STEVE WEBB

AMSTRAD CPC 464 MACHINE CODE

Learn how to write Machine Code programs

Bring your Amstrad to life for only £4.95

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

VIRGIN BOOKS

PRACTICAL AMSTRAD MACHINE CODE PROGRAMMING

Steve Webb

PRACTICAL AMSTRAD MACHINE CODE PROGRAMMING

Steve Webb

Virgin

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

Copyright © 1985 by Steve Webb

ISBN 0 86369 082 3

All rights reserved. No part of this book
may be reproduced in any form or by
any means without the prior permission
of the publisher.

Designed by Ray Hyden.

Layout and illustrations 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.

DEDICATED TO ZOE

I would like to take this opportunity to thank "Dave the Invader" and Cat for helping to produce this book.

Contents

Introduction 13

What Is Machine Code? 17

Storing Opcodes in Memory 33

The Space Invader Program 49

A Few Useful Routines 67

Appendices 75

Introduction

The intention of this book is to introduce you to Machine Code programming on the Amstrad CPC 464 computer. When you have finished reading and understanding it, you should be able to write your own Machine Code programs.

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 quickly be able to grasp the concepts of Machine Code programming.

The chapters in this book are in a logical order, therefore it is important that you read and understand each one before moving on to the next.

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 of the BASIC statements and teaching you about the organization of the Amstrad. The main chapter then goes on to show you how to write a very simple Machine Code game. You are shown how programs can first be written as a flowchart, and the blocks of the flowchart divided into short, simple routines.

This book does not set out to teach you about the intricacies of the Amstrad. The Amstrad, for instance, has three different screen modes, which in themselves would require a whole book to describe them fully. The same also applies to the sophisticated sound facilities of the Amstrad. Rather, this book is an introduction to Machine Code programming. If, after reading the book, you decide that you wish to write some serious Machine Code programs then you will need to buy a few books to supplement the knowledge gained from this one. The sort of books to look out for are those with titles such as "Understanding Graphics on Your Amstrad" or "A Guide to Programming Sounds on Your Amstrad".

Throughout this book you will come across various questions, which you should attempt to answer (the answers are in the back of the book); if you get them wrong then reread the appropriate section until you are able to answer them correctly.

WHAT IS MACHINE CODE?

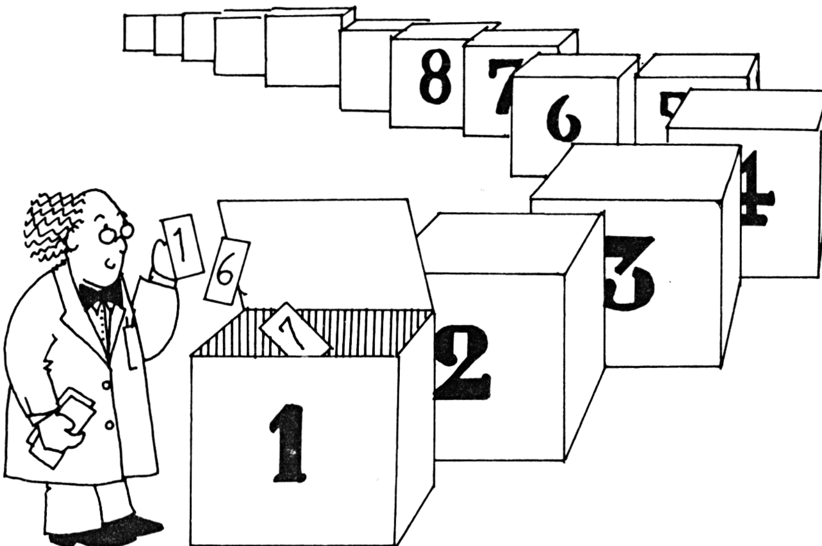
What Is Machine Code?

There are just over 64000 memory locations in your Amstrad, hence the name CPC 64. Each memory location can be considered to be a box, and each box is labelled from 0 to 63999. These labels do not actually exist anywhere; we just refer to the first box as 0, the next as 1 and so on. 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 and a method has to be found of 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 and 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 is then subtracted from the number to be stored (29248).

$$114 \times 256 = 29184$$
$$29248 - 29184 = 64$$

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



Machine Code Equivalents of Basic Statements

In this section I will describe which opcode can be used to simulate various BASIC statements. Quite a lot of BASIC statements such as LIST, RENUM, DELETE, etc., serve no purpose in Machine Code programming, so I will not show any equivalent opcodes for them. Some BASIC statements such as READ, DATA and PRINT have no close equivalent in Machine Code; 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; it can only hold a number between 0 and 255. To enable us to use the registers to store numbers greater than 255, six of the registers have been made into a group of 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 registered 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. For example you can have:

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 using 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 to which you can directly add a number. 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 directly any other combinations of registers other than those shown above. You cannot have, for instance:

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 say 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. Thus 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 appearing 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.

By 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.

Overcoming the limitations of the Z80 is a challenge and with practice you will find 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 from which we can directly subtract a number. 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 5d 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 from which you can directly subtract a value of another register. To subtract the value of a register from a register other than A you will have to use 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 now it is sufficient to regard it as an ordinary subtraction opcode. If you just wish to add one to the value 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; and this opcode and others will then 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 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 one from register A. If the result of subtracting one from register A was NZ (not zero) then we jump to the start of the routine. This continues until finally the result of subtracting one 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, for example, have the following instruction:

LD D,(40000)

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

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 contained 5 and location 40001 contained 15, what will the total value of HL be 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 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 because 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 that we can use. 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. I can hear you all 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, although you can actually PUSH a register pair named AF. F is a special register which cannot be used like the other seven single registers. 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 it. The F register is primarily used by the Z80 processor to indicate the results of various calculations. So, 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 the description of the F register you will get the impression that it should not be toyed with – and that is the way we will leave it.

OPCODE: EX

EX is the opcode to exchange the values of the specified registers. Thus:

```
EX DE,HL
```

means swop over the values of register pairs DE and HL. This opcode could be used when you want to add the value of BC to DE with the answer appearing in DE. Here is how to do it:

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

There are four other EX opcodes. However, since you are a beginner they will not be of any use to you yet, so I will not bother to explain them.

OPCODE: CP r

The opcode CP r allows you to compare register A with either a number or any of the other six single registers. If register A and another register have the same value, you can call, or jump to, a subroutine. Look at the following example:

```
CP A,E
CALL Z,subroutine
Rest of program
```

If register A and E have the same value, the subroutine would be called. This is because compare is very similar to subtract except that the answer is not stored anywhere. The result of a compare opcode would be zero, while the next opcode would be

CALL the specified subroutine if the result of the compare was zero.

Register A can also be compared with a number as shown below:

```
CP A,126
CALL Z, subroutine
Rest of program
```

If register A had a value of 126 then the subroutine would be called.

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. So some opcodes require one memory location, others require two memory locations.

If the first opcode of a program was INC A (increment A) we would first need to know which number represents INC A – that is, 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 get 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 (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 shortly), the processor would look at the contents of the first location and see that the opcode is ADD A. It would then look at the 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 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 which 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 GOTOing line numbers, we GOTO specific memory locations.

When we are inputting a Machine Code program we do not type in numbers using our normal decimal numbering system; instead we use a numbering system call HEXADECIMAL (HEX for short). To give you an idea of what HEX looks like have a look at the following. This 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 a 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 easily be

able to convert from one to another. One advantage of the HEX numbering system is that it is much neater than decimal. For example, HEX numbers between 0 and 255 are always represented by two digits; a decimal number 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 "HEXENT"

```
 10 MODE 1
 20 MEMORY 29999
 30 CLS
 40 LOCATE 1,1
 50 PRINT "PUT CAPS LOCK ON"
 60 LOCATE 1,4
 70 PRINT "PRESS KEY E TO ENTER HEX CODE."
 80 LOCATE 1,8
 90 PRINT "PRESS KEY C TO CHECK HEX CODE."
100 LOCATE 1,12
110 PRINT "PRESS KEY X TO CHECK HEX TOTAL."
120 LOCATE 1,16
130 PRINT "PRESS KEY Q TO STOP."
140 INPUT A$
150 IF A$ = "Q" THEN STOP
160 IF A$ = "E" THEN GOTO 370
170 IF A$ = "C" THEN GOTO 770
180 IF A$ = "X" THEN GOTO 200
190 GOTO 140
200 CLS
210 LOCATE 1,1
220 PRINT "INPUT STARTING ADDRESS."
230 INPUT SA
240 LOCATE 1,5
250 PRINT "INPUT END ADDRESS."
260 INPUT EA
270 LET D=0
280 FOR C = SA TO EA
290 LET D = D + PEEK(C)
300 NEXT C
310 CLS
320 PRINT "TOTAL COUNT = ";D
330 LOCATE 1,20
```

```
340 PRINT "PRESS M TO RETURN TO MENU."
350 IF INKEY$ <> "M" THEN GOTO 350
360 GOTO 30
370 CLS
380 LOCATE 1,1
390 PRINT "PUT CAPS LOCK ON."
400 LOCATE 1,4
410 PRINT "INPUT STARTING ADDRESS."
420 INPUT S
430 IF S<30000 THEN GOTO 750
440 LET A$=""
450 LOCATE 1,25
460 LET ET = S
470 IF A$ = "" THEN INPUT A$
480 LET BAD = 0
490 IF A$ = "M" THEN GOTO 30
500 LET E = LEN (A$)
510 LET E = E-1
520 LET C$ = A$
530 FOR D = 1 TO E STEP 2
540 LET B$ = LEFT$ (C$,2)
550 LET C = VAL ("&H"+ B$)
560 IF C=0 THEN GOSUB 730
570 LET C$ = MID$ (C$,3)
580 NEXT D
590 IF BAD = 1 THEN GOTO 690
600 LOCATE 1,25: PRINT S;"    ";A$
610 LET B$ = LEFT$ (A$,2)
620 IF LEN (B$) = 1 THEN GOTO 690
630 LET C = VAL ("&H"+ B$)
640 POKE (S),C
650 LET S = S + 1
660 LET A$ = MID$ (A$,3)
670 IF A$="" THEN GOTO 460
680 GOTO 610
690 LOCATE 1,25: PRINT "INCORRECT INPUT TRY AGAIN."
700 LET S = ET
710 LET A$ =""
720 GOTO 460
730 IF B$<> "00" THEN LET BAD = 1
740 RETURN
750 PRINT "START ADDRESS MUST BE 30000 OR GREATER."
760 GOTO 400
770 LET ND =0
780 CLS
790 LOCATE 1,1
800 PRINT "INPUT STARTING ADDRESS."
810 INPUT SA
820 LOCATE 1,5
830 PRINT "INPUT END ADDRESS."
```

```
 840  INPUT EA
 850  CLS
 860  IF SA + 20 >EA THEN GOTO 960
 870  FOR C = SA TO SA +20
 880  GOSUB 1000
 890  NEXT C
 900  IF ND = 1 THEN 1050
 910  IF C> EA THEN GOTO 1050
 920  PRINT "PRESS M FOR MORE."
 930  LET SA=C
 940  IF INKEY$ <> "M" THEN GOTO 940
 950  GOT0 860
 960  FOR C=SA TO EA
 970  GOSUB 1000
 980  NEXT C
 990  GOTO 900
1000  IF PEEK (C)< 16 THEN GOTO 1030
1010  PRINT C;"    ";HEX$ (PEEK(C))
1020  RETURN
1030  PRINT C;"    0"; HEX$(PEEK(C))
1040  RETURN
1050  PRINT "PRESS M TO RETURN TO MENU."
1060  IF INKEY$ <> "M" THEN GOTO 1060
1070  GOTO 30
```

Using Hexent

To load the HEXENT program you must type LOAD "" When the program has loaded, type run, and the following menu will appear:

```
Put caps lock on
Press key E to enter HEX code
Press key C to check HEX code
Press key X to check HEX total
Press key Q to stop
```

The following program will do the following:

```
Load register A with 17d
ADD five to register A
Store the new value of A in memory location 30050
Return to BASIC
```

```
3E11      20      LD      A,17
C605      30      ADD     A,5
326275    40      LD      (30050) ,A
C9        50      RET
```

The left-hand column shows the actual HEX code that you will be entering in a short while. The middle column shows four line numbers from 10 to 40. (These line numbers have nothing to do with the actual Machine Code program, but they serve as a reference. For example, I may say "Refer to line 20 of the above program". Remember, Machine Code programs do not have line numbers; if you want to CALL a subroutine or JUMP to a part of your program then you must CALL or JUMP to a specific memory location.) The third column shows a fairly easy to understand listing of what the HEX codes mean. For instance, the first line is LD A,17, which means load register A with a value of 17.

I will now show you exactly how to enter and check the above small program (the same procedure will apply to any of the programs throughout this book). At the start of each program you will see three things – the start address of the program, the end address of the program and the HEX total of the program. In the case of the above program the three things are:

Start address 30000
End address 30007
HEX total 748

Press key E, then enter, while at the menu stage of the HEXENT program. You will now be asked to enter the STARTING ADDRESS (in the above program it is 30000). Having done this, a small question mark will appear in the bottom left-hand corner of the screen. The program is now waiting for you to enter the HEX codes of the left-hand column in the program listing. The first line to be entered is 3E11; then press enter. 3E11 is two bytes of HEX, 3E is the Ld A and 11 is the HEX for 17. The first byte will be stored in location 30000 and the second byte in location 30001. Having pressed enter, you will see confirmation of what you have typed appear on the screen. You will also see on the screen the memory location of the first byte of the line you have just entered. You should now see the following:

30000 3E11

The question mark will also appear, telling you that the

program is ready to accept the next line of HEX – in this example, C605. Having typed this and pressed enter you should see the following:

30000 3E11
30002 C605

This shows that C6 has gone into location 30002 and 05 into location 30003.

Continue to enter the remaining two lines in exactly the same way as shown above and your screen should look like this:

30000 3E11
30002 C605
30004 326275
30007 C9

This shows that the last byte of HEX has gone into location 30007 – known as the end address of the program. Now press key M and enter, and this will return you to the menu. The next thing to do is to check that you have actually entered the HEX codes correctly. This is done by pressing key C and enter.

You will now be asked for the start address of the program (30000) and the end address (30007). On the screen you will now see a listing of the eight memory locations from 30000 to 30007, and the contents of these locations:

30000 3E
30001 11
30002 C6
30003 05
30004 32
30005 62
30006 75
30007 C9

You should carefully check that the printout is identical with what you should have entered. If it is incorrect you will need to re-enter the HEX codes. (Please remember that at this stage it is possible you may have incorrectly typed the HEXENT BASIC listing. You must make sure that the HEXENT program is correct before you proceed with the rest of this book.)

Having checked the listing you can return to the menu by pressing key M. There is one final check to be made before we try out the program, and this is the HEX TOTAL CHECK – initiated by pressing key X, then enter. You will again be asked to enter the start and end addresses of the program to be checked. The following will appear on the screen:

TOTAL COUNT = 748

The number 748 is the result of adding up the contents of the memory locations from 30000 to 30007. If you obtained a different number then you must check again that you have entered the code correctly. Possibly you have incorrectly typed in the part of the HEXENT BASIC program dealing with the HEX TOTAL CHECK. Now press key M to return to the menu.

We are now ready to test the Machine Code program. First, press key Q to stop the HEXENT program. Now type the following lines:

```
1000 CALL 30000
1005 Print PEEK (30050)
```

Now type GOTO 1000, and you should see the number 22 printed on the screen (the result of adding 17 and 5).

The above procedure for entering and checking Machine Code routines applies to all the listings in this book and 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; the first line will always start with line 1000 and you initiate it by GOTO 1000. When the testing has finished, the HEXENT program can be restarted by typing RUN.

If you wish to save any of the Machine Code programs on tape then do the following:

SAVE "Name",B, Start address, Length

Thus, if you wished to save the small program we have just entered, you would type the following:

```
SAVE"Add",B,30000,8
```

To load a Machine Code routine simply type LOAD""

I will now explain a few simple Machine Code routines which will come in useful when writing your own programs. There are nearly 200 routines in the Amstrad's ROM which Machine Code programmers can, and should, make use of. It is beyond the scope of this book, however, to describe what all of these routines do; and if you wish to know more about them then I suggest you purchase a book from Amstrad entitled "The Complete CPC 464 Operating System". I have made use of some of these routines, and I will explain them as necessary.

Screen Mode Set

As you know, the Amstrad has three screen modes −0,1 and 2. Any of these modes can easily be set by calling a routine at 48142. Before calling the routine, register A must be loaded with the screen mode that you require. The following routine shows how to set screen mode 2:

Start address **30000**
End address **30005**
HEX total **672**

```
3E02      10      LD      A,2
CD0EBC    20      CALL    48142
C9        30      RET
```

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

```
1000 CALL 30000
1005 STOP
```

Position Cursor

If you wish to print a character, the first thing to do is to position the cursor. This is done by calling a routine at location 47989. Before calling this routine, register H must contain the column number and register L must contain the row number indicating where you wish to position the cursor. The following routine will ensure that mode 2 is set as in the

previous routine; it will then move the cursor to column 5, row 10.

Start address **30000**
End address **30012**
HEX total **1280**

```
3E02      10     LD      A,2
CD0EBC    20     CALL    48142
2605      30     LD      H,5
2E0A      40     LD      L,10
CD75BB    50     CALL    47989
C9        60     RET
```

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

```
1000 CALL 30000
1005 GOTO 1005
```

In the appendix of this book are three charts showing the row and column numbers of each screen mode.

Print a Character

Having positioned the cursor, the next thing we do is to print a character. This is done by calling the routine at location 47962. Before calling this routine, register A must be loaded with a number between 0 and 255 (representing the character that you wish to print). If you look in the Appendix of your Amstrad's Instruction Manual you will see that there is a complete list of the 256 characters you can print. The capital letter A is represented by the number 65. The following routine will set screen mode 2, position the cursor at column 5 and row 10, then print the letter A:

Start address **30000**
End address **30017**
HEX total **1889**

```
3E02     10    LD    A,2
CD0EBC   20    CALL  48142
2605     30    LD    H,5
2E0A     40    LD    L,10
CD75BB   50    CALL  47989
3E41     60    LD    A,65
CD5ABB   70    CALL  47962
C9       80    RET
```

The routine can be tested with these lines of BASIC:

```
1000 CALL 30000
1005 GOTO 1005
```

Reading the Keyboard

There are very few programs that do not make use of the keyboard, and as you would expect there is a special routine for reading the keyboard. The routine for reading the keyboard is at location 47902. This routine will not immediately tell you which key has been pressed, you must first ask it whether a specific key has been pressed. You must load register A with the key that you wish to test, then call the routine. If the specified key is being pressed then, after calling the key routine, the zero flag will not be set. So the next routine after calling the key routine would be something like CALL IF NOT ZERO a subroutine. The subroutine, of course, is the routine we want to execute if the specified key is being pressed.

In the following example, the instruction after calling the key routine is JUMP IF ZERO to the start of the routine. The routine will therefore stay in a constant loop until the specified key is pressed. Obviously, before you can make use of the key routine you will need to know the number of the key you wish to detect. If you look in the appendix you will see that there is a layout of the keyboard showing the numbers of each key:

Start address **30000**
End address **30008**
HEX total **1121**

```
3E45      20 START   LD      A,69
CD1EBB    30         CALL    47902
CA3075    40         JP      Z,START
C9        50         RET
```

The above program can be tested with the following lines of BASIC. See if you can work out which key needs to be pressed in order to exit from the routine:

```
1000 CALL 30000
1005 Print "HOORAY"
```

Delays

Machine Code programs are very fast and it is often necessary to incorporate delay routines to slow them down. A very simple delay routine is shown below. Register A is loaded with a number which represents the required delay length. The next instruction is DECREMENT register A. Then a check is made to see if the value of A has reached zero; if it has, the program continues. If not, the program jumps back to the decrement instruction. This loop continues until the value of A is zero, thus causing a delay:

```
     LD A,35
LOOP DEC A
     JP NZ,LOOP
```

Even with an initial value of 255, the delay loop is still very fast. A longer delay can be obtained by using the following routine. Register pair BC must be loaded with a number between 1 and 65535 (representing the required delay time):

```
      LD BC,1743
DELAY DEC C
      JP NZ,DELAY
      DEC B
      JP NZ,DELAY
```

In most programs it is necessary to have several delays at different points throughout the program. To save memory space it is better to write a routine which has a variable delay time. Such a routine is shown below:

```
DELAY DEC C
      JP NZ,DELAY
      DEC B
      JP NZ,DELAY
      RET
```

You can see that it is virtually identical to the previous routine, except that the initial value of BC has not been declared and a return instruction has now been added at the end. When we wish to cause a delay we simply load register BC with the required value and call the above delay subroutine. You will see this method being used in the Space Invader program in the next section.

THE SPACE INVADER PROGRAM

The Space Invader Program

I will now describe in great detail how to write a simple Space Invader program. The fact that it is simple does not mean that the principles involved in writing it cannot be applied to writing far more complex programs.

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 routines and 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 to do this in BASIC programming, but it is very important when you first start to develop your own Machine Code routines and programs.

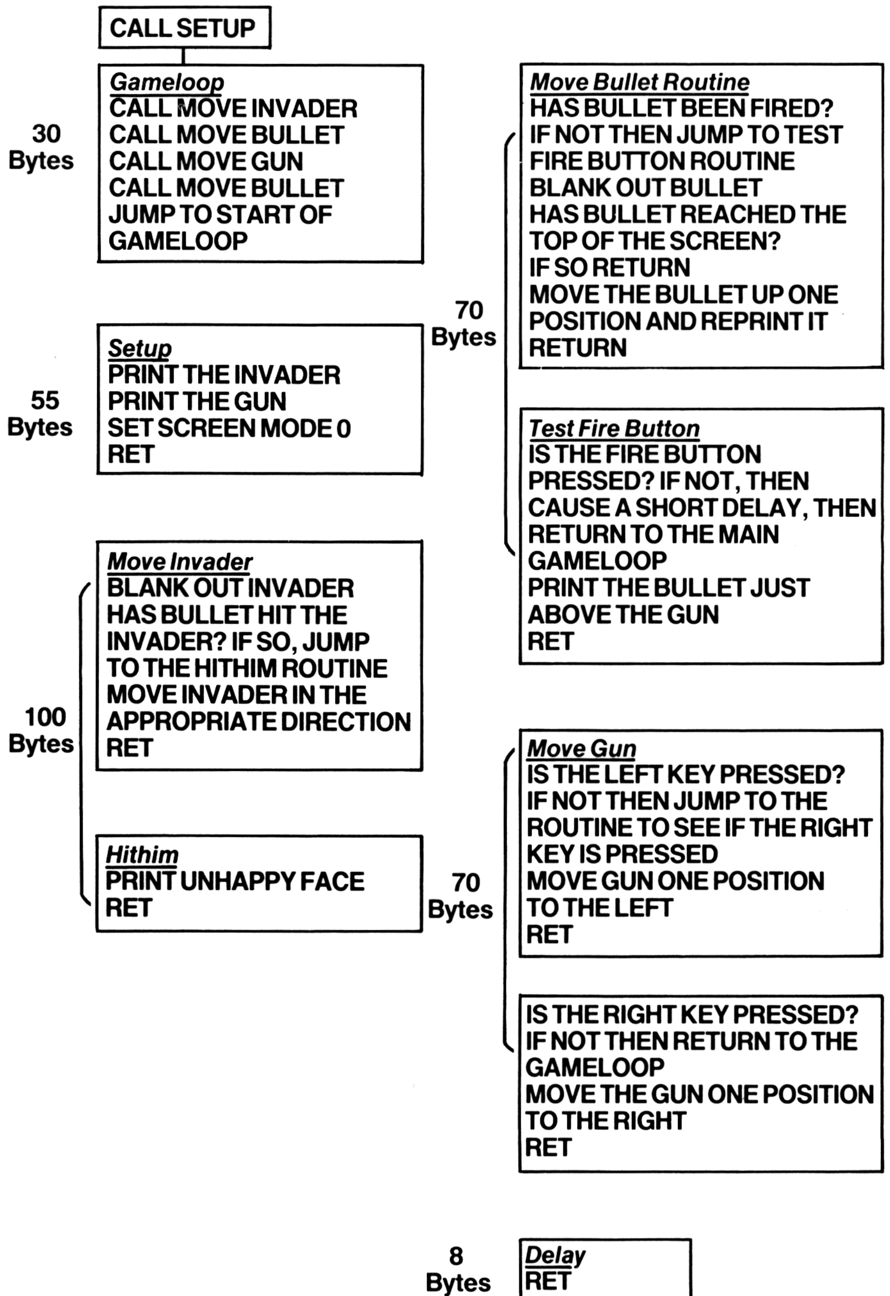
I will then go on to describe each of the flowchart blocks in detail. One of the blocks, for example, is labelled "move bullet up screen". I will describe exactly how this is achieved and, where necessary, I will show a further sub-flowchart of the actual routine.

The routines in this program have been written in such a way that they can be tested either 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 showing clearly how the program works and what it is meant to do. The following diagram is the complete flowchart of the Space Invader program:

FLOWCHART OF THE SPACE INVADER PROGRAM

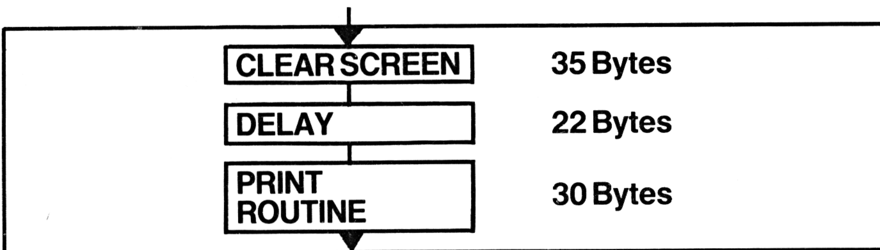


I am sure you will agree that the flowchart is relatively easy to understand. You should study it and get a firm understanding of how the finished program will work.

The Memory Map

I have already explained that in Machine Code programs we do not have line numbers. Therefore, if you need to insert a new instruction you will have to move all the instructions which occur after that particular location. This causes huge problems with absolute jumps and calls. Let's say that you have a delay routine at location 30050. To access the delay you would probably have an instruction somewhere in the program such as CALL 30050. If, for some reason, you needed to add a simple three-byte instruction at the end of the routine (which normally ends at location 30049), then quite obviously you will need to use locations 30050/51/52. So now you will have to move the delay routine to start at location 30053. Having moved the delay routine, you will now have to work through your program and change all the CALL 30050 to CALL 30053. 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 each routine will need. You will then allocate specific memory locations to each routine, leaving a few bytes of memory free between the end of one routine and the start of the next. The free memory spaces will then enable you to extend any routine without affecting the routines which come after it. The following flowchart of a very simple program can be used to demonstrate how to create a memory map.



The routines in the flowchart are each marked with the approximate number of bytes they use. The clear screen routine could be put at location 30000, the delay routine at 30050 and the print routine at location 30080. 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 **30000**
SETUP **30041**
MOVE INVADER **30107**
MOVE BULLET **30227**
MOVE GUN **30311**
DELAY **30395**
VARIABLES **31000**

In the memory map I have allocated an area of memory to store variable, and I will explain this in a short while.

Explanation of Terms

I will now explain a few terms which will make the program a lot easier to understand.

1) **GAMELOOP:** This is the main loop of the program. The loop repeatedly calls subroutines such as move the Invader and move the bullet.
2) **INVPOS:** Two bytes of memory are used to hold the current position of the Invader. This is used when the Invader is moved either to the left or right and updated with the new position.
3) **INVDIR:** This is a single memory location which keeps track of the direction in which the Space Invader is moving. It will contain the number one if it is moving from right to left, and the number zero if it is moving left to right.
4) **GUNPOS:** Two bytes of memory are used to hold the current position of the gun. This variable is used when the gun is

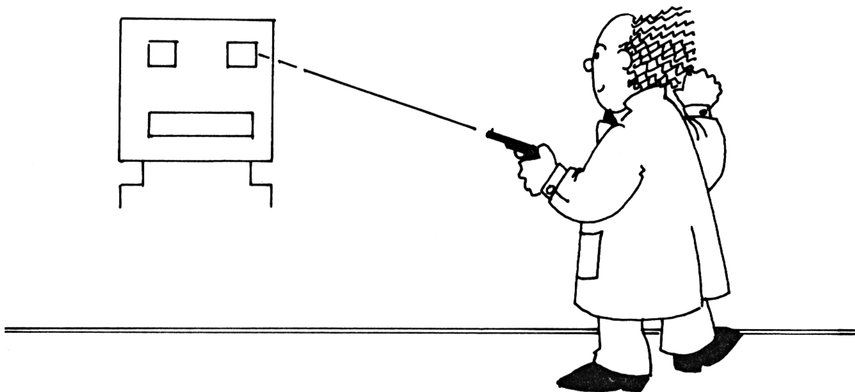
moved and it is then updated with the new position of the gun.

5) **BULPOS:** Two bytes of memory are used to record the current position of the gun. As with the other variables, this is updated each time the bullet moves.

Setup Routine

The first routine we will write is called the SETUP routine; it does the following:

1) Sets the Invader position (INVPOS) to row one and column one.
2) Sets the Invader direction (INVDIR) to one.
3) Sets the gun position (GUNPOS) to row 25 and column ten.
4) Sets the bullet position (BULPOS) to row 255 and column 255. Obviously there is no such row or column as 255 and the program uses this fact to establish that the bullet has not yet been fired. It is therefore unnecessary to move the bullet vertically up the screen.
5) Resets the colours to their default values – a blue background with yellow ink. It does this by calling a routine at location 48128.
6) Sets screen mode 0 by loading register A with zero and calling the routine at location 48142 (I have already explained this).
7) It then prints the Invader. It does this by loading register H with the current column position of the Invader and register L with the current row position of the Invader.



This is achieved with the single instruction LD HL, (INVPOS). The position cursor routine at location 47989 is then called. The character which represents the Space Invader is printed. This is done by loading register A with the number of the character which represents the Invader; then the print a character routine at location 47962 is called.

8) Finally the gun is printed in an identical manner to the Invader, except that the variable GUNPOS is used to position the cursor. The character number which represents the gun is 244.

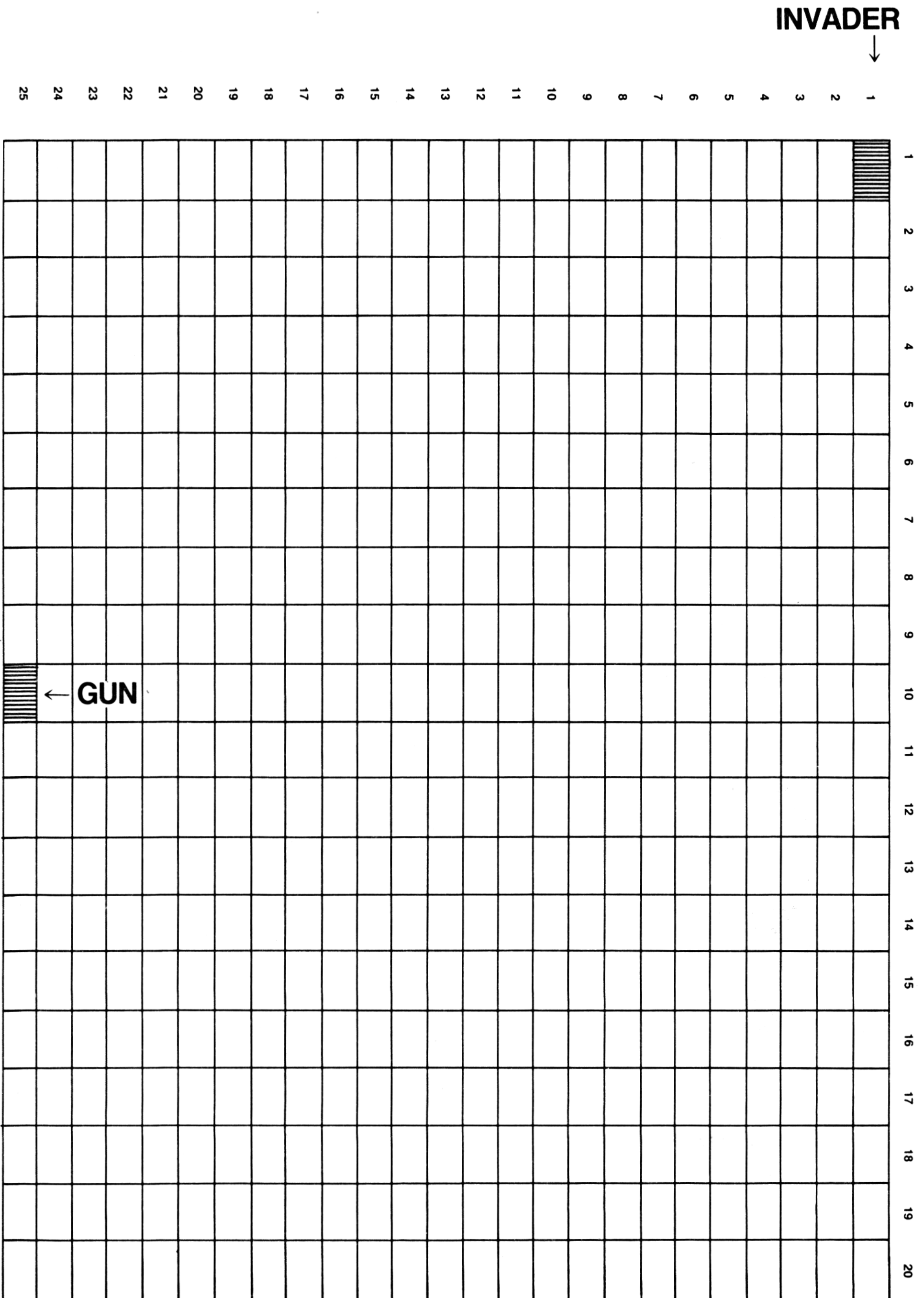
Start address 30041
End address 30097
HEX total 5619

```
2601     260 SETUP  LD     H,1
2E01     270        LD     L,1
221879   280        LD     (INVPOS),HL
3E01     290        LD     A,1
321E79   300        LD     (INVDIR),A
260A     310        LD     H,10
2E19     320        LD     L,25
221A79   330        LD     (GUNPOS),HL
26FF     340        LD     H,255
2EFF     350        LD     L,255
221C79   360        LD     (BULPOS),HL
CD00BC   370        CALL   48128
3E00     380        LD     A,0
CD0EBC   390        CALL   48142
2A1879   400        LD     HL,(INVPOS)
CD75BB   410        CALL   47989
3EE0     420        LD     A,224
CD5ABB   430        CALL   47962
2A1A79   440        LD     HL,(GUNPOS)
CD75BB   450        CALL   47989
3EF4     460        LD     A,244
CD5ABB   470        CALL   47962
C9       480        RET
```

The SETUP routine can be tested with the following lines of BASIC:

```
1000 CALL 30041
1005 GOTO 1005
```

You should see the Invader and gun printed in the positions shown in the diagram below:



Move Gun Routine

This routine determines whether the move left or right keys are being pressed; it then moves the gun in the appropriate direction. (Key A is used to move left and key D is used to move right.) I will now describe the complete action of the move gun routine:

1) Register A is loaded with 69 (key A) and then the test key routine at location 47902 is called. If the key is not being pressed then the program jumps to step 5 below to see if key D is being pressed.
2) Assuming that key A is being pressed, we do the following. We first need to see if the gun is at the extreme left edge of the screen. If it is then it cannot be moved any further and the program returns to the main gameloop. We test for the left-hand edge by loading register A with the current column of the gun. Register A is then compared to the number one using the instruction CP 1. If the answer is zero – meaning that the gun is in column one – then the program returns to the gameloop.
3) Having established that the gun is not at the extreme left edge of the screen we can now move it one position to the left. Registers H and L are already loaded with the current row and column positions from the previous step. So we call the position cursor routine at location 47989. Then, register A is loaded with the number 32 and the print a character routine at location 47962 is called. (The number 32 represents a space, so the above action has the effect of blanking out the Space Invader.)
4) Now the position of the gun needs to be moved one column to the left. This is done by loading HL, (GUNPOS); then the value of the register H, which holds the column position of the gun, is decremented. The new value of HL is then stored back in the gun position variable using the instruction LD (GUNPOS), HL. We then call the cursor positioning routine to put the cursor one position to the left of where the gun was. Register A is then loaded with 244 (the number of the gun character), and the print a character routine is called. The gun has now been moved one position to the left, and the program returns to the main gameloop.
5) If, at the start of the move gun routine, key A is not being

pressed then the program jumps here to see if the move right key is being pressed. If it is not, the program returns to the main gameloop.

6) The routine for moving the gun right is very similar to that for moving the gun left. The obvious differences are that we must check for the extreme right of the screen (that is, column 20) and we must increment the current column of the gun.

Start address 30311
End address 30386
HEX total 9294

```
3E45     1540 MOVGUN LD    A,69
CD1EBB   1550        CALL  47902
CA8E76   1560        JP    Z,MOVGR
2A1A79   1570        LD    HL,(GUNPOS)
7C       1580        LD    A,H
FE01     1590        CP    1
C8       1600        RET   Z
CD75BB   1610        CALL  47989
3E20     1620        LD    A,32
CD5ABB   1630        CALL  47962
2A1A79   1640        LD    HL,(GUNPOS)
25       1650        DEC   H
221A79   1660        LD    (GUNPOS),HL
CD75BB   1670        CALL  47989
3EF4     1680        LD    A,244
CD5ABB   1690        CALL  47962
C9       1700        RET
3E3D     1710 MOVGR  LD    A,61
CD1EBB   1720        CALL  47902
C8       1730        RET   Z
2A1A79   1740        LD    HL,(GUNPOS)
7C       1750        LD    A,H
FE14     1760        CP    20
C8       1770        RET   Z
CD75BB   1780        CALL  47989
3E20     1790        LD    A,32
CD5ABB   1800        CALL  47962
2A1A79   1810        LD    HL,(GUNPOS)
24       1820        INC   H
221A79   1830        LD    (GUNPOS),HL
CD75BB   1840        CALL  47989
3EF4     1850        LD    A,244
CD5ABB   1860        CALL  47962
C9       1870        RET
```

The move gun routines can be tested with the following lines of BASIC:

```
1000 CALL 30041
1005 CALL 30311
1010 GOTO 1005
```

After running the above program, pressing keys A or D will cause the gun to be moved left or right. (Press the escape key twice to stop the program running.)

The Delay Routine

The delay routine used in the Space Invader program is identical to the Universal Delay Routine described in the previous chapter and therefore needs no further explanation.

Start address **30395**
End address **30403**
HEX total **1217**

```
0D        1960 DELAY   DEC    C
C2BB76    1970         JP     NZ,DELAY
05        1980         DEC    B
C2BB76    1990         JP     NZ,DELAY
C9        2000         RET
```

It is very difficult to test the delay routine on its own but the simplest thing to do is CALL 30395. You should notice a delay before the READY cursor appears.

Move Invader Routine

The routine for moving the Space Invader is very similar to the move gun routine. The Invader is moved to the left or right, first by blanking out the Invader with a space character, then by incrementing or decrementing the current column according to the direction of the Invader. Finally the Space Invader is printed in the new position. The complete action of the move Invader routine is shown below:

1) Blank out the Space Invader by first positioning the cursor at the current position, then printing a space.
2) At this point we look to see if the Space Invader has been

hit by a bullet. This is done quite simply by comparing the current row and column positions of the Space Invader with those of the bullet; if they are the same the program jumps to a routine called HITHIM (I will describe this later on).

3) If the Space Invader has not been hit we need to decide in which direction to move it. This is done by looking at the variable Invader direction, (INVDIR). If it is set to one then the Invader is moving from left to right, if it is set to zero it is moving from right to left.
4) Let's say that the Invader is moving right. The procedure for doing this is identical to that for moving the gun right. There is one main difference in that when the right-hand edge of the screen is detected we set the INVDIR direction variable to zero, so that the next time the move Invader routine is called the move left routine will be executed. Again, moving the Invader left is similar to moving the gun left except that when the left-hand edge of the screen is detected the INVDIR variable is set to one.

Start address **30107**
End address **30218**
HEX total **14210**

```
2A1879     580 MOVINV  LD    HL,(INVPOS)
CD75BB     590         CALL  47989
3E20       600         LD    A,32
CD5ABB     610         CALL  47962
2A1879     620         LD    HL,(INVPOS)
ED5B1C79   630         LD    DE,(BULPOS)
A7         640         AND   A
ED52       650         SBC   HL,DE
CAF375     660         JP    Z,HITHIM
3A1E79     670         LD    A,(INVDIR)
FE01       680         CP    1
CAD775     690         JP    Z,MOVRGT
2A1879     700         LD    HL,(INVPOS)
7C         710         LD    A,H
FE01       720         CP    1
C2CA75     730         JP    NZ,LEFT
3E01       740         LD    A,1
321E79     750         LD    (INVDIR),A
C9         760         RET
25         770 LEFT    DEC   H
221879     780         LD    (INVPOS),HL
```

```
CD75BB    790          CALL   47989
3EE0      800          LD     A,224
CD5ABB    810          CALL   47962
C9        820          RET
2A1879    830 MOVRGT   LD     HL,(INVPOS)
7C        840          LD     A,H
FE14      850          CP     20
C2E675    860          JP     NZ,RIGHT
3E00      870          LD     A,0
321E79    880          LD     (INVDIR),A
C9        890          RET
24        900 RIGHT    INC    H
221879    910          LD     (INVPOS),HL
CD75BB    920          CALL   47989
3EE0      930          LD     A,224
CD5ABB    940          CALL   47962
C9        950          RET
2A1879    960 HITHIM   LD     HL,(INVPOS)
CD75BB    970          CALL   47989
3EE1      980          LD     A,225
CD5ABB    990          CALL   47962
01FFFF    1000         LD     BC,65535
CDBB76    1010         CALL   DELAY
CDBB76    1020         CALL   DELAY
CDBB76    1030         CALL   DELAY
C9        1040         RET
```

Move Bullet Routine

The next routine I shall describe is called the move bullet routine. In actual fact it does a little more than just move the bullet up the screen, as you will see:

1) If you refer back to the SETUP routine you will see that we set the column and row positions of the bullet to 255. The fact that the column position is 255 indicates that the bullet has not yet been fired. So the first thing that the move bullet routine must do is to determine whether the column position of the bullet is 255; if it is, then the program jumps to step 5 below (this ascertains whether the key which is used to fire the bullet is being pressed).
2) Assuming that the bullet has been fired, it must be somewhere on the screen. All we have to do now is move it upwards one position. Registers H and L are already loaded

with the current position of the bullet so we simply call the position cursor routine, load register A with 32, then call the print a character routine to blank out the bullet.

3) Next, register L is decremented. This is used to move the cursor up to the next row, but it also checks to see whether the bullet has reached the top of the screen. Let's say that the bullet was last printed on row one; in this case, by decrementing L, the zero flag will be set. The instruction after DEC L is JP NZ,MOVBL1; this is a small routine which reprints the bullet. If, however, the result of decrementing register L was zero, then we know that the bullet has reached the top of the screen and there is no need to reprint it. The row and column positions of the bullet are then set to 255; this indicates that the bullet is not active. The program then returns to the gameloop.
4) Assuming that the bullet is not at the top of the screen all we have to do is to print it in its new position. This is done by positioning the cursor, and then calling the print a character routine. The program then returns to the gameloop.
5) The program jumps to this step if the column position of the bullet was found to be 255 in step 1. A test is made to determine whether the fire key is being pressed. This is done by loading register A with 18 (18 is the code for the ENTER key which we are using to represent the fire button), and calling the test key routine at location 47902. If the key is not being pressed then the program jumps to a small routine called SLOW1. This is a small delay to compensate for the fact that the program does not have to do anything if the key is not being pressed.
6) If the fire button is being pressed then the bullet must be printed directly above the gun. This is done by loading register pair HL with the current gun position, decrementing the value of register L, then loading the variable BULPOS with the value of HL. Finally, the bullet is printed by calling the print a character routine. The program then returns to the gameloop.

Start address 30227
End address 30301
HEX total 9078

```
2A1C79   1130 MOVBUL LD     HL,(BULPOS)
7C       1140        LD     A,H
FEFF     1150        CP     255
CA3F76   1160        JP     Z,TRYBUL
CD75BB   1170        CALL   47989
3E20     1180        LD     A,32
CD5ABB   1190        CALL   47962
2A1C79   1200        LD     HL,(BULPOS)
2D       1210        DEC    L
C23376   1220        JP     NZ,MOVBL1
26FF     1230        LD     H,255
2EFF     1240        LD     L,255
221C79   1250        LD     (BULPOS),HL
C9       1260        RET
221C79   1270 MOVBL1 LD     (BULPOS),HL
CD75BB   1280        CALL   47989
3EEF     1290        LD     A,239
CD5ABB   1300        CALL   47962
C9       1310        RET
3E12     1320 TRYBUL LD     A,18
CD1EBB   1330        CALL   47902
CA5776   1340        JP     Z,SLOW1
2A1A79   1350        LD     HL,(GUNPOS)
2D       1360        DEC    L
221C79   1370        LD     (BULPOS),HL
CD75BB   1380        CALL   47989
3EEF     1390        LD     A,239
CD5ABB   1400        CALL   47962
C9       1410        RET
010002   1420 SLOW1  LD     BC,512
CDBB76   1430        CALL   DELAY
C9       1440        RET
```

The move bullet routine can be tested with the following lines of BASIC:

```
1000 CALL
1005 CALL
1010 GOTO 1005
```

RUN the above tester program, press the enter key and you will see the bullet move up the screen.

Gameloop Routine

The final routine to be entered is the actual gameloop. The purpose of the gameloop is constantly to call all the routines that we have now entered. The actual operation of the

gameloop is shown below:

1) When the game is started by CALL 30000 the first thing to be executed is the SETUP routine.
2) The main part of the gameloop consists of calling three routines as shown here:

CALL MOVE INVADER
CALL MOVE BULLET
CALL MOVE GUN
CALL MOVE BULLET

The move bullet routine is called twice in the gameloop so that it can travel twice as fast as the gun and the Invader.
3) A check is then made to see if key X is being pressed. If key X is being pressed we return to BASIC by jumping to a routine at location 42075. The routine at location 42075 ensures that the characters we have been pressing do not appear on the screen when we return to BASIC.
4) If key X is not being pressed then a long delay is caused by loading register BC with 4000 and calling the delay routine. If this long delay was omitted the game would run so fast that it would be difficult to see what was happening. After the delay, the program jumps back to the start of the gameloop (step 2 above).

Start address **30000**
End address **30031**
HEX total **3978**

```
CD5975    60 START   CALL   SETUP
CD9B75    70 LOOP    CALL   MOVINV
CD1376    80         CALL   MOVBUL
CD6776    90         CALL   MOVGUN
CD1376   100         CALL   MOVBUL
3E3F     110         LD     A,63
CD1EBB   120         CALL   47902
C203BB   130         JP     NZ,47875
01A00F   140         LD     BC,4000
CDBB76   150         CALL   DELAY
C33375   160         JP     LOOP
```

The whole Space Invader game can now be tested with the following lines of BASIC:

```
1000 CALL 3000
1005 STOP
```

I have assumed in this chapter that you will be able to sit down and enter the whole program in one go. However, if you only have time to enter one or two routines, then here is the procedure for saving a routine once it has been entered and tested; it can then be loaded at a later date. Here is the procedure for saving the SETUP routine:

1) Stop the HEXENT program by pressing key Q.
2) SAVE "SETUP",B,30041,57

a) B says that you are saving a block of code.
b) 30041 is the start address of the routine.
c) 57 is the length of the routine which is calculated by subtracting the start address from the end address and adding one.

3) When you wish to carry on entering the rest of the program you must load HEXENT, then load any routines you have saved. You can now RUN HEXENT and continue to enter the rest of the program.

A FEW USEFUL ROUTINES

A Few Useful Routines

This section 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, and it is therefore possible to have more than one routine in memory at the same time.

Name: Scroll a row of text to the left.

Function: This routine will enable you to scroll any of the 25 rows of text in any of the screen modes.

Requirements: You must tell the routine which of the 25 rows to scroll by adding the row number at the end of the CALL instruction. To scroll line 13 you would type CALL 30250,13. If the computer is in mode 2, then calling the routine will move the row one character position to the left. If mode 1 is being used then the routine must be called twice in order to move the row one position. And in mode 0 the routine must be called four times to effect a one character movement.

Start address **30250**
End address **30299**
HEX total **6817**

```
DD6E00    20           LD      L, (IX+0)
2600      30           LD      H,0
CD1ABC    40           CALL    48154
E5        50 LOOP 1    PUSH    HL
7E        60           LD      A, (HL)
F5        70           PUSH    AF
54        80           LD      D,H
5D        90           LD      E,L
EB        100          EX      DE,HL
CD5676    110          CALL    INCPTR
0E4F      120          LD      C,79
7E        130 LOOP 2   LD      A, (HL)
12        140          LD      (DE) ,A
CD5676    150          CALL    INCPTR
EB        160          EX      DE,HL
```

```
CD5676    170          CALL   INCPTR
EB        180          EX     DE,HL
0D        190          DEC    C
20F3      200          JR     NZ,LOOP2
F1        210          POP    AF
12        220          LD     (DE),A
E1        230          POP    HL
7C        240          LD     A,H
C608      250          ADD    A,8
67        260          LD     H,A
E638      270          AND    56
20DD      280          JR     NZ,LOOP1
C9        290          RET
23        300 INCPTR   INC    HL
CBF4      310          SET    6,H
CBFC      320          SET    7,H
C9        330          RET
```

Name: Scroll a row of text to the right.

This routine is very similar to the scroll left routine except, of course, that the text is scrolled to the right. The requirements are the same as the scroll left routine. So to scroll row 22 to the right one character position while in screen mode 0 you would have to type the following:

```
CALL 30350,22
CALL 30350,22
CALL 30350,22
CALL 30350,22
```

Start address **30350**
End address **30406**
HEX total **7903**

```
DD6E00    20           LD     L, (IX+0)
2600      30           LD     H,0
CD1ABC    40           CALL   48154
014F00    50           LD     BL,79
09        60           ADD    HL,BC
CBF4      70           SET    6,H
CBFC      80           SET    7,H
E5        90 LOOP 1    PUSH   HL
7E        100          LD     A, (HL)
F5        110          PUSH   AF
54        120          LD     D,H
```

```
5D        130           LD    E,L
CDC176    150           CALL  DECPTR
0E4F      160           LD    C,79
7E        170 LOOP 2    LD    A, (HL)
12        180           LD    (DE) ,A
CDC176    190           CALL  DECPTR
EB        200           EX    DE,HL
CDC176    210           CALL  DECPTR
EB        220           EX    DE,HL
0D        230           DEC   C
20F3      240           JR    NZ,LOOP2
F1        250           POP   AF
12        260           LD    (DE),A
E1        270           POP   HL
7C        280           LD    A,H
C608      290           ADD   A,8
67        300           LD    H,A
E638      310           ADD   56
20DE      320           JR    NZ,LOOP1
C9        330           RET
2B        340 DECPTR    DEC   HL
CBF4      350           SET   6,H
CBFC      360           SET   7,H
C9        370           RET
```

Name: Laser Sound

Function: When activated by using CALL 30000, this routine will make a noise – similar to sound effects in games – which represents a laser gun being fired.

Requirement: The volume control must be turned to maximum.

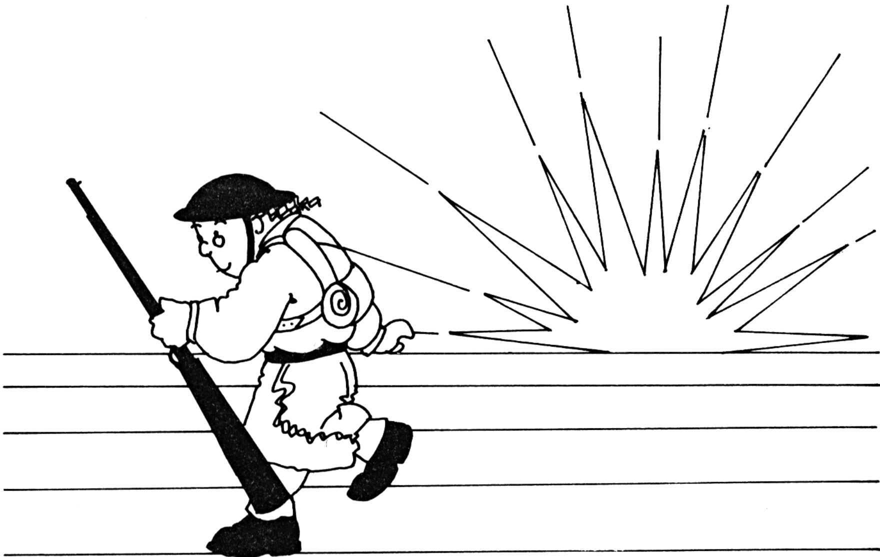
Start address **30000**
End address **30069**
HEX total **7089**

```
3E08      20          LD     A,8
0E0F      30          LD     C,15
CD34BD    40          CALL   48436
3E07      50          LD     A,7
0E3E      60          LD     C,62
CD34BD    70          CALL   48436
3E00      80          LD     A,0
0E6E      90          LD     C,110
CD34BD    100         CALL   48436
3E01      110         LD     A,1
0E00      120         LD     C,0
CD34BD    130         CALL   48436
0E32      140         LD     C,50
3E00      150 LOOP    LD     A,0
C5        155         PUSH   BC
CD34BD    160         CALL   48436
3EC8      170         LD     A,200
F5        180 DELAY   PUSH   AF
3E1E      190         LD     A,30
3D        200 DEL     DEC    A
C25975    210         JP     NZ,DEL
F1        220         POP    AF
3D        230         DEC    A
C25675    240         JP     NZ,DELAY
C1        245         POP    BC
79        250         LD     A,C
C606      260         ADD    A,6
CA6E75    270         JP     Z,END
3D        280         DEC    A
4F        290         LD     C,A
C34E75    300         JP     LOOP
3E07      310 END     LD     A,7
0E3F      320         LD     C,63
CD34BD    330         CALL   48436
C9        340         RET
```

Name: Bomb Sound.

Function: When this routine is activated by using CALL 30100 it will make the whistling sound of a falling bomb, followed by an explosion.

Start address **30100**
End address **30216**
HEX total **14897**



```
3E07     20        LD     A,7
0E3E     30        LD     C,62
CD34BD   40        CALL   48436
3E08     50        LD     A,8
0E0F     60        LD     C,15
CD34BD   70        CALL   48436
0E28     80        LD     C,40
3E00     90 LOOP   LD     A,0
C5       100       PUSH   BC
CD34BD   110       CALL   48436
3E0A     120       LD     A,10
F5       130 DELAY PUSH   AF
3EFF     140       LD     A,255
3D       150 DEL   DEC    A
C2AF75   160       JP     NZ,DEL
F1       170       POP    AF
3D       180       DEC    A
C2AC75   190       JP     NZ,DELAY
C1       200       POP    BC
79       210       LD     A,C
D6C8     220       SUB    200
CAC575   230       JP     Z,NEXT
C6C9     240       ADD    A,201
4F       250       LD     C,A
C3A475   260       JP     LOOP
3E00     270 NEXT  LD     A,0
0E00     280       LD     C,0
CD34BD   290       CALL   48436
3E07     300       LD     A,7
```

```
0E37     310         LD    C,55
CD34BD   320         CALL  48436
3E00     330         LD    A,0
F5       340 NEXT1   PUSH  AF
4F       350         LD    C,A
CD34BD   360         CALL  48436
3E32     370         LD    A,50
F5       380 STEVE   PUSH  AF
3EFF     390         LD    A,255
3D       400 TRACEY  DEC   A
C2DF75   410         JP    NZ,TRACEY
F1       420         POP   AF
3D       430         DEC   A
C2DC75   440         JP    NZ,STEVE
F1       450         POP   AF
D61F     460         SUB   31
CAF375   470         JP    Z,LDEL
C620     480         ADD   A,32
C3D575   490         JP    NEXT1
3E64     500 LDEL    LD    A,100
F5       510 LDEL1   PUSH  AF
3EFF     520         LD    A,255
3D       530 LDEL2   DEC   A
C2F875   540         JP    NZ,LDEL2
F1       550         POP   AF
3D       560         DEC   A
C2F575   570         JP    NZ,LDEL1
3E07     580         LD    A,7
0E3F     590         LD    C,63
CD34BD   600         CALL  48436
C9       610         RET
```

APPENDICES

Appendices

1) Z80 OPCODES
2) DECIMAL TO HEX CONVERSION CHART
3) BINARY
4) KEYBOARD LAYOUT
5) SCREEN MODES
6) CHARACTER DESIGNER
7) ASSEMBLERS
8) MORE MACHINE CODE INSTRUCTIONS AND ROUTINES
9) MULTIPLICATION
10) MORE USEFUL ROM ROUTINES
11) AUTHOR'S NOTE
12) ANSWERS

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

Instruction	Opcode
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

Instruction	Opcode
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	SUB L	95
SRL E	CB3B	SUB n	D6n
SRL H	CB3C	XOR (HL)	AE
SRL L	CB3D	XOR (IX + d)	DDAEd
SUB (HL)	96	XOR (IY + d)	FDAEd
SUB (IX + d)	DD96d	XOR A	AF
SUB (IY + d)	FD96d	XOR B	A8
SUB A	97	XOR C	A9
SUB B	90	XOR D	AA
SUB C	91	XOR E	AB
SUB D	92	XOR H	AC
SUB E	93	XOR L	AD
SUB H	94	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 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, and these are numbered in the following way:

8 bits make one byte	7	6	5	4	3	2	1	0

These eight bits can be thought of as 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

Thus, 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 when bit 1 is on it 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, since 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.)

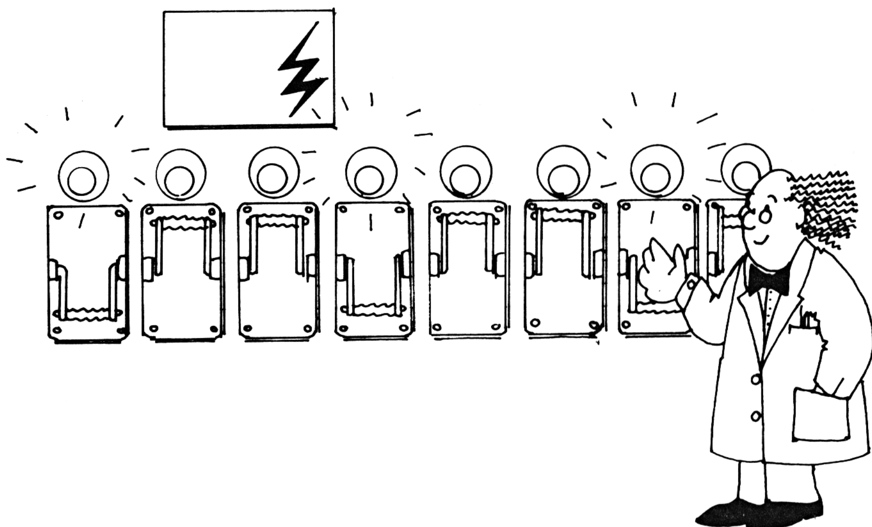
There are two main opcodes which are associated with binary:

OPCODE: SET b,R

R represents any of the seven single registers; b must be a number between zero and seven and is the bit of the register that you wish to set. The following example will show you how to use this opcode. Let's say that register A has a value of zero. If we execute the opcode SET 3,A then register A will now have a value of 8. If you look at the previous diagrams you will see that when bit number 3 is on – that is, when it is SET to one – it represents a value of 8. You can set any of the 8 bits of any of the seven single registers; thus the opcode SET 6,E means set the sixth bit of register E to one. If a bit is already set to one then setting it again will have no effect.

OPCODE: RES b,R

This opcode is exactly the opposite of SET. If you execute the instruction RES 3, A then the third bit of register A will be RESET to zero; if the bit was already zero then the opcode will have no effect.



Keyboard Layout

The following diagram shows the numerical code of all the keys:

MAIN KEYBOARD

66 64 65 57 56 49 48 41 40 33 32 25 24 16 79

68 67 59 58 50 51 43 42 35 34 27 26 17 18

70 69 60 61 53 52 44 45 37 36 29 28 19

21 71 63 62 55 54 46 38 39 31 30 22 21

47 23

NUMERIC KEYPAD

10	11	3
20	12	4
13	14	5
15	7	6

CURSOR KEYS

	0	
8	9	1
	2	

Screen Mode 0 (20 columns × 25 rows)

Screen Mode 1 (40 columns × 25 rows)

Screen Mode 2 (80 columns × 25 rows)

Character Designer

This program will enable you to re-design the shapes of the characters from character number 32 – the SPACE up – to character number 255 – which is a special symbol. You will then be able to save the new character set on tape for future use.

Start address **30000**
End address **30364**
HEX total **43837**

```
AF        20        XOR   A
327A76    30        LD    (CURCHR),A
327B76    40        LD    (CURLIN),A
327C76    50        LD    (CURCOL),A
3E01      60        LD    A,1
CD0EBC    70        CALL  48142
CDB975    80 LOOP   CALL  GRID
CDFA75    90        CALL  CURSOR
CD1776    100       CALL  CHRNUM
CD5576    110       CALL  DRCHAR
CD18BB    120       CALL  47896
FE0D      130       CP    13
C8        140       RET   Z
213F75    150       LD    HL,LOOP
E5        160       PUSH  HL
FEF0      170       CP    240
CA7975    180       JP    Z,CURUP
FEF1      190       CP    241
CA8375    200       JP    Z,CURDWN
FEF2      210       CP    242
CA8E75    220       JP    Z,CURLEF
FEF3      230       CP    243
CA9875    240       JP    Z,CURRIG
FE20      250       CP    " "
CAA375    260       JP    Z,SWITCH
FE4E      270       CP    "N"
CAA975    280       JP    Z,NEXT
FE50      290       CP    "P"
CAB175    300       JP    Z,PREV
C9        310       RET
3A7B76    320 CURUP LD    A,(CURLIN)
A7        330       AND   A
C8        340       RET   Z
3D        350       DEC   A
327B76    360       LD    (CURLIN),A
```

```
C9        370           RET
3A7B76    380 CURDWN    LD      A,(CURLIN)
FE07      390           CP      7
C8        400           RET     Z
3C        410           INC     A
327B76    420           LD      (CURLIN),A
C9        430           RET
3A7C76    440 CURLEF    LD      A,(CURCOL)
A7        450           AND     A
C8        460           RET     Z
3D        470           DEC     A
327C76    480           LD      (CURCOL),A
C9        490           RET
3A7C76    500 CURRIG    LD      A,(CURCOL)
FE07      510           CP      7
C8        520           RET     Z
3C        530           INC     A
327C76    540           LD      (CURCOL),A
C9        550           RET
CD3B76    560 SWITCH    CALL    PIXPOS
AE        570           XOR     (HL)
77        580           LD      (HL),A
C9        590           RET
3A7A76    600 NEXT      LD      A,(CURCHR)
3C        610           INC     A
327A76    620           LD      (CURCHR),A
C9        630           RET
3A7A76    640 PREV      LD      A, (CURCHR)
3D        650           DEC     A
327A76    660           LD      (CURCHR),A
C9        670           RET
3A7A76    680 GRID      LD      A,(CURCHR)
6F        690           LD      L,A
2610      700           LD      H,16
29        710           ADD     HL,HL
29        720           ADD     HL,HL
29        730           ADD     HL,HL
EB        740           EX      DE,HL
2100C0    750           LD      HL,49152
0608      760           LD      B,8
C5        770 GRIDL1    PUSH    BC
D5        780           PUSH    DE
E5        790           PUSH    HL
1A        800           LD      A,(DE)
4F        810           LD      C,A
0608      820           LD      B,8
E5        830  GRIDL2   PUSH    HL
117D76    840           LD      DE,CHAR1
CB11      850           RL      C
3003      860           JR      NC,GRIDL3
```

```
118D76   870          LD    DE,CHAR2
1A       880  GRIDL3  LD    A,(DE)
77       890          LD    (HL),A
13       900          INC   DE
23       910          INC   HL
1A       920          LD    A,(DE)
77       930          LD    (HL),A
13       940          INC   DE
2B       950          DEC   HL
7C       960          LD    A,H
C608     970          ADD   A,8
67       980          LD    H,A
E638     990          AND   56
20F0     1000         JR    NZ,GRIDL3
E1       1010         POP   HL
23       1020         INC   HL
23       1030         INC   HL
10E0     1040         DJNZ  GRIDL2
E1       1050         POP   HL
115000   1060         LD    DE,80
19       1070         ADD   HL,DE
D1       1080         POP   DE
13       1090         INC   DE
C1       1100         POP   BC
10CF     1110         DJNZ  GRIDL1
C9       1120         RET
3A7B76   1130 CURSOR  LD    A,(CURLIN)
3C       1140         INC   A
6F       1150         LD    L,A
3A7C76   1160         LD    A,(CURCOL)
3C       1170         INC   A
67       1180         LD    H,A
CD75BB   1190         CALL  47989
CD3B76   1200         CALL  PIXPOS
A6       1210         AND   (HL)
D601     1220         SUB   1
9F       1230         SBC   A,A
2F       1240         CPL
E676     1250         AND   118
EE9F     1260         XOR   159
CD5ABB   1270         CALL  47962
C9       1280         RET
21010C   1290 CHRNUM  LD    HL,3073
CD75BB   1300         CALL  47989
3A7A76   1310         LD    A,(CURCHR)
4F       1320         LD    C,A
0664     1330         LD    B,100
CD2D76   1340         CALL  CHRNL1
060A     1350         LD    B,10
CD2D76   1360         CALL  CHRNL1
```

```
0601     1370          LD    B,1
162F     1380 CHRNL1   LD    D,47
79       1390          LD    A,C
90       1400 CHRNL2   SUB   B
14       1410          INC   D
30FC     1420          JR    NC,CHRNL2
80       1430          ADD   A,B
4F       1440          LD    C,A
7A       1450          LD    A,D
CD5ABB   1460          CALL  47962
C9       1470          RET
3A7A76   1480 PIXPOS   LD    A,(CURCHR)
6F       1490          LD    L,A
2610     1500          LD    H,16
29       1510          ADD   HL,HL
29       1520          ADD   HL,HL
29       1530          ADD   HL,HL
3A7B76   1540          LD    A,(CURLIN)
85       1550          ADD   A,L
6F       1560          LD    L,A
3A7C76   1570          LD    A,(CURCOL)
47       1580          LD    B,A
3E00     1590          LD    A,0
37       1600          SCF
04       1610          INC   B
1F       1620 PIXPL1   RRA
10FD     1630          DJNZ  PIXPL1
C9       1640          RET
         1650
3A7A76   1660 DRCHAR   LD    A,(CURCHR)
6F       1670          LD    L,A
2610     1680          LD    H,16
29       1690          ADD   HL,HL
29       1700          ADD   HL,HL
29       1710          ADD   HL,HL
EB       1720          EX    DE,HL
215AC1   1730          LD    HL,49498
0608     1740          LD    B,8
1A       1750 DRCHL1   LD    A,(DE)
E6F0     1760          AND   240
77       1770          LD    (HL),A
23       1780          INC   HL
1A       1790          LD    A,(DE)
17       1800          RLA
17       1810          RLA
17       1820          RLA
17       1830          RLA
E6F0     1840          AND   240
77       1850          LD    (HL),A
13       1860          INC   DE
```

```
2B       1870         DEC    HL
7C       1880         LD     A,H
C608     1890         ADD    A,8
67       1900         LD     H,A
10EB     1910         DJNZ   DRCHL1
C9       1920         RET
00       1930 CURCHR  DEFB   0
00       1940 CURLIN  DEFB   0
00       1950 CURCOL  DEFB   0
F0F0C030 1960 CHAR1   DEFB   240,240,192,48
C030C030 1970         DEFB   192,48,192,48
C030C030 1980         DEFB   192,48,192,48
C030F0F0 1990         DEFB   192,48,240,240
F0F0F0F0 2000 CHAR2   DEFB   240,240,240,240
F0F0F0F0 2010         DEFB   240,240,240,240
F0F0F0F0 2020         DEFB   240,240,240,240
F0F0F0F0 2030         DEFB   240,240,240,240
```

Having entered and thoroughly checked the above program you should have it on tape, first by pressing Q while at the menu stage of HEXENT then by typing in the following command:

SAVE"CHDES",B,30000,365

When you wish to reload the character designer simply type in:

LOAD"",30000

To use the character designer type CALL 30000 and you will see an 8 × 8 grid appear on the screen together with the number 0. The number represents the character we can currently re-design. Character numbers 0 to 31 are special control characters which cannot be printed and are not worth re-designing. By pressing key N (next) you can step on to the next character. Pressing key P (previous) will step you on to the previous character.

On the grid you will notice a cursor; by using the four cursor control keys this can be moved around. By pressing the space bar you can turn the pixel – which is under the cursor – on and off. So, by using the cursor control keys and the space bar you can design your own characters. When you have designed all

the characters, you can save them on tape, first by pressing the ENTER key to exit from the designer program then by typing in the following command:

SAVE "CHARS",B,32768,2048

When you wish to use the re-designed characters in your own programs you must do the following:

X = HIMEM + 1 (ENTER)
LOAD"CHARS",X

Assemblers

If you wish to become a serious Machine Code programmer you will have to invest in a programmers' aid called an ASSEMBLER. An Assembler will allow you to enter Machine Code routines in mnemonics that are fairly easily understood. If you look at the program listings in this book you will see that the right-hand columns are easy to understand compared to the HEX numbers on the left that you have had to type in. If you look at the first line of the SETUP routine you will see that the right-hand column says LD H,1; this is far easier to understand than 2601 – its HEX representation. An Assembler allows you to enter the instructions exactly as shown in the right-hand columns – LD H,1 for instance. An Assembler also takes away the task of having to calculate jump addresses. In short, an Assembler is well worth the £10 to £15 that you will have to pay for it.

More Machine Code Instructions and Routines

LDIR

This is a very powerful instruction. It is used when you wish to copy a block of data from one area of memory to another area. Suppose that you want to move 100 bytes which start at location 30000 to the area which starts at location 39000. The following example shows how to do this using the LDIR instruction:

Step 1) Load register HL with the start location of the data to be moved.
Step 2) Load register DE with the start location of the area of memory where the data is to be copied.
Step 3) Load register BC with the length of the data to be moved.
Step 4) Execute the instruction LDIR.

As I mentioned earlier on, LDIR is very powerful and is the same as the following routine:

```
START:  LD A,(HL)
        LD (DE),A
        INC HL
        INC DE
        DEC BC
        If BC has not reached
        zero then jump to the
        start of this routine.
```

When using the LDIR instruction you must take care that the two areas of memory do not overlap in such a way that the original data would be corrupted before it has been copied. The following example shows how this situation would occur.

Suppose that there are five bytes of data starting at location 30000 which you want to move to the area starting at location 30003. To do this we would use the following routine:

```
LD HL, 30000
LD DE,30003
LD BC,5
LDIR
```

The following diagram shows the contents of the five locations before executing the above routine:

Location	Value
30000	5
30001	6
30002	7
30003	8
30004	9

The following diagram shows what would be contained in the five locations starting at location 30003 after executing the above routine:

Location	Value
30003	5
30004	6
30005	7
30006	5
30007	6

It is obvious that the first area has not been successfully copied into the second area. Very simply, the reason why the original values of locations 30003/4 were not copied into locations 30006/7 is that location 30003 was overwritten with what was in location 30000 and location 30004 was overwritten with what was in location 30001. The following instruction LDDR will show you how it is possible to copy an area of memory into another area of memory which overlaps the original area as in the above example.

LDDR

The instruction LDDR is very similar to LDIR, the difference being that registers HL and DE are decremented instead of being incremented. The following example shows how to copy the area of memory which starts at location 35000 into the area starting at location 30000. The length of the data to be moved is 50 bytes.

Step 1) Load register HL with the end address of the data to be copied. In this example it is 35049.
Step 2) Load register DE with the end address of the area of

memory into which the data is to be copied.

Step 3) Load register BC with the length of the data to be moved.

Step 4) Execute the instruction LDDR.

The LDDR instruction can also be used to copy one area of memory into another area of memory when – using the LDIR instruction – it would normally corrupt the original data before it had been copied. If we look at the example where we wanted to copy the five bytes of data starting at location 30000 to the area of memory starting at location 30003 – but which we were unable to do using the LDIR instruction – we can now do this by using the LDDR instruction as shown in the following example:

Step 1) Load register HL with the end address of the data to be copied, which in this example is 30004.

Step 2) Load register DE with the end address of the area of memory into which the data is to be copied; in this example it is 30007.

Step 3) Load register BC with the length of the data to be moved, which in this case is 5.

Step 4) Execute the instruction LDDR.

The original data will now have been successfully copied into the new area. The original data will still have been corrupted, however, but not before it had been copied.

Multiplication

In Machine Code programming there are no specific instructions for performing multiplication, so we have to write our own routines to do so. I will first show you how to multiply two numbers where it is known that the answer will not be greater than 255; in this case we can use a single register to perform the calculation. The following examples show how to multiply the value of register A with the answer appearing in register A:

```
X2:   ADD A,A

X4:   ADD A,A
      ADD A,A

X8:   ADD A,A
      ADD A,A
      ADD A,A

X16:  ADD A,A
      ADD A,A
      ADD A,A
      ADD A,A

X3:   LD B,A
      ADD A,A
      ADD A,B

X5:   LD B,A
      ADD A,A
      ADD A,A
      ADD A,A

X6:   ADD A,A
      LD B,A
      ADD A,A
      ADD A,A

X7:   LD B,A
      ADD A,A
      ADD A,A
      ADD A,A
      SUB B
```

I'm sure that you can see from the previous examples that any numbers can be multiplied by a series of additions and

subtractions.

The previous examples also apply to register pairs. So to multiply the value of register pair HL simply do the following:

```
ADD HL,HL
ADD HL,HL
```

To multiply the value of HL by 5 do the following:

```
LD D,H
LD E,L
ADD HL,HL
ADD HL,HL
ADD HL,DE
```

If you are writing a program which requires a lot of multiplication to be performed then the following routine will be of use to you. It can be used to multiply any two numbers between 1 and 255 inclusive. The two numbers to be multiplied must be contained in registers HL and BC; you then call the subroutine and the answer will be returned in register HL.

```
START: DEC BC
       LD A,C
       OR B
       RET Z
       ADD HL,HL
       JP START
```

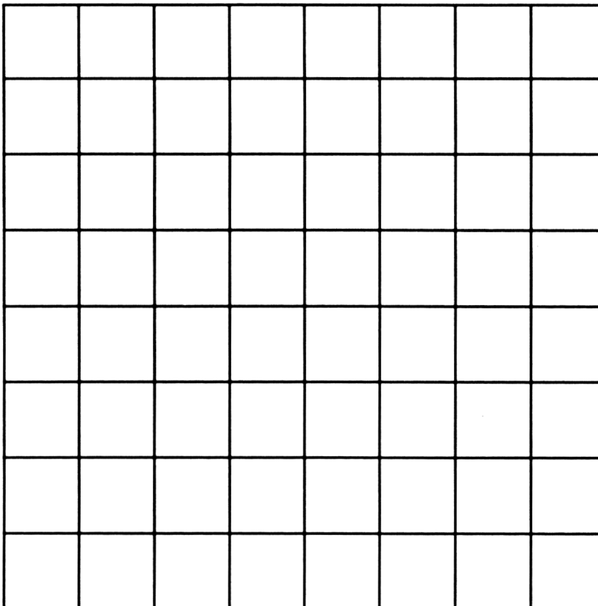
More Useful ROM Routines

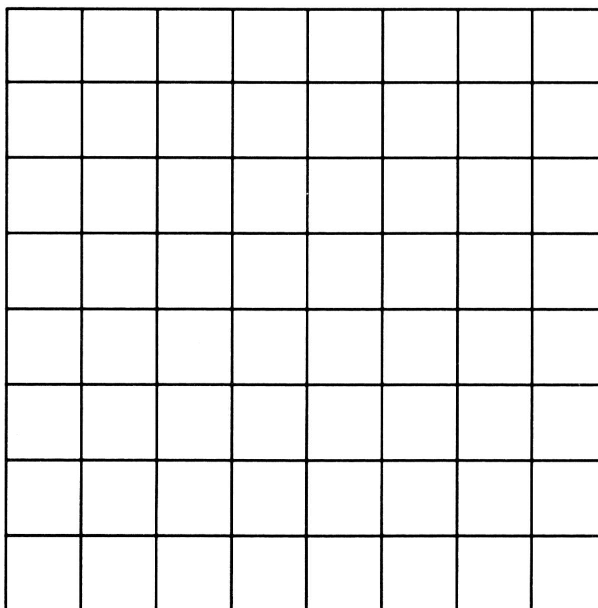
There is a routine in the ROM which you can call to determine the state of the CAPS LOCK and SHIFT LOCK. To access the routine simply CALL 47905. After calling the routine, register L will contain 0 if the SHIFT LOCK is off and 255 if it is on. Register H will contain 0 if the CAPS LOCK is off and 255 if it is on.

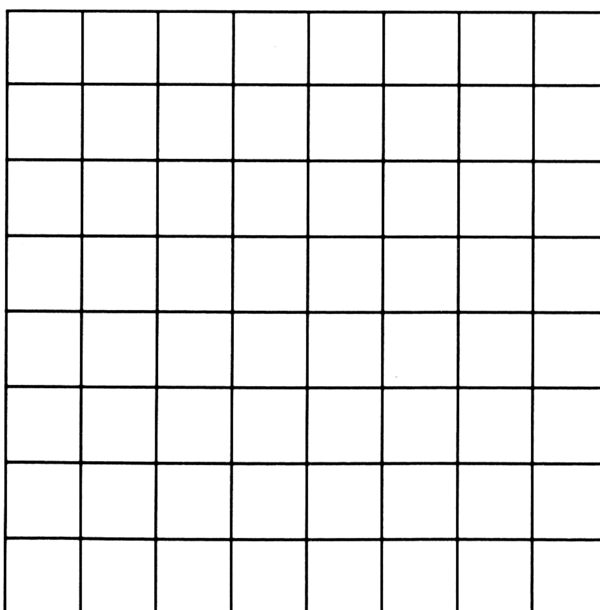
The routine at location 47968 can be used to find out what character is printed at the current cursor position. After calling the routine, register A will contain the ASCII value of the character at the cursor position. If there was not a recognisable character at the cursor position then the carry flag will not be set.

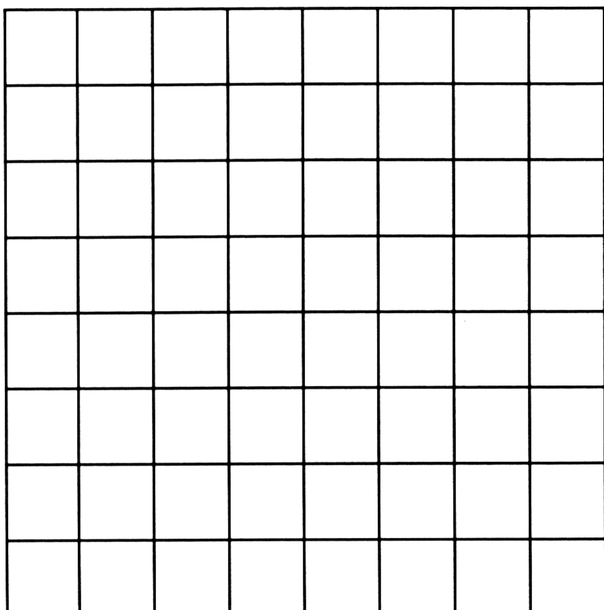
The routine at location 47992 can be used to find the current row and column position of the cursor. After calling the routine, register H will contain the column position and register L will contain the row position.

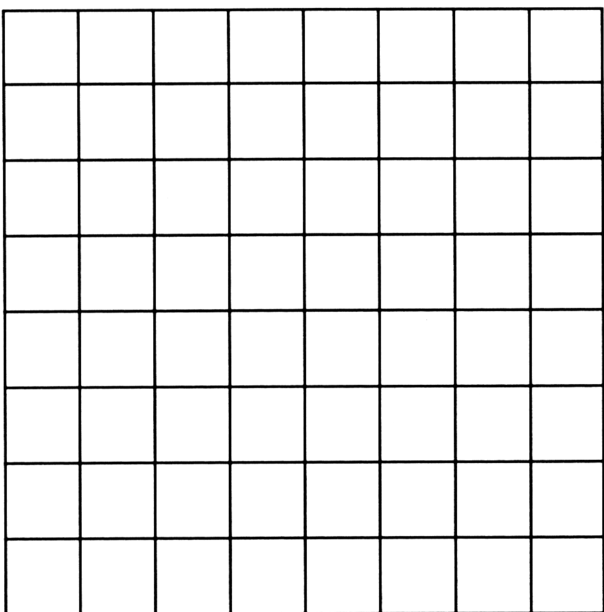
The following grids can be used to help you design your own characters:

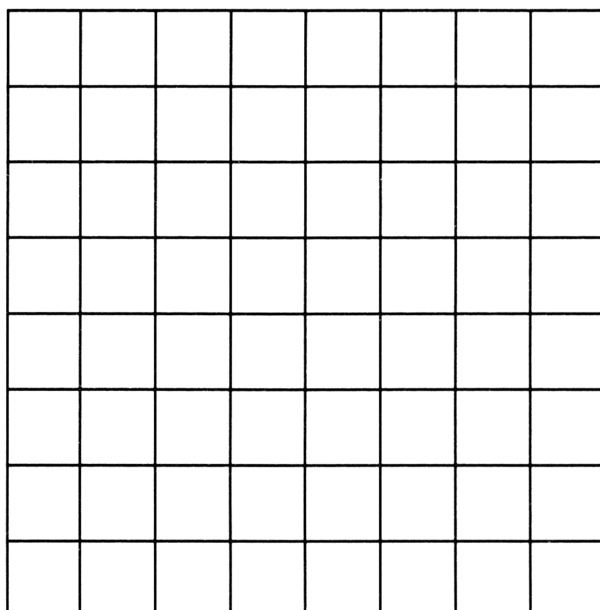


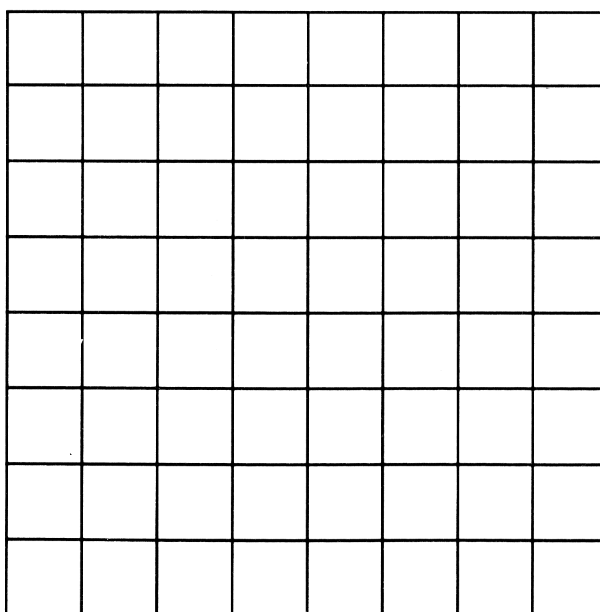


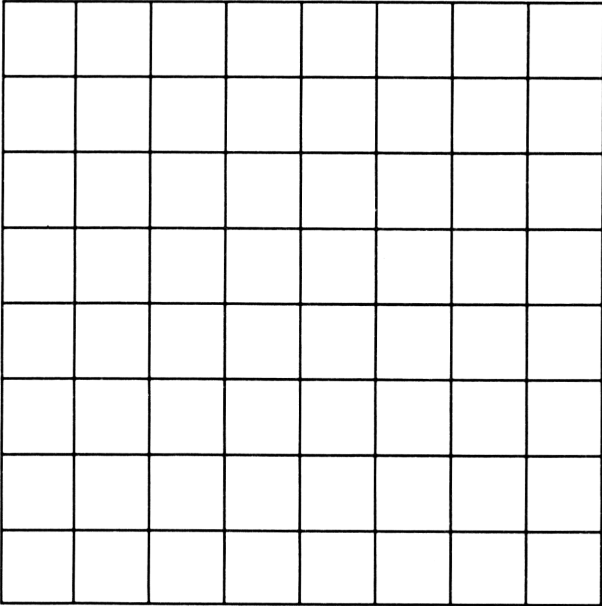


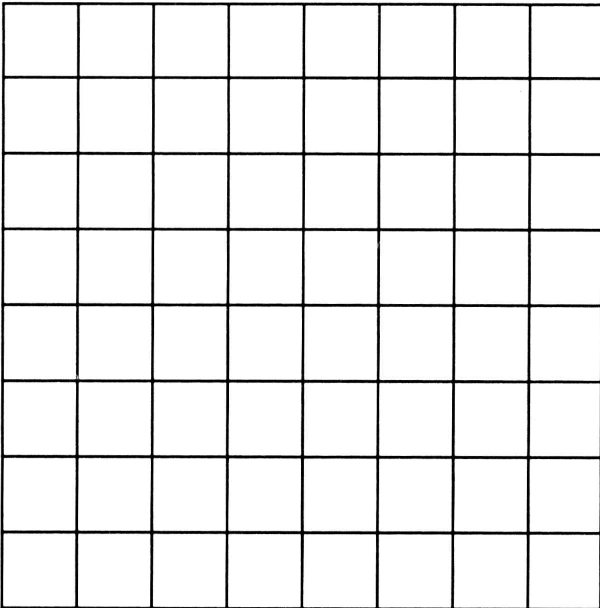


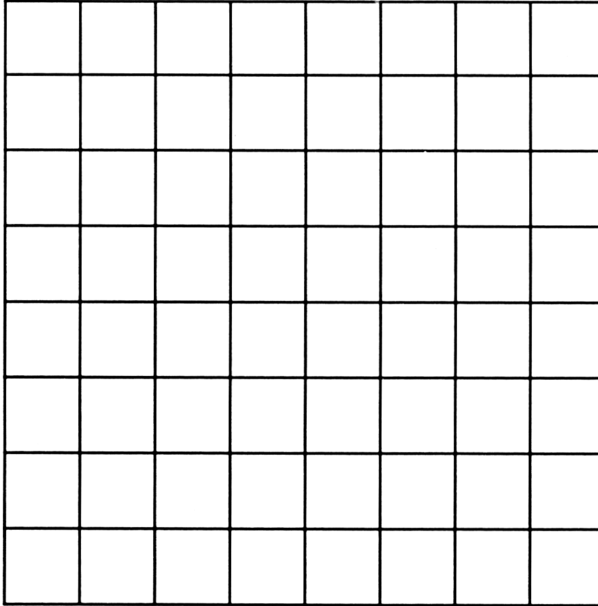


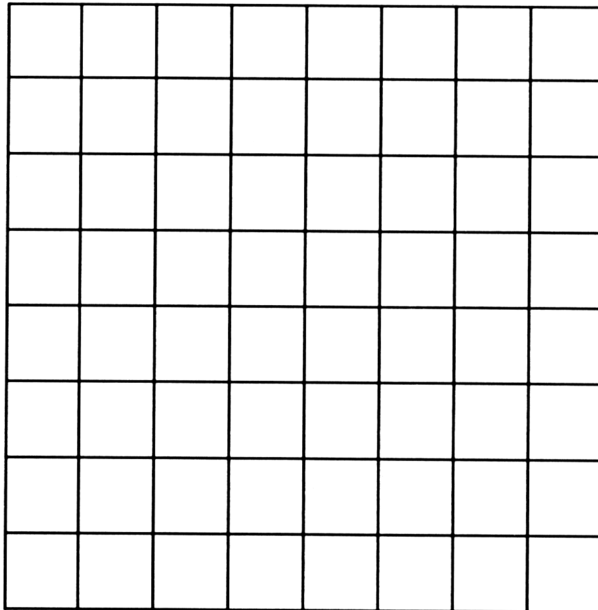


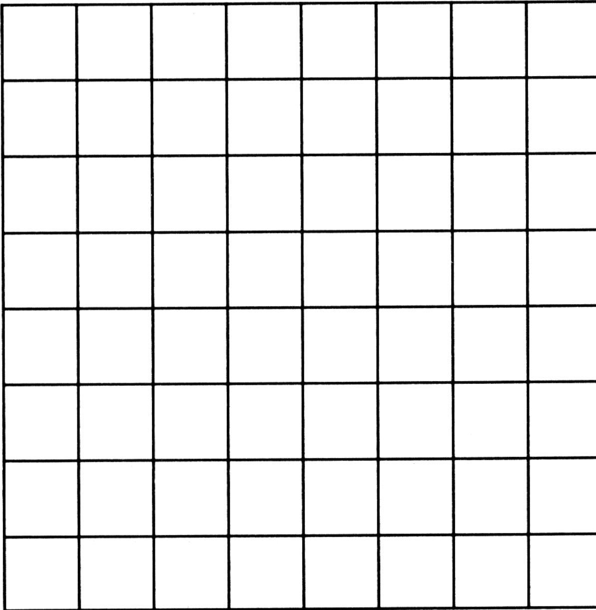


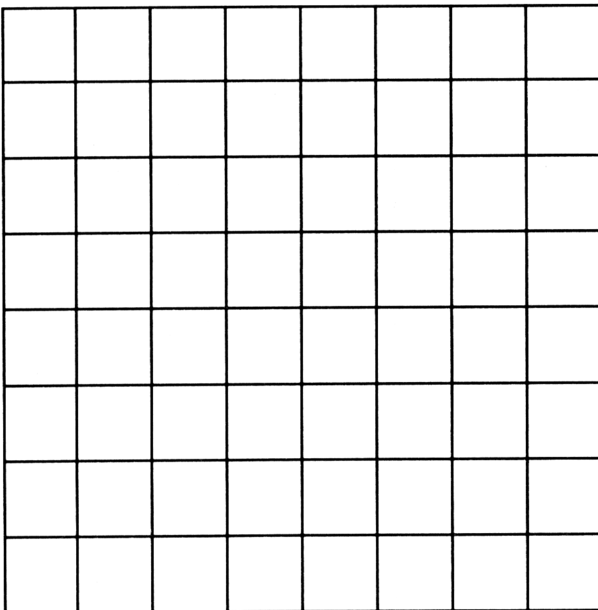


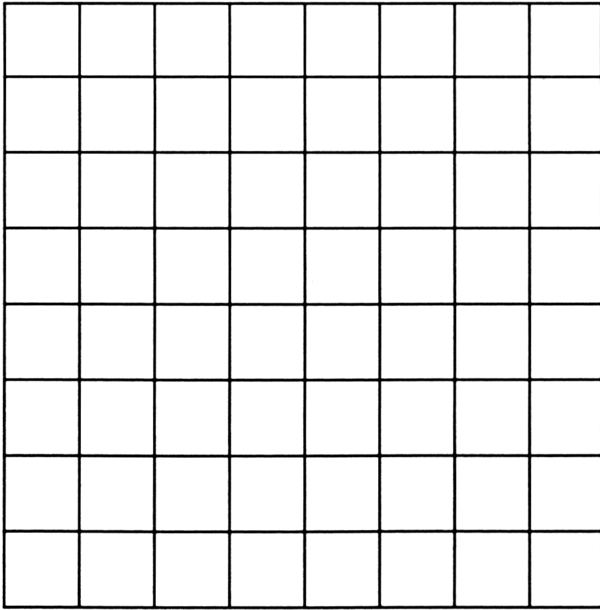


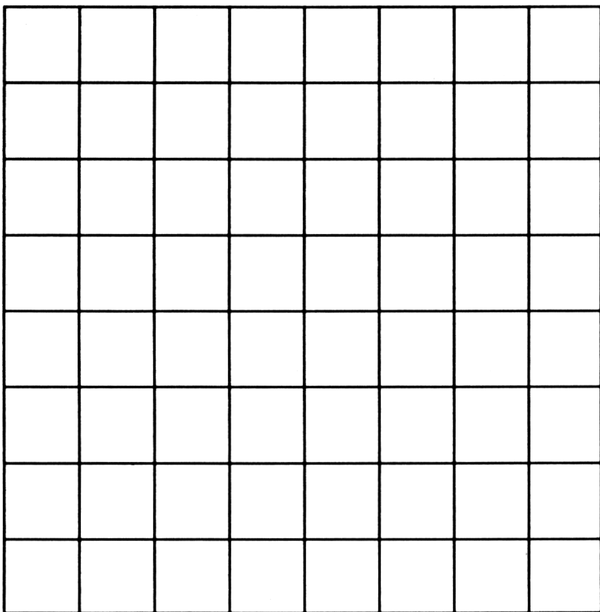


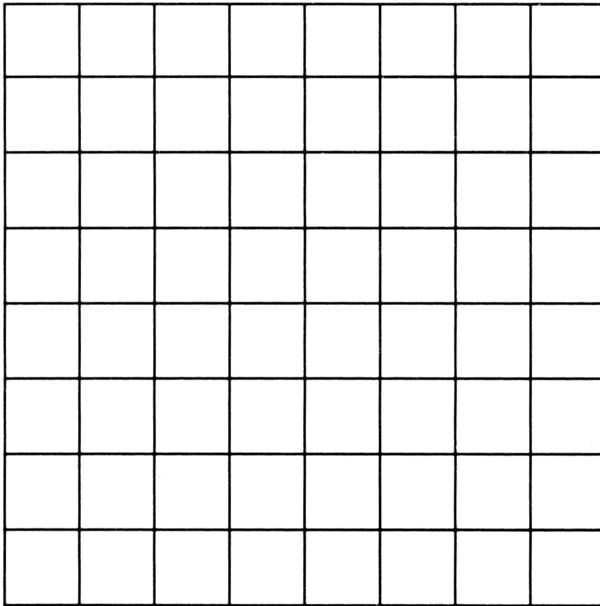


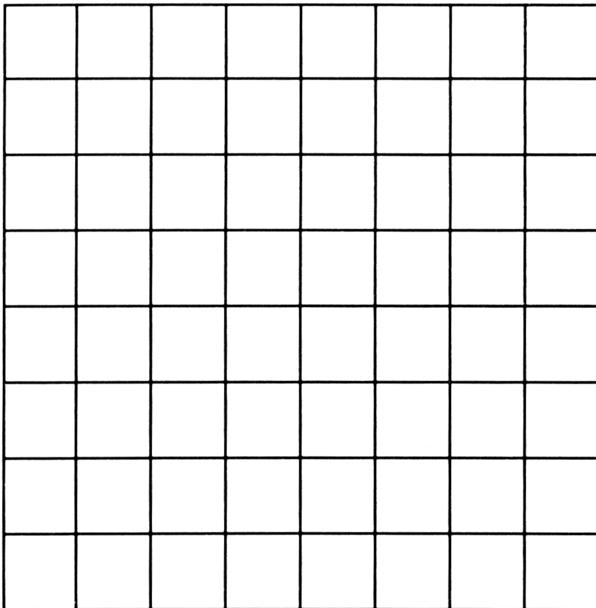












Author's Note

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 moving the gun 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.*
7) *The key to be pressed in order to escape from the keyboard reading routine is key A.*
8) *If bits 1, 4 and 6 are on then the value of the byte will be 82.*
9) *To give a value of 67 bits, 6, 2 and 1 should be set.*

SORCERY

Let this arcade adventure transport you back to a land of breath-taking beauty, a place and time where treachery and evil dominate.

- **Five random starting locations**
- **Marvellously detailed graphics and animation**
- **40 screens to master and explore**
- **Many objects to collect and use and exchange**
- **Atmospheric music and sound effects**
- **Developed from the CBM 64 version utilising the Amstrad's features to the full**

"Get Sorcery or you'll turn into a frog" – Which Micro (CBM 64).
"An excellent arcade game adventure" – HCW (CBM 64).
"Virgin's best game since the screen-scorching Falcon Patrol I & II" – Commodore User (CBM 64).
"I have played this game every night since I received it" – TV Gamer (CBM 64).
"Fast, wacky and very polished" – Commodore Horizons (CBM 64).

SAVE £1!

YOU'VE READ THE BOOK, NOW PLAY THE GAME!

As a special offer available to purchasers of this book, simply clip the coupon below and send it to **'I've Read The Book' Department, Virgin Games, 2-4 Vernon Yard, Portobello Road, London W.11,** enclosing a cheque/p.o. for £7.95 saving £1 — Yes! £1!

NAME____________________________

ADDRESS____________________________

FOR THE AMSTRAD CPC 464

AMSTRAD CPC 464 MACHINE CODE

Without a doubt, the Amstrad CPC 464 is the best computer in its price range; it has excellent colour, graphics and sound and a large usable memory. To take full advantage of these facilities you will need to learn Machine Code, particularly if you want to write fast arcade-type games. Machine Code has two main advantages over BASIC. Firstly, it is between 50 and 100 times faster; secondly, it can use far less memory.

If you have a reasonable understanding of BASIC programming you will find that you can quickly grasp the fundamentals of Machine Code by studying this book. The Machine Code equivalents of BASIC statements such as FOR/NEXT, PRINT, GOTO, GOSUB, ADDITION and SUBTRACTION are all explained. The book then goes on to demonstrate, in great detail, how to write a simple Space Invader game. There are also a number of questions throughout the book which will test your knowledge of Machine Code programming.

And even if you're not interested in writing 100 per cent Machine Code programs, this book will show you how to use small Machine Code routines to enhance your BASIC programs.

UK £4.95 ISBN 0 86369 082 3

AMSTRAD CPC 464 MACHINE CODE Steve Webb

Document numérisé avec amour par

AMSTRAD

MÉMOIRE ÉCRITE

https://acpc.me/