characters, purely for convenience. Therefore the ASCII for the letter A is 65 decimal, and line 46 loads register A with 65 as the letter A is to be our sprite 1. This is then loaded into the third byte in the attributes, ATTR1+2. Note that the although the decimal value was used for clarity, the assembler converted it to hex, as one will see in the left column.

Lastly the colour must be defined and loaded into the fourth attribute byte ATTR1+3, and line 49 defines the colour number 1, which is black and this concludes the set up for sprite 1, having its co-ordinates, character and colour stored in 1B00 to 1B03H.

The procedure is duplicated for setting up sprite 2 except the information must be stored at 1B04 to 1B07H, which was labelled as ATTR2 in line 13. The vertical position is the same as sprite 1, the only differences being the horizontal (30) in line 57, the character which this time is the letter B (ASCII 66 decimal) and the colour which is 15 for white.

Once the program is running these co-ordinates can be altered for different effects, providing one remembers to assemble each time.

Lines 67/68 zero register A and loads this into COUNT.

Lines 70/71 is the CHSNS routine which checks for a key being pressed and jumps to the QUIT routine, and has been included so that one can stop the program before it finishes to modify it just in case one has slowed it down too much.

Line 72 loads the value stored in COUNT into register A and if it has not reached 1 line 74 jumps back to check it again. This loop continues until register A is equal or greater than the value in line 73. Remember the hook at FD9F is incrementing COUNT 50 times a second, therefore CP 1 will only cause the loop to continue for 1/50th of a second before carrying on with the program and moving the sprite by 1 pixel. If line 73 was altered to CP 50 the sprite would move by 1 pixel only once a second, very slow. Without the delay at hook HTIMI the sprite would move so fast it would simply appear at the finishing point.

Lines 77/82 move sprite 2 and zero the counter for the next delay before moving again. The only attribute we are changing is the horizontal position, ATTR2+1, therefore this must be loaded into HL and a call to read VRAM (RDVRM) will return the value in the A register (its current horizontal position). To move the sprite we INCrement A and write it back to the VRAM address still pointed to by HL by WRTVRM.

Lines 84/86 check for a collision of the 2 sprites. STATFL (F3E7H) holds the status register of the VDP and bit 5 is set to 1 if a collision, 2 sprites overlap by at least 1 pixel, has occured. Therefore line 85 tests bit 5 of the status flag and line 86 jumps if it was zero back to CKMOVE again. If a collision had taken place bit 5 would be 1 therefore the zero flag would not be set and the test would fail and the program would fall through to the next line.

Line 90 is only reached after a collision and will shoot sprite 2 to near the top of the screen by loading HL by the vertical attribute ATTR2, leaving the horizontal ATTR2+1 intact, loading A with 40 and calling WRTVRM again to re-position it.

Line 95 QUIT is reached after a collision or if a key has been pressed while the program was running. It simply replaces the contents of hook HTIMI with its original byte the code for RET (C9H), in case one is going to run another program as you will not want the hook accessing COUNT 50 times a second, and jumps back to ZEN.

Line 99 is the storage byte for the counter.

Lines 102/104 are accessed only from the hook HTIMI, and simply add 1 to count each time the interrupt occurs.

Once running experiment with altering the positions of the sprites and time delays and try adding a routine for moving each sprite on the press of certain keys.

LOADER PROGRAM

When loading a machine code program one may have seen a different screen message displayed than the usual one or, as is becoming more popular, the complete display could alter to a graphics title while the program appears to be still loading.

The answer can lie in the fact that two programs have been loaded, the second automatically. The first short program contains the titles and a jump to the loading routine for the second larger program. When the first program has loaded it executes immediately so printing the titles on screen and enters a loading routine for the second. Execution is so fast that the tape stops for a minimal time and starts again almost without being noticed. Only one 'Found:program name' message appears on screen whilst the first program is loading.

The loader program begins with the screen title message, in the example it will display 'NOW LOADING MAIN PROGRAM', but this can be expanded upon as will be explained. It is advised to only add the loader jump section after fully debugging and testing the graphics titles. Although the ORG is set at 9000H, which is ideal for testing, before saving the object file it could be altered to another memory location, this also means that the second program could be set to the same ORG before saving and the loader program will be overwritten and dissappear from memory as the main program loads in. If one has recorded an ASCII file of one of the earlier programs it will be simple to test this loader. First complete the entries on the next page, making sure it runs correctly, then add in the loader section carefully and save as a binary file by the 'WB' command. Verify the tape and do not rewind as the main program will be recorded starting from where the first finished, on the next section of the tape. Kill the loader program from memory and load in a program from the earlier chapter Assemble and save the second program with the 'WB' command onto the tape, and one should possess two programs on the tape, the second will automatically load and run.

```
 1                          ORG   9000H
 2                          LOAD  9000H
 3                INIT32:   EQU   006FH
 4                ERAFNK:   EQU   00CCH
 5                FORCLR:   EQU   0F3E9H
 6                BAKCLR:   EQU   0F3EAH
 7                BDRCLR:   EQU   0F3EBH
 8                CHGCLR:   EQU   0062H
 9                T32NAM:   EQU   1800H
10                LDIRVM:   EQU   005CH
11                FILVRM:   EQU   0056H
12                ;
13 9000 CDCC00              CALL  ERAFNK
14 9003 3E09                LD    A,9
15 9005 32EAF3              LD    (BAKCLR),A
16 9008 32EBF3              LD    (BDRCLR),A
17 900B 3E01                LD    A,1
18 900D 32E9F3              LD    (FORCLR),A
19 9010 CD6F00              CALL  INIT32
20 9013 210018              LD    HL,T32NAM
21 9016 3ED1                LD    A,0D1H
22 9018 010003              LD    BC,768
23 901B CD5600              CALL  FILVRM
24 901E 212D90              LD    HL,DISPL
25 9021 116319              LD    DE,T32NAM+355
26 9024 011A00              LD    BC,26
27 9027 CD5C00              CALL  LDIRVM
28 902A C300A0              JP    0A000H
29                ;
30 902D 204E6F77  DISPL:    DB    " Now Loading"
30 9031 204C6F61
30 9035 64696E67
31 9039 204D6169            DB    " Main Program "
31 903D 6E205072
31 9041 6F677261
31 9045 6D20
32                          END
```

Analysis

Line 13 which has been used before erases the function keys.

Lines 14/18 set up the colours. Register A is first loaded with the code for the colour light red, and this is loaded into the storage bytes for background (BAKCLR) and border (BDRCLR) colours at F3EA and F3BE respectively. The character colours (FORCLR) is loaded with register A, this time 1 for black, at F3E9.

Line 19 initialises screen 1 which will clear the screen and change it to the new colours.

> NOTE If one wished to simply alter the existing colours whilst remaining in the same screen mode and maintaining what was currently displayed on screen, a call to 0062H (CHGCLR) would suffice after setting up the colours.

Line 20 loads HL with the start of the Name table for screen 1 (1800H), the top left position.

Line 21 loads register A with the code for the character which will cover the screen whilst line 22 loads our byte counter (register BC) with the number of positions we will write to. As screen 1 contains 24 lines of 32 characters BC is loaded with 768 decimal, it could of course been converted to hex to read LD BC,300H.

Line 23 calls a routine (FILVRM) at 0056H which writes the data in register A to VRAM, which has its source address in HL and the number of bytes in BC, all of which has been done. If one only wanted to write to the centre 4 lines of the screen then HL would instead have required loading with the start of the name table plus the offset. The tenth screen line starts at 288 positions (32x9 as first line is 0) higher than the start of the screen, therefore progam line 20 could have read:- LD HL,T32NAM+288. But obviously the byte counter would require reducing too, for 4 lines of print it should be loaded with 128 decimal (32x4).

Lines 24/26 load the start of our actual screen message ' Now Loading Main Program ' (labelled DISPL) into HL, load the destination address of its position into DE, and load the length of the message into BC (26 bytes including the leading and trailing spaces). As the message should commence 3 positions in from the left on line 12 the actual screen position is calculated thus:- 32x11+3, the top (first) line is 0 therefore to calculate the 12th line, the line width of 32 is multiplied by 11.

Line 27 calls LDIRVM at 005CH which was used in chapter one to load VRAM directly with the contents of an area of RAM.

Line 28 jumps back to the ZEN mainloop so the titles can be altered and run until it reaches your satisfaction. This jump address will be altered when we add the loading section.

After entering G9000H the display will switch into screen 1 mode and cover the screen with the ASCII character of 'D1' and the centre line will print our message. Get it running first and then make your alterations to colour, length of message etc., remembering to assemble after each change in the program otherwise the changes will not be entered into memory. After each test the 'ZEN' message will appear at the top of the screen and in order to clear the screen and return to screen 0, which is the usual screen in ZEN, enter the 'RETURN' key on its own.

The Loader

To append the loader routine make the line which contains the 'END' message the current line, if one has not altered the program it should be line 32, and enter 'E' and 'Return' and add the lines listed on the following page.

```
                TO
DISPLAYED     ENTER

   32         LOADER:LD A,0FEH
   33         LD (0F41CH),A
   34         LD HL,STRING
   35         JP 6EC6H
   36         STRING:DB 22H,"CAS:"
   37         DB 22H,2CH,"R",0
   38
ZEN
```

Line 28 which is the jump to Zen requires altering to:-
JR LOADER

this will then make the program jump to the loader routine after the titles have been displayed. Make sure the last line is still 'END' and assemble the program to (V)ideo and make a note of the last byte of the program. If one has not altered the program it should be 905AH but obviously your program could be longer. The final section of the program would have assembled like this:-

```
32 9047 3EFE      LOADER:     LD    A,0FEH
33 9049 321CF4                LD    (0F41CH),A
34 904C 215290                LD    HL,STRING
35 904F C3C66E                JP    6EC6H
36 9052 22434153 STRING:      DB    22H,"CAS:"
36 9056 3A
37 9057 222C5200              DB    22H,2CH,"R",0
38                            END
```

Quite simply we have loaded the value FE Hex into the contents of F41CH which stops a second 'Found:' message being printed when the second program is loading, which would spoil the display. Line 34 loads HL with the start of a string which are the characters that could normally follow a BLOAD command i.e. "CAS:",R. We then jump to the BLOAD routine in ROM at 6EC6H.

To save the program as a binary file enter:-

```
WB
START 9000H
STOP 905AH
LOAD 9000H
EXEC 9000H
```

and give the loader the name of the main program that will follow it on tape. Afterwhich rewind the tape and verify by entering 'VB' and 'RETURN', and if all is well move the tape on slightly to allow a gap between the end of the loader and the start of the second program.

Also, as a safety measure, save the ASCII file of this loader program on a separate tape for other programs and also in case it does not operate correctly when the second program is appended.

Now kill the file by entering 'K' and 'RETURN', similar to 'NEW' in Basic, and load in or enter from the keyboard the main program. If one saved the ASCII file of one of the earlier example programs in chapter 3 this will be ideal for testing. Alter the ORG and LOAD of the second program to 9000H, the same address as the loader used, and assemble to (V)ideo in order to note the last byte of the program and save as previously shown onto the same tape that contains the loader program, but press 'RETURN' when prompted for a name.

To test switch off for a few seconds, and when turned back on enter:-

```
BLOAD"CAS:",R
```

and the first program found on the tape will load and run, and displaying the titles, and in doing so should automatically load and run the second, main program.

Argument Transfer using USR

The USR function was used in chapter 1 with what is termed as a dummy argument within the brackets i.e. USR(0). It is possible to pass up to a machine code routine an integer, string, single or double precision variable. A=USR(&H1234) would store in a storage area in RAM the hex value 1234 and call up the machine code routine. Misuse of this function could cause a program to crash.

The type of argument passed to the routine is always stored at address F663, so the routine can check what type has been passed up, and would contain:-

2 for an integer
3 a string
4 a single precision real type variable
8 a double precision real type variable.

Take an example:-
First one needs to DEFine the USeR address as was shown in chapter 1 (i.e. 10 DEF USR2=&HE000). Our machine code routine is called from Basic to execute a certain task, it could be to move a block of memory, and requires the destination address (for example A020H) to be passed from the Basic program and stored in order to load into register DE. The Basic line could be:-

```
   100 A=USR2(&HA020)
```

This would then call the machine code routine at address E000H and carry out any tasks until it RETurned to the Basic program. The integer (or address) A020H is always stored at address F7F8 and F7F9 hex. Therefore to load the integer into register DE all the machine code program needs to do is:-

```
   LD DE,(0F7F8H)
```

and DE will contain A020H. If, while still in the machine code routine, one wished to alter the integer and pass it back down to the Basic program when it RETurns simply:-

```
   LD (F7F8H),DE
```

and the new value will be placed into variable A on returning to the Basic program.

Strings operate in a slightly different manner in that F7F8 and F7F9 hex will contain NOT the string, but an address where the string descriptor is located. This string descriptor contains 3 bytes. The first signifies the string's length and the second and third the address where it is stored, and could be used thus:-

```
100 B$="MSX"
110 A$=USR2(B$)
```

This time, as it is a string, F663H contains the value 3. Addresses F7F8 and F7F9 hex contain the address of the descriptor, for example 802D, and at address 802D will be the length of the string, 3, while 802E and 802F will contain the actual location where the string is stored in reverse order naturally. Did I mention it was complicated?

Single precision values, type 4, are stored at F7F6 to F7F9 hex and Double precision, type 8, at F7F6 to F7FD hex.

<u>WARNING</u> One cannot access the storage addresses (F7F6H onwards) after the routine has returned to the Basic program (they cannot be PEEKed), as they will not be stored. Only the variable which was used in the USR line (in the last example A$) will contain the data.

For an example of how an integer would be passed into a machine code routine load ZEN in the normal way, and enter this short program:-

```
1 ORG 0E000H
2 LOAD 0E000H
3 LOOP:EQU 0A003H
4 LD DE,(0F7F8H)
5 JP LOOP
6 END
7 .
```

and assemble.

If all is well enter 'B' to return to Basic and enter this Basic program:-

```
   10 DEFUSR2=&HA000
   20 A=USR2(&H1234)
   30 PRINT HEX$(A)
```

and RUN.

The program will enter the ZEN mainloop so enter:-

GE000H and 'RETURN' twice. The short program will execute immediately and return with the ZEN prompt. Now enter 'X' to examine the registers and one will find DE contains 1234, which proves that an integer can be passed up to a routine.

To discover how it passes back an integer enter 'MF7F8H' and the contents will be displayed as 34. Enter 35H and 'RETURN' followed by the full stop '.' and 'RETURN' which alters the contents of memory, as you will obviously know by now. Enter 'B' to return to Basic and line 30 will then execute and print the hex value of variable A, which one will see has changed to 1235H.

The 'Q' command in ZEN is useful for displaying the contents of memory and could be used here to discover how the single and double precision variables are stored, as once control is passed back to Basic the storage area is corrupted.

This section lists some of the useful routines found in ROM which can be used by your own machine code program, some have been used already in previous chapters, and some may be too advanced for ones immediate use.

The start of ROM contains a table of jumps to the various routines, some of the more straightforward have been listed to assist if one wishes to disassemble certain sections of memory, but normally a call to the appropriate location in the table is all that will be required, providing one knows which registers should be loaded with the relevant data before the call is made. Each routine carries an abbreviated name, or label up to six characters in length, as used in the MSX specification.

Addr	Jump	Name	Function
0000	02D7	CHKRAM	The first byte disables the interrupts and a jump to 02D7 checks RAM and sets slots for the command area, this is followed by the address of the character generator table and also the ports for VDP read and write.
0008	2683	SYNCHR	Called by RST 8. Checks the current character pointed to by HL is the required one and falls into CHRGTR if true, else gives Syntax error. Character to be checked must be the next byte after this RST. Carry flag set if it is a number, Z flag set if end of statement. Modifies AF,HL
0010	2686	CHRGTR	Gets next character, or basic token, in basic text. Entry HL. Exits with HL pointing to next char, A contains char, carry flag set if number, Z flag set if end of statement. Modifies AF,HL

Addr	Jump	Name	Function
0018	1B45	OUTDO	Outputs contents of reg A to current device, VDU, printer etc. Called by RST 18H
0020	146A	DCOMPR	Compares HL with DE and sets Zero flag if matched. Modifies AF
0038	0C3C	KEYINT	Performs hardware interrupt procedure 50 times a second. Modifies nothing.
0041	0577	DISSCR	Disables screen, blanks out screen. Modifies AF, BC
0044	0570	ENASCR	Enables screen, switches it back and restores characters which were previously displayed. Modifies AF, BC
0047	057F	WRTVDP	Writes data to VDP register. Enter with reg in C, data in B Modifies AF, BC
004A	07D7	RDVRM	Reads VRAM pointed to by HL, returns data in reg A. Modifies AF
004D	07CD	WRTVRM	Writes to VRAM pointed by HL, data in reg A. Modifies AF
0050	07EC	SETRD	Sets up VDP for Read. HL on entry. Modifies AF
0053	07DF	SETWRT	Sets up VDP for Write. HL on entry. Modifies AF
0056	0815	FILVRM	Fills VRAM starting at HL with data in reg A and length in BC. Modifies AF, BC

Addr	Jump	Name	Function
0059	070F	LDIRMV	Moves block of VRAM to memory. VRAM source in HL to destination in DE and length in BC. Modifies all.
005C	0744	LDIRVM	Moves block of memory to VRAM from source in HL to VRAM destination in DE and length in BC. Modifies all.
005F	084F	CHGMOD	Initialises screen according to value of reg A, 0 to 3. Stores A at FCAFH. Modifies all.
0062	07F7	CHGCLR	Changes colour of screen to colours specified in:- Foreground colour (FORCLR) at F3E9, Background (BAKCLR) at F3EA and Border (BDRCLR) at F3EB. Modifies all.
0066	1398	NMI	Performs non-maskable Interrupt procedure. Entry none. Modifies none.
0069	06A8	CLRSPR	Initialises all sprites. Patterns are set to nulls.
006C	050E	INITXT	Initialises screen to text mode, screen 0 and sets VDP. Modifies all.
006F	0538	INIT32	Initialises for screen 1, and sets VDP. Modifies all.
0072	05D2	INIGRP	Initialises to screen 2, and sets VDP. Modifies all.
0075	061F	INIMLT	Initialises to screen 3, and sets VDP. Modifies all.

Addr	Jump	Name	Function
0078	0594	SETTXT	Sets VDP for screen 0
007B	05B4	SETT32	Sets VDP for screen 1
007E	0602	SETGRP	Sets VDP for screen 2
0081	0659	SETMLT	Sets VDP for screen 3
0084	06E4	CALPAT	Returns address of sprite pattern table in HL. Entry reg A = sprite no. Modifies AF, DE, HL.
0087	06F9	CALATR	Returns address of sprite attribute table in HL. Entry sprite no. in reg A. Modifies AF, DE, HL.
008A	0704	GSPSIZ	Returns current sprite size in reg A (no. of bytes) Returns carry flag set if 16x16 sprite otherwise reset. Modifies AF.
008D	1510	GRPPRT	Prints a character on graphic screen in reg A.
0090	04BD	GICINI	Initialises PSG. Modifies all.
0093	1102	WRTPSG	Write data in reg E to PSG register number in A.
0096	110E	RDPSG	Reads data from PSG register in A, returns with data in A. Modifies AF.
0099	11C4	STRTMS	Checks and starts the background music.
009C	0D6A	CHSNS	Checks the keyboard for pressed key. Returns with Z flag set if key in buffer. Modifies AF.

Addr	Jump	Name	Function
009F	10CB	CHGET	Waits until a key is typed. Returns with ASCII of key in reg A. Modifies AF.
00A2	08BC	CHPUT	Outputs contents of reg A to screen.
00A5	085D	LPTOUT	Outputs contents of reg A to printer. Carry flag set if aborted. Modifies F.
00A8	0884	LPTSTT	Checks printer status. Returns FFhex in A and Z flag reset if printer ready, 0 in A and Z flag set if not ready. Modifies AF.
00AE	23BF	PINLIN	Stores line of input from keyboard in buffer, terminates when RETURN entered. Returns start of buffer in HL, carry flag set if STOP was entered. Modifies all.
00B7	046F	BREAKX	Checks for CTRL/STOP keys. Carry flag set if pressed. Modifies AF.
00C0	1113	BEEP	Sounds bell.
00C3	0848	CLS	Clears screen if Z flag set.
00C6	088E	POSIT	Positions the cursor. Entry H=column, L=line. Modifies AF.
00C9	0B26	FNKSB	Checks if function keys should be on, if so displays them, else does nothing. Modifies all.
00CC	0B15	ERAFNK	Turns off function key display. Modifies all.

Addr	Jump	Name	Function
00CF	0B2B	DSPFNK	Turns on function key display. Modifies all.
00D2	083B	TOTEXT	Forces screen into text mode. Modifies all.
0132	0F3D	CHGCAP	Switches CAPS light on/off, but does not affect CAP status. Entry 0 in reg A turns on, any other turns off. Modifies AF.
0156	0468	KILBUF	Clears keyboard buffer. Modifies HL.

Addresses F380H upwards are assigned to storage areas for accessing from ROM or equally from your own program in RAM. The more common of which are listed below followed by their MSX name and amount of bytes and purpose.

For example the current line length of screen 0 is held at F3AEH, and is usually set to 25H (37 dec) and can naturally be altered as this is in RAM. To check on the contents of a location in memory one could enter a Basic line:- ?PEEK(&HF3AE) or from ZEN :-QF3AEH.

Addr	Name	Size	Function
F39A	USRTAB	20	Addresses assigned to the 10 USR functions (0 to9). Until a DEF USR statement been initialised these addresses all contain 475A which loads error 5 into the error flag.
F3AE	LINL40	1	Line width in screen 0
F3AF	LINL32	1	Line width in screen 1

Addr	Name	Size	Function	
F3B0	LINLEN	1	Line length.	
F3B2	CLMLST	1	Lines on screen.	
			Screen 0	
F3B3	TXTNAM	2	Name address table start.	(0000H)
F3B5	TXTCOL	2	Colour " " "	(unused)
F3B7	TXTCGP	2	Character Generator table start	(0800H)
F3B9	TXTATR	2	Attribute Table start	(unused)
F3BB	TXTPAT	2	Sprite Pattern Generator table start	(unused)
			Screen 1	
F3BD	T32NAM	2	Name address table start.	(1800H)
F3BF	T32COL	2	Colour " " "	(2000H)
F3C1	T32CGP	2	Character Generator table start	(0000H)
F3C3	T32ATR	2	Attribute Table start	(1B00H)
F3C5	T32PAT	2	Sprite Pattern Generator table start	(3800H)
			Screen 2	
F3C7	GRPNAM	2	Name address table start.	(1800H)
F3C9	GRPCOL	2	Colour " " "	(2000H)
F3CB	GRPCGP	2	Character Generator table start	(0000H)
F3CD	GRPATR	2	Attribute Table start	(1B00H)
F3CF	GRPPAT	2	Sprite Pattern Generator table start	(3800H)

Addr	Name	Size	Function
			Screen 3
F3D1	MLTNAM	2	Name address table start. (0800H)
F3D3	MLTCOL	2	Colour " " " (unused)
F3D5	MLTCGP	2	Character Generator table start (0000H)
F3D7	MLTATR	2	Attribute Table start (1B00H)
F3D9	MLTPAT	2	Sprite Pattern Generator table start (3800H)
F3DB	CLIKSW	1	Key click switch. 0=off, any other=on
F3DC	CSRY	1	Cursor Y position (line)
F3DD	CSRX	1	Cursor X position (column)
F3DE	CNSDFG	1	Function key display switch. 0=off
			VDP Register values
F3DF to F3E6		8	Stores VDP 0 to VDP 7
F3E7	STATFL	1	Stores VDP Status register
F3E9	FORCLR	1	Foreground colour
F3EA	BAKCLR	1	Background colour
F3EB	BDRCLR	1	Border colour
F55E	BUF	256	Input Buffer
F672	MEMSIZ	2	Highest location in memory

5

Bytesearcher

This utility program is loaded from ZEN and simply appends a byte search routine which is useful when disassembling sections of memory. One can either search for a two byte address or string.

Two byte search

The keyboard routine within ZEN commences at address A742H, and let us suppose one wanted to discover where and how often the keyboard routine was referred to within the memory area which ZEN occupies. One would enter:-

YA742H

Note that the address is entered correctly, not as it would be found in memory low byte first, as the search routine adjusts for this. The command 'Y' has been used as most other letters are already utilised, although this could be altered in line 11 to a lower case (small) letter such as 's' which is not otherwise used.

One will be prompted to enter the 'START' address followed by 'H', so if ZEN was to be searched enter the first memory location of ZEN:-

A000H

Logically the next prompt is for 'END', so in this example one could enter the last address of ZEN:-

BB5CH

The final prompt is for 'OPTION' and for the screen to display the locations one would enter 'V'. The screen will then display:-

```
Occurences of A742H
between:-A000 and BB5C

A7F4 A92A
ZEN
```

String search

Strings may be searched for by entering the string within quotes:-
Y"Ok"
which will search for the 'Ok' message. To find its location within the ROM one would enter the Start address as '0000' or simply '0', and for the End address enter the top of ROM '8000H'. The screen would then display:-

```
Occurences of "Ok"
between:-0000 and 8000

3FD7
ZEN
```

which is the location in ROM where this message resides.

The two byte search could then be used to discover which areas of the ROM access the 'Ok' message by searching for 3FD7 between 0 and 8000H, and the display would reveal that it is referred to at:-
412F 53FB and 7072

Bytesearcher accesses many routines within ZEN only once calling a routine outside at 0020H which is a ROM routine to compare HL with DE, and the routines are listed in the comment field and may be checked against your ZEN reference manual.

The program can be saved as an ASCII file, where one simply enters 'W' and enters the filename, and can be loaded back in and assembled only when one requires the extra byte search facility, for disassembling.

> NOTE After entering the code, or loading from tape, it is essential to assemble the bytesearcher BEFORE modifying the 3 bytes at A251H, as this area is within the mainloop of ZEN and a jump is made to E000H to discover if the key pressed was 'Y', and if it has not been assembled the bytesearcher program will not be at E000H and ZEN could crash and the program lost.

```
                ;BYTESEARCHER
                ;AFTER ASSEMBLY ALTER
                ;ZEN BY:-   M0A251H
                ;and enter these 3 bytes
                ;0C3H   00   0E0H
                ;
                            ORG   0E000H
                            LOAD  0E000H
                ;
E000 CAA5A3     EXTRA:      JP    Z,0A3A5H          ;Orig routine
E003 FE59                   CP    "Y"               ;For Bytesearcher
E005 2803                   JR    Z,BYTSCH          ;It's what we want
                ;
                ;New commands go here
                ;
E007 C354A2                 JP    0A254H            ;Back to Zen
                ;
E00A 118AA1     BYTSCH:     LD    DE,0A18AH         ;(TBUFF+1)
E00D 21D6E0                 LD    HL,SCHSTR         ;Store Input
E010 010000                 LD    BC,0              ;String counter
                ;
                ;Transfer the string
                ;
E013 1A         TRSTR:      LD    A,(DE)
E014 77                     LD    (HL),A
E015 23                     INC   HL
E016 13                     INC   DE
E017 03                     INC   BC
E018 FE0D                   CP    0DH               ;Return?
E01A 20F7                   JR    NZ,TRSTR          ;No-keep transfering
                ;
                ;Transfer complete-Check that
                ;something is there
                ;
E01C 78                     LD    A,B
E01D 0D                     DEC   C                 ;Don't count 'CR'
E01E B1                     OR    C
```

```
E01F CAD5A8                      JP   Z,0A8D5H       ;Error, so 'HUH?'
                      ;
                      ;Now get START/STOP parameters
                      ;
E022 CDC5A8                      CALL 0A8C5H         ;'STARTSTOP'
E025 2B                          DEC  HL
E026 19                          ADD  HL,DE
E027 ED532CA1                    LD   (0A12CH),DE    ;=START
E02B 222EA1                      LD   (0A12EH),HL    ;=STOP
E02E CD39AB                      CALL 0AB39H
                      ;
                      ;Print title
                      ;
E031 21E0E0                      LD   HL,MSG1
E034 CDDCA7                      CALL 0A7DCH         ;ZEN "STR1"
E037 21D6E0                      LD   HL,SCHSTR
E03A CDDCA7                      CALL 0A7DCH
E03D 21EFE0                      LD   HL,MSG2
E040 CDC4A6                      CALL 0A6C4H
E043 CDDCA7                      CALL 0A7DCH
E046 2A2CA1                      LD   HL,(0A12CH)    ;Start of Data
E049 CD95A9                      CALL 0A995H         ;ZEN "WORDSP"
E04C 21F9E0                      LD   HL,MSG3
E04F CDDCA7                      CALL 0A7DCH
E052 2A2EA1                      LD   HL,(0A12EH)    ;End of Data
E055 CD95A9                      CALL 0A995H
E058 CDC4A6                      CALL 0A6C4H         ;ZEN "CRLF"
E05B CDC4A6                      CALL 0A6C4H         ;another CRLF
                      ;
                      ;Check for H or Quote at end
                      ;
E05E 21D6E0                      LD   HL,SCHSTR
E061 0600                        LD   B,0            ;Counter for convert
E063 7E               FENDS:     LD   A,(HL)         ;Find string end
E064 FE0D                        CP   0DH
E066 2804                        JR   Z,COMP
E068 23                          INC  HL
```

```
E069 04                    INC   B
E06A 18F7                  JR    FENDS
               ;
E06C 2B        COMP:       DEC   HL               ;Back-up to 'H' or "
E06D 7E                    LD    A,(HL)
E06E FE22                  CP    22H              ;It's a quote string
E070 2816                  JR    Z,SEEK
E072 FE48                  CP    "H"
E074 C2D5A8                JP    NZ,0A8D5H        ;Not hex
E077 23                    INC   HL               ;Back to end
E078 11D6E0                LD    DE,SCHSTR
E07B CDDAA8                CALL  0A8DAH           ;ZEN convert routine
E07E 22D7E0                LD    (SCHSTR+1),HL
E081 3E0D                  LD    A,0DH
E083 32D9E0                LD    (SCHSTR+3),A
E086 1802                  JR    FIND
               ;
E088 360D      SEEK:       LD    (HL),0DH
E08A 2A2CA1    FIND:       LD    HL,(0A12CH)
E08D ED5B2EA1              LD    DE,(0A12EH)
E091 2B                    DEC   HL
E092 D5                    PUSH  DE
E093 E5                    PUSH  HL
               ;
E094 AF                    XOR   A
               ;
E095 32D5E0                LD    (COUNT),A
E098 E1        FINDIT:     POP   HL
E099 D1                    POP   DE
E09A 23                    INC   HL
E09B D5                    PUSH  DE
E09C E5                    PUSH  HL
E09D CD2000                CALL  0020H            ;ROM Compare HL'DE
E0A0 2008                  JR    NZ,LOOK
E0A2 E1                    POP   HL
E0A3 D1                    POP   DE
E0A4 CDC4A6                CALL  0A6C4H           ;Finished so CRLF
```

```
E0A7 C300A0                JP    0A000H          ;Back to ZEN
                  ;
E0AA 11D7E0       LOOK:    LD    DE,SCHSTR+1
E0AD 1A           LOOKIT:  LD    A,(DE)
E0AE FE0D                  CP    0DH
E0B0 2807                  JR    Z,FOUND
E0B2 BE                    CP    (HL)
E0B3 20E3                  JR    NZ,FINDIT
E0B5 13                    INC   DE
E0B6 23                    INC   HL
E0B7 18F4                  JR    LOOKIT
                  ;
E0B9 E1           FOUND:   POP   HL              ;Address this srch
E0BA E5                    PUSH  HL              ;Restack it
E0BB CD95A9                CALL  0A995H
E0BE 3AD5E0                LD    A,(COUNT)
E0C1 3C                    INC   A
E0C2 FE05                  CP    5
E0C4 32D5E0                LD    (COUNT),A
E0C7 20CF                  JR    NZ,FINDIT
E0C9 AF                    XOR   A
E0CA 32D5E0                LD    (COUNT),A
E0CD CDC4A6                CALL  0A6C4H
E0D0 18C6                  JR    FINDIT
                  ;
E0D2 C300A0                JP    0A000H
E0D5 00           COUNT:   DB    0
                  SCHSTR:  DS    10
E0E0 4F636375     MSG1:    DB    "Occurences of ",0DH
E0E4 72656E63
E0E8 6573206F
E0EC 66200D
E0EF 62657477     MSG2:    DB    "between:-",0DH
E0F3 65656E3A
E0F7 2D0D
E0F9 616E6420     MSG3:    DB    "and ",0DH
E0FD 0D
                           END
```

Appendix

HEX to OPCODE Conversion Table

This first table is to assist when one knows the Hex value and wishes to know the opcode and the amount of bytes it should be followed by. When one attempts to convert decimal values in Basic DATA statements to Opcodes and Operands be sure to start with the first byte in the routine, else one could get false information.
As an example take program 1 in chapter 1. The first byte in the DATA line has the decimal value of 62, convert this to hex and one will see it is 3Ehex. Now look in the table below to find what 3E signifies. It is LD A,nn which means load register A with the value of the next byte which is 66 dec (42hex). Now continue with the third value in the DATA line which is 33 which converts to 21hex. On checking below one will see it signifies LD HL,aabb and must have the next two bytes loaded into HL and so on. If one began converting at the wrong place, say at the second byte, and tried to convert 66 to hex (42) and then looked in the table below it equals on its own LD B,D which would be totally wrong, therefore it is essential to start at the beginning.
In the table nn equals a one byte value in the range 00h to FFh (0 to 255 dec) and bb aa two bytes in the same range.

00	NOP
01 bb aa	LD BC,aabb
02	LD (BC),A
03	INC BC
04	INC B
05	DEC B
06 nn	LD B,nn
07	RLCA
08	EX AF,AF'
09	ADD HL,BC
0A	LD A,(BC)
0B	DEC BC
0C	INC C
0D	DEC C
0E nn	LD C,nn
0F	RRCA
10 nn	DJNZ nn
11 bb aa	LD DE,aabb
12	LD (DE),A
13	INC DE
14	INC D
15	DEC D
16 nn	LD D,nn
17	RLA

18 nn	JR nn
19	ADD HL,DE
1A	LD A,(DE)
1B	DEC DE
1C	INC E
1D	DEC E
1E nn	LD E,nn
1F	RRA
20 nn	JR NZ,nn
21 bb aa	LD HL,aabb
22 bb aa	LD (aabb),HL
23	INC HL
24	INC H
25	DEC H
26 nn	LD H,nn
27	DAA
28 nn	JR Z,nn
29	ADD HL,HL
2A bb aa	LD HL,(nn)
2B	DEC HL
2C	INC L
2D	DEC L
2E nn	LD L,nn
2F	CPL
30 nn	JR NC,nn
31 bb aa	LD SP,aabb
32 bb aa	LD (aabb),A
33	INC SP
34	INC (HL)
35	DEC (HL)
36 nn	LD (HL),nn
37	SCF
38 nn	JR C,nn
39	ADD HL,SP
3A bb aa	LD A,(aabb)
3B	DEC SP
3C	INC A
3D	DEC A
3E nn	LD A,nn
3F	CCF
40	LD B,B
41	LD B,C
42	LD B,D
43	LD B,E
44	LD B,H
45	LD B,Ln
46	LD B,(HL)
47	LD B,A
48	LD C,B
49	LD C,C
4A	LD C,D
4B	LD C,E
4C	LD C,H
4D	LD C,L
4E	LD C,(HL)
4F	LD C,A
50	LD D,B
51	LD D,C
52	LD D,D
53	LD D,E
54	LD D,H
55	LD D,L
56	LD D,(HL)
57	LD D,A
58	LD E,B
59	LD E,C
5A	LD E,D
5B	LD E,E
5C	LD E,H
5D	LD E,L
5E	LD E,(HL)
5F	LD E,A
60	LD H,B
61	LD H,C

62	LD H,D
63	LD H,E
64	LD H,H
65	LD H,L
66	LD H,(HL)
67	LD H,A
68	LD L,B
69	LD L,C
6A	LD L,D
6B	LD L,E
6C	LD L,H
6D	LD L,L
6E	LD L,(HL)
6F	LD L,A
70	LD (HL),B
71	LD (HL),C
72	LD (HL),D
73	LD (HL),E
74	LD (HL),H
75	LD (HL),L
76	HALT
77	LD (HL),A
78	LD A,B
79	LD A,C
7A	LD A,D
7B	LD A,E
7C	LD A,H
7D	LD A,L
7E	LD A,(HL)
7F	LD A,A
80	ADD A,B
81	ADD A,C
82	ADD A,D
83	ADD A,E
84	ADD A,H
85	ADD A,L
86	ADD A,(HL)
87	ADD A,A
88	ADC A,B
89	ADC A,C
8A	ADC A,D
8B	ADC A,E
8C	ADC A,H
8D	ADC A,L
8E	ADC A,(HL)
8F	ADC A,A
90	SUB B
91	SUB C
92	SUB D
93	SUB E
94	SUB H
95	SUB L
96	SUB (HL)
97	SUB A
98	SBC A,B
99	SBC A,C
9A	SBC A,D
9B	SBC A,E
9C	SBC A,H
9D	SBC A,L
9E	SBC A,(HL)
9F	SBC A,A
A0	AND B
A1	AND C
A2	AND D
A3	AND E
A4	AND H
A5	AND L
A6	AND (HL)
A7	AND A

A8	XOR B
A9	XOR C
AA	XOR D
AB	XOR E
AC	XOR H
AD	XOR L
AE	XOR (HL)
AF	XOR A
B0	OR B
B1	OR C
B2	OR D
B3	OR E
B4	OR H
B5	OR L
B6	OR (HL)
B7	OR A
B8	CP B
B9	CP C
BA	CP D
BB	CP E
BC	CP H
BD	CP L
BE	CP (HL)
BF	CP A
C0	RET NZ
C1	POP BC
C2 bb aa	JP NZ,aabb
C3 bb aa	JP aabb
C4 bb aa	CALL NZ,aabb
C5	PUSH BC
C6 nn	ADD A,nn
C7	RST 00
C8	RET Z
C9	RET
CA bb aa	JP Z,aabb

CB 00	RLC B
CB 01	RLC C
CB 02	RLC D
CB 03	RLC E
CB 04	RLC H
CB 05	RLC L
CB 06	RLC (HL)
CB 07	RLC A
CB 08	RRC B
CB 09	RRC C
CB 0A	RRC D
CB 0B	RRC E
CB 0C	RRC H
CB 0D	RRC L
CB 0E	RRC (HL)
CB 0F	RRC A
CB 10	RL B
CB 11	RL C
CB 12	RL D
CB 13	RL E
CB 14	RL H
CB 15	RL L
CB 16	RL (HL)
CB 17	RL A
CB 18	RR B
CB 19	RR C
CB 1A	RR D
CB 1B	RR E
CB 1C	RR H
CB 1D	RR L
CB 1E	RR (HL)
CB 1F	RR A
CB 20	SLA B
CB 21	SLA C
CB 22	SLA D

CB 23	SLA E
CB 24	SLA H
CB 25	SLA L
CB 26	SLA (HL)
CB 27	SLA A
CB 28	SRA B
CB 29	SRA C
CB 2A	SRA D
CB 2B	SRA E
CB 2C	SRA H
CB 2D	SRA L
CB 2E	SRA (HL)
CB 2F	SRA A
CB 30	SLI B
CB 31	SLI C
CB 32	SLI D
CB 33	SLI E
CB 34	SLI H
CB 35	SLI L
CB 36	SLI (HL)
CB 37	SLI A
CB 38	SRL B
CB 39	SRL C
CB 3A	SRL D
CB 3B	SRL E
CB 3C	SRL H
CB 3D	SRL L
CB 3E	SRL (HL)
CB 3F	SRL A
CB 40	BIT 0,B
CB 41	BIT 0,C
CB 42	BIT 0,D
CB 43	BIT 0,E
CB 44	BIT 0,H
CB 45	BIT 0,L
CB 46	BIT 0,(HL)
CB 47	BIT 0,A
CB 48	BIT 1,B
CB 49	BIT 1,C
CB 4A	BIT 1,D
CB 4B	BIT 1,E
CB 4C	BIT 1,H
CB 4D	BIT 1,L
CB 4E	BIT 1,(HL)
CB 4F	BIT 1,A
CB 50	BIT 2,B
CB 51	BIT 2,C
CB 52	BIT 2,D
CB 53	BIT 2,E
CB 54	BIT 2,H
CB 55	BIT 2,L
CB 56	BIT 2,(HL)
CB 57	BIT 2,A
CB 58	BIT 3,B
CB 59	BIT 3,C
CB 5A	BIT 3,D
CB 5B	BIT 3,E
CB 5C	BIT 3,H
CB 5D	BIT 3,L
CB 5E	BIT 3,(HL)
CB 5F	BIT 3,A
CB 60	BIT 4,B
CB 61	BIT 4,C
CB 62	BIT 4,D
CB 63	BIT 4,E
CB 64	BIT 4,H
CB 65	BIT 4,L
CB 66	BIT 4,(HL)
CB 67	BIT 4,A
CB 68	BIT 5,B

CB 69	BIT 5,C
CB 6A	BIT 5,D
CB 6B	BIT 5,E
CB 6C	BIT 5,H
CB 6D	BIT 5,L
CB 6E	BIT 5,(HL)
CB 6F	BIT 5,A
CB 70	BIT 6,B
CB 71	BIT 6,C
CB 72	BIT 6,D
CB 73	BIT 6,E
CB 74	BIT 6,H
CB 75	BIT 6,L
CB 76	BIT 6,(HL)
CB 77	BIT 6,A
CB 78	BIT 7,B
CB 79	BIT 7,C
CB 7A	BIT 7,D
CB 7B	BIT 7,E
CB 7C	BIT 7,H
CB 7D	BIT 7,L
CB 7E	BIT 7,(HL)
CB 7F	BIT 7,A
CB 80	RES 0,B
CB 81	RES 0,C
CB 82	RES 0,D
CB 83	RES 0,E
CB 84	RES 0,H
CB 85	RES 0,L
CB 86	RES 0,(HL)
CB 87	RES 0,A
CB 88	RES 1,B
CB 89	RES 1,C
CB 8A	RES 1,D
CB 8B	RES 1,E
CB 8C	RES 1,H
CB 8D	RES 1,L
CB 8E	RES 1,(HL)
CB 8F	RES 1,A
CB 90	RES 2,B
CB 91	RES 2,C
CB 92	RES 2,D
CB 93	RES 2,E
CB 94	RES 2,H
CB 95	RES 2,L
CB 96	RES 2,(HL)
CB 97	RES 2,A
CB 98	RES 3,B
CB 99	RES 3,C
CB 9A	RES 3,D
CB 9B	RES 3,E
CB 9C	RES 3,H
CB 9D	RES 3,L
CB 9E	RES 3,(HL)
CB 9F	RES 3,A
CB A0	RES 4,B
CB A1	RES 4,C
CB A2	RES 4,D
CB A3	RES 4,E
CB A4	RES 4,H
CB A5	RES 4,L
CB A6	RES 4,(HL)
CB A7	RES 4,A
CB A8	RES 5,B
CB A9	RES 5,C
CB AA	RES 5,D
CB AB	RES 5,E
CB AC	RES 5,H
CB AD	RES 5,L
CB AE	RES 5,(HL)

CB AF	RES 5,A
CB B0	RES 6,B
CB B1	RES 6,C
CB B2	RES 6,D
CB B3	RES 6,E
CB B4	RES 6,H
CB B5	RES 6,L
CB B6	RES 6,(HL)
CB B7	RES 6,A
CB B8	RES 7,B
CB B9	RES 7,C
CB BA	RES 7,D
CB BB	RES 7,E
CB BC	RES 7,H
CB BD	RES 7,L
CB BE	RES 7,(HL)
CB BF	RES 7,A
CB C0	SET 0,B
CB C1	SET 0,C
CB C2	SET 0,D
CB C3	SET 0,E
CB C4	SET 0,H
CB C5	SET 0,L
CB C6	SET 0,(HL)
CB C7	SET 0,A
CB C8	SET 1,B
CB C9	SET 1,C
CB CA	SET 1,D
CB CB	SET 1,E
CB CC	SET 1,H
CB CD	SET 1,L
CB CE	SET 1,(HL)
CB CF	SET 1,A
CB D0	SET 2,B
CB D1	SET 2,C
CB D2	SET 2,D
CB D3	SET 2,E
CB D4	SET 2,H
CB D5	SET 2,L
CB D6	SET 2,(HL)
CB D7	SET 2,A
CB D8	SET 3,B
CB D9	SET 3,C
CB DA	SET 3,D
CB DB	SET 3,E
CB DC	SET 3,H
CB DD	SET 3,L
CB DE	SET 3,(HL)
CB DF	SET 3,A
CB E0	SET 4,B
CB E1	SET 4,C
CB E2	SET 4,D
CB E3	SET 4,E
CB E4	SET 4,H
CB E5	SET 4,L
CB E6	SET 4,(HL)
CB E7	SET 4,A
CB E8	SET 5,B
CB E9	SET 5,C
CB EA	SET 5,D
CB EB	SET 5,E
CB EC	SET 5,H
CB ED	SET 5,L
CB EE	SET 5,(HL)
CB EF	SET 5,A
CB F0	SET 6,B
CB F1	SET 6,C
CB F2	SET 6,D
CB F3	SET 6,E
CB F4	SET 6,H

CB F5	SET 6,L
CB F6	SET 6,(HL)
CB F7	SET 6,A
CB F8	SET 7,B
CB F9	SET 7,C
CB FA	SET 7,D
CB FB	SET 7,E
CB FC	SET 7,H
CB FD	SET 7,L
CB FE	SET 7,(HL)
CB FF	SET 7,A
CC bb aa	CALL Z,aabb
CD bb aa	CALL aabb
CE nn	ADC A,nn
CF	RST 08
D0	RET NC
D1	POP DE
D2 bb aa	JP NC,aabb
D3 nn	OUT (nn),A
D4 bb aa	CALL NC,aabb
D5	PUSH DE
D6 nn	SUB nn
D7	RST 10
D8	RET C
D9	EXX
DA bb aa	JP C,aabb
DB nn	IN A,(nn)
DC bb aa	CALL C,nn
DD 09	ADD IX,BC
DD 19	ADD IX,DE
DD 21 bb aa	LD IX,aabb
DD 22 bb aa	LD (aabb),IX
DD 23	INC IX
DD 29	ADD IX,IX
DD 2A bb aa	LD IX,(aabb)
DD 2B	DEC IX
DD 34 nn	INC (IX+nn)
DD 35 nn	DEC (IX+nn)
DD 36 nn n1	LD (IX+nn),n1
DD 39	ADD IX,SP
DD 46 nn	LD B,(IX+nn)
DD 4E nn	LD C,(IX+nn)
DD 56 nn	LD D,(IX+nn)
DD 5E nn	LD E,(IX+nn)
DD 66 nn	LD H,(IX+nn)
DD 6E nn	LD L,(IX+nn)
DD 70 nn	LD (IX+nn),B
DD 71 nn	LD (IX+nn),C
DD 72 nn	LD (IX+nn),D
DD 73 nn	LD (IX+nn),E
DD 74 nn	LD (IX+nn),H
DD 75 nn	LD (IX+nn),L
DD 77 nn	LD (IX+nn),A
DD 7E nn	LD A,(IX+nn)
DD 86 nn	ADD A,(IX+nn)
DD 8E nn	ADC A,(IX+nn)
DD 96 nn	SUB (IX+nn)
DD 9E nn	SBC A,(IX+nn)
DD A6 nn	AND (IX+nn)
DD AE nn	XOR (IX+nn)
DD B6 nn	OR (IX+nn)
DD BE nn	CP (IX+nn)
DD CB nn 06	RLC (IX+nn)
DD CB nn 0E	RRC (IX+nn)
DD CB nn 16	RL (IX+nn)
DD CB nn 1E	RR (IX+nn)
DD CB nn 26	SLA (IX+nn)
DD CB nn 2E	SRA (IX+nn)
DD CB nn 36	SLI (IX+nn)
DD CB nn 3E	SRL (IX+nn)

DD CB nn 46	BIT 0,(IX+nn)
DD CB nn 4E	BIT 1,(IX+nn)
DD CB nn 56	BIT 2,(IX+nn)
DD CB nn 5E	BIT 3,(IX+nn)
DD CB nn 66	BIT 4,(IX+nn)
DD CB nn 6E	BIT 5,(IX+nn)
DD CB nn 76	BIT 6,(IX+nn)
DD CB nn 7E	BIT 7,(IX+nn)
DD CB nn 86	RES 0,(IX+nn)
DD CB nn 8E	RES 1,(IX+nn)
DD CB nn 96	RES 2,(IX+nn)
DD CB nn 9E	RES 3,(IX+nn)
DD CB nn A6	RES 4,(IX+nn)
DD CB nn AE	RES 5,(IX+nn)
DD CB nn B6	RES 6,(IX+nn)
DD CB nn BE	RES 7,(IX+nn)
DD CB nn C6	SET 0,(IX+nn)
DD CB nn CE	SET 1,(IX+nn)
DD CB nn D6	SET 2,(IX+nn)
DD CB nn DE	SET 3,(IX+nn)
DD CB nn E6	SET 4,(IX+nn)
DD CB nn EE	SET 5,(IX+nn)
DD CB nn F6	SET 6,(IX+nn)
DD CB nn FE	SET 7,(IX+nn)
DD E1	POP IX
DD E3	EX (SP),IX
DD E5	PUSH IX
DD E9	JP (IX)
DD F9	LD SP,IX
DE nn	SBC A,nn
DF	RST 18
E0	RET PO
E1	POP HL
E2 bb aa	JP PO,aabb
E3	EX (SP),HL
E4 bb aa	CALL PO,aabb
E5	PUSH HL
E6 nn	AND nn
E7	RST 20
E8	RET PE
E9	JP (HL)
EA bb aa	JP PE,aabb
EB	EX DE,HL
EC bb aa	CALL PE,aabb
ED 40	IN B,(C)
ED 41	OUT (C),B
ED 42	SBC HL,BC
ED 43 bb aa	LD (aabb),BC
ED 44	NEG
ED 45	RETN
ED 46	IM 0
ED 47	LD I,A
ED 48	IN C,(C)
ED 49	OUT (C),C
ED 4A	ADC HL,BC
ED 4B bb aa	LD BC,(aabb)
ED 4D	RETI
ED 4F	LD R,A
ED 50	IN D,(C)
ED 51	OUT (C),D
ED 53 bb aa	LD (aabb),DE
ED 56	IM 1
ED 57	LD A,I
ED 58	IN E,(C)
ED 59	OUT (C),E
ED 5A	ADC HL,DE
ED 5B bb aa	LD DE,(aabb)
ED 5E	IM 2
ED 5F	LD A,R
ED 60	IN H,(C)

ED 61	OUT (C),H
ED 62	SBC HL,HL
ED 67	RRD
ED 68	IN L,(C)
ED 69	OUT (C),L
ED 6A	ADC HL,HL
ED 6F	RLD
ED 70	IN F,(C)
ED 72	SBC HL,SP
ED 73 bb aa	LD (aabb),SP
ED 78	IN A,(C)
ED 79	OUT (C),A
ED 7A	ADC HL,SP
ED 7B bb aa	LD SP,(aabb)
ED A0	LDI
ED A1	CPI
ED A2	INI
ED A3	OUTI
ED A8	LDD
ED A9	CPD
ED AA	IND
ED AB	OUTD
ED B0	LDIR
ED B1	CPIR
ED B2	INIR
ED B3	OTIR
ED B8	LDDR
ED B9	CPDR
ED BA	INDR
ED BB	OTDR
EE nn	XOR nn
EF	RST 28
F0	RET P
F1	POP AF
F2 bb aa	JP P,aabb
F3	DI
F4 bb aa	CALL P,aabb
F5	PUSH AF
F6 nn	OR nn
F7	RST 30
F8	RET M
F9	LD SP,HL
FA bb aa	JP M,aabb
FB	EI
FC bb aa	CALL M,aabb
FD 09	ADD IY,BC
FD 19	ADD IY,DE
FD 21 bb aa	LD IY,aabb
FD 22 bb aa	LD (aabb),IY
FD 23	INC IY
FD 29	ADD IY,IY
FD 2A bb aa	LD IY,(aabb)
FD 2B	DEC IY
FD 34 nn	INC (IY+nn)
FD 35 nn	DEC (IY+nn)
FD 36 nn n1	LD (IY+nn),n1
FD 39	ADD IY,SP
FD 46 nn	LD B,(IY+nn)
FD 4E nn	LD C,(IY+nn)
FD 56 nn	LD D,(IY+nn)
FD 5E nn	LD E,(IY+nn)
FD 66 nn	LD H,(IY+nn)
FD 6E nn	LD L,(IY+nn)
FD 70 nn	LD (IY+nn),B
FD 71 nn	LD (IY+nn),C
FD 72 nn	LD (IY+nn),D
FD 73 nn	LD (IY+nn),E
FD 74 nn	LD (IY+nn),H
FD 75 nn	LD (IY+nn),L
FD 77 nn	LD (IY+nn),A

FD 7E nn	LD A,(IY+nn)
FD 86 nn	ADD A,(IY+nn)
FD 8E nn	ADC A,(IY+nn)
FD 96 nn	SUB (IY+nn)
FD 9E nn	SBC A,(IY+nn)
FD A6 nn	AND (IY+nn)
FD AE nn	XOR (IY+nn)
FD B6 nn	OR (IY+nn)
FD BE nn	CP (IY+nn)
FD CB nn 06	RLC (IY+nn)
FD CB nn 0E	RRC (IY+nn)
FD CB nn 16	RL (IY+nn)
FD CB nn 1E	RR (IY+nn)
FD CB nn 26	SLA (IY+nn)
FD CB nn 2E	SRA (IY+nn)
FD CB nn 36	SLI (IY+nn)
FD CB nn 3E	SRL (IY+nn)
FD CB nn 46	BIT 0,(IY+nn)
FD CB nn 4E	BIT 1,(IY+nn)
FD CB nn 56	BIT 2,(IY+nn)
FD CB nn 5E	BIT 3,(IY+nn)
FD CB nn 66	BIT 4,(IY+nn)
FD CB nn 6E	BIT 5,(IY+nn)
FD CB nn 76	BIT 6,(IY+nn)
FD CB nn 7E	BIT 7,(IY+nn)
FD CB nn 86	RES 0,(IY+nn)
FD CB nn 8E	RES 1,(IY+nn)
FD CB nn 96	RES 2,(IY+nn)
FD CB nn 9E	RES 3,(IY+nn)
FD CB nn A6	RES 4,(IY+nn)
FD CB nn AE	RES 5,(IY+nn)
FD CB nn B6	RES 6,(IY+nn)
FD CB nn BE	RES 7,(IY+nn)
FD CB nn C6	SET 0,(IY+nn)
FD CB nn CE	SET 1,(IY+nn)
FD CB nn D6	SET 2,(IY+nn)
FD CB nn DE	SET 3,(IY+nn)
FD CB nn E6	SET 4,(IY+nn)
FD CB nn EE	SET 5,(IY+nn)
FD CB nn F6	SET 6,(IY+nn)
FD CB nn FE	SET 7,(IY+nn)
FD E1	POP IY
FD E3	EX (SP),IY
FD E5	PUSH IY
FD E9	JP (IY)
FD F9	LD SP,IY
FE nn	CP nn
FF	RST 38

Instruction set in Alphabetical order

8E	ADC A,(HL)
DD 8E nn	ADC A,(IX+nn)
FD 8E nn	ADC A,(IY+nn)
8F	ADC A,A
88	ADC A,B
89	ADC A,C
8A	ADC A,D
8B	ADC A,E
8C	ADC A,H
8D	ADC A,L
CE nn	ADC A,nn
ED 4A	ADC HL,BC
ED 5A	ADC HL,DE
ED 6A	ADC HL,HL
ED 7A	ADC HL,SP

86	ADD A,(HL)
DD 86 nn	ADD A,(IX+nn)
FD 86 nn	ADD A,(IY+nn)
87	ADD A,A
80	ADD A,B
81	ADD A,C
82	ADD A,D
83	ADD A,E
84	ADD A,H
85	ADD A,L
C6 nn	ADD A,nn
09	ADD HL,BC
19	ADD HL,DE
29	ADD HL,HL
39	ADD HL,SP
DD 09	ADD IX,BC
DD 19	ADD IX,DE
DD 29	ADD IX,IX
DD 39	ADD IX,SP
FD 09	ADD IY,BC
FD 19	ADD IY,DE
FD 29	ADD IY,IY
FD 39	ADD IY,SP

A6	AND (HL)
DD A6 nn	AND (IX+nn)
FD A6 nn	AND (IY+nn)
A7	AND A
A0	AND B
A1	AND C
A2	AND D
A3	AND E
A4	AND H
A5	AND L
E6 nn	AND nn

CB 46	BIT 0,(HL)
DD CB nn 46	BIT 0,(IX+nn)
FD CB nn 46	BIT 0,(IY+nn)
CB 47	BIT 0,A
CB 40	BIT 0,B
CB 41	BIT 0,C
CB 42	BIT 0,D
CB 43	BIT 0,E
CB 44	BIT 0,H
CB 45	BIT 0,L

CB 4E	BIT 1,(HL)
DD CB nn 4E	BIT 1,(IX+nn)
FD CB nn 4E	BIT 1,(IY+nn)
CB 4F	BIT 1,A
CB 48	BIT 1,B

CB 49	BIT 1,C
CB 4A	BIT 1,D
CB 4B	BIT 1,E
CB 4C	BIT 1,H
CB 4D	BIT 1,L

CB 56	BIT 2,(HL)
DD CB nn 56	BIT 2,(IX+nn)
FD CB nn 56	BIT 2,(IY+nn)
CB 57	BIT 2,A
CB 50	BIT 2,B
CB 51	BIT 2,C
CB 52	BIT 2,D
CB 53	BIT 2,E
CB 54	BIT 2,H
CB 55	BIT 2,L

CB 5E	BIT 3,(HL)
DD CB nn 5E	BIT 3,(IX+nn)
FD CB nn 5E	BIT 3,(IY+nn)
CB 5F	BIT 3,A
CB 58	BIT 3,B
CB 59	BIT 3,C
CB 5A	BIT 3,D
CB 5B	BIT 3,E
CB 5C	BIT 3,H
CB 5D	BIT 3,L

CB 66	BIT 4,(HL)
DD CB nn 66	BIT 4,(IX+nn)
FD CB nn 66	BIT 4,(IY+nn)
CB 67	BIT 4,A
CB 60	BIT 4,B

CB 61	BIT 4,C
CB 62	BIT 4,D
CB 63	BIT 4,E
CB 64	BIT 4,H
CB 65	BIT 4,L

CB 6E	BIT 5,(HL)
DD CB nn 6E	BIT 5,(IX+nn)
FD CB nn 6E	BIT 5,(IY+nn)
CB 6F	BIT 5,A
CB 68	BIT 5,B
CB 69	BIT 5,C
CB 6A	BIT 5,D
CB 6B	BIT 5,E
CB 6C	BIT 5,H
CB 6D	BIT 5,L

CB 76	BIT 6,(HL)
DD CB nn 76	BIT 6,(IX+nn)
FD CB nn 76	BIT 6,(IY+nn)
CB 77	BIT 6,A
CB 70	BIT 6,B
CB 71	BIT 6,C
CB 72	BIT 6,D
CB 73	BIT 6,E
CB 74	BIT 6,H
CB 75	BIT 6,L

CB 7E	BIT 7,(HL)
DD CB nn 7E	BIT 7,(IX+nn)
FD CB nn 7E	BIT 7,(IY+nn)
CB 7F	BIT 7,A
CB 78	BIT 7,B

CB 79	BIT 7,C
CB 7A	BIT 7,D
CB 7B	BIT 7,E
CB 7C	BIT 7,H
CB 7D	BIT 7,L

DC bb aa	CALL C,aabb
FC bb aa	CALL M,aabb
D4 bb aa	CALL NC,aabb
CD bb aa	CALL aabb
C4 bb aa	CALL NZ,aabb
F4 bb aa	CALL P,aabb
EC bb aa	CALL PE,aabb
E4 bb aa	CALL PO,aabb
CC bb aa	CALL Z,aabb

3F	CCF

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

ED A9	CPD
ED B9	CPDR
ED A1	CPI
ED B1	CPIR

2F	CPL

27	DAA

35	DEC (HL)
DD 35 nn	DEC (IX+nn)
FD 35 nn	DEC (IY+nn)
3D	DEC A
05	DEC B
0B	DEC BC
0D	DEC C
15	DEC D
1B	DEC DE
1D	DEC E
25	DEC H
2B	DEC HL
DD 2B	DEC IX
FD 2B	DEC IY
2D	DEC L
3B	DEC SP

F3	DI

10 nn	DJNZ nn

FB	EI

E3	EX (SP),HL
DD E3	EX (SP),IX
FD E3	EX (SP),IY
08	EX AF,AF'
EB	EX DE,HL
D9	EXX

76	HALT

ED 46	IM 0
ED 56	IM 1
ED 5E	IM 2
ED 78	IN A,(C)
DB nn	IN A,(nn)
ED 40	IN B,(C)
ED 48	IN C,(C)
ED 50	IN D,(C)
ED 58	IN E,(C)
ED 70	IN F,(C)
ED 60	IN H,(C)
ED 68	IN L,(C)
34	INC (HL)
DD 34 nn	INC (IX+nn)
FD 34 nn	INC (IY+nn)
3C	INC A
04	INC B
03	INC BC
0C	INC C
14	INC D
13	INC DE
1C	INC E
24	INC H
23	INC HL
DD 23	INC IX
FD 23	INC IY
2C	INC L
33	INC SP
ED AA	IND
ED BA	INDR
ED A2	INI
ED B2	INIR

E9	JP (HL)
DD E9	JP (IX)
FD E9	JP (IY)
DA bb aa	JP C,aabb
FA bb aa	JP M,aabb
D2 bb aa	JP NC,aabb
C3 bb aa	JP aabb
C2 bb aa	JP NZ,aabb
F2 bb aa	JP P,aabb
EA bb aa	JP PE,aabb
E2 bb aa	JP PO,aabb
CA bb aa	JP Z,aabb
38 nn	JR C,nn
18 nn	JR nn
30 nn	JR NC,nn
20 nn	JR NZ,nn
28 nn	JR Z,nn
02	LD (BC),A
12	LD (DE),A
77	LD (HL),A
70	LD (HL),B
71	LD (HL),C
72	LD (HL),D
73	LD (HL),E
74	LD (HL),H
75	LD (HL),L
36 nn	LD (HL),nn
DD 77 nn	LD (IX+nn),A
DD 70 nn	LD (IX+nn),B
DD 71 nn	LD (IX+nn),C
DD 72 nn	LD (IX+nn),D
DD 73 nn	LD (IX+nn),E

DD 74 nn	LD (IX+nn),H
DD 75 nn	LD (IX+nn),L
DD 36 nn n1	LD (IX+nn),n1

FD 77 nn	LD (IY+nn),A
FD 70 nn	LD (IY+nn),B
FD 71 nn	LD (IY+nn),C
FD 72 nn	LD (IY+nn),D
FD 73 nn	LD (IY+nn),E
FD 74 nn	LD (IY+nn),H
FD 75 nn	LD (IY+nn),L
FD 36 nn n1	LD (IY+nn),n1

32 bb aa	LD (aabb),A
ED 43 bb aa	LD (aabb),BC
ED 53 bb aa	LD (aabb),DE
22 bb aa	LD (aabb),HL
DD 22 bb aa	LD (aabb),IX
FD 22 bb aa	LD (aabb),IY
ED 73 bb aa	LD (aabb),SP

0A	LD A,(BC)
1A	LD A,(DE)
7E	LD A,(HL)
DD 7E nn	LD A,(IX+nn)
FD 7E nn	LD A,(IY+nn)
3A bb aa	LD A,(aabb)
7F	LD A,A
78	LD A,B
79	LD A,C
7A	LD A,D
7B	LD A,E
7C	LD A,H
ED 57	LD A,I
7D	LD A,L
3E nn	LD A,nn
ED 5F	LD A,R

46	LD B,(HL)
DD 46 nn	LD B,(IX+nn)
FD 46 nn	LD B,(IY+nn)
47	LD B,A
40	LD B,B
41	LD B,C
42	LD B,D
43	LD B,E
44	LD B,H
45	LD B,L
06 nn	LD B,nn

ED 4B bb aa	LD BC,(aabb)
01 bb aa	LD BC,aabb

4E	LD C,(HL)
DD 4E nn	LD C,(IX+nn)
FD 4E nn	LD C,(IY+nn)
4F	LD C,A
48	LD C,B
49	LD C,C
4A	LD C,D
4B	LD C,E
4C	LD C,H
4D	LD C,L
0E nn	LD C,nn

56	LD D,(HL)
DD 56 nn	LD D,(IX+nn)
FD 56 nn	LD D,(IY+nn)

57	LD D,A
50	LD D,B
51	LD D,C
52	LD D,D
53	LD D,E
LD D,L	
16 nn	LD D,nn

ED 5B bb aa	LD DE,(aabb)
11 bb aa	LD DE,aabb

5E	LD E,(HL)
DD 5E nn	LD E,(IX+nn)
FD 5E nn	LD E,(IY+nn)
5F	LD E,A
58	LD E,B
59	LD E,C
5A	LD E,D
5B	LD E,E
5C	LD E,H
5D	LD E,L
1E nn	LD E,nn

66	LD H,(HL)
DD 66 nn	LD H,(IX+nn)
FD 66 nn	LD H,(IY+nn)
67	LD H,A
60	LD H,B
61	LD H,C
62	LD H,D
63	LD H,E
64	LD H,H
65	LD H,L
26 nn	LD H,nn

2A bb aa	LD HL,(aabb)
21 bb aa	LD HL,aabb

ED 47	LD I,A

DD 21 bb aa	LD IX,aabb

FD 2A bb aa	LD IY,(aabb)
FD 21 bb aa	LD IY,aabb

6E	LD L,(HL)
DD 6E nn	LD L,(IX+nn)
FD 6E nn	LD L,(IY+nn)
6F	LD L,A
68	LD L,B
69	LD L,C
6A	LD L,D
6B	LD L,E
6C	LD L,H
6D	LD L,L
2E nn	LD L,nn

ED 4F	LD R,A

ED 7B bb aa	LD SP,(aabb)
F9	LD SP,HL
DD F9	LD SP,IX
FD F9	LD SP,IY
31 bb aa	LD SP,aabb

ED A8	LDD
ED B8	LDDR
ED A0	LDI
ED B0	LDIR

ED 44	NEG

00	NOP

B6	OR (HL)
DD B6 nn	OR (IX+nn)
FD B6 nn	OR (IY+nn)
B7	OR A
B0	OR B
B1	OR C
B2	OR D
B3	OR E
B4	OR H
B5	OR L
F6 nn	OR nn

ED BB	OTDR
ED B3	OTIR

ED 79	OUT (C),A
ED 41	OUT (C),B
ED 49	OUT (C),C
ED 51	OUT (C),D
ED 59	OUT (C),E
ED 61	OUT (C),H
ED 69	OUT (C),L
D3 nn	OUT (nn),A

ED AB	OUTD
ED A3	OUTI

F1	POP AF
C1	POP BC
D1	POP DE
E1	POP HL
DD E1	POP IX
FD E1	POP IY

F5	PUSH AF
C5	PUSH BC
D5	PUSH DE
E5	PUSH HL
DD E5	PUSH IX
FD E5	PUSH IY

CB 86	RES 0,(HL)
DD CB nn 86	RES 0,(IX+nn)
FD CB nn 86	RES 0,(IX+nn)
CB 87	RES 0,A
CB 80	RES 0,B
CB 81	RES 0,C
CB 82	RES 0,D
CB 83	RES 0,E
CB 84	RES 0,H
CB 85	RES 0,L

CB 8E	RES 1,(HL)
DD CB nn 8E	RES 1,(IX+nn)
FD CB nn 8E	RES 1,(IY+nn)
CB 8F	RES 1,A
CB 88	RES 1,B
CB 89	RES 1,C
CB 8A	RES 1,D
CB 8B	RES 1,E
CB 8C	RES 1,H
CB 8D	RES 1,L

CB 96	RES 2,(HL)
DD CB nn 96	RES 2,(IX+nn)
FD CB nn 96	RES 2,(IY+nn)

CB 97	RES 2,A
CB 90	RES 2,B
CB 91	RES 2,C
CB 92	RES 2,D
CB 93	RES 2,E
CB 94	RES 2,H
CB 95	RES 2,L

CB 9E	RES 3,(HL)
DD CB nn 9E	RES 3,(IX+nn)
FD CB nn 9E	RES 3,(IY+nn)
CB 9F	RES 3,A
CB 98	RES 3,B
CB 99	RES 3,C
CB 9A	RES 3,D
CB 9B	RES 3,E
CB 9C	RES 3,H
CB 9D	RES 3,L

CB A6	RES 4,(HL)
DD CB nn A6	RES 4,(IX+nn)
FD CB nn A6	RES 4,(IY+nn)
CB A7	RES 4,A
CB A0	RES 4,B
CB A1	RES 4,C
CB A2	RES 4,D
CB A3	RES 4,E
CB A4	RES 4,H
CB A5	RES 4,L

CB AE	RES 5,(HL)
DD CB nn AE	RES 5,(IX+nn)
FD CB nn AE	RES 5,(IY+nn)
CB AF	RES 5,A
CB A8	RES 5,B
CB A9	RES 5,C
CB AA	RES 5,D
CB AB	RES 5,E
CB AC	RES 5,H
CB AD	RES 5,L

CB B6	RES 6,(HL)
DD CB nn B6	RES 6,(IX+nn)
FD CB nn B6	RES 6,(IY+nn)
CB B7	RES 6,A
CB B0	RES 6,B
CB B1	RES 6,C
CB B2	RES 6,D
CB B3	RES 6,E
CB B4	RES 6,H
CB B5	RES 6,L

CB BE	RES 7,(HL)
DD CB nn BE	RES 7,(IX+nn)
FD CB nn BE	RES 7,(IY+nn)
CB BF	RES 7,A
CB B8	RES 7,B
CB B9	RES 7,C
CB BA	RES 7,D
CB BB	RES 7,E
CB BC	RES 7,H
CB BD	RES 7,L

C9	RET
D8	RET C
F8	RET M
D0	RET NC
C0	RET NZ
F0	RET P
E8	RET PE

E0	RET PO
C8	RET Z
ED 4D	RETI
ED 45	RETN
CB 16	RL (HL)
DD CB nn 16	RL (IX+nn)
FD CB nn 16	RL (IY+nn)
CB 17	RL A
CB 10	RL B
CB 11	RL C
CB 12	RL D
CB 13	RL E
CB 14	RL H
CB 15	RL L
17	RLA
CB 06	RLC (HL)
DD CB nn 06	RLC (IX+nn)
FD CB nn 06	RLC (IY+nn)
CB 07	RLC A
CB 00	RLC B
CB 01	RLC C
CB 02	RLC D
CB 03	RLC E
CB 04	RLC H
CB 05	RLC L
07	RLCA
ED 6F	RLD
CB 1E	RR (HL)
DD CB nn 1E	RR (IX+nn)
FD CB nn 1E	RR (IY+nn)
CB 1F	RR A
CB 18	RR B
CB 19	RR C
CB 1A	RR D
CB 1B	RR E
CB 1C	RR H
CB 1D	RR L
1F	RRA
CB 0E	RRC (HL)
DD CB nn 0E	RRC (IX+nn)
FD CB nn 0E	RRC (IY+nn)
CB 0F	RRC A
CB 08	RRC B
CB 09	RRC C
CB 0A	RRC D
CB 0B	RRC E
CB 0C	RRC H
CB 0D	RRC L
0F	RRCA
ED 67	RRD
C7	RST 0
CF	RST 8h
D7	RST 10h
DF	RST 18h
E7	RST 20h
EF	RST 28h
F7	RST 30h
FF	RST 38h

9E	SBC A,(HL)
DD 9E nn	SBC A,(IX+nn)
FD 9E nn	SBC A,(IY+nn)
9F	SBC A,A
98	SBC A,B
99	SBC A,C
9A	SBC A,D
9B	SBC A,E
9C	SBC A,H
9D	SBC A,L
DE nn	SBC A,nn

ED 42	SBC HL,BC
ED 52	SBC HL,DE
ED 62	SBC HL,HL
ED 72	SBC HL,SP

37	SCF

CB C6	SET 0,(HL)
DD CB nn C6	SET 0,(IX+nn)
FD CB nn C6	SET 0,(IY+nn)
CB C7	SET 0,A
CB C0	SET 0,B
CB C1	SET 0,C
CB C2	SET 0,D
CB C3	SET 0,E
CB C4	SET 0,H
CB C5	SET 0,L

CB CE	SET 1,(HL)
DD CB nn CE	SET 1,(IX+nn)
FD CB nn CE	SET 1,(IY+nn)
CB CF	SET 1,A
CB C8	SET 1,B
CB C9	SET 1,C
CB CA	SET 1,D
CB CB	SET 1,E
CB CC	SET 1,H
CB CD	SET 1,L

CB D6	SET 2,(HL)
DD CB nn D6	SET 2,(IX+nn)
FD CB nn D6	SET 2,(IY+nn)
CB D7	SET 2,A
CB D0	SET 2,B
CB D1	SET 2,C
CB D2	SET 2,D
CB D3	SET 2,E
CB D4	SET 2,H
CB D5	SET 2,L

CB DE	SET 3,(HL)
DD CB nn DE	SET 3,(IX+nn)
FD CB nn DE	SET 3,(IY+nn)
CB DF	SET 3,A
CB D8	SET 3,B
CB D9	SET 3,C
CB DA	SET 3,D
CB DB	SET 3,E
CB DC	SET 3,H
CB DD	SET 3,L

CB E6	SET 4,(HL)
DD CB nn E6	SET 4,(IX+nn)
FD CB nn E6	SET 4,(IY+nn)
CB E7	SET 4,A
CB E0	SET 4,B
CB E1	SET 4,C
CB E2	SET 4,D

CB E3	SET 4,E
CB E4	SET 4,H
CB E5	SET 4,L

CB EE	SET 5,(HL)
DD CB nn EE	SET 5,(IX+nn)
FD CB nn EE	SET 5,(IY+nn)
CB EF	SET 5,A
CB E8	SET 5,B
CB E9	SET 5,C
CB EA	SET 5,D
CB EB	SET 5,E
CB EC	SET 5,H
CB ED	SET 5,L

CB F6	SET 6,(HL)
DD CB nn F6	SET 6,(IX+nn)
FD CB nn F6	SET 6,(IY+nn)
CB F7	SET 6,A
CB F0	SET 6,B
CB F1	SET 6,C
CB F2	SET 6,D
CB F3	SET 6,E
CB F4	SET 6,H
CB F5	SET 6,L

CB FE	SET 7,(HL)
DD CB nn FE	SET 7,(IX+nn)
FD CB nn FE	SET 7,(IY+nn)
CB FF	SET 7,A
CB F8	SET 7,B
CB F9	SET 7,C
CB FA	SET 7,D
CB FB	SET 7,E
CB FC	SET 7,H
CB FD	SET 7,L

CB 26	SLA (HL)
DD CB nn 26	SLA (IX+nn)
FD CB nn 26	SLA (IY+nn)
CB 27	SLA A
CB 20	SLA B
CB 21	SLA C
CB 22	SLA D
CB 23	SLA E
CB 24	SLA H
CB 25	SLA L

CB 36	SLI (HL)
DD CB nn 36	SLI (IX+nn)
FD CB nn 36	SLI (IY+nn)
CB 37	SLI A
CB 30	SLI B
CB 31	SLI C
CB 32	SLI D
CB 33	SLI E
CB 34	SLI H
CB 35	SLI L

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

CB 3E	SRL (HL)
DD CB nn 3E	SRL (IX+nn)
FD CB nn 3E	SRL (IY+nn)

CB 3F	SRL A
CB 38	SRL B
CB 39	SRL C
CB 3A	SRL D
CB 3B	SRL E
CB 3C	SRL H
CB 3D	SRL L

96	SUB (HL)
DD 96 nn	SUB (IX+nn)
FD 96 nn	SUB (IY+nn)
97	SUB A
90	SUB B
91	SUB C
92	SUB D
93	SUB E
94	SUB H
95	SUB L
D6 nn	SUB nn

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

HEX	DEC *256	DEC	H	D *256	D	H	D *256	D	H	D *256	D	H	D *256	D
00	00000	0	34	13312	52	68	26624	104	9C	39936	156	D0	53248	208
01	00256	1	35	13568	53	69	26880	105	9D	40192	157	D1	53504	209
02	00512	2	36	13824	54	6A	27136	106	9E	40448	158	D2	53760	210
03	00768	3	37	14080	55	6B	27392	107	9F	40704	159	D3	54016	211
04	01024	4	38	14336	56	6C	27648	108	A0	40960	160	D4	54272	212
05	01280	5	39	14592	57	6D	27904	109	A1	41216	161	D5	54528	213
06	01536	6	3A	14848	58	6E	28160	110	A2	41472	162	D6	54784	214
07	01792	7	3B	15104	59	6F	28416	111	A3	41728	163	D7	55040	215
08	02048	8	3C	15360	60	70	28672	112	A4	41984	164	D8	55296	216
09	02304	9	3D	15616	61	71	28928	113	A5	42240	165	D9	55552	217
0A	02560	10	3E	15872	62	72	29184	114	A6	42496	166	DA	55808	218
0B	02816	11	3F	16128	63	73	29440	115	A7	42752	167	DB	56064	219
0C	03072	12	40	16384	64	74	29696	116	A8	43008	168	DC	56320	220
0D	03328	13	41	16640	65	75	29952	117	A9	43264	169	DD	56576	221
0E	03584	14	42	16896	66	76	30208	118	AA	43520	170	DE	56832	222
0F	03840	15	43	17152	67	77	30464	119	AB	43776	171	DF	57088	223
10	04096	16	44	17408	68	78	30720	120	AC	44032	172	E0	57344	224
11	04352	17	45	17664	69	79	30976	121	AD	44288	173	E1	57600	225
12	04608	18	46	17920	70	7A	31232	122	AE	44544	174	E2	57856	226
13	04864	19	47	18176	71	7B	31488	123	AF	44800	175	E3	58112	227
14	05120	20	48	18432	72	7C	31744	124	B0	45056	176	E4	58368	228
15	05376	21	49	18688	73	7D	32000	125	B1	45312	177	E5	58624	229
16	05632	22	4A	18944	74	7E	32256	126	B2	45568	178	E6	58880	230
17	05888	23	4B	19200	75	7F	32512	127	B3	45824	179	E7	59136	231
18	06144	24	4C	19456	76	80	32768	128	B4	46080	180	E8	59392	232
19	06400	25	4D	19712	77	81	33024	129	B5	46336	181	E9	59648	233
1A	06656	26	4E	19968	78	82	33280	130	B6	46592	182	EA	59904	234
1B	06912	27	4F	20224	79	83	33536	131	B7	46848	183	EB	60160	235
1C	07168	28	50	20480	80	84	33792	132	B8	47104	184	EC	60416	236
1D	07424	29	51	20736	81	85	34048	133	B9	47360	185	ED	60672	237
1E	07680	30	52	20992	82	86	34304	134	BA	47616	186	EE	60928	238
1F	07936	31	53	21248	83	87	34560	135	BB	47872	187	EF	61184	239
20	08192	32	54	21504	84	88	34816	136	BC	48128	188	F0	61440	240
21	08448	33	55	21760	85	89	35072	137	BD	48384	189	F1	61696	241
22	08704	34	56	22016	86	8A	35328	138	BE	48640	190	F2	61952	242
23	08960	35	57	22272	87	8B	35584	139	BF	48896	191	F3	62208	243
24	09216	36	58	22528	88	8C	35840	140	C0	49152	192	F4	62464	244
25	09472	37	59	22784	89	8D	36096	141	C1	49408	193	F5	62720	245
26	09728	38	5A	23040	90	8E	36352	142	C2	49664	194	F6	62976	246
27	09984	39	5B	23296	91	8F	36608	143	C3	49920	195	F7	63232	247
28	10240	40	5C	23552	92	90	36864	144	C4	50176	196	F8	63488	248
29	10496	41	5D	23808	93	91	37120	145	C5	50432	197	F9	63744	249
2A	10752	42	5E	24064	94	92	37376	146	C6	50688	198	FA	64000	250
2B	11008	43	5F	24320	95	93	37632	147	C7	50944	199	FB	64256	251
2C	11264	44	60	24576	96	94	37888	148	C8	51200	200	FC	64512	252
2D	11520	45	61	24832	97	95	38144	149	C9	51456	201	FD	64768	253
2E	11776	46	62	25088	98	96	38400	150	CA	51712	202	FE	65024	254
2F	12032	47	63	25344	99	97	38656	151	CB	51968	203	FF	65280	255
30	12288	48	64	25600	100	98	38912	152	CC	52224	204			
31	12544	49	65	25856	101	99	39168	153	CD	52480	205			
32	12800	50	66	26112	102	9A	39424	154	CE	52736	206			
33	13056	51	67	26368	103	9B	39680	155	CF	52992	207			

The left column is the Hex code.
The centre column is the decimal equivalent multiplied by 256 for calculating the M.S.B
The third column is for use with the L.S.B. or single byte.

INDEX

Even with a good knowledge of machine code programming the user still requires additional information in how to access the inbuilt routines of a particular micro in order to achieve the simplest of tasks such as displaying messages on screen. Starting Machine Code on the MSX not only shows the ways the Z80 instructions are used but demonstrates these fundamental, but nevertheless crucial, routines in action in a way that even a first time user will find straightforward. From how to access machine code routine from Basic to using an Assembler, moving sprites and playing music in machine code on the MSX are all explained and demonstrated so lessening the imaginary and sometimes daunting divide between Basic and machine code programming.

7.95net

Published by

Kuma MSX

Kuma Computers Ltd., Pangbourne, Berkshire, England
Telephone 07357-4335 Telex 849462 TELFAC