PRACTICAL MSX MACHINE CODE PROGRAMMING

Bring your MSX to life for only £4.95

Learn how to write Machine Code programs

Uses less memory and runs up to 100 times faster than ordinary BASIC

£££££'s of practical information

By Steve Webb

PRACTICAL MSX MACHINE CODE PROGRAMMING

PRACTICAL MSX MACHINE CODE PROGRAMMING

BY

STEVE WEBB

First published in Great Britain in 1985
by Virgin Books Ltd, 61-63 Portobello
Road, London W11 3DD.

ISBN 0 86369 074 2

Designed by Ray Hyden.

Illustrations and layout by Sue Walliker.

Book production services by
Book Production Consultants, Cambridge.

Photoset by Keyline Graphics.

Printed and bound in Great Britain by
Richard Clay (The Chaucer Press) Ltd,
Suffolk.

Distributed by Arrow Books.

Contents

I would like to take this opportunity to thank Cat who has spent a great deal of time and effort in editing this book. My thanks are also extended to Sue Davies and to ZOE who makes each day a happier one.

Introduction

The intention of this book is to introduce you to Machine Code programming on MSX computers. The chapters have been written in a logical order and it is important that you read and understand each one before moving on to the next.

Machine Code programming is not as complex as you may have been led to believe. If you have a working knowledge of BASIC then you will be able to grasp the concepts of Machine Code programming very quickly.

The first few chapters deal with the theory of Machine Code, showing you how numbers are stored, answering the question "What is Machine Code?", describing the Machine Code equivalents of most BASIC statements, and teaching you about the organisation of the MSX computer. The main chapter then goes on to explain how to write a very simple Machine Code game. You are shown how a program can first be written as a flowchart, then the blocks of that flowchart divided into short, simple routines.

In the last few chapters you will be shown some useful routines and how to take advantage of the advanced facilities of the MSX. Throughout the book you will come across various questions, which you should attempt to answer (you will find the answers in the back of the book); if you get them wrong, re-read the appropriate section until you are able to answer them correctly.

The MSX Standard

MSX is a world-wide standard which, at the time of writing, has been accepted by over 25 large electronics companies. The standard is a huge step forward in the world of home computing, and it offers many exciting opportunities for the user and programmer. The standard applies to the hardware as well as to the software.

Let's suppose you have a Sony MSX and you have just written a program on it in BASIC or Machine Code and saved it on tape. It is now possible, because of the standard, to take that tape and load it into any other MSX computer. The program will now

work perfectly without modification – providing it has been written in accordance with the standard.

This interchangeability of software is possible due to the standardisation of the hardware and firmware. The system variables are all in a standard location in memory; just as importantly the screen layouts are the same. So how do you go about deciding which MSX computer to buy? As in the case of choosing a hi-fi system, you may like the look of one make more than another, or you may wish to remain loyal to a particular brand-name. You may buy a certain MSX because of its enhancements. These enhancements are made by individual manufacturers. One manufacturer may decide to fit a light-pen facility to his computer. The one thing to bear in mind about these enhancements is that they may or may not be available on other machines. If you have written a program which only works with a light-pen, then it is obvious that the program will not work on an MSX without a light-pen enhancement. And while we're on the subject of enhancements, these and peripherals such as printers are all governed by the standard.

Unfortunately, there is one thing which is not governed by the standard and that is memory size. You may have a 32k, 48k or 64k, depending on the manufacturer. A program will work on any MSX as long as it has the same or more memory than the computer it was written on. I personally predict that the 64k version will rapidly become accepted as the standard and that the others will fall by the wayside.

You should remember that MSX is a world standard. So if you've written a program in Britain, it will work just as well in Japan or any other part of the world for that matter.

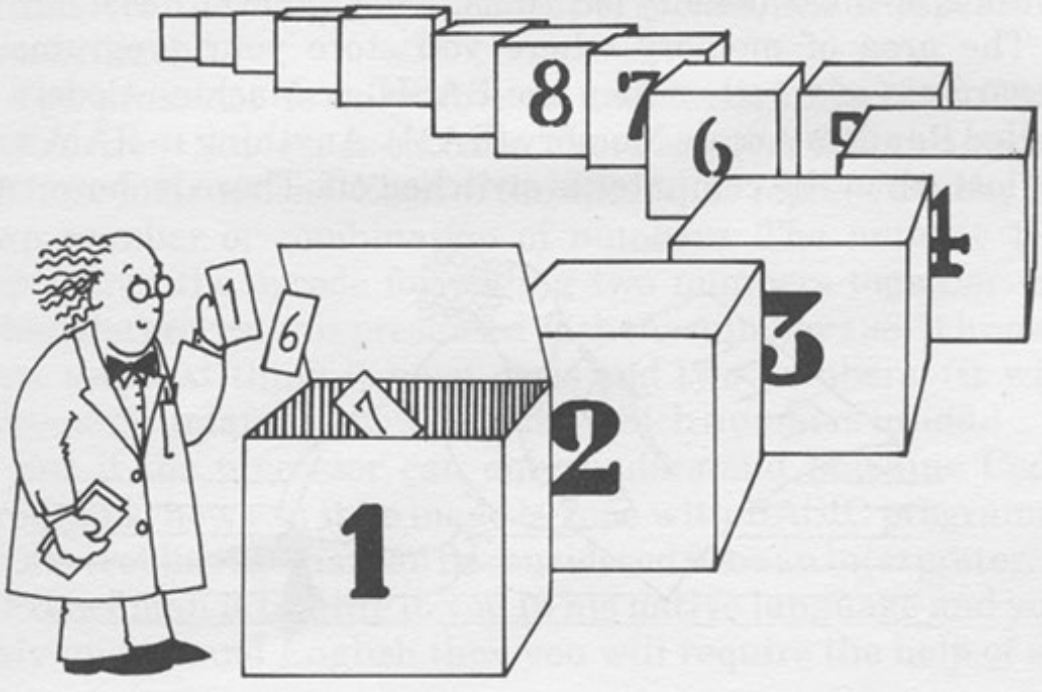
WHAT IS MACHINE CODE?

What Is Machine Code?

It is most likely that you have a 64k MSX computer. k is an abbreviation of 1000, so 64k means 64000. A 64k computer has got 64000 locations of memory. Each memory location can be considered to be a box, each box labelled from 0 to 63999. These labels do not actually exist anywhere, but we just refer to the first box as 0 and the next as 1 and so on. When you first switch on the computer you will have slightly over 28000 memory locations which you can use for either BASIC programs or Machine Code. This is what is left after 4000 of 32000 has been taken up by system variables. In a 64k computer, a further 32k is available in addition to the 28k described above. However, advanced Machine Code programming techniques are needed if you want to use this extra 32k – and this goes beyond the scope of this book. All the programs in this book will work on a 32k (or greater) MSX computer.

Each of the memory locations can hold a number between 0 and 255 inclusive. Only being able to store a maximum number of 255 is obviously a limitation, so a method has to be found for storing larger numbers. The following example shows how this is done.

Let's say that the number you wish to store is 29248. First we divide this number by 256, then round the answer down to the



nearest whole number. Thus: 29248 divided by 256 = 114.25, rounded down = 114. We call this, (114), the high part of the number to be stored. It represents the total number of 256s in the number. The high part, 114 in this example, is then multiplied by 256 and the answer subtracted from the number to be stored (29248).

114 x 256 = 29184
29248 – 29184 = 64

64 is called the low part of the number to be stored.

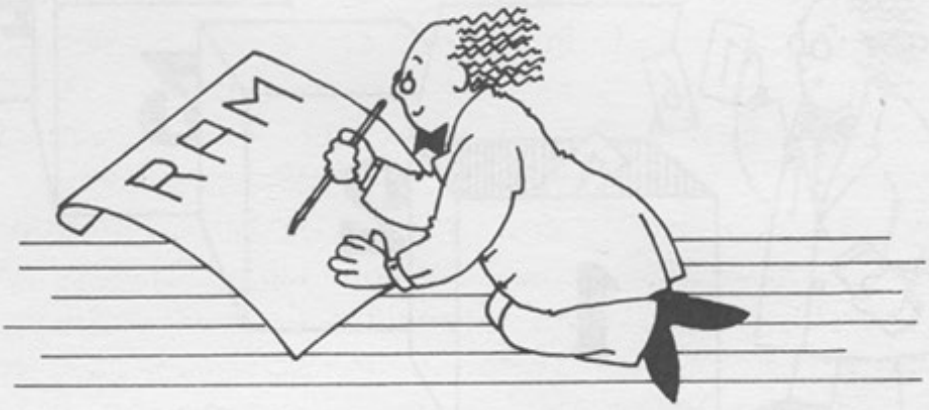
To store the number 29248 in memory, we put the low part, 64, in one memory location and the high part, 114, in the following location. So to store a number greater than 255 requires the use of two memory locations. Thus if you read in this book or anywhere else that a number greater than 255 is stored at location 50000, you will now know that it is in fact stored at locations 50000 and 50001.

What are the high and low parts of the number 45621?

If the low part of a number is 31 and the high part is 64, then what is the number?

Numbers between 0 and 255 are often referred to as bytes. So if you read that a program is 5000 bytes long, this means that it takes up 5000 memory locations.

The area of memory where you store your programs – regardless of whether they are BASIC or Machine Code – is called Random Access Memory (RAM). Anything in RAM will be lost when the computer is switched off. There is, however,



another area of memory called Read Only Memory (ROM). As its name suggests, you can only read the contents of the locations. The contents of the ROM will not be lost when the computer is switched off. This is just as well since it contains the operating system – and without it the computer would just be a useless junk of metal. The operating system contains many Machine Code routines which deal with the running of the computer, such as reading the keyboard, doing calculations and keeping a check on the syntax in BASIC programs.

At the centre of your computer is a processor called the Z80, and this is the part of the computer which does most of the hard work. It is not a brain; it is not intelligent. In fact it can only do very simple arithmetic. But the advantage it has over a human is that it can do many calculations per second, and so appears to be intelligent.

The processor is only able to understand Machine Code instructions, known as opcodes. There are over 600 variations of these instructions. Each of the opcodes is represented by its own number or combination of numbers. The number 198 represents the opcode for adding two numbers together. So when the processor is presented with the number 198, it knows that the next thing it must do is add two numbers. (It will become clear later on how it knows which numbers to add.)

But if the processor can only understand Machine Code programs, how can it be made to cope with BASIC programs? The secret lies in what can be considered to be an interpreter. If a Frenchman is talking to you in his native language and you only understand English then you will require the help of an

interpreter. The Frenchman will talk to the interpreter and the interpreter will translate for you. The same thing happens between a BASIC program and the processor. The BASIC program converses with an interpreter and the interpreter talks to the processor. This use of an interpreter obviously lengthens the time that it takes to have a "conversation", whether it is between a Frenchman and an Englishman or a BASIC program and the processor.

If we write our programs in Machine Code then we will be able to talk directly to the processor without the need for an interpreter. "Speaking" directly to the processor will greatly increase the speed at which the program is executed. In general, a Machine Code program is 50 to 100 times faster than the equivalent BASIC program.

MACHINE CODE EQUIVALENTS OF BASIC STATEMENTS

Machine Code Equivalents of BASIC Statements

In this section I will describe which opcodes can be used to simulate BASIC statements. Quite a lot of BASIC statements, such as LIST, NEW, RENUM, DELETE and AUTO, serve no purpose in Machine Code programming so I won't show any equivalent opcodes for them. Some BASIC statements, such as RUN, READ, DATA, PRINT, VPOKE and VPEEK, have no close equivalent in Machine Code but I will describe how to simulate them if and when needed.

The opcodes which I will describe here are the most commonly used ones. They are quite sufficient to enable you to write simple Machine Code programs.

In BASIC programming you will have been accustomed to using variables such as A, B, C. . .X, Y, Z. In Machine Code programming there are no variables as such; their nearest equivalents are registers. There are very few registers and the ones that you will mainly be using are labelled:

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

There is another register called F but we will not concern ourselves with it at this moment. Each register can be considered to be similar to a memory location; only a number between 0 and 255 can be stored. To enable us to use the registers to store numbers greater than 255, six of the registers have been grouped into three pairs of registers as follows:

HL
BC
DE

This pairing does not prevent the registers being used individually.

You will be familiar with the following BASIC statement:

LET A = 5

The equivalent opcode is:

LD A,5

LD is an abbreviation of the word LOAD. The complete opcode is read as "Load register A with a value of 5".

Any of the other six registers can be loaded with a number between 0 and 255, in a similar manner to register A.

LD H, 199 (Load register H with a value of 199)
LD D, 2 (Load register D with a value of 2)

I have already mentioned that each individual register can be considered as a memory location. So, remembering how numbers are stored, we know that the high part of 827 is 3, (827 divided by 256, rounded down), and the low part is 59. When the opcode LD HL, 827 is executed, the high part of the number is put into register H and the low part into register L. (This can easily be remembered by thinking of H = High and L = Low.) In register pair BC, B is the high part. In register pair DE, D is the high part.

BASIC: LET A = B
Meaning: Let variable A have the same value as variable B.
Opcode: LD A,B
Meaning: Load register A with the same value as contained by register B.

It is possible to load any of the single registers with the value of any other single register. You can have, for example:

LD A,H
LD E,A
LD H,C

The opcode LD is probably the most commonly used, and you will find it difficult to write a Machine Code program without it.

BASIC: LET A = A + 5
Meaning: Increase the present value of A by 5.
Opcode: ADD A,5
Meaning: Increase the value contained by register A by 5.

Register A is the only register you can directly add a number to. You cannot have opcodes such as:

```
ADD B,9
ADD E,3
```

This is obviously a limitation of the Z80 processor. However, it is easily overcome as you will discover in a short while.

BASIC: LET A = A + B
Meaning: Increase the value of A by the value of B.
Opcode: ADD A,B
Meaning: Increase the value contained by register A by the value contained by register B.

Any of the seven registers can be added to register A with the answer always appearing in register A.

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

The last opcode above is ADD A,A. This has the effect of doubling the present value of A. It is not possible to add any other combinations of registers other than those shown above. Thus you cannot have:

```
ADD B,H
ADD D,C
```

Register pairs can also be added together, but only in the following combinations:

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

The answer will always appear in HL. The last opcode, ADD HL,HL, obviously has the effect of doubling the value of HL. It is not possible to add a single register directly to a register pair.

Earlier on I mentioned that you cannot add a number directly to any register other than register A. I will now show you how to get round this limitation. Let's suppose that register B has a present value of 14 and you want to add 9 to it. The following example shows how to do this:

Step 1) **LD A,B**
Register A now has the same value as B.
Step 2) **ADD A,9**
The required addition is done with the answer in register A.
Step 3) **LD B,A**
The answer is transferred from register A to B.

With the three previous instructions it is now possible to add a number to any of the seven registers.

Another limitation of the Z80 which I pointed out was that you can only add a register to register A. So you cannot have:

ADD B,H

We get round this limitation in the following way. Let's say that B has a value of 5 and H has a value of 7 and that we want to add the registers together with the answer appearing in register B. The following example shows how to do this:

Step 1) **LD A,B**
Register A now has the same value as B.
Step 2) **ADD A,H**
The addition is complete with the answer in register A.
Step 3) **LD B,A**
The answer is transferred to register B.

With the above three steps it is now possible to add any combination of the registers. You can even add the value of one register to itself.

Using similar steps we can now add any register pair to any other register pair. To add the values of BC and DE, with the answer appearing in BC, we do the following:

Step 1) **LD H,B**
LD L,C
Register pair HL now has the same value as BC.
Step 2) **ADD HL,DE**
The addition is done with the answer in HL.
Step 3) **LD B,H**
LD C,L
The answer is transferred back into register pair BC.

Getting over the limitations of the Z80 is a challenge and with practice you will learn the most efficient way round them to suit your particular requirements.

BASIC: LET A = A – 5
Meaning: Decrease the value of A by 5.
Opcode: SUB A,5
Meaning: Decrease the value of register A by 5.

As with addition, register A is the only register we can directly subtract a number from. If you wish to subtract a number from any of the other six registers you must perform the following. This example will show you how to subtract 5 from register D.

Step 1) **LD A,D**
Register A now has the same value as D.
Step 2) **SUB A,5**
The subtraction is done with the answer in register A.
Step 3) **LD D,A**
The answer is transferred from A to D.

BASIC: LET A = A – B
Meaning: Decrease the value of A by the value of B.
Opcode: SUB B
Meaning: Subtract the value of register B from register A.

Again, register A is the only register you can directly subtract a value of another register from.

To subtract the value of a register from a register other than A you will have to use the three steps similar to those I have shown you.

Register pairs can also be subtracted from each other but only in the following combinations:

SBC HL,BC
SBC HL,DE
SBC HL,HL

SBC is a special form of subtraction. For some obscure reason the straightforward SUB opcode for register pairs was left out of the Z80. I will explain exactly what SBC means later. For the moment it is sufficient to regard it as an ordinary subtraction opcode. If you just wish to add one to the values of any register or register pair, there is a very useful opcode called INC. This means increment the value of the specified register by one. The possible combinations are:

INC A
INC B
INC C
INC D
INC E
INC H
INC L
INC HL
INC BC
INC DE

Similarly, if you wish to subtract one from the value of any register or register pair there is an opcode called DEC. This means decrement the specified register by one.

BASIC: GOTO (line number)
Meaning: Go to specified line number.
Opcode: JP (memory location)
Meaning: Jump to the specified memory location.

Jump to a specified memory location is virtually the same as go to a specified line number. In the following chapter I will explain how opcodes are stored in memory; then this opcode as well as others will become much clearer.

BASIC: GOSUB (line number)
Meaning: Go to a subroutine which starts at the specified line number. The subroutine is terminated with a return instruction.
Opcode: CALL (memory location)
Meaning: Go to a subroutine which starts at the specified memory location. As with a BASIC subroutine,

Machine Code subroutines must also be terminated with a return instruction.

BASIC: IF A = 5 THEN (GOTO/GOSUB/LET/etc.)
Meaning: If A = 5 then do whatever is specified.

In Machine Code programming we do have IF instructions but they are implemented in a different way than in BASIC.

The IF statement in Machine Code relates to the result of the last calculation carried out. A typical IF statement is:

CALL Z, (specified memory location)

This means that if the result of the last calculation was zero then call the subroutine at the specified memory location.

The following is a list of the most commonly used "IF" opcodes:

CALL NZ,nn
If the last result calculated was not zero then call the subroutine at the specified memory location nn.

CALL M,nn
If the last result calculated was minus then call the subroutine at memory location nn.

CALL P,nn
If the last result calculated was positive then call the subroutine at memory location nn.

JP Z,nn
If the last result was zero then jump to memory location nn.

JP NZ,nn
If the last result calculated was not zero then jump to memory location nn.

JP M,nn
If the last result calculated was minus then jump to memory location nn.

JP P,nn
If the last result calculated was positive then jump to memory location nn.

BASIC: **FOR A = 1 TO 100**
Routine which needs to be done 100 times.
NEXT A
Meaning: **Do the routine within the loop 100 times.**
Machine Code: **LD A,100**
Routine which needs to be done 100 times.
SUB A,1
JP NZ, start of routine.

To simulate a FOR/NEXT loop, we first load register A with 100, or however many times we need to perform the routine within the loop. At the end of the routine we subtract 1 from register A. If the result of subtracting 1 from register A was NZ (not zero) then we jump to the start of the routine. This continues until finally the result of subtracting 1 from A is zero, which means that the routine has been executed the required number of times.

When programming in BASIC, PEEK and POKE are the nearest that you normally get to Machine Code. This is because these two instructions are dealing directly with memory.

BASIC: **LET A = PEEK (40000)**
Meaning: **Let variable A have the same value as contained by memory location 40000.**

Opcode: **LD A, (40000)**
Meaning: **Load register A with the value contained by memory location 40000.**

If location 40000 contained 81 and we executed the opcode LD A, (40000), register A would now contain 81. Register A is the only single register which can be directly loaded with the contents of a memory location. You cannot have the following instruction, for example:

LD D, (40000)

Any of the three register pairs can be used to PEEK at memory location.

Opcode: LD HL, (40000)
The effect of this opcode is to load register L with the contents of location 40000 and to load register H with the contents of location 40001. You should be able to see how this is identical to how numbers greater than 255 are stored in memory locations.

If location 40000 contains 5 and location 40001 contains 15, what will be the total value of HL after the opcode LD HL, (40000)?

BASIC: POKE (40000),A
Meaning: Put the value of variable A into memory location 40000.
Opcode: LD (40000),A
Meaning: Put the value of register A into memory location 40000.

Register A is the only single register which can be directly loaded into a memory location.

Any of the three register pairs can be POKEd into memory locations.

Opcode: LD (40000),HL
Meaning: Load location 40000 with the value contained by register L and load location 40001 with the value contained by register H.

If HL contains the value 35621, what will be the values held in locations 40000 and 40001 after the opcode LD (40000),HL?

BASIC: LET A = 5
LET B = 40000
POKE (B),A
Meaning: Put the value contained by A into the memory location contained by B.
Opcodes: LD A,5
LD HL,40000
LD (HL),A
Meaning: Put the value contained by register A into the memory location contained by HL.

BASIC: LET B = 40000
LET A = PEEK(B)
Meaning: Let variable A have the value contained in the memory location held by B.
Opcodes: LD HL,40000
LD A,(HL)
Meaning: Load register A with the value contained by the memory location held by HL.

Register A is the only single register which can be loaded from the three register pairs as shown here:

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

The other six single registers can only be loaded from register pair HL as shown below:

```
LD B,(HL)
LD C,(HL)
LD D,(HL)
LD E,(HL)
LD H,(HL)
LD L,(HL)
```

Opcodes: PUSH
POP

These two opcodes do not really have BASIC equivalents. However, they are very important in Machine Code programming and the following example will help to explain what they are used for.

Suppose that we have a routine which uses all of the seven registers as shown below:

```
HL
BC
DE
 A
```

In BASIC if we wanted to execute this routine several times we would use a FOR/NEXT loop. I have already explained how to simulate a FOR/NEXT loop in Machine Code, so now let's try it.

```
                LD A,8 (number of times to repeat loop)
(start of routine) HL
                BC
                DE
                A
                SUB A,1
                JP NZ, (start of routine)
                END
```

I am sure that you can see that the above program will not work. It is like setting up a FOR/NEXT loop such as FOR A = 1 TO 8, and then using the variable A in the routine inside the loop. Very strange things would happen; the value of A controlling the loop would be destroyed by the use of A inside the routine. In BASIC this problem is easy to avoid because there are so many variables which can be used. In Machine Code programming it is a real problem and we use the PUSH and POP opcodes to overcome it.

The following shows how the routine should be rewritten using PUSH and POP:

```
                LD A,8
(start of routine) PUSH A
                HL
                BC
                DE
                A
                POP A
                SUB A,1
                JP NZ, (start of routine)
```

After we have loaded register A with 8 we then PUSH it. In very simple terms this means that we put the current value of A in a "safe" place called a stack. We can then execute the main routine without fear of destroying the original value of A. At the end of the routine we then POP A. This recovers the original value of A.

We can PUSH and POP any of the register pairs, but we cannot PUSH or POP individual registers. Now I can hear you saying "But you've just shown us how to PUSH A which is an individual register". I admit that I did but it was to avoid confusion. There is no such opcode as PUSH A, but you can PUSH a register pair named AF. F is a special register which cannot be used like the other seven single registers, although

before I briefly describe the F register, let me say that it is entirely possible to write very sophisticated Machine Code programs without one reference to the F register.

This F register is primarily used by the Z80 processor to indicate the results of various calculations. Depending on the value contained by the F register, the processor can tell such things as whether the last calculation was zero, whether it was positive or negative or whether the calculation caused a carry. It is this carry which makes the difference between the opcodes SUB (Subtract) and SBC (Subtract with carry). Very simply SBC means that if the carry was set then it will be included in the subtraction. I think that from this description of the F register you will realize that it should not be used by an inexperienced programmer – so I won't be mentioning it again.

STORING OPCODES IN MEMORY

Storing Opcodes in Memory

By now you will have a firm understanding of how numbers are stored in memory. That is, you can only store numbers between 0 and 255 in any memory location. I have also mentioned that opcodes are represented by a unique number or combination of numbers. In practice what this means is that some opcodes are represented by a number between 0 and 255, while others are represented by two numbers between 0 and 255.

The numbers which represent opcodes are stored in memory just like any other numbers. Some opcodes require one memory location, others require two memory locations.

If the first opcode of a program was INC A (increment A) then we would first need to know which number represents INC A. It is in fact 60. Next we would need to know where the program should start in memory, let's say 40000. The number 60 then has to be put into location 40000 (I'll explain how this is done later on). This is then the first opcode of the program in memory. If the next opcode was INC HL we would need to put its number, 52, into location 40001.

The whole program is built up by storing the numbers which represent the required opcodes. The two opcodes I have just mentioned are only one byte long – that is, they are stored in one memory location. However, some opcodes such as ADD A, 5 require two memory locations. One location holds the actual opcode which is ADD A. The next consecutive location must hold the number which you wish to add to A. If ADD A,5 was the first opcode of a program starting at location 40000 then we would need to put 198 (which is the number for ADD A) into location 40000. In location 40001 we would have to put 5.

When the program is started (I will show you how to do this in a while), the processor would look at the contents of the first location and see that the opcode is ADD A. It would then look at location 40001 to see how much it should add to A. Once it had completed the addition it would move on to the next opcode which starts at location 40002.

Some opcodes are two bytes long and may need to be followed by one or two bytes, making the total instruction three or four bytes long. Again, these bytes must be stored consecutively in the memory.

By now you will have noticed that, unlike BASIC instructions, Machine Code instructions do not have line numbers. The instructions of Machine Code programs are stored directly one after another in memory. The processor keeps a check on the address of the instruction it is currently executing. When the instruction is completed, the processor moves on to the next one. Despite the lack of line numbers it is still possible to have instructions such as GOTO and GOSUB. Instead of having to GOTO line numbers we can GOTO specific memory locations.

When we input a Machine Code program we do not type in numbers using our normal decimal numbering system, instead we use a numbering system called HEXADECIMAL (HEX for short). To give you an idea of what HEX looks like, the following shows a few decimal numbers and their HEX equivalents:

DECIMAL	HEX
0	00
9	09
10	0A
15	0F
16	10
255	FF

A complete table of numbers and their HEX equivalents is given in the appendix. Throughout this book, where it is necessary to avoid confusion, I have used d and h to distinguish between a decimal and HEX number.

8d = 8 decimal
12h = 12 HEX

What is the decimal equivalent of E3h?

If FBh is the high part of a number and CBh is the low part, then what is the number in decimal?

Use the decimal/HEX table at the back of this book to help you with the above two questions.

Do not worry too much about the HEX numbering system. With the use of the table in the appendix you will be able to convert very easily from one to another. One advantage of the HEX numbering system is that it is much neater than decimal. All HEX numbers between 0 and 255 are represented by two

digits; a decimal number, however, between 0 and 255 can be one, two, or three digits long.

There are several methods of actually entering numbers and opcodes into memory. I have written a small BASIC program which will enable you to enter and check Machine Code programs very quickly.

Enter the following program and check it very thoroughly. Then SAVE it on tape using:

SAVE "CAS:HEXENT"

```
 10 CLEAR200,39999
 15 CLS
 20 LOCATE 0,0
 25 PRINT "PUT CAPS LOCK ON"
 30 LOCATE 0,4
 35 PRINT "PRESS KEY E TO ENTER HEX CODE"
 40 LOCATE 0,8
 45 PRINT "PRESS KEY C TO CHECK HEX CODE"
 50 LOCATE 0,12
 55 PRINT "PRESS KEY X TO CHECK HEX TOTAL"
 60 LOCATE 0,16
 65 PRINT "PRESS KEY Q TO STOP"
 70 A$=INKEY$
 75 IF A$="E"THEN GOTO 185
 80 IF A$="C"THEN GOTO 380
 85 IF A$="X"THEN GOTO 100
 90 IF A$="Q"THEN STOP
 95 GOTO70
100 CLS
105 LOCATE 0,0
110 PRINT "INPUT STARTING ADDRESS"
115 INPUT SA
120 LOCATE 0,5
125 PRINT "INPUT END ADDRESS"
130 INPUT EA
135 LET D=0
140 FOR C=SA TO EA
145 LET D=D+PEEK(C)
150 NEXT C
155 CLS
160 PRINT "TOTAL COUNT = ";D
165 LOCATE 0,20
170 PRINT "PRESS M TO RETURN TO MENU"
175 IF INKEY$<>"M"THEN GOTO 175
180 GOTO15
185 CLS
190 LOCATE 0,0
```

```
195 PRINT "PUT CAPS LOCK ON"
200 LOCATE 0,4
205 PRINT "INPUT STARTING ADDRESS"
210 INPUT S
215 IF S<40000 THEN GOTO 370
220 LET A$=""
225 LOCATE 0,23
230 LET ET = S
235 IF A$="" THEN INPUT A$
240 LET BAD=0
245 IF A$="M"THEN GOTO15
250 LET E=LEN(A$)
255 LET E=E-1
260 LET C$=A$
265 FOR D=1 TO E STEP 2
270 LET B$=LEFT$(C$,2)
275 LET C=VAL ("&H"+B$)
280 IF C=0 THEN GOSUB 360
285 LET C$=MID$(C$,3)
290 NEXT D
295 IF BAD =1 THEN GOTO 345
300 LOCATE 0,21:PRINT S;" "A$
305 LET B$=LEFT$(A$,2)
310 IF LEN (B$)=1 THEN GOTO345
315 LET C=VAL("&H"+B$)
320 POKE (S) ,C
325 LET S=S+1
330 LET A$=MID$(A$,3)
335 IF A$=""THEN GOTO 230
340 GOTO 305
345 LOCATE 0,22:PRINT "INCORRECT INPUT TRY AGAIN"
350 LET S=ET
355 GOTO 220
360 IF B$<>"00"THEN LET BAD =1
365 RETURN
370 PRINT "START ADDRESS MUST BE 40000 OR GREATER"
375 GOTO 205
380 LET ND =0
385 CLS
390 LOCATE 0,0
395 PRINT "INPUT STARTING ADDRESS"
400 INPUT SA
405 LOCATE 0,5
410 PRINT "INPUT END ADDRESS"
415 INPUT EA
420 CLS
425 IF SA+20>EA THEN GOTO490
430 FOR C=SA TO SA + 20
435 IF PEEK(C)<16 THEN GOTO480
440 PRINT C;" ";HEX$(PEEK(C))
```

```
445 NEXT C
450 IF ND=1 THEN GOTO 505
455 IF C>EA THEN GOTO 505
460 PRINT "PRESS M FOR MORE"
465 LET SA=C
470 IF INKEY$ <> "M" THEN GOTO 470
475 GOTO425
480 PRINT C;" 0";HEX$(PEEK(C))
485 GOTO 445
490 FOR C=SA TO EA
495 LET ND=1
500 GOTO 435
505 PRINT "PRESS M TO RETURN TO THE MENU"
510 IF INKEY$ <>"M" THEN GOTO 510
515 GOTO 15
```

When you wish to load the program type the following:

LOAD "CAS:",r

When it has loaded it will automatically RUN and you will see the following menu:

Press key E to input code.
Press key C to check code.
Press key X to start the count.
Press key Q to stop.

To enter Machine Code you must press key E. You will then be asked to put the CAPS LOCK on.

Here is the first Machine Code program for you to enter. It will add two numbers together then put the result into location 40100. This is really just a check to ensure that you have correctly entered the HEXENT program and to make you familiar with using it. The correct sequence for entering this program is shown below:

1) Press key E.
2) Put the CAPS LOCK on.
3) You will then be asked to enter the start address of the program, in this case it is 40000.
4) You can then start to enter the HEX codes. The first line is 3E05 (press enter).

This is two bytes of HEX which will go into locations 40000 and 40001.

5) Enter the other three lines in a similar manner, pressing enter after each line.
6) After you have entered the last line and pressed enter, you will need to get back to the menu by pressing M then enter.

Start address **40000**
End address **40007**
Hex total **852**

```
3E05        LD A,5
C610        ADD A,16
32A49C      LD (40100),A
C9          RET
```

The next thing to do is to check that you have entered the code correctly. This is done by pressing C. You will then be asked to enter the start address of the code you wish to check, which in this case is 40000. Then you will be asked to enter the end address of the code, which in this case is 40007. The screen will then display the memory locations and their contents. You should check this very carefully; if there is an error you should go back to the menu and re-enter the code.

There is another checking facility which is obtained by pressing key X. You will then be asked for the start and end address again. The HEXENT program will then add up the contents of all these locations and print out the total, in this case it should be 852. If it is not, then you have either entered the code incorrectly or, less likely, you have mistyped the HEXENT program. Make sure that you cure the problem before proceeding.

All this checking is vitally important because, unlike BASIC – which will just stop if it has an error – an error in Machine Code will usually cause the computer to tie itself in knots and you will have to switch it off.

To test the small program you have just entered, first stop HEXENT by pressing key Q. Now enter the following lines:

```
1000 def usr = 40000
1005 a = usr(1)
```

These two lines can be regarded as meaning GOSUB the

Machine Code routine at memory location 40000. If you look at the last instruction of the Machine Code program you will find that it is RET, meaning return. When the processor reaches this opcode it will return to where it came from, in this case back to BASIC. This is exactly how a subroutine in BASIC works.

Now type GOTO 1000 to execute the Machine Code routine. Now type PRINT PEEK(40100). The number 21 should be printed, the result of adding 16 and 5.

The above procedure for entering and checking Machine Code routines applies to all the listings in this book, so I will not explain it again. It is important to note the start and end addresses and the total count, and these are given at the start of each listing. The procedure for testing the routines in each case is also similar. You will be given some lines of BASIC to enter which will always start with line 1000; you initiate it by GOTO 1000. When the testing has finished, the HEXENT program can be restarted by typing RUN.

UNDERSTANDING THE SCREEN DISPLAY

Understanding the Screen Display

One very useful feature of the MSX is its four different display modes. These four modes are:

MODE 0 24 ROWS 40 CHARACTERS PER ROW
MODE 1 24 ROWS 32 CHARACTERS PER ROW
MODE 2 HIGH RESOLUTION MODE
MODE 3 MULTICOLOUR MODE

Each mode has its own advantage and is selected according to the requirements of the program.

Throughout the rest of this book I will deal exclusively with SCREEN MODE 1, and unless otherwise stated everything will relate to MODE 1. In a later chapter I will describe the other modes in greater detail. Mode 1 is probably the best when writing games programs.

The first question to ask is "How does a picture appear on the screen?" The characters or text that you wish to display are held in memory as a series of numbers. The computer hardware is constantly looking at the area of memory that contains your picture and then does the hard work of turning it into a TV picture.

So where is the picture stored in memory? I have already described the area of memory called RAM where your BASIC or Machine Code programs are stored. Somewhere deep within your MSX machine is another area of memory called VIDEO RAM (VRAM). The VRAM is 16384 bytes long and all MSX computers have it.

The VRAM is used to store your picture as well as other things connected with it, such as the shape of characters and sprites (I will deal with this later). In screen mode 1 there is a map in VRAM which is 32 bytes wide by 24 deep, making a total of 768 bytes. Each of the 768 positions of the map correspond to a position on the screen where it is possible to print a character. The number 65 represents the letter A; if the first position of the map contained 65 then the letter A would appear in the top left-hand corner of the screen. Let's try and make this actually

happen. The first thing to do is to find the address of the first position of the map; this is done by entering the following lines:

```
SCREEN 1
PRINT BASE (5)
```

The first line ensures that you are in mode 1. The second line will print a number which is the start address of your screen map. You should make a note of this number; we will refer to it as SCREENSTART. In my MSX the SCREENSTART is 6144. Now let's put the number 65 into this location by entering the following line:

```
VPOKE (SCREENSTART),65
```

You should now see the letter A in the top left-hand corner. It is a simple matter to calculate the address of any other character position on the screen. The top right-hand corner of the screen, for example, is SCREENSTART + 31. (NB: on some televisions it may not be possible to see the extreme corners of the screen, so try VPOKEing SCREENSTART+5 instead.)

The screen map can be VPOKEd with any number between 0 and 255 and the corresponding character will appear on the screen. The shapes of the characters are stored in VRAM. Each character requires 8 bytes to store its shape, so that a total of 2048 bytes is required to store the shapes of all the 256 characters. The start address of the characters' shapes can be found by entering the following line:

```
PRINT BASE (7)
```

This will print a number which is the start address in VRAM of the characters' shapes. In my MSX, the address is zero. We will refer to this address as CHARSTART. The start address of any of the other characters can be found in the following way:

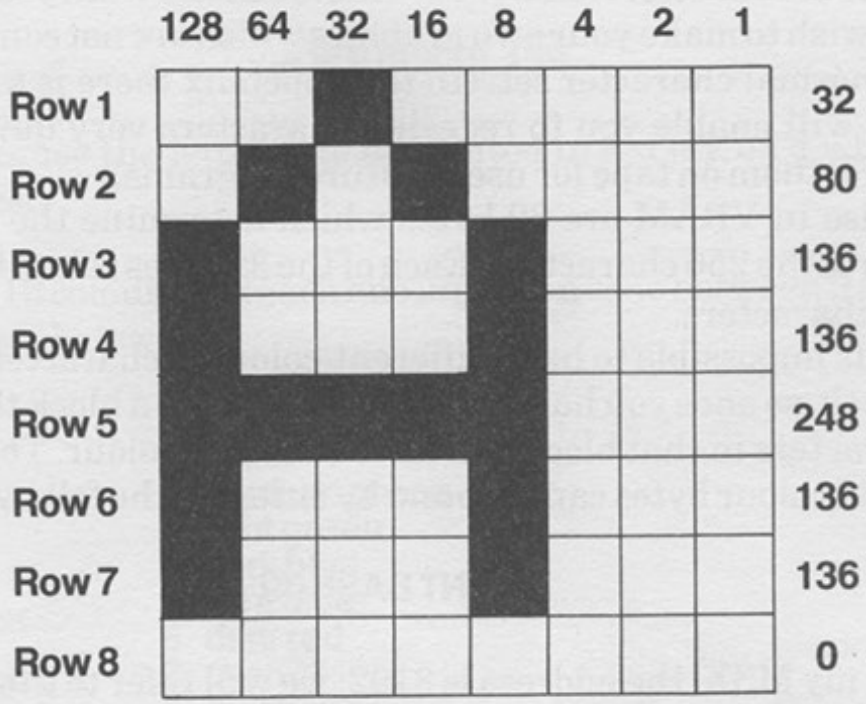
```
(Character number * 8) + CHARSTART
```

Now let's examine how a character shape is stored in VRAM. We will use the capital letter A as an example. In my MSX, the 8 bytes for the letter A start at location:

(65 * 8) + CHARSTART = 520

(65 is the code for the letter A.)

Each of the 256 characters can be drawn on an 8 by 8 grid, as shown below for the letter A.



Each of the eight rows represent one of the 8 bytes needed to store the character shape. Each of the rows can be converted into a number. (If you wish to know how this is done, then read the section in the appendix on binary.) It is these 8 numbers which are stored in VRAM and which make up the character SHAPE. The following example shows how the shapes are stored:

VPOKE (SCREENSTART+5),65

This will print letter A on the screen. What we will do now is alter the actual shape of character number 65. Enter the following lines and watch the character appear on screen each time that you press return:

```
VPOKE (CHARSTART + 520), 255
VPOKE (CHARSTART + 521), 129
VPOKE (CHARSTART + 522), 129
```

```
VPOKE(CHARSTART + 523), 129
VPOKE(CHARSTART + 524), 129
VPOKE(CHARSTART + 525), 129
VPOKE(CHARSTART + 526), 129
VPOKE(CHARSTART + 527), 255
```

You should now see that we have completely redesigned the shape of character number 65. This is exactly what you do when you wish to make your own graphics which are not contained in the normal character set. (In the appendix there is a program that will enable you to redesign characters very quickly and SAVE them on tape for use in future programs.)

Also in VRAM are 32 bytes which determine the colour of each of the 256 characters. Each of the 32 bytes relate to a block of 8 characters.

It is impossible to have different-coloured characters within a block, so once you have assigned a colour to a block then the 8 characters in that block will have the same colour. The start of the 32 colour bytes can be found by entering the following line:

```
PRINT BASE(6)
```

In my MSX, the address is 8192; we will refer to this address as COLSTART. To demonstrate how the colour table controls the colour of the character, type in the following example:

```
SCREEN 1
VPOKE(SCREENSTART +5), 65
```

This will put the letter on the screen. We will now alter the byte in the colour table relating to the block of 8 characters which the letter A is in. To calculate which of the 32 bytes controls the colour of character A, type in the following:

```
COLSTART + (CHARACTER NUMBER / 8)
```

Ignore the remainder in the answer. In my MSX, the address of the byte which relates to the letter A is:

```
8192 + (65/8) = 8200
```

In the following example I will be VPOKEing location 8200;

you, of course, must VPOKE the address you got from the above calculation.

VPOKE (8200),241

This will cause the letter A to be printed in white on a black background.

VPOKE (8200),159

This will cause the letter A to be printed in red ink on a white background.

There are 16 colours and each one is given a code between 0 and 15, as shown below:

0 transparent
1 black
2 medium green
3 light green
4 dark blue
5 light blue
6 dark red
7 cyan
8 medium red
9 light red
10 dark yellow
11 light yellow
12 dark green
13 magenta
14 grey
15 white

The code to be VPOKEd into VRAM to produce the required character colours can be calculated in the following way:

Get the code for the ink colour. Multiply it by 16 then add the code for the paper colour. Thus the code for magenta ink is 13 multiplied by 16 = 208, and the code for white paper is 15. Therefore the code for white paper and magenta ink is 208 + 15 = 223.

SPRITES

Sprites

One of the very useful features of the MSX is its sprites. What are sprites? Let's suppose you are writing a game and you need to move something smoothly across the screen. If you didn't have sprites, you would first have to print a character in one position, then overprint it with a space, then print the character in the next position. This process is not only tedious, it is also slow and the movement of the object would be very jerky.

With the use of sprites it is a joy moving objects around the screen. Imagine that a sprite is like an ordinary character; you can define its colour and shape. A sprite can be printed anywhere on the screen – it is not confined to being printed within a character cell. Sprites do not have to be rubbed out before they can be moved. And to move a sprite from BASIC or Machine Code only involves changing the contents of a memory location.

It is possible to have up to 32 sprites on the screen (although there is a limitation to this which I will deal with later). Sprites can also be one of four sizes, but it is not possible to have different-sized sprites on the screen. In the following example we will use small-size sprites, which are in fact the same size as actual characters. The 32 sprites are numbered from 0 to 31. To set the sprite size type in the following:

```
VDP(1) = 224
```

I will deal with the other three sizes in a later chapter. In VRAM there is an area of memory which holds the shapes of the sprites. The start address of this area can be found by typing in:

```
PRINT BASE(9)
```

In my MSX the start address is 14336. I will call this address SPRITE SHAPE START. As with ordinary characters, the shapes of these small sprites are stored in 8 bytes. In my MSX, the 8 bytes of data for sprite number 0 will be from 14336 to 14343. We will now fill these 8 bytes with data to represent a Space Invader.

```
VPOKE SPRITE SHAPE START + 0,60
VPOKE SPRITE SHAPE START + 1,90
VPOKE SPRITE SHAPE START + 2,255
VPOKE SPRITE SHAPE START + 3,231
VPOKE SPRITE SHAPE START + 4,128
VPOKE SPRITE SHAPE START + 5,36
VPOKE SPRITE SHAPE START + 6,66
VPOKE SPRITE SHAPE START + 7,36
```

It is unlikely that anything has appeared on the screen; this is because we have not yet defined where we want the sprite to appear. In VRAM there is an area of memory which contains information about the position and colour of the sprite. Each of the 32 sprites has four variables associated with it:

1) Vertical position
2) Horizontal position
3) Sprite number
4) Colour

The start address of these variables can be found by typing in:

```
PRINT BASE (8)
```

In my MSX the start address is 6912. I will call this SPRITE VARS START. The start address of the four variables for any sprite can be found by using the following calculation:

```
(SPRITE NUMBER * 4) + SPRITE VARS START
```

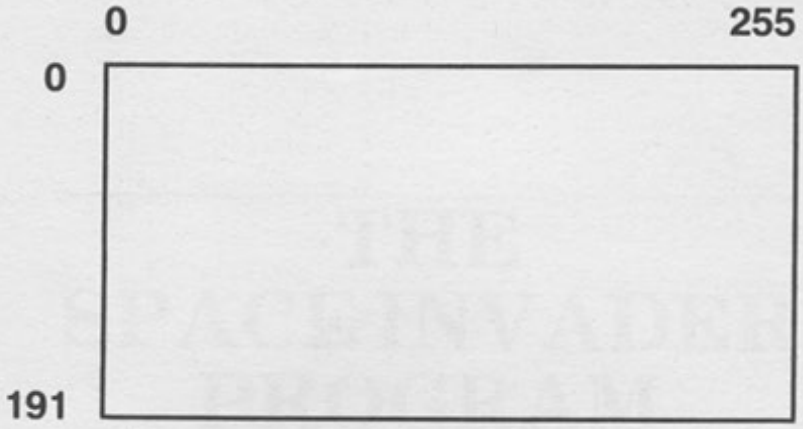
So in my MSX the start of the four variables for sprite 15 is:

```
(15*4) + 6912 = 6972
```

If you enter the four following lines, the sprite we have designed will appear on the screen:

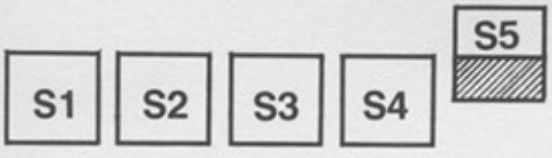
```
VPOKE (SPRITE VARS START + 0), 100
VPOKE (SPRITE VARS START + 1), 50
VPOKE (SPRITE VARS START + 2), 0
VPOKE (SPRITE VARS START + 3), 10
```

The horizontal and vertical variables refer to where the top left-hand corner of the sprite will appear on the screen. For the purpose of positioning sprites, the screen is numbered in the following way:



Try VPOKEing the horizontal and vertical variables with different numbers and you will see the sprite move around. If you poke the vertical variable with a number larger than 191, then the sprite will disappear off the screen. Associated with the sprites is a register; this can be read to see if any sprites have collided. I will show you how this can be used in the next chapter in the Space Invader program.

Earlier on, I mentioned that it is possible to have 32 different sprites on the screen. There is a very important limitation to this which you should bear in mind when writing programs, particularly games programs. If you have more than four sprites on a horizontal line, the fifth and subsequent sprites on that line will disappear. This also applies to sprites which are only partially lined up. The following diagram shows how the bottom half of the fifth sprite cannot be seen:



(This shaded bit of sprite 5 will not be seen.)

The whole subject of sprites is very complex. If you wish to know more about them then I advise you to buy a book which deals specifically with the subject. In the appendix to this book there is a program that will enable you to design shapes of sprites and save them on tape for future use.

THE SPACE INVADER PROGRAM

The Space Invader Program

Before I describe the Space Invader program I would like to mention a very important feature of MSX computers called the BASIC INPUT OPERATING SYSTEM (BIOS). In the ROM are a whole host of Machine Code routines; these can be used to facilitate the writing of Machine Code programs. A full description of the most important routines will be given in a later chapter. In the Space Invader program I use a few of these routines and I will briefly describe them as and when necessary.

I will now describe in great detail how to write a simple Space Invader program. However, the fact that it is simple doesn't mean that the principles involved in writing it cannot be applied to writing a far more complex program. Any program can be described as a set of blocks – and these blocks can then be broken down into a set of routines. At the end of this chapter you should have gained enough knowledge to enable you to start writing your own Machine Code programs.

I will begin by describing the flowchart for the program. If you've ever written a program in BASIC then you will have no difficulty in understanding the flowchart.

The next thing I will discuss is how to form a memory map. You will not have had to do this in BASIC programming, but it is quite important when you first start to develop your own Machine Code programs.

I will then go on to describe each of the flowchart blocks in detail. For example, one of the blocks is labelled "move bullet up screen". I will describe exactly how this is achieved and, where necessary, I will show you a further sub-flowchart of the actual routine.

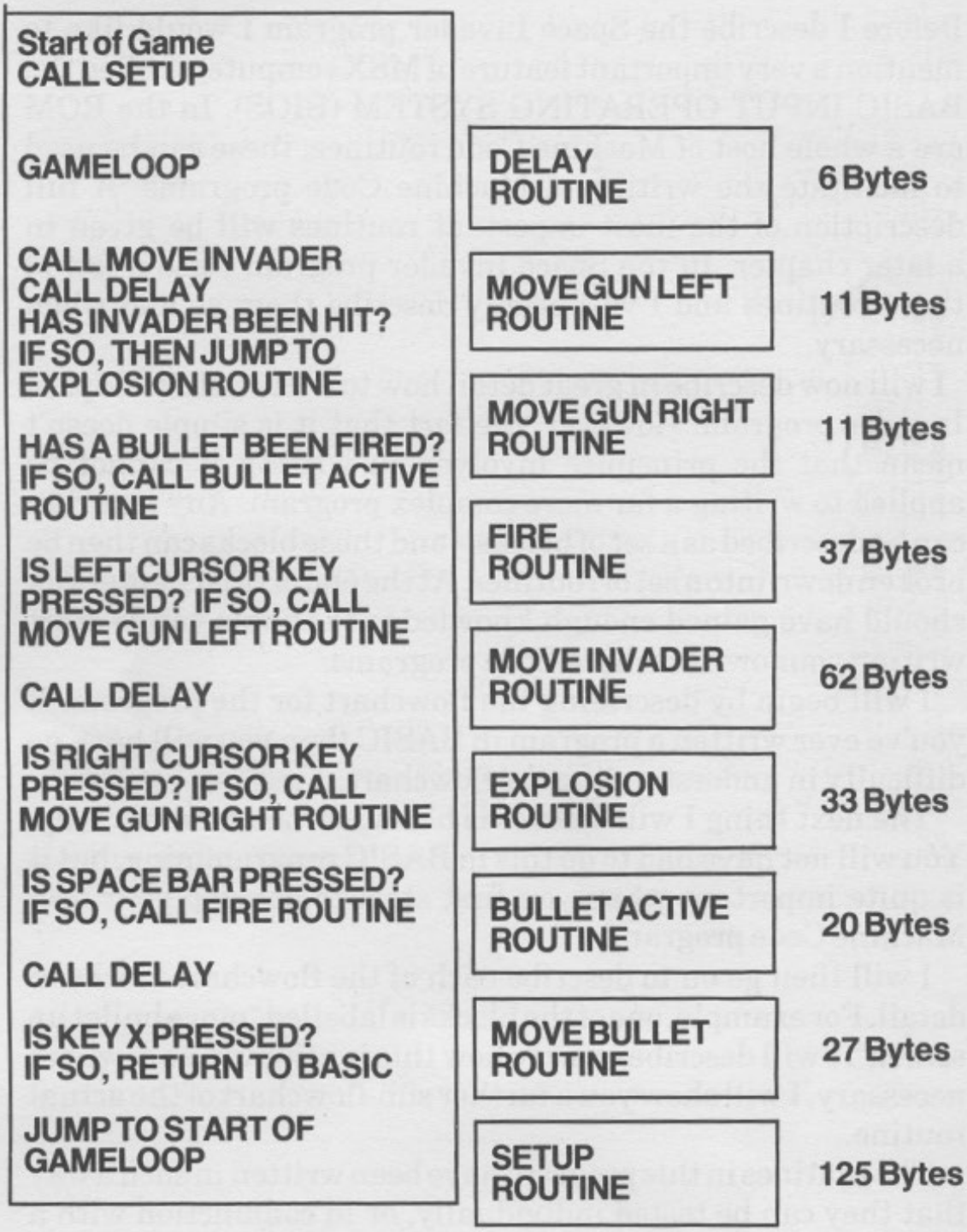
The routines in this program have been written in such a way that they can be tested individually, or in conjunction with a previously tested routine. This will enable you to test the program step by step and see exactly what each routine does.

THE FLOWCHART

When developing a program, one of the first things to do is to draw a flowchart which shows clearly how the program works

and what it is meant to do. The following diagram shows the flowchart of the Space Invader program.

SPACE INVADER FLOWCHART

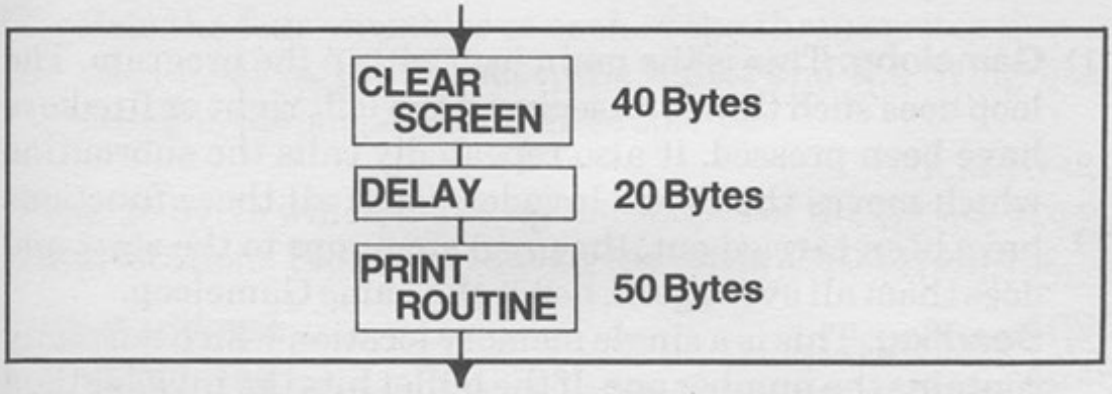


I am sure you will agree that the flowchart is relatively easy to understand. You should study it carefully and get a firm understanding of how the finished program will work.

THE MEMORY MAP

I have already explained that in Machine Code programs we do not have line numbers. So if you need to insert a new instruction you will have to move all the instructions which follow on from it. This then causes huge problems with absolute jumps and calls. Let us suppose that you have a delay routine at location 40000 in the program. To access the delay you would probably have an instruction somewhere else such as CALL 40000. If, for some reason, you needed to add a simple three-byte instruction at the end of the routine which normally ends at location 39999, then you will obviously need to use locations 40000/40001/40002. It will then be necessary to move the delay routine so that it starts at location 40003. Having done that, you will then have to work through your program and change all the CALL 40000 to CALL 40003. Writing a program in this way is tedious and will take a very long time.

These problems can be overcome by using your main flowchart to create a memory map. To form a memory map you need to know approximately how many bytes there are in each routine. You then allocate specific memory locations to each routine, while also leaving a few bytes free between the end of one routine and the start of the next. The free memory spaces will enable you to extend any routine without affecting the following routines. The following flowchart – for a very simple program – demonstrates how to create a memory map:



In the flowchart each of the three routines are marked with the approximate number of bytes they use. The clear screen routine could be put at location 40000, the delay routine at 40050 and the print routine at location 40080. This will leave a few bytes between each routine so that each one can be extended if necessary without the need to reshuffle the whole program.

If you look at the flowchart for the Space Invader program you will see that each routine is marked with the approximate number of bytes it will use. Using this information, I have created the following memory map. The left-hand column shows the name of each routine, and the right-hand column shows its start address.

GAMELOOP	40000
BULACT	40082
EXPLO	40133
FIRE	40157
MOVBUL	40205
MOVINV	40243
MOVLEF	40316
MOVRIG	40338
SETUP	40363
DELAY	40474

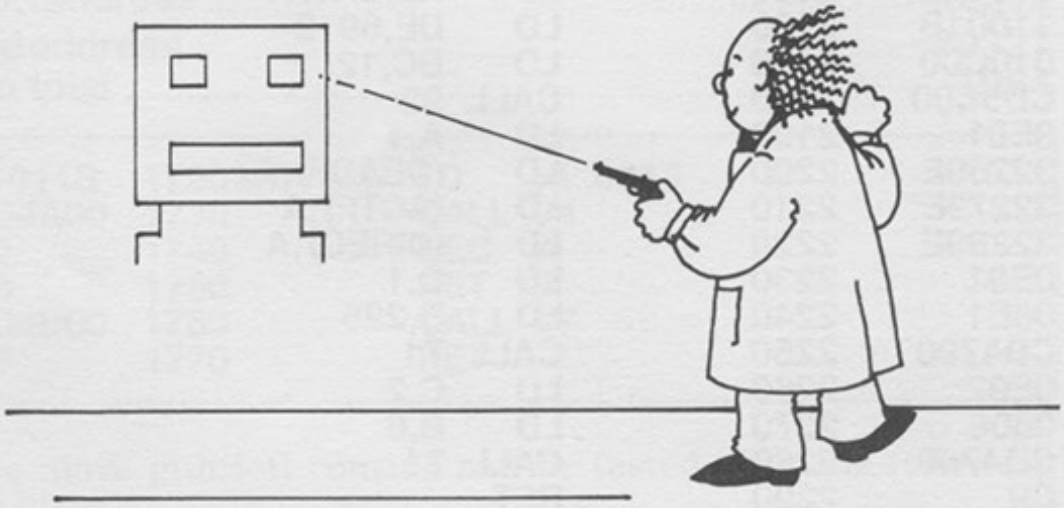
In the memory map I have also assigned an area of memory to variables. (I will explain the purpose of these variables shortly.)

EXPLANATION OF TERMS

I would now like to explain a few terms which will make the overall explanation of the game a lot easier:

1) **Gameloop:** This is the main loop within the program. The loop does such things as seeing if the left, right or fire keys have been pressed. It also repeatedly calls the subroutine which moves the Space Invader. Once all these functions have been carried out, the program jumps to the start and does them all over again, hence the name Gameloop.
2) **Deadflag:** This is a single memory location which normally contains the number one. If the bullet hits the Invader then the memory location is set to zero. The location of the Deadflag is 41000.
3) **Bullet Active Flag:** This is a single memory location which normally contains the number one. When the bullet has been fired this location is set to zero. It is used to prevent a bullet being fired when there is already one on the screen. The location of the Bullet Active Flag is 41001.

4) **Invader Direction Flag:** This is a single memory location which contains the number one if the Invader is moving from right to left, and the number zero if the Invader is moving from left to right. The location of the Invader Direction Flag is 41002.



INITIALISATION ROUTINE

The first routine which we will write is called the initialisation routine. This does the following:

Sets the sprite size to twice normal size.
Sets the SPRITE SHAPE START to 14336.
Sets the SPRITE VARS START to 6912.
Sets the four variables of each of the three sprites that are used (that is, the Invader, the bullet and the firing base).
Designs the shapes of the three sprites.

The 24 bytes which make up the shapes of the sprites are put into locations 14336 to 14359 by the initialisation routine:

Start address **40363**
End address **40463**
Hex total **6910**

```
3E01      2050 SETUP:  LD    A,1
32AFFC    2060         LD    (64687),A
CD5F00    2070         CALL  95
21EC9D    2080         LD    HL,DATA
110038    2090         LD    DE,14336
```

```
011800    2100          LD    BC,24
CD5C00    2110          CALL  92
0E05      2120          LD    C,5
0636      2130          LD    B,54
CD4700    2140          CALL  71
21049E    2150          LD    HL,DATA1
11001B    2160          LD    DE,6912
010C00    2170          LD    BC,12
CD5C00    2180          CALL  92
3E01      2190          LD    A,1
32269E    2200          LD    (DEADF),A
32279E    2210          LD    (ACTF),A
32289E    2220          LD    (DIREC),A
0E01      2230          LD    C,1
06E1      2240          LD    B,225
CD4700    2250          CALL  71
0E02      2260          LD    C,2
0606      2270          LD    B,6
CD4700    2280          CALL  71
C9        2290          RET
1818187E  2300 DATA:    DEFB  24,24,24,126
FFFFFF00  2310          DEFB  255,255,255,0
3C7E99FF  2320          DEFB  60,126,153,255
663C4224  2330          DEFB  102,60,66,36
18181818  2340          DEFB  24,24,24,24
18181818  2350          DEFB  24,24,24,24
AA64000F  2360 DATA1:   DEFB  170,100,0,15
0000010C  2370          DEFB  0,0,1,12
C8000201  2380          DEFB  200,0,2,1
```

The above initialisation routine can be tested with the following few lines of BASIC:

```
1000 DEF USR = 40363
1005 A = USR(1)
1010 GOTO 1010
```

You should now see the Invader printed in the top left-hand corner of the screen and the gun at the bottom of the screen in the middle.

MOVE GUN LEFT ROUTINE

The following routine is called when the correct key has been pressed to move the gun left. The routine first checks to see if the gun is at the extreme left edge of the screen; if it is, then obviously the gun cannot be moved any further and the routine returns to the gameloop. If the gun is not at the extreme left edge then it is repositioned to the left. This is done by finding out

what the gun's current horizontal position is, subtracting one from it, and then putting this new value into the correct memory location in VRAM which corresponds to the horizontal position of the gun.

Start address **40316**
End address **40327**
Hex total **1084**

```
21011B  1720 MOVLEF  LD    HL,6913
CD4A00  1730         CALL  74
3D      1740         DEC   A
C8      1750         RET   Z
CD4D00  1760         CALL  77
C9      1770         RET
```

The move gun left routine can be tested with the following BASIC program:

```
1000 DEF USR = 40363
1005 A = USR(1)
1010 DEF USR = 40316
1015 A = USR(1)
1020 FOR B = 1 to 100
1025 NEXT B
1030 GOTO 1015
```

You should see the gun move as far as it can to the left of the screen.

MOVE GUN RIGHT ROUTINE

The routine for moving the gun right is very similar to that for moving it left. The only difference is that a check is made to see if the gun is at the extreme right of the screen; and, secondly, one is added – not subtracted – to the current horizontal position.

Start address **40338**
End address **40352**
Hex total **1916**

```
21011B  1880 MOVRIG  LD    HL,6913
CD4A00  1890         CALL  74
D6F0    1900         SUB   240
C8      1910         RET   Z
```

```
C6F1    1920    ADD  A,241
CD4D00  1930    CALL 77
C9      1940    RET
```

The move gun right routine can be tested by changing line 1010 in the previous BASIC program to:

```
1010 DEF USR = 40338
```

MOVE INVADER ROUTINE

This routine is the most complex of all. The following list shows what happens each time it is called:

1) If the Invader Direction Flag is set to one then jump to the move Invader right to left routine (6).
2) The Direction Flag must be set to zero so the Invader is currently moving left to right.
3) If the Invader is at the extreme right-hand edge of the screen, set the direction flag to one and return to the gameloop.
4) The Invader is not at the extreme right edge, so it is moved one position to the right. The Invader is moved in a very similar way to the gun; that is, we first get its current horizontal position and, in the case of moving right, we add one to it. This new value is then put into the correct place in VRAM which represents the horizontal position of the Invader sprite.
5) Return to the gameloop.
6) The Invader Direction Flag is set to one so the Invader is currently moving from right to left.
7) If the Invader is at the extreme left of the screen, then the Direction Flag is set to zero and the program returns to the gameloop.
8) The Invader is not at the extreme left edge of the screen, so it is moved one position to the left. This is done by subtracting one from the horizontal position of the Invader sprite.
9) Return to the gameloop.

Start address **40243**
End address **40305**
Hex total **5797**

```
3A289E  1350 MOVINV  LD    A,(DIREC)
3D      1360         DEC   A
CA569D  1370         JP    Z,ILEFT
21051B  1380         LD    HL,6917
CD4A00  1390         CALL  74
D6F0    1400         SUB   240
C24B9D  1410         JP    NZ, IRIG1
3E01    1420         LD    A,1
32289E  1430         LD    (DIREC) ,A
C9      1440         RET
21051B  1450 IRIG1:  LD    HL,6917
CD4A00  1460         CALL  74
3C      1470         INC   A
CD4D00  1480         CALL  77
C9      1490         RET
21051B  1500 ILEFT:  LD    HL,6917
CD4A00  1510         CALL  74
D601    1520         SUB   1
C2679D  1530         JP    NZ,ILEF1
3E00    1540         LD    A,0
32289E  1550         LD    (DIREC) ,A
C9      1560         RET
21051B  1570 ILEF1:  LD    HL,6917
CD4A00  1580         CALL  74
3D      1590         DEC   A
CD4D00  1600         CALL  77
C9      1610         RET
```

The move invader routine can be tested with the following lines:

```
1000 DEF USR = 40363
1005 A = USR (0)
1010 DEF USR = 40243
1015 A = USR (0)
1020 GOTO 1015
```

FIRE BUTTON ROUTINE

This routine is called when the Space Bar is pressed. The first thing to do is to check that there is not already a bullet on its way up the screen. If there is, then the program returns to the gameloop. This check is made by looking at the Bullet Active Flag; if it is zero then a bullet is on the screen somewhere – another one cannot be fired so the program returns to the gameloop. If the Bullet Active Flag is one, then a bullet has not yet been fired and we can continue with the rest of the routine.

The next thing to be done is to set the horizontal and vertical variables of the bullet sprite so that the bullet appears to be coming out of the end of the gun. The horizontal position of the bullet will be the same as the gun. The vertical position of the bullet will be 16 less than the gun which is 154.

The Bullet Active Flag is then set to zero. This serves two purposes. Firstly, another bullet cannot be fired; and, secondly, the gameloop knows that it must constantly call the move bullet routine (as will be described later on).

Start address **40157**
End address **40194**
Hex total **2697**

```
3A279E    870 FIRE:   LD    A,(ACTF)
3C        880         INC   A
3D        890         DEC   A
C8        900         RET   Z
21011B    910         LD    HL,6913
CDCE4A00  920         CALL  74
21091B    930         LD    HL,6921
CD4D00    940         CALL  77
21001B    950         LD    HL,6912
CD4A00    960         CALL  74
D611      970         SUB   17
21081B    980         LD    HL,6920
CD4D00    990         CALL  77
3E00     1000         LD    A,0
32279E   1010         LD    (ACTF),A
C9       1020         RET
```

The fire button routine can be tested with the following lines. When run you should see the bullet appear just above the gun:

```
1000 DEF USR = 40363
1005 A = USR(0)
1010 DEF USR = 40157
1015 A = USR(0)
```

MOVE BULLET ROUTINE

This routine will move the bullet one position up the screen each time it is called. This is done by subtracting one from the current value of the vertical variable of the sprite. The routine also checks to see if the bullet has reached the top of the screen.

If it has, then the Bullet Active Flag is set to one, and the vertical variable of the bullet sprite is set so that the sprite cannot be seen.

Start address **40205**
End address **40232**
Hex total **2400**

```
21081B  1130 MOVBUL  LD    HL,6920
CD4A00  1140         CALL  74
3D      1150         DEC   A
CA1B9D  1160         JP    Z,TOP
CD4D00  1170         CALL  77
C9      1180         RET
3EC8    1190 TOP:    LD    A,200
21081B  1200         LD    HL,6920
CD4D00  1210         CALL  77
3E01    1220         LD    A,1
32279E  1230         LD    (ACTF),A
C9      1240         RET
```

The move bullet routine can be tested with the following lines. You will see the bullet constantly move from the bottom of the screen to the top.

```
1000 DEF USR = 40363
1015 A = USR (0)
1020 DEF USR = 40157
1025 A = USR (0)
1030 DEF USR = 40205
1035 A = USR (0)
1040 GOTO 1035
```

BULLET ACTIVE ROUTINE

The gameloop constantly checks to see if the Bullet Active Flag is set to zero; if it is then the following routine is called. The first thing this routine does is to call the MOVE BULLET ROUTINE four times. This means that the bullet can travel at a fast speed. It then checks to see if the bullet has collided with the Invader. This is done by checking a register in VRAM known as the Collision Flag. This Collision Flag is set to one if any of the sprites collide, so that whenever the Collision Flag is set to a zero we know the bullet has hit the Invader. If the Collision Flag

is set to a zero then the DEADFLAG is set to zero; this indicates to the gameloop that the Invader has been hit and that an explosion should occur. (I will describe this later.)

Start address **40082**
End address **40102**
Hex total **2416**

```
CD0D9D  420 BULACT  CALL  MOVBUL
CD0D9D  430         CALL  MOVBUL
CD0D9D  440         CALL  MOVBUL
CD3E01  450         CALL  318
CB6F    460         BIT   5,A
C8      470         RET   Z
3E00    480         LD    A,0
32269E  490         LD    (DEADF),A
C9      500         RET
```

EXPLOSION ROUTINE

The gameloop constantly checks the DEADFLAG; if it is set to zero then the program jumps to the explosion routine. The way in which the explosion is caused is simple but dramatic. Firstly, the screen is set to mode 3 – the multicolour mode. The memory map is then filled with random numbers which give a very colourful display. This is repeated several times, causing a spectacular explosion effect.

Start address **40113**
End address **40146**
Hex total **3587**

```
3E03    610 EXPLO:  LD    A,3
32AFFC  620         LD    (64687), A
CD5F00  630         CALL  95
3E64    640         LD    A,100
210000  650         LD    HL,0
E5      660 EXPLO1  PUSH  HL
F5      670         PUSH  AF
110008  680         LD    DE,2048
01E803  690         LD    BC,1000
CD5C00  700         CALL  92
F1      710         POP   AF
```

```
E1        720        POP   HL
24        730        INC   H
3D        740        DEC   A
C2BE9C    750        JP    NZ,EXPLO1
C3409C    760        JP    START
```

DELAY ROUTINE

As can be seen from the flowchart, a delay routine is included. This is required to slow down the whole program so that you can actually see what is happening. The delay loop works by loading register A with a value of 255 and repeatedly decrementing it until zero is reached. When register A finally reaches zero, the program returns to the gameloop. It is not practical to test this routine, since even with a maximum possible delay value of 255 it is still too fast for the average human to detect. At the end of this chapter I will demonstrate how you can prove that the delay is actually working.

Start address **40474**
End address **40480**
Hex total **959**

```
3EFF      2490 DELAY:  LD    A,255
3D        2500 DEL:    DEC   A
C21C9E    2510         JP    NZ,DEL
C9        2520         RET
```

The explosion routine can be tested with the following lines:

```
1000 POKE 40000,201
1005 DEF USR = 40113
1010 A = USR (0)
```

THE GAMELOOP

I will now describe the gameloop in great detail. Firstly, here is a list of the things which the gameloop does:

1) Call MOVE INVADER ROUTINE.
2) See if the DEADFLAG is set to zero, if so jump to the explosion routine.

3) See if the BULLET ACTIVE FLAG is set to zero, if so call the bullet active routine.
4) If the left-hand cursor key is being pressed then call the move gun left routine.
5) If the right-hand cursor key is being pressed then call the move gun right routine.
6) If the space bar is pressed then call the FIRE BUTTON ROUTINE.
7) If key X is being pressed then return to BASIC.

The gameloop also calls the DELAY ROUTINE several times. This is to slow down the whole program. Without this delay things would move so fast that you would hardly be able to see them.

The calls to the various routines do not need any explanation. I will now explain how it is possible to detect which key is being pressed. The program does not do the equivalent of asking "Which key is being pressed?" – but it does ask "Is a specific key being pressed?" In the BIOS there is a routine we can use to see if a specific key is being pressed. In order for the routine to work we have to provide it with a small piece of information before calling it. Look at the following diagram; it shows 9 rows numbered from 0 to 8 which each contain 8 keys:

	7	6	5	4	3	2	1	0
0	7	6	5	4	3	2	1	0
1	;	]	[	\	=	–	9	8
2	B	A	£	/	.	`	'	`
3	J	I	H	G	F	E	D	C
4	R	Q	P	O	N	M	L	K
5	Z	Y	X	W	V	U	T	S
6	F3	F2	F1	CODE	CAP	GRAPH	CTRL	SHIFT
7	RETURN	SEL	BACK SPACE	STOP	TAB	ESC	F5	F4
8	RIGHT CURSOR	DOWN CURSOR	UP CURSOR	LEFT CURSOR	DEL	INS	HOME	SPACE

Let's suppose we wish to detect the left-hand cursor key, then this is the procedure for doing so:

Load register A with 8 (8 is the number of the row in the above diagram which contains the left-hand cursor key.)
Call 321d (This is the location of the routine which will read the row you have specified.)

The routine will now put a value into register A relating to which of the 8 keys in the specified row you pressed. If you look at the above diagram you will see that each of the 8 columns is numbered from 0 to 7. The left-hand cursor key is in the column marked 4. To test if this key has been pressed we use the following instruction:

BIT 4,A

The next instruction will be:

CALL Z, MOVE GUN LEFT ROUTINE

Thus if the result of the instruction BIT 4,A was zero (that is, the key was pressed) then call the MOVE GUN LEFT ROUTINE. The Machine Code for the gameloop is shown below.

Start address **40000**
End address **40071**
Hex total **8681**

```
CDAB9D   20 START:   CALL  SETUP
CD339D   30 LOOP:    CALL  MOVINV
CD1A9E   40          CALL  DELAY
3A269E   50          LD    A, (DEADF)
3C       60          INC   A
3D       70          DEC   A
CAB19C   80          JP    Z,EXPLO
3A279E   90          LD    A,(ACTF)
3C      100          INC   A
3D      110          DEC   A
CC929C  120          CALL  Z,BULACT
3E08    130          LD    A,8
CD4101  140          CALL  321
CB67    150          BIT   4,A
CC7C9D  160          CALL  Z,MOVLEF
CD1A9E  170          CALL  DELAY
```

```
3E08      180     LD      A,8
CD4101    190     CALL    321
CB7F      200     BIT     7,A
CC929D    210     CALL    Z,MOVRIG
3E08      220     LD      A,8
CD4101    230     CALL    321
CB47      240     BIT     0,A
CCDD9C    250     CALL    Z,FIRE
CD1A9E    260     CALL    DELAY
3E05      270     LD      A,5
CD4101    280     CALL    321
CB6F      290     BIT     5,A
C8        300     RET     Z
C3439C    310     JP      LOOP
```

The whole game can now be tested by entering the following lines:

```
1000 DEF USR = 40000
1005 A = USR(0)
```

If you wish to see what the game would be like without a delay then type in the following line, then run the above program:

```
POKE 40475,1
```

Location 40475 should normally be 255; it represents the length of the delay.

ADVANCED MSX FACILITIES

Advanced MSX Facilities

In this chapter I will deal with some of the subjects I felt weren't really necessary to know about while learning Machine Code. In such a broadly-based book as this, it is impossible to cover each subject in great detail; if you wish to know more about a particular subject you will need to purchase a specialised book. In many cases it isn't necessary to understand something before you can use it. A favourite catchphrase of mine is "If it works, use it"; a large number of people drive cars, for instance, without the slightest idea of how they work. This is the theme of this chapter. I will discuss some of the advanced features of the MSX and how to use them, but I will not describe in great detail how they work.

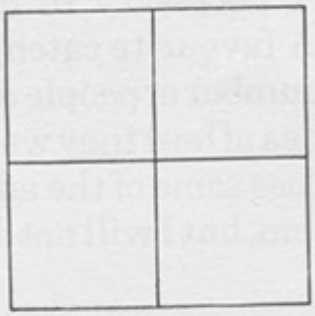
THE FOUR SCREEN MODES

The screen mode we have used throughout this book is MODE 1, and this is probably the most useful mode to use for arcade-type games.

In SCREEN MODE 0 you can have 24 rows, each containing 40 characters. Type SCREEN 0 to see this mode. In this mode it is only possible to have two colours, the INK and PAPER colour. The border colour will take on the same colour as the paper. This mode is of most use for text programs such as word processing applications. It is not possible to have sprites in MODE 0.

SCREEN MODE 2 is very similar to MODE 1. Up to 768 character shapes can be displayed on the screen at any one time. Moreover, each of the eight bytes, which make up a character, can be a different combination of any two colours.

SCREEN MODE 3 is referred to as multicolour mode and is extremely complex. As in MODE 1, the screen is divided into 24 rows containing 32 characters, giving 768 character positions. Instead of printing characters within these positions we actually print colours. Each character position is divided as in the following diagram:



Each of the quarters can be any one of the 16 possible colours.

SPRITES

In an earlier chapter, one of the sprite modes was covered in detail. I will now show you the other three modes. Firstly we will put an ordinary-size sprite on the screen, by typing in the following:

```
SCREEN 1
VDP(1) = 224
VPOKE(SPRITE VARS START + 0),100
VPOKE(SPRITE VARS START + 1),50
VPOKE(SPRITE VARS START + 2),0
VPOKE(SPRITE VARS START + 3),10
VPOKE(SPRITE SHAPE START + 0),255
VPOKE(SPRITE SHAPE START + 1),129
VPOKE(SPRITE SHAPE START + 2),129
VPOKE(SPRITE SHAPE START + 3),129
VPOKE(SPRITE SHAPE START + 4),129
VPOKE(SPRITE SHAPE START + 5),129
VPOKE(SPRITE SHAPE START + 6),129
VPOKE(SPRITE SHAPE START + 7),255
```

You should now have a box-shaped sprite in the middle of the screen. Now type in:

```
VDP(1) = 225
```

VIDEO RAM (VRAM)

The VRAM contains 16384 bytes which can be configured in several different ways. When writing a game, it is up to you to decide how to allocate VRAM. The chip which controls the VRAM has 8 registers, numbered from 0 to 7, called VDPs. The value of each VDP determines how the VRAM will be configured. Setting the values of the VDPs to random numbers could result in the machine locking up and you will then have to switch off the machine. So, do not set the VDPs to values which have not been carefully calculated and checked.

The first VDP, VDP 0, is not of any interest to us and you are advised not to alter its value.

The main use of VDP 1 is to control the type and size of the sprites. If VDP 1 is given a value of 224, then each of the 32 sprites will require 8 bytes to define its shape – each sprite will appear on the screen as a normal-sized character. When VDP 1 is given a value of 225 then the 8 bytes which make up its shape will still be printed on the screen – but twice the size of the previous mode. If VDP 1 has a value of 226 then it uses 32 bytes to define its shape. The sprite will appear on the screen as four normal-sized characters stuck together in a square. When VDP 1 has a value of 227 – as opposed to a value of 226 – then the sprites are doubled in size.

The value of VDP 2 determines the start address of the screen map. There are 16 possible positions for the screen map, as shown in the following chart:

VDP 2	Start address of the screen map
0	0
1	1024
2	2048
3	3072
4	4096
5	5120
6	6144
7	7168
8	8192
9	9216
10	10240
11	11264
12	12288
13	13312
14	14336
15	15360

The value of VDP 3 determines the start address of the colour table. VDP 3 can be set to any value between 0 and 255. The colour table will start at the address of VDP 3 multiplied by 64. So if VDP 3 has a value of 21, the colour table will start at 21 x 64 = 1344.

The value of VDP 4 determines the start address of the character shapes. It can only be set to one of the following eight possibilities:

VDP 4	Start address of the character shapes
0	0
1	2048
2	4096
3	6144
4	8192
5	10240
6	12288
7	14336

The value of VDP 5 determines the start address of the sprite variables. Its value can be set to anything between 0 and 127 inclusive. The start address of the sprite variables is then calculated by multiplying the value of VDP 5 by 128. So if VDP 5 has a value of 45, the sprite variables will start at 45 x 128 = 5760.

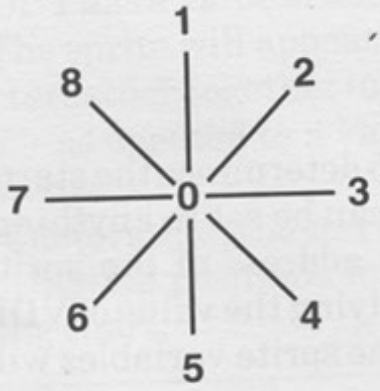
The value of VDP 6 determines the start address of the sprite shapes. It can be set to any one of the eight values shown in the following chart:

VDP 6	Start address of the sprite shapes
0	0
1	2048
2	4096
3	6144
4	8192
5	10240
6	12288
7	14336

VDP 7 is not of much interest to the Machine Code programmer. Its value will determine what colour the ink and paper will be set to in SCREEN MODE 0 when a CLS command is issued from BASIC.

JOYSTICKS

All MSX machines have the facility to plug in one or two joysticks. There are two routines in the ROM which can be used to determine the condition of each joystick. The first routine deals with finding out the position of the specified joystick. To use this routine you must first load register A with either 1 or 2 – which is the joystick that you wish to read – then you CALL the routine which is at location 213. The routine will calculate the position of the joystick, then put a value into register A that corresponds to this value. The following diagram shows the values which represent the eight directions of the joystick. If the joystick is in a central position, then register A will contain 0:



Here is a small Machine Code routine to demonstrate the reading of the joystick. First register A is loaded with 1 (the number of the joystick that we will be reading). Then the routine at location 213 is called. The value of register A is then put into location 50000:

Start address **40000**
End address **40008**
Hex total **1007**

```
3E01
CDD500
3250C3
C9
```

The following BASIC program can be used to test the above routine:

```
1000 SCREEN 1
1005 DEF USR = 40000
1010 A = USR (0)
1015 B = PEEK (50000)
1020 PRINT B
1025 GOTO 1010
```

Now run the above program. Printed on the screen will be a value that corresponds to the position of joystick one.

The second routine associated with the joysticks is used to see if the fire button has been pressed. The routine is at location 216 and, as with the previous routine, register A must be loaded with either 1 or 2 according to which joystick you want to read. The routine will then test the appropriate fire button; if it is pressed then it will load register A with 255, if the button is not pressed then register A will be loaded with 0. The following Machine Code routine will demonstrate the fire button detection routine. (Do not type it in unless you actually have a joystick, because the only way to exit the routine and return to BASIC is by pressing the fire button.)

Start address **40000**
End address **40008**
Hex total **1025**

```
3E01
CDD800
3C
20F8
C9
```

Enter and run the following BASIC program to test the above routine:

```
1000 DEF USR = 40000
1005 A = USR (0)
1010 CLS
1015 PRINT "FIRE BUTTON HAS BEEN PRESSED"
```

BASIC INPUT OPERATING SYSTEM (BIOS)

The BIOS was briefly described in the chapter dealing with the Space Invader program. The BIOS contains many Machine

Code routines which we can use, thus making the writing of our own Machine Code routines a great deal easier. The routines I have listed here are the ones that I consider to be important to know about while learning Machine Code programming.

Name: Fill VRAM

Function: When called, this routine will fill a specified area of VRAM with a particular piece of data. For instance it could be used to fill the 768 bytes of the screen map in SCREEN MODE 1 with character 32, which would have the effect of clearing the screen.

Requirements: The start address of the block of VRAM to be filled must be held in register HL. The length of the block must be held in BC and the data, 32 in the above example, must be held in register A.

CALL: CD 56 00

Name: Move block of VRAM to ordinary memory.

Function: This routine will move a specified block of VRAM into ordinary user memory. It could be used for copying the screen memory map into RAM.

Requirements: Register HL must hold the start address of the block of VRAM to be moved. Register DE must hold the start address in RAM of where the block is being moved to, and BC must hold the length of the block.

CALL: CD 59 00

Name: Move block of memory into VRAM.

Function: This routine is used to copy a specified area of memory into VRAM. It could be used to copy a newly-designed set of characters, which are somewhere in RAM, into their normal position in VRAM.

Requirements: Register HL must hold the start address of the block of memory in RAM to be moved. Register DE must hold the start address of where the block of memory must be moved to

in VRAM. Register BC must hold the length of the block.

CALL: CD 5C 00

Name: Get character.

Function: When this routine is called it will wait until a key has been pressed and then return with the character code in register A. A typical example is when you want someone to input some information such as their name.

CALL: CD 9F 00

Name: Position cursor.

Function: This routine will position the cursor at a specified position on the screen. It could be used to indicate the start of where a person should begin to answer a question you have printed on the screen.

Requirements: Register H must hold the column number in which you wish to position the cursor. In SCREEN MODE 1 this would be in the range 0 to 31. Register L must hold the row in which you wish to position the cursor.

CALL: CD C6 00

Name: Erase function key display.

Function: When this routine is called it will cause the function key display at the bottom of the screen to be turned off.

CALL: CD CC 00

Name: Turn on function key display.

Function: When this routine is called it will cause the function key display at the bottom of the screen to be turned on.

CALL: CD CF 00

Name: Write to VRAM.

Function: This routine is the Machine Code equivalent of the BASIC instruction VPOKE (VRAM address), n – where VRAM address is a

	memory address between 0 and 16383 in VRAM and n is a number between 0 and 255 which you wish to put into that location.
Requirements:	Register HL must hold the address in VRAM that you want to write to. Register A must contain a number between 0 and 255.
CALL:	CD 4D 00

Name:	Read VRAM.
Function:	This routine is the Machine Code equivalent of the BASIC instruction VPEEK (VRAM address). After this routine has been called, register A will contain the value of whatever was in the specified VRAM address.
Requirements:	Register HL must hold the address of the VRAM address that you wish to read.
CALL:	CD 4A 00

A FEW USEFUL ROUTINES

A Few Useful Routines

This chapter contains a few routines for you to type in. They effectively demonstrate the power of Machine Code programming. Each routine is located at a different position in memory; it is therefore possible, if you wish, to have more than one routine in memory at the same time. A book titled *Useful Utilities for Your MSX* is also available from Virgin Books. It contains many useful routines which you can use if you do not feel like writing your own.

Name: Scroll a line of text to the right.

Function: This routine will enable you to scroll any of the 24 rows of text in SCREEN MODE 1 to the right.

Requirements: Memory location 50000 must hold a number between 0 and 23 which represents the row that you wish to scroll.

Start address **40000**
End address **40044**
Hex total **4158**

```
3A50C3   20          LD    A,(50000)
012000   30          LD    BC,32
211F18   40          LD    HL,6175
3C       50          INC   A
3D       60 LOOP:    DEC   A
CA529C   70          JP    Z,LOOP1
09       80          ADD   HL,BC
C34A9C   90          JP    LOOP
CD4A00  100 LOOP1:   CALL  74
F5      110          PUSH  AF
011F00  120          LD    BC,31
2B      130 LOOP2:   DEC   HL
CD4A00  140          CALL  74
23      150          INC   HL
CD4D00  160          CALL  77
2B      170          DEC   HL
0B      180          DEC   BC
79      190          LD    A,C
B0      200          OR    B
C2599C  210          JP    NZ,LOOP2
F1      220          POP   AF
CD4D00  230          CALL  77
C9      240          RET
```

Name: Scroll a line of text to the left.

Function: This routine will enable you to scroll any of the 24 rows of text in SCREEN MODE 1 to the left.

Requirements: Memory location 50001 must contain a number between 0 and 23 which represents the row that you wish to scroll.

Start address **40100**
End address **40144**
Hex total **4420**

```
3A51C3   20          LD    A,(50001)
012000   30          LD    BC,32
210018   40          LD    HL,6144
3C       50          INC   A
3D       60 LOOP:    DEC   A
CAB69C   70          JP    Z,LOOP1
09       80          ADD   HL,BC
C3AE9C   90          JP    LOOP
CD4A00   100 LOOP1:  CALL  74
F5       110         PUSH  AF
011F00   120         LD    BC,31
23       130 LOOP2:  INC   HL
CD4A00   140         CALL  74
2B       150         DEC   HL
CD4D00   160         CALL  77
23       170         INC   HL
0B       180         DEC   BC
79       190         LD    A,C
B0       200         OR    B
C2BD9C   210         JP    NZ,LOOP2
F1       220         POP   AF
CD4D00   230         CALL  77
C9       240         RET
```

Name: Scroll a column of text in SCREEN MODE 1 upwards.

Function: This routine will enable you to scroll any of the 32 columns of text in SCREEN MODE 1 up.

Requirements: Memory location 50002 must contain a number between 0 and 31 which is the number of the column that you wish to scroll upwards.

Start address **40300**
End address **40345**
Hex total **4540**

```
3A52C3   20          LD    A,(50002)
3C       30          INC   A
210018   40          LD    HL,6144
3D       50 LOOP:    DEC   A
CA7B9D   60          JP    Z,LOOP1
23       70          INC   HL
C3739D   80          JP    LOOP
CD4A00   90 LOOP1:   CALL  74
F5      100          PUSH  AF
011700  110          LD    BC,23
112000  120          LD    DE,32
19      130 LOOP2:   ADD   HL,DE
CD4A00  140          CALL  74
ED52    150          SBC   HL,DE
CD4D00  160          CALL  77
19      170          ADD   HL,DE
0B      180          DEC   BC
79      190          LD    A,C
B0      200          OR    B
C2859D  210          JP    NZ,LOOP2
F1      220          POP   AF
CD4D00  225          CALL  77
C9      230          RET
```

Name: Scroll a column of text down.

Function: This routine will allow you to scroll any of the 32 columns in SCREEN MODE 1 downwards.

Requirements: Memory location 50003 must contain a number between 0 and 31 which represents the number of the column that you wish to scroll downwards.

Start address **40200**
End address **40246**
Hex total **4761**

```
3A53C3   20          LD    A,(50003)
3C       30          INC   A
21E01A   40          LD    HL,6880
3D       50 LOOP:    DEC   A
CA179D   60          JP    Z,LOOP1
```

```
23          70              INC   HL
C30F9D      80              JP    LOOP
CD4A00      90 LOOP1:       CALL  74
F5         100              PUSH  AF
011700     110              LD    BC,23
112000     120              LD    DE,32
ED52       130 LOOP2:       SBC   HL,DE
CD4A00     140              CALL  74
19         150              ADD   HL,DE
CD4D00     160              CALL  77
ED52       170              SBC   HL,DE
0B         180              DEC   BC
79         190              LD    A,C
B0         200              OR    B
C2219D     210              JP    NZ,LOOP2
F1         220              POP   AF
CD4D00     230              CALL  77
C9         240              RET
```

```
1000 SCREEN 1
1005 VDP(2)=6
1010 X=65
1015 FOR A=6144 TO 6880 STEP 32
1020 Y=X
1025 FOR B=0 TO 31
1030 VPOKE (A+B),Y
1035 Y=Y+1
1040 NEXT B
1045 X=X+1
1050 NEXT A
1055 POKE (50000),10
1060 DEF USR=40000
1065 A=USR(1)
1070 GOTO 1065
```

The above BASIC program can be used to test any of the four scrolling routines. You will need to alter lines 1055 and 1060 depending on which scrolling routine you are testing and which column or row is to be scrolled.

Name: Laser gun sound effect.

Function: When called, this routine will make a sound which is typical of the sound associated with arcade Space Invader machines.

Start address **40600**
End address **40667**
Hex total **7249**

```
3E08      20            LD    A,8
1E0F      30            LD    E,15
CD9300    40            CALL  147
3E07      50            LD    A,7
1EFE      60            LD    E,254
CD9300    70            CALL  147
3E00      80            LD    A,0
1E6E      90            LD    E,110
CD9300    100           CALL  147
3E01      110           LD    A,1
1E00      120           LD    E,0
CD9300    130           CALL  147
1E6E      140           LD    E,110
3E00      150 LOOP:     LD    A,0
CD9300    160           CALL  147
3EC8      170           LD    A,200
F5        180 DELAY:    PUSH  AF
3E0A      190           LD    A,10
3D        200 DEL:      DEC   A
C2C09E    210           JP    NZ,DEL
F1        220           POP   AF
3D        230           DEC   A
C2BD9E    240           JP    NZ,DELAY
7B        250           LD    A,E
C606      260           ADD   A,6
CAD49E    270           JP    Z,END
3D        280           DEC   A
5F        290           LD    E,A
C3B69E    300           JP    LOOP
3E07      310 END:      LD    A,7
1EFF      320           LD    E,255
CD9300    330           CALL  147
C9        340           RET
```

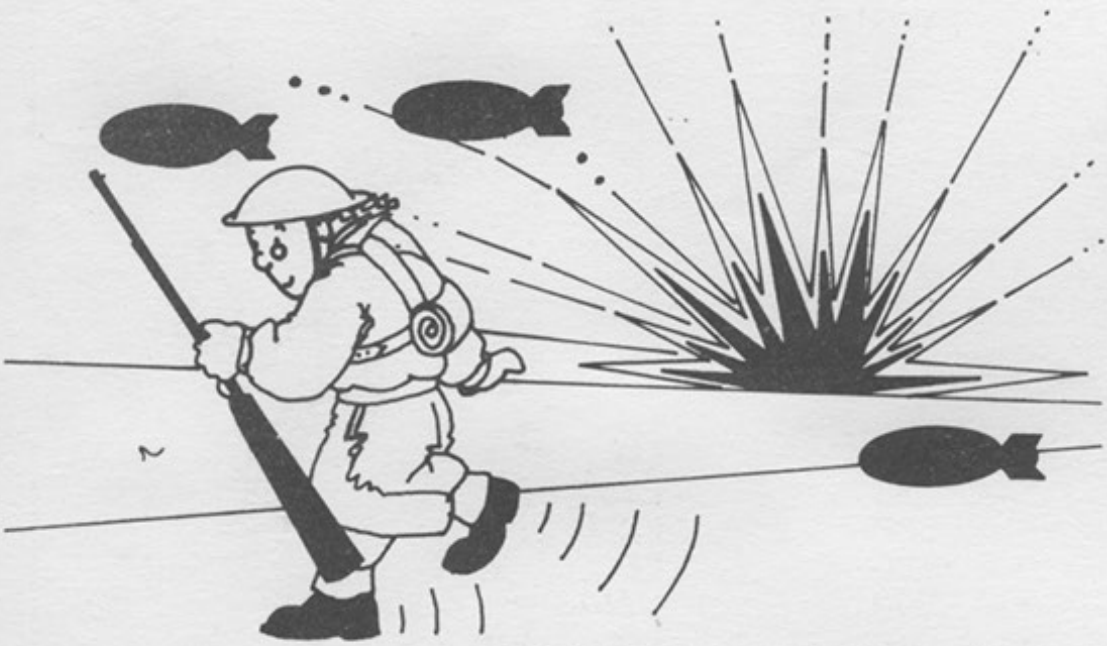
Name: Dropping bomb sound effect.

Function: This routine will produce the whistling sound that a falling bomb makes and then produce an explosion effect.

Start address **40400**
End address **40514**
Hex total **13909**

```
3E07      20          LD    A,7
1EFE      30          LD    E,254
CD9300    40          CALL  147
3E08      50          LD    A,8
1E0F      60          LD    E,15
CD9300    70          CALL  147
1E28      80          LD    E,40
3E00      90 LOOP:    LD    A,0
CD9300   100          CALL  147
3E0A     110          LD    A,10
F5       120 DELAY:   PUSH  AF
3EFF     130          LD    A,255
3D       140 DEL:     DEC   A
C2EA9D   150          JP    NZ,DEL
F1       160          POP   AF
3D       170          DEC   A
C2E79D   180          JP    NZ,DELAY
7B       190          LD    A,E
D696     200          SUB   150
CAFF9D   210          JP    Z,NEXT
C697     220          ADD   A,151
5F       230          LD    E,A
C3E09D   240          JP    LOOP
3E00     250 NEXT:    LD    A,0
1E00     260          LD    E,0
CD9300   270          CALL  147
3E07     280          LD    A,7
1EF7     290          LD    E,247
CD9300   300          CALL  147
3E00     310          LD    A,0
F5       320 NEXT1:   PUSH  AF
5F       330          LD    E,A
CD9300   340          CALL  147
3E32     350          LD    A,50
F5       360 STEVE:   PUSH  AF
3EFF     370          LD    A,255
3D       380 TRACEY:  DEC   A
```

```
C2199E   390          JP    NZ,TRACEY
F1       400          POP   AF
3D       410          DEC   A
C2169E   420          JP    NZ,STEVE
F1       430          POP   AF
D61F     440          SUB   31
CA2D9E   450          JP    Z,LDEL
C620     460          ADD   A,32
C30F9E   470          JP    NEXT1
3E64     480 LDEL:    LD    A,100
F5       490 LDEL1:   PUSH  AF
3EFF     500          LD    A,255
3D       510 LDEL2:   DEC   A
C2329E   520          JP    NZ,LDEL2
F1       530          POP   AF
3D       540          DEC   A
C22F9E   550          JP    NZ,LDEL1
3E07     560          LD    A,7
1EFF     570          LD    E,255
CD9300   580          CALL  147
C9       590          RET
```



APPENDICES

Appendices

1) Z80 OPCODES
2) DECIMAL TO HEX CONVERSION CHART
3) BINARY
4) CHARACTER/SPRITE DESIGNER

Opcodes

ADC A, (HL)	8E
ADC A, (IX + d)	DD8Ed
ADC A, (IY + d)	FD8Ed
ADC A, A	8F
ADC A, B	88
ADC A, C	89
ADC A, D	8A
ADC A, E	8B
ADC A, H	8C
ADC A, L	8D
ADC A, n	CEn
ADC HL, BC	ED4A
ADC HL, DE	ED5A
ADC HL, HL	ED6A
ADC HL, SP	ED7A
ADD A, (HL)	86
ADD A, (IX + d)	DD86d
ADD A, (IY + d)	FD86d
ADD A, A	87
ADD A, B	80
ADD A, C	81
ADD A, D	82
ADD A, E	83
ADD A, H	84
ADD A, L	85
ADD A, n	C6n
ADD HL, BC	09
ADD HL, DE	19
ADD HL, HL	29
ADD HL, SP	39
ADD IX, BC	DD09
ADD IX, DE	DD19
ADD IX, IX	DD29
ADD IX, SP	DD39
ADD IY, BC	FD09
ADD IY, DE	FD19
ADD IY, IY	FD29
ADD IY, SP	FD39
AND(HL)	A6
AND (IX + d)	DDA6d
AND (IY + d)	FDA6d
AND A	A7
AND B	A0
AND C	A1
AND D	A2
AND E	A3
AND H	A4
AND L	A5
AND n	E6n
BIT 0, (HL)	CB46
BIT 0, (IX + D)	DDCBd46
BIT 0, (IY + d)	FDCBd46
BIT 0, A	CB47
BIT 0, B	CB40
BIT 0, C	CB41
BIT 0, D	CB42
BIT 0, E	CB43
BIT 0, H	CB44
BIT 0, L	CB45
BIT 1, (HL)	CB4E
BIT 1, (IX + d)	DDCBd4E
BIT 1, (IY + d)	FDCBd4E
BIT 1, A	CB4F
BIT 1, B	CB48
BIT 1, C	CB49
BIT 1, D	CB4A
BIT 1, E	CB4B
BIT 1, H	CB4C
BIT 1, L	CB4D
BIT 2, (HL)	CB56
BIT 2, (IX + d)	DDCBd56
BIT 2, (IY + d)	FDCBd56
BIT 2, A	CB57
BIT 2, B	CB50
BIT 2, C	CB51
BIT 2, D	CB52
BIT 2, E	CB53
BIT 2, H	CB54
BIT 2, L	CB55
BIT 3, (HL)	CB5E
BIT 3, (IX + d)	DDCBd5E
BIT 3, (IY + d)	FDCBd5E
BIT 3, A	CB5F
BIT 3, B	CB58
BIT 3, C	CB59
BIT 3, D	CB5A
BIT 3, E	CB5B
BIT 3, H	CB5C
BIT 3, L	CB5D
BIT 4, (HL)	CB66
BIT 4, (IX + d)	DDCBd66
BIT 4, (IY + d)	FDCBd66
BIT 4, A	CB67
BIT 4, B	CB60
BIT 4, C	CB61
BIT 4, D	CB62
BIT 4, E	CB63
BIT 4, H	CB64
BIT 4, L	CB65
BIT 5, (HL)	CB6E
BIT 5, (IX + d)	DDCBd6E
BIT 5, (IY + D)	FDCBd6E
BIT 5, A	CB6F
BIT 5, B	CB68
BIT 5, C	CB69
BIT 5, D	CB6A
BIT 5, E	CB6B
BIT 5, H	CB6C

BIT 5, L	CB6D
BIT 6, (HL)	CB76
BIT 6, (IX + d)	DDCBd76
BIT 6, (IY + d)	FDCBd76
BIT 6, A	C877
BIT 6, B	CB70
BIT 6, C	CB71
BIT 6, D	CB72
BIT 6, E	CB73
BIT 6, H	CB74
BIT 6, L	CB75
BIT 7, (HL)	CB7E
BIT 7, (IX + d)	DDCBd7E
BIT 7, (IY + d)	FDCBd7E
BIT 7, A	CB7F
BIT 7, B	CB78
BIT 7, C	CB79
BIT 7, D	CB7A
BIT 7, E	CB7B
BIT 7, H	CB7C
BIT 7, L	CB7D
CALL C, nn	DCnn
CALL M, nn	FCnn
CALL NC, nn	D4nn
CALL nn	CDnn
CALL NZ, nn	C4nn
CALL P, nn	F4nn
CALL PE, nn	ECnn
CALL PO, nn	E4nn
CALL Z, nn	CCnn
CCF	3F
CP (HL)	BE
CP (IX + d)	DDBEd
CP (IY + d)	FDBEd
CP A	BF
CP B	B8
CP C	B9
CP D	BA
CP E	BB
CP H	BC
CP L	BD
CP n	FEn
CPD	EDA9
CPDR	EDB9
CPI	EDA1
CPIR	EDB1
CPL	2F
DAA	27
DEC (HL)	35
DEC (IX + d)	DD35d
DEC (IY + d)	FD35d
DEC A	3D
DEC B	05
DEC BC	0B
DEC C	0D
DEC D	15
DEC DE	1B
DEC E	1D
DEC H	25
DEC HL	2B
DEC IX	DD2B
DEC IY	FD2B
DEC L	2D
DEC SP	3B
DI	F3
DJNZ, d	10d
E1	FB
EX (SP), HL	E3
EX (SP), IX	DDE3
EX (SP), IY	FDE3
EX AF, AF	08
EX DE, HL	EB
EXX	D9
HALT	76
IM 0	ED46
IM 1	ED56
IM 2	ED5E
IN A, (C)	ED78
IN A, (n)	DBn
IN B, (C)	ED40
IN C, (C)	ED48
IN D, (C)	ED50
IN E, (C)	ED58
IN H, (C)	ED60
IN L, (C)	ED68
INC (HL)	34
INC (IX + d)	DD34d
INC (IY + d)	FD34d
INC A	3C
INC B	04
INC BC	03
INC C	0C
INC D	14
INC DE	13
INC E	1C
INC H	24
INC HL	23
INC IX	DD23
INC IY	FD23
INC L	2C
INC SP	33
IND	EDAA
INDR	EDBA
INI	EDA2
INIR	EDB2
JP (HL)	E9
JP (IX)	DDE9
JP (IY)	FDE9
JP C, nn	DAnn
JP M, nn	FAnn
JP NC, nn	D2nn
JP nn	C3nn

JP NZ, nn	C2nn
JP P, nn	F2nn
JP PE, nn	EAnn
JP PO, nn	E2nn
JP Z, nn	CAnn
JR C, d	38d
JR, d	18d
JR NC, d	30d
JR NZ, d	20d
JR Z, d	28d
LD (BC), A	02
LD (DE), A	12
LD (HL), A	77
LD (HL), B	70
LD (HL), C	71
LD (HL), D	72
LD (HL), E	73
LD (HL), H	74
LD (HL), L	75
LD (HL), n	36n
LD (IX + d), A	DD77d
LD (IX + d), B	DD70d
LD (IX + d), C	DD71d
LD (IX + d), D	DD72d
LD (IX + d), E	DD73d
LD (IX + d), H	DD74d
LD (IX + d), L	DD75d
LD (IX + d), n	DD36dn
LD (IY + d), A	FD77d
LD (IY + d), B	FD70d
LD (IY + d), C	FD71d
LD (IY + d), D	FD72d
LD (IY + d), E	FD73d
LD (IY + d), H	FD74d
LD (IY + d), L	FD75d
LD (IY + d), n	FD36dn
LD (nn), A	32nn
LD (nn), BC	ED43nn
LD (nn), DE	ED53nn
LD (nn), HL	22nn
LD (nn), IX	DD22nn
LD (nn), IY	FD22nn
LD (nn), SP	ED73nn
LD A, (BC)	0A
LD A, (DE)	1A
LD A, (HL)	7E
LD A, (IX + d)	DD7Ed
LD A, (IY + d)	FD7Ed
LD A, (nn)	3Ann
LD A, A	7F
LD A, B	78
LD A, C	79
LD A, D	7A
LD A, E	7B
LD A, H	7C
LD A, I	ED57
LD A, L	7D
LD A, n	3En
LD B, (HL)	46
LD B, (IX + d)	DD46d
LD B, (IY + d)	FD46d
LD B, A	47
LD B, B	40
LD B, C	41
LD B, D	42
LD B, E	43
LD B, H	44
LD B, L	45
LD B, n	06n
LD B, C (nn)	ED4Bnn
LD B, C nn	01nn
LD C, (HL)	4E
LD C, (IX + d)	DD4Ed
LD C, (IY + d)	FD4Ed
LD C, A	4F
LD C, B	48
LD C, C	49
LD C, D	4A
LD C, E	4B
LD C, H	4C
LD C, L	4D
LD C, n	0En
LD D, (HL)	56
LD D, (IX + d)	DD56d
LD D, (IY + d)	FD56d
LD D, A	57
LD D, B	50
LD D, C	51
LD D, D	52
LD D, E	53
LD D, H	54
LD D, L	55
LD D, n	16n
LD DE, (nn)	ED58nn
LD DE, nn	11nn
LD E, (HL)	5E
LD E, (IX + d)	DD5Ed
LD E, (IY + d)	FD5Ed
LD E, A	5F
LD E, B	58
LD E, C	59
LD E, D	5A
LD E, E	5B
LD E, H	5C
LD E, L	5D
LD E, n	1En
LD H, (HL)	66
LD H, (IX + d)	DD66d
LD H, (IY + d)	FF66d
LD H, A	67
LD H, B	60
LD H, C	61

LD H, D	62
LD H, E	63
LD H, H	64
LD H, L	65
LD H, n	26n
LD HL, (nn)	2Ann
LD HL, nn	21nn
LD I, A	ED47
LD IX, (nn)	DD2Ann
LD IX, nn	DD21nn
LD IY, (nn)	FD2Ann
LD IY, nn	FD21nn
LD L, (HL)	6E
LD L, (IX + d)	DD6Ed
LD L, (IY + d)	FD6Ed
LD L, A	6F
LD L, B	68
LD L, C	69
LD L, D	6A
LD L, E	6B
LD L, H	6C
LD L, L	6D
LD L, n	2En
LD SP, (nn)	ED7Bnn
LD SP, HL	F9
LD SP, IX	DDF9
LD SP, IY	FDF9
LD SP, nn	31nn
LDD	EDA8
LDDR	EDB8
LDI	EDAO
LDIR	EDB0
NEG	ED44
NOP	00
OR (HL)	B6
OR (IX + d)	DDB6d
OR (IY + d)	FDB6d
OR A	B7
OR B	B0
OR C	B1
OR D	B2
OR E	B3
OR H	B4
OR L	B5
OR n	F6n
OTDR	EDBB
OTIR	EDB3
OUT (C), A	ED79
OUT (C), B	ED41
OUT (C), C	ED49
OUT (C), D	ED51
OUT (C), E	ED59
OUT (C), H	ED61
OUT (C), L	ED69
OUT (n), A	D3n
OUTD	EDAB
OUTI	EDA3
POPAF	F1
POP BC	C1
POP DE	D1
POP HL	E1
POP IX	DDE1
POP IY	FDE1
PUSH AF	F5
PUSH BC	C5
PUSH DE	D5
PUSH HL	E5
PUSH IX	DDE5
PUSH IY	FDE5
RES 0, (HL)	CB86
RES 0, (IX + d)	DDCBd86
RES 0, (IY + d)	FDCBd86
RES 0, A	CB87
RES 0, B	CB80
RES 0, C	CB81
RES 0, D	CB82
RES 0, E	CB83
RES 0, H	CB84
RES 0, L	CB85
RES 1, (HL)	CB8E
RES 1, (IX + d)	DDCBd8E
RES 1, (IY + d)	FDCBd8E
RES 1, A	CB8F
RES 1, B	CB88
RES 1, C	CB89
RES 1, D	CB8A
RES 1, E	CB8B
RES 1, H	CB8C
RES 1, L	CB8D
RES 2, (HL)	CB96
RES 2, (IX + d)	DDCBd96
RES 2, (IY + d)	FDCBd96
RES 2, A	CB97
RES 2, B	CB90
RES 2, C	CB91
RES 2, D	CB92
RES 2, E	CB93
RES 2, H	CB94
RES 2, L	CB95
RES 3, (HL)	CB9E
RES 3, (IX + d)	DDCBd9E
RES 3, (IY + d)	FDCBd9E
RES 3, A	CB9F
RES 3, B	CB98
RES 3, C	CB99
RES 3, D	CB9A
RES 3, E	CB9B
RES 3, H	CB9C
RES 3, L	CB9D
RES 4, (HL)	CBA6
RES 4, (IX + d)	DDCBdA6
RES 4, (IY + d)	FDCBdA6

RES 4, A	CBA7
RES 4, B	CBA0
RES 4, C	CBA1
RES 4, D	CBA2
RES 4, E	CBA3
RES 4, H	CBA4
RES 4, L	CBA5
RES 5, (HL)	CBAE
RES 5, (IX + d)	DDCBdAE
RES 5, (IY + d)	FDCBdAE
RES 5, A	CBAF
RES 5, B	CBA8
RES 5, C	CBA9
RES 5, D	CBAA
RES 5, E	CBAB
RES 5, H	CBAC
RES 5, L	CBAD
RES 6, (HL)	CBB6
RES 6, (IX + d)	DDCBdB6
RES 6, (IY + d)	FDCBdB6
RES 6, A	CBB7
RES 6, B	CBB0
RES 6, C	CBB1
RES 6, D	CBB2
RES 6, E	CBB3
RES 6, H	CBB4
RES 6, L	CBB5
RES 7, (HL)	CBBE
RES 7, (IX + d)	DDCBdBE
RES 7, (IY + d)	FDCBdBE
RES 7, A	CBBF
RES 7, B	CBB8
RES 7, C	CBB9
RES 7, D	CBBA
RES 7, E	CBBB
RES 7, H	CBBC
RES 7, L	CBBD
RET	C9
RET C	D8
RET M	F8
RET NC	D0
RET NZ	C0
RET P	F0
RET PE	E8
RET P0	E0
RET Z	C8
RETI	ED4D
RETN	ED45
RL (HL)	CB16
RL (IX + d)	DDCBd16
RL (IY + d)	FDCBd16
RL A	CB17
RL B	CB10
RL C	CB11
RL D	CB12
RL E	CB13
RL H	CB14
RL L	CB15
RLA	17
RLC (HL)	CB06
RLC (IX + d)	DDCBd06
RLC (IY + d)	FDCBd06
RLC A	CB07
RLC B	CB00
RLC C	CB01
RLC D	CB02
RLC E	CB03
RLC H	CB04
RLC L	CB05
RLCA	07
RLD	ED6F
RR (HL)	CB1E
RR (IX + d)	DDC8d1E
RR (IY + d)	FDCBd1E
RR A	CB1F
RR B	CB18
RR C	CB19
RR D	CB1A
RR E	CB1B
RR H	CB1C
RR L	CB1D
RRA	1F
RRC (HL)	CB0E
RRC (IX + d)	DDCBd0E
RRC (IY + d)	FDCBd0E
RRC A	CB0F
RRC B	CB08
RRC C	CB09
RRC D	CB0A
RRC E	CB0B
RRC H	CB0C
RRC L	CB0D
RRC A	0F
RRD	ED67
RST 0	C7
RST10H	D7
RST 18H	DF
RST 20H	E7
RST 28H	EF
RST 30H	F7
RST 38H	FF
RST 8	CF
SBC A, (HL)	9E
SBC A, (IX + d)	DD9Ed
SBC A, (IY + d)	FD9Ed
SBC A, A	9F
SBC A, B	98
SBC A, C	99
SBC A, D	9A
SBC A, E	9B
SBC A, H	9C
SBC A, L	9D

SBC A, n	DEn
SBC HL, BC	ED42
SBC HL, DE	ED52
SBC HL, HL	ED62
SBC HL, SP	ED72
SCF	37
SET 0, (HL)	CBC6
SET 0, (IX + d)	DDCBdC6
SET 0, (IY + d)	FDCBdC6
SET 0, A	CBC7
SET 0, B	CBC0
SET 0, C	CBC1
SET 0, D	CBC2
SET 0, E	CBC3
SET 0, H	CBC4
SET 0, L	CBC5
SET 1, (HL)	CBCE
SET 1, (IX + d)	DDCBdCE
SET 1, (IY + d)	FDCBdCE
SET 1, A	CBCF
SET 1, B	CBC8
SET 1, C	CBC9
SET 1, D	CBCA
SET 1, E	CBCB
SET 1, H	CBCC
SET 1, L	CBCD
SET 2, (HL)	CBD6
SET 2, (IX + d)	DDCBdD6
SET 2, (IY + d)	FDCBdD6
SET 2, A	CBD7
SET 2, B	CBD0
SET 2, C	CBD1
SET 2, D	CBD2
SET 2, E	CBD3
SET 2, H	CBD4
SET 2, L	CBD5
SET 3, (HL)	CBDE
SET 3, (IX + d)	DDCBdDE
SET 3, (IY + d)	FDCBdDE
SET 3, A	CBDF
SET 3, B	CBD8
SET 3, C	CBD9
SET 3, D	CBDA
SET 3, E	CBDB
SET 3, H	CBDC
SET 3, L	CBDD
SET 4, (HL)	CBE6
SET 4, (IX + d)	DDCbE6
SET 4, (IY + d)	FDCBdE6
SET 4, A	CBE7
SET 4, B	CBE0
SET 4, C	CBE1
SET 4, D	CBE2
SET 4, E	CBE2
SET 4, H	CBE4
SET 4, L	CBE5
SET 5, (HL)	CBEE
SET 5, (IX + d)	DDCBdEE
SET 5, (IY + d)	FDCBdEE
SET 5, A	CBEF
SET 5, B	CBE8
SET 5, C	CBE9
SET 5, D	CBEA
SET 5, E	CBEB
SET 5, H	CBEC
SET 5, L	CBED
SET 6, (HL)	CBF6
SET 6, (IX + d)	DDCBdF6
SET 6, (IY + d)	FDCBdF6
SET 6, A	CBF7
SET 6, B	CBF0
SET 6, C	CBF1
SET 6, D	CBF2
SET 6, E	CBF3
SET 6, H	CBF4
SET 6, L	CBF5
SET 7, (HL)	CBFE
SET 7, (IX + d)	DDCBdFE
SET 7, (IY + d)	FDCBdFE
SET 7, A	CBFF
SET 7, B	CBF8
SET 7, C	CBF9
SET 7, D	CBFA
SET 7, E	CBFB
SET 7, H	CBFC
SET 7, L	CBFD
SLA (HL)	CB26
SLA (IX + d)	DDCBd26
SLA (IY + d)	FDCBd26
SLA A	CB27
SLA B	CB20
SLA C	CB21
SLA D	CB22
SLA E	CB23
SLA H	CB24
SLA L	CB25
SRA (HL)	CB2E
SRA (IX + d)	DDCBd2E
SRA (IY + d)	FDCBd2E
SRA A	CB2F
SRA B	CB28
SRA C	CB29
SRA D	CB2A
SRA E	CB2B
SRA H	CB2C
SRA L	CB2D
SRL (HL)	CB3E
SRL (IX + d)	DDCBd3E
SRL (IY + d)	FDCBd3E
SRL A	CB3F
SRL B	CB38
SRL C	CB39

SRL D	CB3A
SRL E	CB3B
SRL H	CB3C
SRL L	CB3D
SUB (HL)	96
SUB (IX + d)	DD96d
SUB (IY + d)	FD96d
SUB A	97
SUB B	90
SUB C	91
SUB D	92
SUB E	93
SUB H	94
SUB L	95
SUB n	D6n
XOR (HL)	AE
XOR (IX + d)	DDAEd
XOR (IY + d)	FDAEd
XOR A	AF
XOR B	A8
XOR C	A9
XOR D	AA
XOR E	AB
XOR H	AC
XOR L	AD
XOR n	EEn

HEX/Decimal Conversion Table

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

Binary

If this is the first time that you have encountered Binary you will probably find it slightly confusing. It may even put you off wanting to learn Machine Code, and I certainly wouldn't want that to happen. Let me stress that you do not need to understand Binary in order to write Machine Code programs. In the section on storing numbers, you will have learnt that each memory location can hold a number between 0 and 255. We refer to these numbers as bytes. Each byte is divided into eight parts, known as bits; these are numbered in the following way:

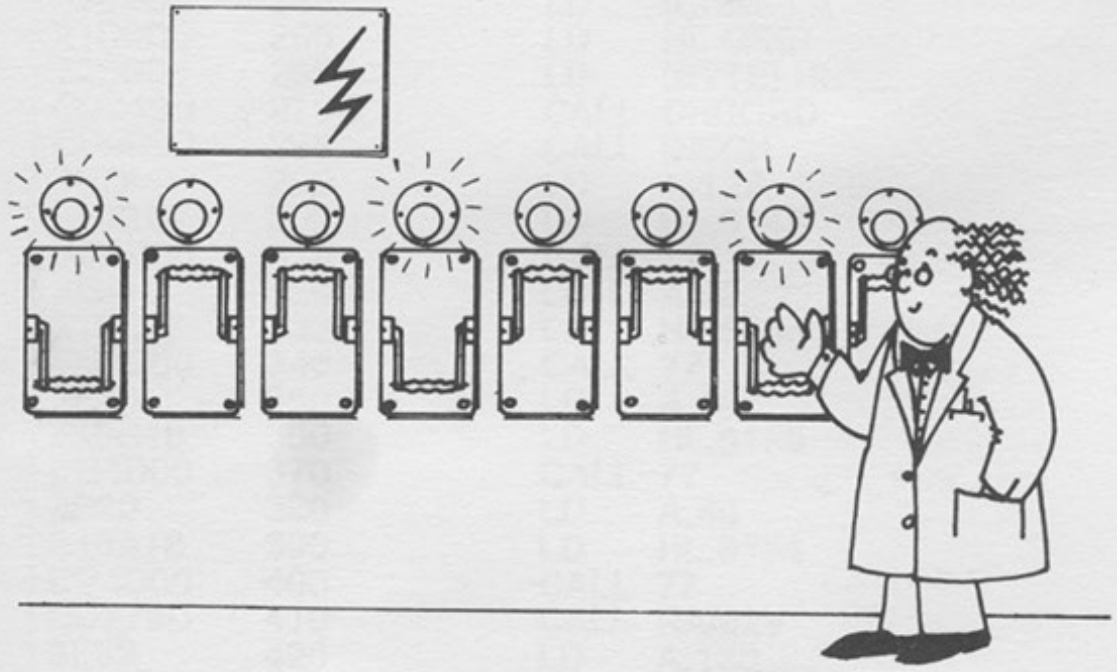
Eight bits make one byte

7	6	5	4	3	2	1	0

These eight bits can be thought of as being eight little switches. Each switch can be either on or off. When a switch is off we say that it has a value of 0. When it is on, then it represents a value as shown in the following diagram:

Bit values	128	64	32	16	8	4	2	1
Bit numbers	7	6	5	4	3	2	1	0

So it can be seen that when bit 4 is on it represents a value of 16. If bits 1 and 3 were on, and the rest were off, the byte would have a total value of 10. This is because bit 1, when it is on, has a value of 2 and bit 3 has a value of 8. Any number between 0 and



255 is stored in a byte as a combination of bits. Fortunately, you do not really have to concern yourself with the practicality of this: when you POKE and PEEK – or the equivalent in Machine Code – the processor takes care of the bits.

If bits 1, 4 and 6 of a byte are on then what is the value of the byte?

Which bits of a byte should be on to give a value of 67?

(I haven't actually shown you how to calculate this, but see if you can work it out.)

Character/Sprite Designer

The following program will enable you to design your own character and sprite shapes for use in future programs. In an earlier chapter I showed you how each character was made up of eight bytes; this program lets you design the eight bytes. Remember, this designer is specifically for use with SCREEN MODE 1. Enter the program using HEXENT and check it in the normal way:

Start address **40000**
End address **40702**
Hex total **85482**

```
011800  110 START:  LD    BC,24
11F803  120         LD    DE,1016
215F9D  130         LD    HL,IXSTR
CD5C00  140         CALL  92
3E01    150         LD    A,1
32E3D6  160         LD    (CURCHR),A
32E8D6  170         LD    (VER),A
32E9D6  180         LD    (HOR),A
210218  190         LD    HL,6146
22E6D6  200         LD    (CURSOR),HL
21A08C  210         LD    HL,SETSTR
22E4D6  220         LD    (CHRADD),HL
3E80    230         LD    A,128
32E2D6  240         LD    (CHBIT),A
21D8D6  250         LD    HL,GRID
22E0D6  260         LD    (BYTE),HL
CD849D  270         CALL  CHRGRD
CD909D  280         CALL  DISCH
3E7F    290         LD    A,127
210218  300         LD    HL,6146
CD4D00  310         CALL  77
3E31    320         LD    A,49
210C18  330         LD    HL,6156
CD4D00  340         CALL  77
3E30    350         LD    A,48
210B18  360         LD    HL,6155
CD4D00  370         CALL  77
3E30    380         LD    A,48
210A18  390         LD    HL,6154
CD4D00  400         CALL  77
CD779D  410         CALL  RAM2V
3E82    420         LD    A,130
```

```
218B18   430          LD    HL,6283
CD4D00   440          CALL  77          WRTVRM
00       450 LOOP:    NOP
3E07     460          LD    A,7
CD4101   470          CALL  321
CB57     480          BIT   2,A
C8       490          RET   Z
3E08     500          LD    A,8
CD4101   510          CALL  321         SNSMAT
CB47     520          BIT   0,A
CAEC9D   530          JP    Z,SWITCH
CB7F     540          BIT   7,A
CA3D9D   550          JP    Z,RIGHT
CB77     560          BIT   6,A
CAFB9C   570          JP    Z,CDOWN
CB6F     580          BIT   5,A
CAD89C   590          JP    Z,CUP
CB67     600          BIT   4,A
CA1F9D   610          JP    Z,LEFT
3E04     620          LD    A,4
CD4101   630          CALL  321         SNSMAT
CB6F     640          BIT   5,A
CA8B9E   650          JP    Z,PREV
CB5F     660          BIT   3,A
CA329E   670          JP    Z,NEXT
C39F9C   680          JP    LOOP
3AE8D6   690 CUP:     LD    A,(VER)
3D       700          DEC   A
CA9F9C   710          JP    Z,LOOP
32E8D6   720          LD    (VER),A
2AE0D6   730          LD    HL,(BYTE)
2B       740          DEC   HL
22E0D6   750          LD    (BYTE),HL
2AE6D6   760          LD    HL,(CURSOR)
012000   770          LD    BC,32
A7       780          AND   A
ED42     790          SBC   HL,BC
22E6D6   800          LD    (CURSOR),HL
CD909D   810          CALL  DISCH
C39F9C   820          JP    LOOP
3AE8D6   830 CDOWN:   LD    A,(VER)
D608     840          SUB   8
CA9F9C   850          JP    Z,LOOP
C609     860          ADD   A,9
32E8D6   870          LD    (VER),A
2AE0D6   880          LD    HL,(BYTE)
23       890          INC   HL
22E0D6   900          LD    (BYTE),HL
2AE6D6   910          LD    HL,(CURSOR)
012000   920          LD    BC,32
```

```
09          930          ADD   HL,BC
22E6D6      940          LD    (CURSOR),HL
CD909D      950          CALL  DISCH
C39F9C      960          JP    LOOP
3AE9D6      970 LEFT:    LD    A,(HOR)
3D          980          DEC   A
CA9F9C      990          JP    Z,LOOP
32E9D6      1000         LD    (HOR),A
3AE2D6      1010         LD    A,(CHBIT)
87          1020         ADD   A,A
32E2D6      1030         LD    (CHBIT),A
2AE6D6      1040         LD    HL,(CURSOR)
2B          1050         DEC   HL
22E6D6      1060         LD    (CURSOR),HL
CD909D      1070         CALL  DISCH
C39F9C      1080         JP    LOOP
3AE9D6      1090 RIGHT:  LD    A,(HOR)
D608        1100         SUB   8
CA9F9C      1110         JP    Z,LOOP
C609        1120         ADD   A,9
32E9D6      1130         LD    (HOR),A
3AE2D6      1140         LD    A,(CHBIT)
CB0F        1150         RRC   A
32E2D6      1160         LD    (CHBIT),A
2AE6D6      1170         LD    HL,(CURSOR)
23          1180         INC   HL
22E6D6      1190         LD    (CURSOR),HL
CD909D      1200         CALL  DISCH
C39F9C      1210         JP    LOOP
FFC3A599    1220 IXSTR:  DEFB  255,195,165,153
99A5C3FF    1330         DEFB  153,165,195,255
FF81BDBD    1240         DEFB  255,129,189,189
BDBD81FF    1250         DEFB  189,189,129,255
FF818181    1260         DEFB  255,129,129,129
818181FF    1270         DEFB  129,129,129,255
010800      1280 RAM2V:  LD    BC,8
111004      1290         LD    DE,1040
2AE4D6      1300         LD    HL,(CHRADD)
CD5C00      1310         CALL  92
C9          1320         RET
010800      1330 CHRGRD  LD    BC,8
11D8D6      1340         LD    DE,GRID
2AE4D6      1350         LD    HL,(CHRADD)
EDB0        1360         LDIR
C9          1370         RET
210218      1380 DISCH:  LD    HL,6146
11D8D6      1390         LD    DE,GRID
3E08        1400         LD    A,8
F5          1410 DISCH1  PUSH  AF
1A          1420         LD    A,(DE)
```

```
CB7F      1430          BIT   7,A
CDDB9D    1440          CALL  BIT
CB77      1450          BIT   6,A
CDDB9D    1460          CALL  BIT
CB6F      1470          BIT   5,A
CDDB9D    1480          CALL  BIT
CB67      1490          BIT   4,A
CDDB9D    1500          CALL  BIT
CB5F      1510          BIT   3,A
CDDB9D    1520          CALL  BIT
CB57      1530          BIT   2,A
CDDB9D    1540          CALL  BIT
CB4F      1550          BIT   1,A
CDDB9D    1560          CALL  BIT
CB47      1570          BIT   0,A
CDDB9D    1580          CALL  BIT
13        1590          INC   DE
011800    1600          LD    BC,24
09        1610          ADD   HL,BC
F1        1620          POP   AF
3D        1630          DEC   A
C2989D    1640          JP    NZ,DISCH1
CD779D    1650          CALL  RAM2V
3E7F      1660          LD    A,127
2AE6D6    1670          LD    HL,(CURSOR)
CD4D00    1680          CALL  77
CDE39E    1690          CALL  DELAY
C9        1700          RET
F5        1710 BIT:     PUSH  AF
CAE49D    1720          JP    Z,BIT1
3E80      1730          LD    A,128
C3E69D    1740          JP    BIT2
3E81      1750 BIT1:    LD    A,129
CD4D00    1760 BIT2:    CALL  77
F1        1770          POP   AF
23        1780          INC   HL
C9        1790          RET
00        1800 SWITCH   NOP
2AE0D6    1810          LD    HL,(BYTE)
7E        1820          LD    A,(HL)
47        1830          LD    B,A
3AE2D6    1840          LD    A,(CHBIT)
4F        1850          LD    C,A
3E01      1860          LD    A,1
CB79      1870 SWIT1:   BIT   7,C
C2059E    1880          JP    NZ,SWIT2
CB00      1890          RLC   B
CB01      1900          RLC   C
3C        1910          INC   A
C3F89D    1920          JP    SWIT1
```

```
CB78      1930 SWIT2:  BIT   7,B
CA0F9E    1940         JP    Z,SWIT3
CBB8      1950         RES   7,B
C3119E    1960         JP    SWIT4
CBFF      1970 SWIT3:  SET   7,A
3D        1980 SWIT4:  DEC   A
CA1A9E    1990         JP    Z,SWIT5
CB08      2000         RRC   B
C3119E    2010         JP    SWIT4
78        2020 SWIT5:  LD    A,B
77        2030         LD    (HL),A
CDF29E    2040         CALL  GRDCHR
CD909D    2050         CALL  DISCH
CDE39E    2060 WAIT:   CALL  DELAY
3E08      2070         LD    A,8
CD4101    2080         CALL  321
CB47      2090         BIT   0,A
CA229E    2100         JP    Z,WAIT
C39F9C    2110         JP    LOOP
00        2120 NEXT:   NOP
3AE3D6    2130         LD    A,(CURCHR)
D67E      2140         SUB   126
CA9F9C    2150         JP    Z,LOOP
C67F      2160         ADD   A,127
32E3D6    2170         LD    (CURCHR),A
2AE4D6    2180         LD    HL,(CHRADD)
010800    2190         LD    BC,8
09        2200         ADD   HL,BC
22E4D6    2210         LD    (CHRADD),HL
CD849D    2220         CALL  CHRGRD
CD909D    2230         CALL  DISCH
210C18    2240         LD    HL,6156
CD4A00    2250         CALL  74
D639      2260         SUB   57
CA639E    2270         JP    Z,NEXT1
C63A      2280         ADD   A,58
CD4D00    2290         CALL  77
C39F9C    2300         JP    LOOP
3E30      2310 NEXT1:  LD    A,48
CD4D00    2320         CALL  77
210B18    2330         LD    HL,6155
CD4A00    2340         CALL  74
D639      2350         SUB   57
CA7B9E    2360         JP    Z,NEXT2
C63A      2370         ADD   A,58
CD4D00    2380         CALL  77
C39F9C    2390         JP    LOOP
3E30      2400 NEXT2:  LD    A,48
CD4D00    2410         CALL  77
210A18    2420         LD    HL,6154
```

```
3E31      2430            LD    A,49
CD4D00    2440            CALL  77          WRTVRM
C39F9C    2450            JP    LOOP
00        2460 PREV:      NOP
3AE3D6    2470            LD    A,(CURCHR)
3D        2480            DEC   A
CA9F9C    2490            JP    Z,LOOP
32E3D6    2500            LD    (CURCHR),A
2AE4D6    2510            LD    HL,(CHRADD)
010800    2520            LD    BC,8
A7        2530            AND   A
ED42      2540            SBC   HL,BC
22E4D6    2550            LD    (CHRADD),HL
CD849D    2560            CALL  CHRGRD
CD909D    2570            CALL  DISCH
210C18    2580            LD    HL,6156
CD4A00    2590            CALL  74          RDVRM
D630      2600            SUB   48
CABB9E    2610            JP    Z,PREV1
C62F      2620            ADD   A,47
CD4D00    2630            CALL  77          WRTVRM
C39F9C    2640            JP    LOOP
3E39      2650 PREV1:     LD    A,57
CD4D00    2660            CALL  77          WRTVRM
210B18    2670            LD    HL,6155
CD4A00    2680            CALL  74          RDVRM
D630      2690            SUB   48
CAD39E    2700            JP    Z,PREV2
C62F      2710            ADD   A,47
CD4D00    2720            CALL  77          WRTVRM
C39F9C    2730            JP    LOOP
3E39      2740 PREV2:     LD    A,57
CD4D00    2750            CALL  77          WRTVRM
210A18    2760            LD    HL,6154
3E30      2770            LD    A,48
CD4D00    2780            CALL  77          WRTVRM
C39F9C    2790            JP    LOOP
3E32      2800 DELAY:     LD    A,50
F5        2810 DEL:       PUSH  AF
3EFF      2820            LD    A,255
3D        2830 DEL1:      DEC   A
C2E89E    2840            JP    NZ,DEL1
F1        2850            POP   AF
3D        2860            DEC   A
C2E59E    2870            JP    NZ,DEL
C9        2880            RET
010800    2890 GRDCHR     LD    BC,8
ED5BE4D6  2900            LD    DE,(CHRADD)
21D8D6    2910            LD    HL,GRID
EDB0      2920            LDIR
C9        2930            RET
```

Having entered the program and checked it, you can SAVE it on tape using:

```
BSAVE"CAS:DESIN", 40000,40800
```

The program can now be tested by entering and running the following lines:

```
1000 DEF USR = 40000
1005 A = USR(1)
```

If you have entered the program correctly you will see an 8 x 8 grid appear on the screen. In the top left-hand corner of the screen you will see an X. This X can be moved around the grid with the four cursor (arrow) keys. Think of the 64 squares as being switches; each one can be turned on or off. (Read the section on Binary for a greater explanation of this.) To turn a square on or off, first move the X over that square. By pressing the space bar you will alter the condition of that square. That is, if it was on, then pressing the space bar will turn it off; if it was off, then it will be turned on. Just to the right of the grid you will actually see the character you are designing. By moving the X

around the grid you can design a complete character.

This program can be used to design 126 characters. At the top of the screen you will see a number; this is the character currently displayed in the large grid. To move on to any other characters press N (Next). Holding down key N will step through all the characters up to 126. Of course, you won't actually see anything interesting until you have designed the characters. Pressing key P (Previous) will allow you to go back to character 1.

When you have designed some characters and want to SAVE them on tape, press the ESC key. Now type in:

```
BSAVE"CAS:CHARS" 41000,42007
```

I advise you to save the characters twice on two separate cassettes. This will prevent a lot of hard work being lost if one of the cassettes is damaged.

I will now recap on the procedure for designing your own characters:

1) Type in: `CLEAR 200,35999`
2) Type in: `BLOAD"CAS:"`

This will load up the character designer program.

3) At this stage you may wish to load a set of characters which you have already started to design and saved on tape. Do this by typing in:

```
BLOAD"CAS:"
```

4) Enter and run the following lines:

```
1000 DEF USR = 40000
1005 A = USR (1)
```

5) Design your characters.
6) Press ESC.
7) Save your character set on tape using:

```
BSAVE"CAS:CHARS", 36000,38007
```

Having designed your own characters you now need to know

how to use them in your own program. The sequence for doing this is as follows:

1) Type in: **CLEAR 200,35999**
2) Type in: **SCREEN 1**
3) Load your character set using:

BLOAD"CAS:"

The 126 characters are now in RAM; however, to be of any use they need to be moved into VRAM. The following BASIC program will transfer the characters into VRAM:

```
10 FOR A = 0 TO 2007
20 VPOKE(CHARSTART + A), PEEK (36000 + A)
30 NEXT A
40 STOP
```

The previous BASIC program will put your characters into a position whereby the first character you designed will have a code of 126, and the 126th character you designed will have a code of 251. It is these codes which you VPOKE into the screen memory map to make the characters appear on the screen.

Do not change SCREEN MODE when your characters are in VRAM or else they will be overwritten with the normal character set.

The following BASIC program will show that all your designed characters are actually in VRAM:

```
10 FOR A = 126 TO 251
20 VPOKE (SCREENSTART + A),A
30 NEXT A
40 STOP
```

You can now use the characters in your own BASIC or Machine Code programs.

Although the character designer only allows you to design 126 characters, the following method will allow you to redesign and use 252 characters:

1) **CLEAR 200,35999**
2) Load the character designer using:

BLOAD"CAS:"

3) Type in and run the following lines:

```
1000 DEF USR = 40000
1005 A = USR (1)
```

4) Design 126 characters.
5) Press ESC.
6) Save the characters on tape using the following:

```
BSAVE"CAS:SET1",36000,38007
```

7) Type RUN.
8) Design another 126 characters.
9) Press ESC.
10) Save the characters on tape using the following:

```
BSAVE"CAS:SET2", 36000,38007
```

When you wish to use the 252 characters in your program you must do the following:

1)
```
CLEAR 200,35999
```
2)
```
BLOAD "CAS:SET 1"
```
3) Enter and run the following program:

```
10 FOR A = 0 TO 2007
20 VPOKE(CHARSTART + A + 1008),PEEK (36000 + A)
30 NEXT A
40 STOP
```

The above program will put the 126 characters of set 1 into positions in VRAM where their codes are 126 to 251.

4)
```
BLOAD "CAS:SET2"
```
5) Change line 20 of the above BASIC program to:

```
20 VPOKE(CHARSTART + A), PEEK (A + 36000)
```

6) Now type RUN. This will transfer character set 2 into positions in VRAM where their codes will be 0 to 125. Having done this you may find that what is on the screen is very difficult to read, unless you actually designed some letters and numbers in the correct positions in character set 2.

Obviously, if you are going to write a program that uses a lot

of redesigned characters then it is best to leave the actual designing of the characters until you have completed the program.

SPRITE DESIGNER

The character designer can also be used to design the shapes of sprites. The following procedure will show you how to design the shapes of 32 sprites:

1) **CLEAR 200,35999**
2) Load the character design by typing in:

BLOAD"CAS:"

3) Enter and run the following two lines:

```
1000 DEF USR = 40000
1005 A = USR(1)
```

4) Design the 32 sprite shapes in the positions where you would normally design characters 1 to 32.
5) Press ESC.
6) Save the 32 sprite shapes on tape by typing in the following:

BSAVE"CAS:SP32",36000,38007

When you wish to use the sprite shapes follow this procedure:

1) **CLEAR 200,35999**

2) Load the sprite shapes by typing in the following:

BLOAD"CAS:SP32"

3) The sprite shapes are now transferred into their correct position in VRAM with the following program:

```
10 FOR A = 0 TO 255
20 VPOKE (SPRITE SHAPE START + A),
   PEEK (A + 36000)
30 NEXT A
40 STOP
```

Now you can VPOKE some of the sprite variables and try to make the sprites appear on screen.

Author's Note

I hope that I have now achieved what I set out to do, which was to give you a gentle introduction into the world of Machine Code programming. You should have discovered by now whether you and Machine Code make suitable partners. There are many Z80 opcodes which I have not covered in this book; should you wish to learn about them there are a few books available which cover the subject of Z80 opcodes in great detail. When you begin writing your own Machine Code programs you may come across a few problems which you are unable to solve. If you write to me via the publishers of this book then I will be pleased to try and help you (but I won't reply if you don't enclose a stamped addressed envelope!) Please do not write to me with such letters as "I have found a faster way of creating the explosion in the Space Invader program." Any such letters will be passed straight on to the dustman!

Steve Webb
61-63 Portobello Road
London W11
January 1985

Answers

1) *The high part of 45621 is 178, the low part is 53.*
2) *If the low part of a number is 31 and the high part is 64, then the number is 16415.*
3) *If location 40000 contained 5d and location 40001 contained 15d then after the instruction LD HL,(40000), HL will contain the value 3845.*
4) *If HL contains 35621, then after the instruction LD (40000),HL, location 40000 will contain 37 and location 40001 will contain 139.*
5) *The decimal equivalent of E3h is 227d.*
6) *If FBh is the high part of a number and CBh is the low part then the number is 64459.*

PRACTICAL MSX MACHINE CODE PROGRAMMING

The advent of the MSX standard marks a significant step forward in the world of home computing. It offers many exciting opportunities for programmers and users.

MSX computers have many advanced features such as sprites and sound. If you wish to take full advantage of these facilities then you will need to program in Machine Code.

The book assumes that you have no knowledge of Machine Code. It begins by explaining the Machine Code equivalents of the main BASIC instructions such as A=, IF, FOR/NEXT, PRINT, GOTO, GOSUB, ADDITION and SUBTRACTION. It goes on to describe in great detail the individual routines of a simple Space Invader game and how these routines are linked. Finally you will be shown a few routines which can be used with BASIC programs. There are also a number of questions throughout the book which will test your knowledge of Machine Code programming.

★ Learn how easy it is to incorporate Machine Code routines within your BASIC programs.
★ Use Machine Code to create stunning sound and graphics.
★ Design your own characters and sprites with two sophisticated programs.

Whether you simply wish to write a routine to enhance your BASIC programs or to write a fully-fledged Machine Code program, you will find this book extremely useful. It removes the theory surrounding Machine Code, presenting it in a way that is both practical and enjoyable to learn.

Virgin

ISBN 0 86369 074 2

United Kingdom £4.95
Australia $13.95 (recommended)