DR WATSON series

AMSTRAD ASSEMBLY LANGUAGE COURSE

Tim Herbertson

AMSTRAD ASSEMBLY LANGUAGE COURSE

DR WATSON series

AMSTRAD ASSEMBLY LANGUAGE COURSE

Tim Herbertson

JULY 1986

All programs in this book have been written expressly to illustrate specific teaching points. They are not warranted as being suitable for any particular application. Every care has been taken in the writing and presentation of this book but no responsibility is assumed by the author or publishers for any errors or ommisions contained herein.

COPYRIGHT ©Glentop Publishers Ltd 1986
World Rights reserved. 2nd Edition.

No part of this publication may be copied, transmitted or stored in a retrieval system or reproduced in any way including but not limited to photography, photocopy, magnetic or other recording means, without the prior permission from the publishers, with the exception of material entered and executed on a computer system for the reader's own use.

Edited by: Martin Thompson
Graphics: Samantha Borland and Hans Harbord
Wordprocessing: Jane Grant
Program by: Graham Rounce

ISBN 1 85181 112 5

Published by: Glentop Publishers Ltd
Standfast House
Bath Place
High Street Barnet
Herts EN5 5XE
Tel: 01 441 4130

Dr Watson is a Trademark of Glentop Publishers Ltd.
Z80 is a registered trademark of ZILOG Inc.

Introduction

This book forms part of the Dr Watson Assembly Language Series. It has been written so as to provide a completely self-paced course in Z80 Assembly Language Programming. Throughout the course, new instructions are illustrated with example programs. Also, numerous exercises have been included to help test the understanding of the concepts involved.

After reading and working thorugh the first couple of chapters the reader might find it helpful to read Appendix 6. This Appendix summarises the use of the assembler, including how to save and load programs on tape.

Whilst working your way through the book don't be afraid to experiment and try out any ideas you have, you will not harm the computer or Assembler in any way.

Well that's the speil finished: you now have ten action-packed chapters, not to mention the appendices, to work through! I hope you enjoy it.

T.Herbertson
LONDON
February 1985

CONTENTS

CHAPTER 1

Getting Started

Having used a computer before, you are probably well aware that things called 'machine code' and 'assembly language' exist. Quite simply, machine code is the language that the microprocessor chip, a Z80, in your computer understands. As an example, take a simple addition sum - adding one to 83.

In English you would say:

Add one to eighty three; what's the answer?

In BASIC you might say something like:

```
10 LET A=83
20 LET A=A+1
30 PRINT A
```

In Z80 machine code you could say:

```
3E 53
C6 1
CD 5A BB
C9
```

This is pretty-well unintelligible, isn't it? Well, that's why we use assembly language. The same problem is given below in assembly language along with a brief comment on each line.

```
LD A,83                 Load the Accumulator 'A' with '83'
ADD A,1                 Add 1 to the Accumulator
CALL 47962              Store the contents of A on the
                        screen
RET                     Return from the machine-code
                        subroutine
```

That's much easier to read than the machine code, isn't it? With an assembler you can enter your program in assembly language and be able to read though it and understand it more readily. All the assembler does is to change the assembly language into machine code. Thus, when it sees 'LD A' it changes this command into machine code '3E' and puts this into memory in the right place.

The Z80 chip contains various registers, which are in effect internal memory locations, where data can be stored. Most of the Z80's instructions use these registers. One of the most frequently used registers is the accumulator. The accumulator is capable of storing numbers from 0 to 255. Let's now look at an instruction which allows us to LoaD a value into the accumulator. The instruction for this is:-

LD A,n	LoaD the Accumulator with the immediate value n

You will notice that the letters chosen for the instruction reflect what it accomplishes. Instructions of this form are called 'mnemonics'.

There are many different ways of getting numbers into and out of the accumulator and other parts of the computer; these different ways are known as 'addressing modes'. The method above, which uses an unmodified typed-in number, is known as 'immediate-mode' addressing.

The next instruction in the program is ADD:

ADD A,n	ADD the immediate value n to the Accumulator

This instruction takes the number following the comma, in the example '1' (on page 1), and adds it to the value currently held in the accumulator (A). The answer is left in the accumulator, deleting the old number, in our case 83.

Looking back at the example, probably the most baffling instruction is

```
    CALL 47962
```

This tells the computer to store the contents of the accumulator, 84, on the screen - i.e. to display the contents. Whilst true, this is really only half the story.

The computer has many built-in machine-code programs or 'routines' stored in its chips. Without these, it would not be able to understand BASIC, for instance. One of these routines is designed to take a value from the accumulator and display it on a monitor.

To get at these routines in order to use them, CALL is used. Inside the computer, the first instruction in every built-in routine is stored in a specific memory location, these are numbered from 0 to 65535. The number above, 47962, is the memory location of the first instruction in the 'print the value in the accumulator on the screen' routine.

Right then! Let's have a go at running a machine-code program!

A couple of points about the assembler: when you start to write a program the assembler should be told where you want the program to be placed in the computer's memory. For the moment the assembler will decide where to store the program. It is told to do this by the ENT command.

Thus the first line of the program will be:

```
10 ENT
     .
     .
     .
   program
     .
     .
     .
```

Assembly language programs may be run (or called) from BASIC or called from the assembler using the Call program option.

Either way you must tell the computer to return from machine code to BASIC (the assembler program). The command that does this is RET.

RET	RETurn from machine-code subroutine

Right, to make the example into a program, we must:

1. Tell the assembler to decide where to store the program.

2. LoaD immediate value '83' into the Accumulator, i.e. 'LD A,83'

3. Add 1 to the contents of the accumulator. The mnemonic for this is 'ADD A,1'

4. CALL up the built-in machine-code routine to display the new value in the accumulator on the screen, i.e. 'CALL 47962'

5. RETurn from our machine-code p.'ogram to BASIC, i.e. 'RET'

6. Tell the assembler (not the Z80) that the program has finished, using a '@'.

or

PROGRAM 1.1 (do not type this in just yet)

```
10 ENT
20 LD A,83
30 ADD A,1
40 CALL 47962
50 RET
```

Now to enter this:

a) Load the ASSEMBLER program in the computer as follows:

   1) Perform a full reset of the computer by holding down the SHIFT, CTRL and ESC keys together, then release them.

   2) Fully rewind the tape (Side A).

   3) Press CTRL and the small blue ENTER key, then release them.

      The following message should now appear on the screen:

      RUN"
      Press PLAY then any key:

      If it does not appear start from step 1 again.

   4) Press the PLAY key on the data recorder and then press the space bar.

   5) A message will then appear on the screen after a few seconds informing you that the assembler is being loaded.

b) Press CAPS LOCK so you're typing in capital letters, then type 'I' without the quotes, of course, to begin entering a new program. Enter 10 in response to the prompt "Enter start and increment", then press ENTER.

c) A 10 will appear; type in 'ENT' and then press the ENTER key.

d) A new line number will now appear, 20. Type in the next line of the program, 'LD A,83' and press the ENTER key as before. Don't leave out the space between the 'LD' and the 'A'. Also, do not insert a space between the comma and the number, or indeed anywhere else.

e) 30 will now appear. Type in 'ADD A,1', then the ENTER key. Again, don't leave out the space between ADD and A.

f) On line 40, type in 'CALL 47962' and then press the ENTER key. There should be a space between CALL and 47962.

g) On line 50, type in 'RET' and then press the ENTER key.

h) Now as we have finished typing in the program, and want to return to the command mode, press @ and then the ENTER key.

i) Now press M to return to the menu, then L to list the program, entering 10 as the starting line number. (Note: you could have selected the List option straight away without returning to the Menu.) The program should now appear as below:-

```
10          ENT
20          LD    A,83
30          ADD   A,1
40          CALL  47962
50          RET
```

If for any reason it doesn't then select the replace mode 'R' and replace the line(s) containing the error(s). When your program appears as above, when listed, press A for assemble. The assembler will now assemble the program into machine code. Select option 2.

j) After the assembler has assembled the program it will return to the '>' prompt. If there were any errors in the program the assembler would have informed you. the replace or insert function would then be used to correct these errors.

k) To see the program run, select the Call program option.

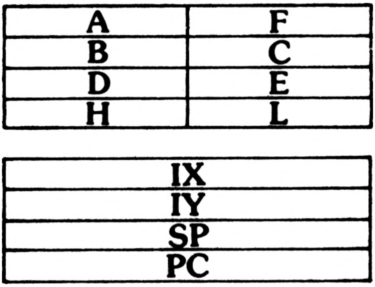
In case you're wondering why the program puts a capital T on the screen and not the answer, 84, have a look at Appendix III in the Amstrad Basic manual showing the 'ASCII character set'. You will see that the code for 'T' is 84.

ASCII stands for the American Standard Code for Information Interchange, and every symbol which the computer can put on the screen has its own ASCII code number. Thus when asked to put 84 onto the screen, the routine called using CALL 47962 puts character number 84 onto the screen.

This is true in BASIC as well, even when doing a sum like 'PRINT 83+1'. In this case, although the computer puts '84' on the screen, it has had to find out what the characters '83+1' mean, do the appropriate sum and convert the answer using the ASCII codes for the characters '8' and '4'. This is in fact quite complicated to do. The computer has a built-in routine which does this automatically for BASIC.

Z80 Chip Architecture

The Z80 has various registers other than the accumulator, and a simplified diagram of the arrangement is shown below, in Figure 1.1.



Simplified Z80 Architecture

FIGURE 1.1

The accumulator, or A register, you have already met. It is somewhat special in that there are more Z80 instructions which can operate on it than there are for the other registers.

All of the registers (just memory locations within the Z80 chip) are 'eight-bit registers', that is, they can only store numbers from 0 to 255. This may not seem terribly useful, but there are ways of using the registers in combination to store much bigger numbers, as will be seen later.

The H and L registers taken together are known as the 'primary data pointer'. Registers B and C, and D and E when taken as pairs are called 'secondary data pointers'. The main point about this for now though is that instructions which use H and L are often shorter when assembled into machine code, and therefore run faster.

Immediate Addressing Of The Registers

Immediate addressing of the accumulator was discussed earlier, where a number was loaded into the accumulator using the instruction

```
        LD A,83
```

Exactly the same form of instruction can be used to load numbers from 0 to 255 into any of the other registers, e.g.

```
        LD B,83
```

would load the number 83 into the B register.

In general,

LD r,n	LoaD register r with the immediate value n

The 'r' in the above box stands for any register, i.e. any one of A, B, C, D, E, H, or L.

Register - Register Addressing

Not only can any register be loaded with a number, but any register can be copied into any other register, or into the accumulator. For example, Program 1.2 puts 43 into the L register, copies this into the accumulator, and displays it on the screen - an ASCII '+'.

PROGRAM 1.2

```
        10 ENT
        20 LD L,43                 Immediate-mode load of L with 43.
        30 LD A,L                  Load A with contents of L.
        40 CALL 47962              Display the contents of the
                                   accumulator.
        50 RET                     Return to BASIC (the assembler)
```

Line 30 contains the new instruction:

```
        LD A,L
```

This copies the contents of register L into the accumulator.

Another example could be:

```
        LD B,E
```

or

```
        LD C,A
```

In general,

```
    LD r1,r2                Copy the contents of register r2
                            into register r1
```

Some authors refer to register-register addressing as 'inherent' addressing, and others refer to it as 'implied' addressing. If you refer to other books, beware of their terminology, as this can lead to confusion.

Right then! Let's type in Program 1.2. If you make a mistake typing in any line before pressing the ENTER key, just use the DELETE key to rub out the mistake and type in the correct version. If you notice a mistake on a line after you've pressed the ENTER key, you can replace those lines after pressing @ and before assembling the program.

1. Load and run the assembler program unless it is loaded already, in which case return to the menu with the M command, if necessary.
2. Select 'I' to begin entering the program, and press CAPS LOCK to set it into capital letters mode.
3. Tell the assembler where in memory the program is to start, i.e. type in 'ENT' and press the ENTER key.
4. Type in 'LD L,43' and press the ENTER key, (press the ENTER key after each entry).
5. Type in 'LD A,L'
6. Type in 'CALL 47962'
7. Type in 'RET'
8. Press @
9. Select 'L' to list the program: check it!
10. Select 'A' to assemble the program.
11. Select 'C' to run the program.

After that, you should be capable of writing some programs on your own. Have a go at the following short exercise. Don't forget to put in the 'RET' instruction, as otherwise the computer will trundle on through its memory after the program itself, looking for instructions to execute. It will almost certainly find something, and that something will cause the system to 'crash'. In other words, there will be no option but to switch off and start again from scratch - including reloading the assembler program.

EXERCISE 1.1

Using immediate mode addressing, load the accumulator with 65, and then display this on the screen. 65 is the ASCII code for capital A. A possible answer is given in the solutions chapter.

More About CALL 47962

If this call is used several times in succession, several things can be displayed on the screen in consecutive screen locations. Try this program:

PROGRAM 1.3

```
10 ENT
20 LD A,72            Load Accumulator with ASCII 'H'
30 CALL 47962         Display 'H'
40 LD A,69            Load Accumulator with 'E'
50 CALL 47962         Display 'E'
60 LD A,76            Load Accumulator with 'L'
70 CALL 47962         Display 'L'
80 CALL 47962         Display 'L' again
90 LD A,79            Load Accumulator with 'O'
100 CALL 47962        Display 'O'
110 RET               Return to BASIC
```

That program should display the word 'HELLO' on the screen, after it has been assembled and run.

EXERCISE 1.2

Write your name in the top left corner of the screen. An answer for 'FRED' is given in the solutions chapter.

That's the end of this chapter. It wasn't that bad was it? By now you should know or be able to interpret the following:

```
LD A,n
LD r1,r2
RET
ENT
@
CALL 47962
ADD A,n
immediate mode addressing
register-register addressing.
```

How about this (r is any register)?

```
ADD A,r
```

CHAPTER 2

Jumping,Subroutines and Labels

Few real life programs proceed along a smooth uninterrupted path without jumping or branching at some stage. This chapter looks at these jump commands and their uses. After this, the chapter goes on to examine the flags that enable these jumps to be controlled. The chapter then looks at both unconditional and conditional calls to subroutines. Symbolic labels are also introduced. These are a very powerful and useful feature of the assembler and aid program development considerably.

Unconditional Jumps

These tell the program to jump willy-nilly - no conditions.

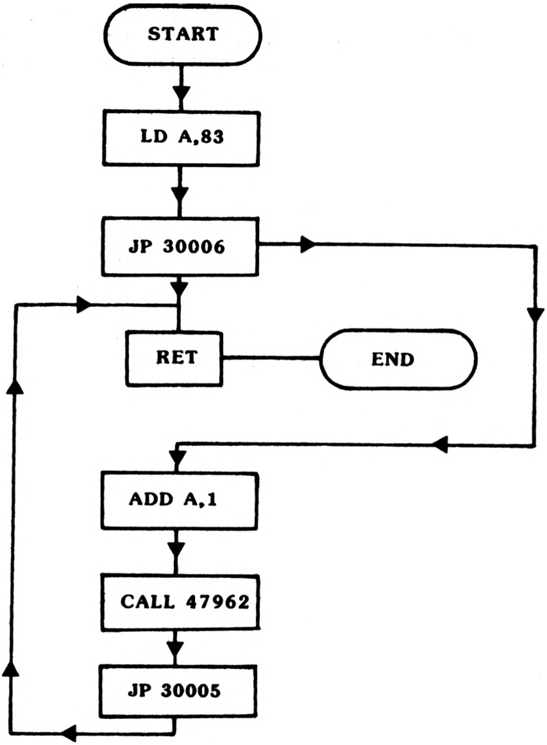
The Z80 instruction set contains five such jumps; for the moment we shall only be concerned with two of these. The other three will be introduced later.

The first to be considered is:-

```
        JP nn       JumP to the specified address nn
```

For example, JP 200 means jump to memory location 200.

Put into a program JP will look like this:



Using JP in a program

FIGURE 2.1

Such a jump routine doesn't really achieve a lot but it could, for instance, be used to patch a piece of code into a program. In Figure 2.1 for instance, the commands 'ADD A,1,' 'CALL 47962' and 'JP 30005' have effectively been inserted into the program.

We shall now type this program into the computer. Remember to type ENTER at the end of each line, and to select 'I' to start entering a program, and enter the starting line number as 10.

Note that the first line in the program is not ENT. As the program is jumping to specific memory locations, the exact start of the program needs to be known. The ORG 30000 causes the assembler to store the machine code starting at memory location 30000. See Appendix 6 for more details.

PROGRAM 2.1

```
        ORG 30000
        LD A,83
        JP 30006
        RET
        ADD A,1
        CALL 47962
        JP 30005
```

Now Assemble the program with Option A, select option 2.

The screen display of the program should now look something like this:-

Memory location	Line number	Object code	Source code
7530	10		ORG 30000
7530	20	3E53	LD A,83
7532	30	C33675	JP 30006
7535	40	C9	RET
7536	50	C601	ADD A,1
7538	60	CD5ABB	CALL 47962
753B	70	C33575	JP 30005

FIGURE 2.2

The first number in the memory location column doesn't seem to correspond to 30000, where the program should begin. This is because the memory locations are represented in Hexadecimal. Don't worry about this for now, just remember that 30000 is 7530 in Hexadecimal, 30006 is 7536 and 30005 is 7535.

When jumps are used in this way it is necessary to tell the program where to jump to, i.e. an address, hence JP 30006. Calculating these addresses is quite straightforward as long as it is done systematically, as will be explained below.

Looking at Figure 2.2, the numbers and letters in the second column, the 'object code', are what the assembly language looks like after it has been assembled - i.e. machine code. Each pair of alphanumeric characters (i.e. numbers and/or letters) in this column represents one item from the original assembly language. Thus, '3E' represents 'LD A' and '53' represents '83' (don't worry about why! It's Hexadecimal and will be explained in a later chapter.) For numbers in assembly language bigger than 255, four machine-code alphanumerics are required. Thus 'C3' represents 'JP' and '3075' represents '30000'. How does this help us calculate addresses to jump to? Well, each pair of digits fills up one memory location - one byte of memory. Thus, as it is known that the first instruction in the program is in memory location 30000 (because of the 'ORG 30000'), we can simply count pairs of machine-code alphanumerics.

For instance, to jump to the 'ADD A,1' instruction, we would need to jump to the memory location containing 'C6'. So, starting at 30000, which contains '3E', we have 53, C3, 36, 75, C9, and then C6. That's 6 memory locations along from 30000, or 30006. Hence the program should jump to 30006. Admittedly, before you've entered the 30006 in the JP instructions, the '3675' won't be there for you to count. However, as you know that the address in machine code is going to be four characters (two bytes), long, this is not a problem provided you write your programs out on paper before typing them in, - which is good practise anyway.

The appendices contain tables detailing how many bytes each instruction requires to help in calculating jump addresses.

Here is a breakdown of the program:

ORG — Uses no memory, it is there for the assembler's use, and is called an 'assembler directive'.

LD A,83 — Takes up 2 bytes - the first is the object code for LD A, and the second is the data.

JP 30006 — Takes up 3 bytes - one for its object code, C3, the other two are the memory location to jump to.

RET — Only requires one byte for its object code, C9.

ADD A,1 Requires 2 bytes. The first, C6, is the object code for 'ADD A', the second is the data.

CALL 47962 This instruction requires 3 bytes, CD is the object code and 5A and BB the memory location being called.

JP 30005 This is the same as the JP 30006 except that the memory location to jump to is different.

By counting up the number of bytes we can calculate the length of the program: Program 2.1 is 14 bytes long.

Using the JP operator we can jump to any memory location from 0 to 65536. If the memory location to jump to is less than +129 or -126 bytes from the current memory location the Jump Relative can be used instead of the Jump instruction. The advantage of using this alternative instruction is that, as it only requires two bytes to store as opposed to three for the JP instruction, it is quicker.

JR e Jump Relative to address e

For example, if you are at address 30000, and you want to jump ahead to address 30045, you would write

JR 30045

You would not say JR 39, the computer will calculate the single-byte offset automatically. Remember, JR can only be used if you are jumping to an address within +129 or -126 of the memory location containing the 'JR' itself.

The Program Counter

Before going much further, the way in which a jump instruction accomplishes its task must be examined. Examining Figure 2.2 it can be seen that the program starts at memory location 30000. In the body of the program we require it to jump to memory location 30006 and 30005. How does the computer keep track of where it is? Inside the Z80 chip there is a '16 bit' register called the Program Counter (PC). This register contains the address of the current instruction. When a machine code program is run, the program counter is set to the first memory location of the program. After execution of the first instruction, the program counter is updated, so that it points to the next instruction. Thus a jump instruction loads the PC with the address to jump to i.e: it 'points' it to the new instruction. The effect of this is that the program continues execution from instruction contained in the memory location pointed to by the jump address. Other instructions are available to alter the contents of the PC hence altering the programs course and will be introduced when required.

Figure 2.3 illustrates the execution of the previous program.

Program	PC before execution	PC after execution
ORG 30000	?	30000
LD A, 83	30000	30002
JP 30006	30002	30006
ADD A, 1	30006	30008
CALL 47962	30008	30011
JP 30005	30011	30005
RET	30005	Back to assembler

FIGURE 2.3

So far the computer has jumped to another section of code unconditionally. Although useful, it would be more useful if the computer could be made to jump upon a certain condition being met. This can be accomplished with the conditional jump group of instructions.

Conditional Jumps

Any program that needs to test for conditions needs CONDITIONAL JUMPS. In BASIC, the analogy is the IF.... THEN command.

i.e. 10 IF X=Y THEN GOTO 500

This line causes the computer to compare the variables X and Y and if they are the same go to line 500.

The Z80 carries out this operation by using a special register, the flag register. The flag register is an eight-bit register like the accumulator and the B,C,D,E,H and L registers, but is used quite differently from these. Whereas the other registers are used to store and manipulate bytes, the flags register is treated as if it contained eight individual bits which are used as signals or flags. The Z80 normally only handles one flag at a time, either setting the bit value to 0 or 1. It can also test a flag to determine whether it is set (1) or reset (0).

For example one of the flags is the Z flag or Zero flag. Whenever an arithmetic process is carried out that produces a result of zero, the zero flag is set to '1' - otherwise it will be reset to '0', indicating that the previous operation did not result in zero.

Several different instructions can set this flag, one of these being:

```
        DEC   d               DECrement register d
```

This instruction decrements the contents of 'd', where 'd' is one of the following registers:

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

If, after decrementing the specified register, the result is equal to zero, then the zero flag is set, otherwise it is reset.

Note the instruction DEC is formally called an OPERATOR, and d, the OPERAND; thus the OPERATOR operates upon the OPERAND. Some OPERATORS, such as LD A,10 require two OPERANDS, in this case, A and 10.

Now on with the programming. Program 2.2 will print 10 A's on the screen.

PROGRAM 2.2

```
        ORG  30000
        LD C,10
        LD A,65
        CALL  47962
        DEC C
        JR NZ,30004
        RET
```

Type this program in and assemble it using option A, then option 2. To run the program exit to BASIC with X and type in CALL 30000. You will see ten A's appear upon the screen. To return to the assembler press the decimal point key on the numeric keypad.

Examining the program, the only line which hasn't been met yet is JR NZ. This operator tests the current state of the zero flag and 'Jumps Relative' to 30004 if the last arithmetic instruction results in a non-zero answer. Thus the program puts the contents of the accumulator upon the screen, DECrements register C and tests to see if the zero flag has been set by the DECrement instruction. If it hasn't, i.e. it is Non Zero, then it Jumps Relative to 30004 otherwise it goes on to the next line of the program where it RETurns.

JR NZ,e	Jump Relative on Non Zero result to address e

EXERCISE 2.1

Rewrite Program 2.2 to use the B register in place of the C register.

EXERCISE 2.2

Why have we used JR NZ,e instead of the alternative JP NZ,e instruction?

Possible answers to both problems are given in the solutions chapter.

Note, as might be expected an instruction to INCrement an operand exists as well:

INC d	INCrement operand d

So far the computer has been programmed to jump unconditionally and on the result of an operation being zero. In both cases it is necessary to know which memory location to jump to. While this situation is bearable within short programs, when programs start increasing in size it becomes increasingly difficult to calculate these jump addresses, not to say time consuming. To overcome this situation, labels are used.

Labels

The use of labels enables a program to be directed to a named instruction, without the necessity of calculating jump addresses. A fancier term for label is SYMBOLIC LABEL as the label itself is symbolic of a location in memory. For instance, the instruction

```
        JP  LOOP:
```

Will cause the assembler to replace LOOP with an address, which was previously assigned to LOOP, whenever it occurs.

To tell the assembler that a label is a label it is necessary to follow the label with a colon. Also labels must be less than or equal to six characters in length. Thus, for example if the program should jump to the memory location containing the instruction DEC C, the following program lines would be used.

```
              .
              .
              .
              .
         JP LOOP:
              .
              .
              .
              .
       LOOP:  DEC C
              .
              .
              .
```

Further conventions must be observed when using labels. The colon for instance must follow the last character of the label (no space between them). Also, a space must follow the colon. This is simply to allow the assembler to work out where the label ends. Don't worry if you forget any of this, the assembler will pick up any errors and inform you.

To summarise:

LABELS

1. A label must consist of six or fewer characters.

2. A colon must immediately follow the label.

3. One space must follow the colon.

4. The label must not contain a space.

For example:

```
LOOP:

TEST:

NXTCHR:
```

are all valid labels. The following are not:-

```
LOOP :

BACKONE:

NEXT L:
```

To illustrate the use of labels we will rewrite Program 2.1 using labels.

Note that as no absolute jumps are being used the ENT directive can be used instead of the ORG directive.

PROGRAM 2.3

```
        ENT
        LD A,83
        JP NXT:
        END: RET
        NXT: ADD A,1
        CALL 47962
        JP END:
```

When you list the above program, you will see that it is displayed with the labels out to the left of the main body of the program, like this, making it a lot easier to read:

```
        ENT
        LD A,83
        JP NXT:
END:    RET
NXT:    ADD A,1
        CALL 47962
        JP END:
```

To ease typing in programs they will be displayed in this format throughout the rest of the book. However, you should, of course, type them in in the normal way.

Using the instructions introduced so far it is possible to jump unconditionally or conditionally to anywhere in memory, or using a jump relative command to any location within +129 or -126 of the current position. Labels can also be used to facilitate calculating jump addresses.

All of these commands will now be combined to print the numbers one to nine on the screen. Remember in Chapter 1 we saw that the ASCII for T is 84? Well, the numbers one to nine also have a code associated with them. Look at the ASCII table (in the BASIC manual appendices 3) and see if you agree that the code for one is 49 and that for nine is 57.

Now to combine what has been learnt so far to solve the problem, i.e: how to put the contents of the accumulator on the screen, load it, and increment it. Also, how to decrement a register and test for zero. If you have any ideas about how to get the computer to display to numbers 1 to 9 on the screen, close the book now and try, if not, don't worry all will be explained.

The following flow chart represents the solution:-

Problem: Display the numbers 1 to 9 on the screen.

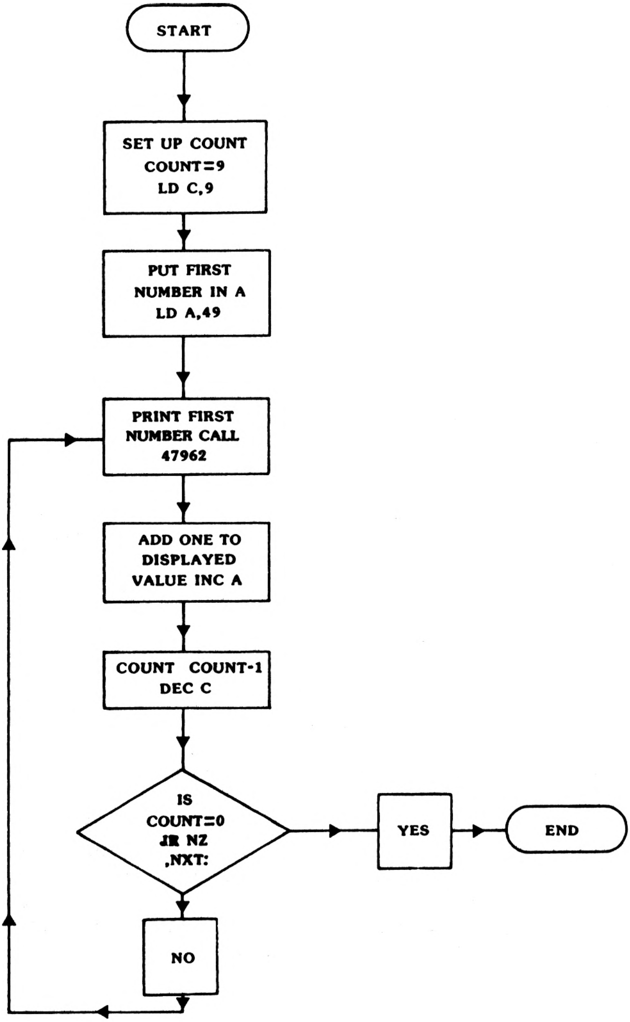


FIGURE 2.4

Translated into a program it looks like this:

PROGRAM 2.4

```
        ENT                     Set start of program.
        LD C,9                  Load C with number of
                                characters to
                                print, i.e. nine.
        LD A,49                 Load A with the code
                                for "1".
NXT:    CALL 47692              Print contents of Accumulator on
                                the screen.
        INC A                   Add one to contents of
                                A, thus loading A with
                                code for "2".
        DEC C                   Decrement the count register.
        JR NZ,NXT:              If DEC C results in anything
                                but zero jump relative
                                to the location addressed by
                                the label NXT.
        RET                     If C=0 then return to the
                                assembler.
```

Before running it, let's steo through it, each step in one loop of the program.

Step Number	Accumulator	Register C	Z flag
1	49	9	0
2	50	8	0
3	51	7	0
4	52	6	0
5	53	5	0
6	54	4	0
7	55	3	0
8	56	2	0
9	57	1	0
10	58	0	1

When the Z flag is set (1) the program terminates and returns to the assembler. (Step 10)

Now assemble and run the program.

EXERCISE 2.3

Write a program which will print the alphabet on the screen. (Hint: the code for A is 65.)

A possible answer is given in the solutions chapter.

So far only the zero flag has been used, one of the seven flags available. Here are the rest.

The Flags

The Carry Flag (C)

This flag is set whenever an addition or subtraction results in a carry (or borrow). It is also used in certain shift and rotate instructions.

Subtract Flag (N)

This flag is mainly used by the Z80 rather than by programs for certain arithmetic operations.

Parity/Overflow Flag (P/V)

This flag has two distinct functions, the first to indicate the parity of a result. The 'parity' of the result is obtained by adding up all the ones in its binary representation. If the result is even then the parity but is set, otherwise reset, i.e.=0. The flag is also set when during certain arithmetic operations 'overflow' occurs.

The Half Carry Flag (H)

The half carry flag is used by the Z80 for 'Binary Coded Decimal' instructions. Don't worry about this for the moment.

Zero Flag (Z)

The zero flag is set when an operation results in zero, and as such, is used a lot for compares.

The Sign Flag (S)

This flag indicates the sign of the arithmetic result or of a byte being transferred. It basically tests bit 7 of a byte, and is set if it equals 1 and reset otherwise.

Subroutines

As the JP instruction was compared with the BASIC statement IF X=Y THEN 500, a Z80 instruction similar to the BASIC statement GOSUB 500 exists. This instruction, CALL, has already been used to print the contents of the accumulator on the screen.

CALL nn	CALL the routine commencing at the memory address nn (nn can be represented by a label).

As in BASIC, a return instruction is required at the end of the subroutine.

RET	Return from the subroutine last called.

Let's see how these instructions can be used in a program.

Problem: Print the ASCII characters 200 to 250 on the screen using a subroutine print the characters.

PROGRAM 2.5

```
        ENT
        LD A,200          A= 1st ASCII code
        LD B,50           B= count
NXT:    CALL PRINT:       Call printing routine
        DEC B             Decrement count
        INC A             Next ASCII code
        JR NZ,NXT:        If B=0 jump to NXT
        RET
PRINT:  CALL 47962
        RET
```

Assemble then run it.

To aid understanding the program the print subroutine is assigned a label PRINT. As well as unconditional calls, conditional calls exist, much the same as conditional and unconditional jumps.

CALL cc,nn	Call the subroutine starting at memory location nn, if condition cc is met.

The conditions on which the routine is called are exactly the same as those used for conditional jumps. When a CALL instruction is executed the current contents of the program counter are saved. The program counter is then loaded with the call address nn. When the Z80 has completed the subroutine, i.e. reached a RET instruction the program counter is loaded with the saved value of the program counter.

Problem Print 100 A's on the screen, but for every 10 printed, print a space.

Representing the solution as a flow chart:

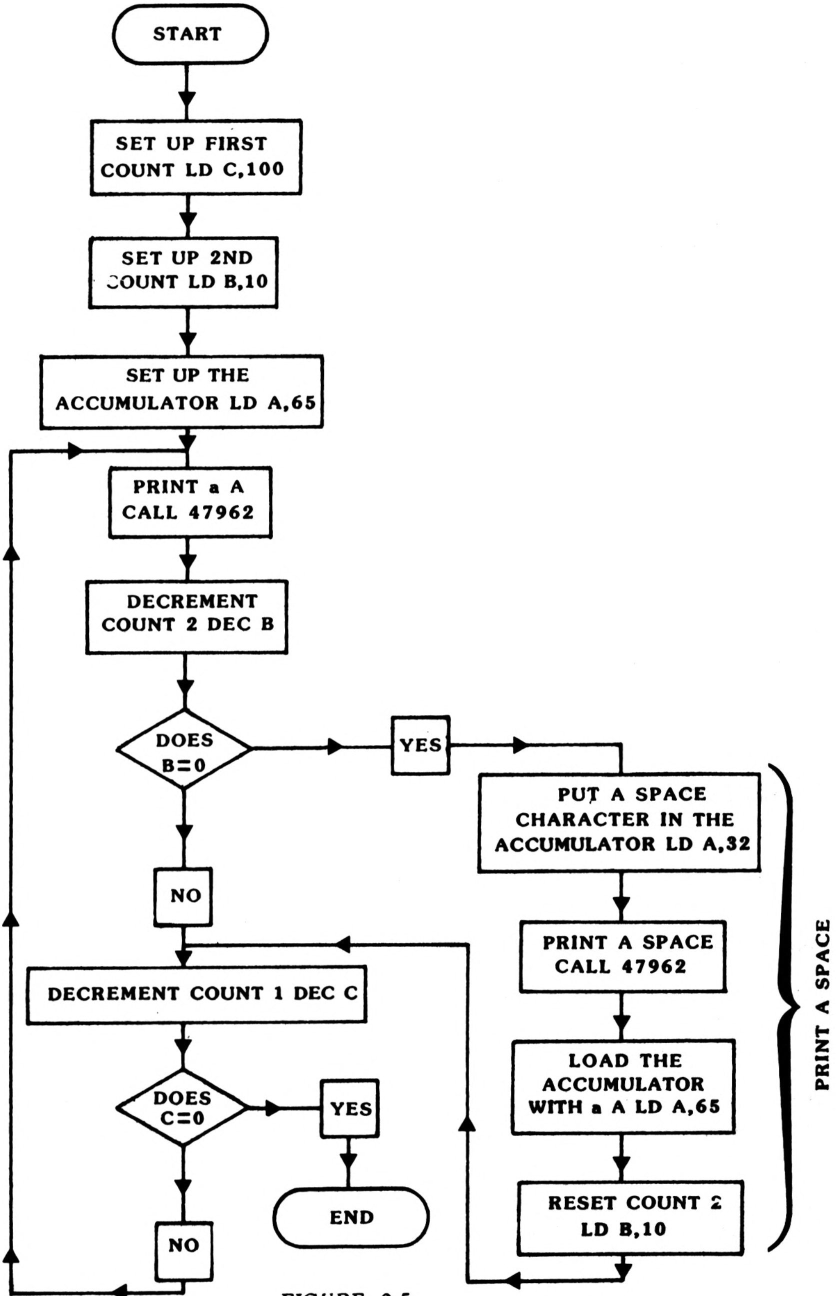

START
SET UP FIRST COUNT LD C,100
SET UP 2ND COUNT LD B,10
SET UP THE ACCUMULATOR LD A,65
PRINT a A CALL 47962
DECREMENT COUNT 2 DEC B
DOES B=0
YES
NO
PUT A SPACE CHARACTER IN THE ACCUMULATOR LD A,32
PRINT A SPACE CALL 47962
LOAD THE ACCUMULATOR WITH a A LD A,65
RESET COUNT 2 LD B,10
PRINT A SPACE
DECREMENT COUNT 1 DEC C
DOES C=0
YES
END
NO

FIGURE 2.5

Converting this into a program:

PROGRAM 2.6

```
        ENT
        LD C,100          C= 1st count
        LD B,10           B= 2nd count
        LD A,65           A="A"
NXT:    CALL 47962        Print a 'A'
        DEC B
        CALL Z,PSPC:      If B=0 print a space
        DEC C
        JR NZ,NXT:        Have a 100 A's been
                          printed? If no jump to
                          NXT space.
        RET               Return to Assembler

PSPC:   LD A,32           A=ASCII for a space.
        CALL 47962        Print a space
        LD A,65           A="A"
        LD B,10           Reload B as count
        RET               Return to main program
```

Try running this program now; lo and behold a hundred A's appear upon the screen separated every 10 by a space. This form of conditional call is very useful as it considerably eases writing structured programs.

Summary

The program counter (PC) is a 16-bit register, which contains the memory address of the current instruction from within a program.

The flags register contains 8 bits, which reflect the status of the arithmetic section of the Z80.

Labels can be used in jump instructions. A label must be less than seven characters long, terminated by a colon which must be followed immediately by one space.

Two main forms of jumps exist, conditional and unconditional. These two forms can be divided further into absolute jumps, i.e. JP NXT or relative jumps, i.e. JR NXT.

Two forms of CALL exist, conditional and unconditional. Every call instruction must have associated with it a RET instruction.

The following list of instructions should be understood in context, even if you don't quite understand all the flag conditions.

```
JP    nn
JP    cc,nn
JR    e
JR    Z,e
JR    NZ,e
```

Where:-

nn   is an absolute address or a label.

cc   is one of the operands signifying what flag to test for.

e   is an absolute address, except that it is limited to -126 to +129 bytes from the current address. A label can also be used.

cc	condition
NZ	Non-Zero
Z	Zero
NC	No-Carry
C	Carry
PO	Parity Odd
PE	Parity Even
P	Positive
M	Negative

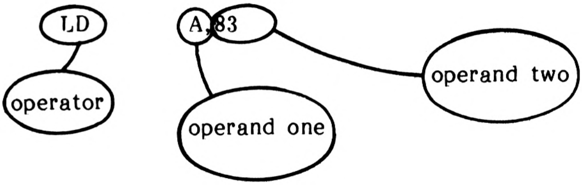
Also:-

```
INC  r

DEC  r
```

Where r is an 8-bit register, and any one of the following

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

The definition of operand and operator is best shown with an example:



CHAPTER 3

Register pairs and addressing modes

So far we have only been able to load a register with an 8-bit number. The Z80 allows certain registers to be combined into pairs, thus allowing 16-bit numbers to be stored and manipulated.

The following registers may be paired together.

B and C
D and E
H and L

We have already met the instructions that enable us to load 8-bit data into individual registers. The instruction which allows 16-bit loading is:

LD dd,nn	Load register paid dd with the 16-bit data nn.

Where dd is any of the following registers: BC, DE, HL, or SP. 'nn' is a 16-bit number.

Note: The SP register is a special purpose register, called the stack pointer. We will return to it at a later stage.

When loading a register pair in this way we are loading it immediately with the data value given, hence the formal term for this type of addressing, IMMEDIATE MODE.

All the programs so far have stored data in registers. This situation is fine if we don't have very much data to process. In real-life situations though, we will require more data storage than the seven 8-bit or three 16-bit registers available.

A need arises for intructions to load memory with data, and to read the contents of a memory location. The Z80 allows the contents of registers to be stored in memory, and the contents of memory locations to be copied into registers.

This form of addressing is termed DIRECT MODE, because it uses the contents of memory locations and registers directly - i.e. without doing anything with them first.

LD (nn),dd	LoaD memory location nn with the contents of register pair dd.

An example of this instruction is:

```
LD (200),BC
```

This will copy the contents of register pair BC into memory location 200. Note that the memory location has to be included in brackets.

The opposite instruction, to copy the contents of a memory location into a register pair is:

LD dd,(nn)	Load register pair dd with the contents of memory location nn.

An example of this instruction is:

```
LD BC,(200)
```

What will this do? It will load the contents of memory location 200 into BC.

Okay, let's now use these instructions.

Up until now characters have been printed on the screen with a CALL instruction. This is not the only built in routine available. The internal memory (ROM) also contains some graphics routines, which amongst other things allow us to draw lines on the screen. For the moment we will not be concerned with how this is accomplished, but with how to use it. When in BASIC to draw a line from the origin to the point 400,200 the following command would be typed in (assuming that the graphics cursor is at 0,0).

```
DRAW 400,200
```

If you try this, ensure that you reset the graphics cursor to 0,0 with the following command:

```
PLOT 0,0
```

By using the graphics routine starting in memory location 48118, we can also plot a line to 400,200 using machine code. In BASIC, the coordinates are typed in following the DRAW statement. When using machine-code routines, the data is passed (X,Y coordinates in this case) in registers. The DE register is loaded with the X cordinate, HL with the Y coordinate.

Let's now summarise the information needed to use this routine:

Start address 48118

Parameters X,Y

X stored in register pair DE
Y stored in register pair HL

Right, let's now plot a line to the point 400,200 on the screen.

Here is the program:-

PROGRAM 3.1

```
ENT
LD DE,400
LD (35000),DE
LD HL,200
LD (35002),HL
LD DE,(35000)
LD HL,(35002)
CALL 48118
RET
```

Type it in now and run it .....

Examing the program. The first point to notice is that the contents of DE are stored in memory location 35000, but the contents of HL in 35002. Why not store the contents of HL in 35001? The reason is that any one memory location can only store an 8-bit number and we are storing the contents of a 16-bit register i.e. 16 bits. What happens is that the 16-bit number is split into two 8-bit numbers which are then stored concurrently. Thus to store a 16-bit number requires two memory locations. See Figure 3.1.

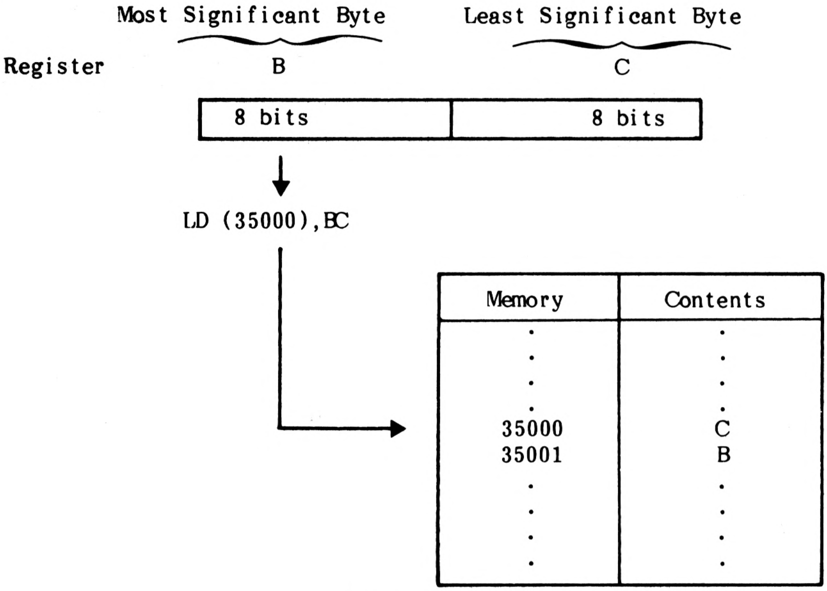


Figure 3.1

The 'Least Significant Byte' (LSB) is stored first, followed immediately by the 'Most Significant Byte' (MSB).

EXERCISE 3.1

Write down the order in which the registers would be stored if the following sub-program was executed.

```
LD (200),DE
LD (202),HL
```

See the solutions chapter for the answer.

When paired, the C, E and L registers are classed as the Least Significant Bytes, hence the Most Significant Bytes are registers B, D and H.

Now another exercise -

EXERCISE 3.2

Write a program to load memory location 35000 with 100 and location 35002 with 400 and then plot a line to these points.

Note the graphics cursor will have to be reset to 0,0. Either jump into BASIC by selecting X to do this with PLOT 0,0 or for the more adventurous; the Amstrad contains a machine-code routine to plot a point, thus we could incorporate this into the program. This would save using BASIC to reset the graphics cursor.

The format of this call is as follows:-

Calling address 48106.

DE = X coordinate
HL = Y coordinate

(Pretty much the same as the line plotting routine)

So to plot a point at 0,0 we have to load DE with 0 and HL with 0, then call 48106.

A possible answer is given in the solutions chapter.

EXERCISE 3.3

Write a program which will join the following points together with lines:

X	Y
200	300
400	200
0	0

N.B. Start with 200,300, remember, you load DE with the X coordinate and HL with the Y coordinate.

Possible solutions are given in the solutions chapter.

The Z80 chip also supports various other addressing modes. So far, the immediate mode for both 8 and 16-bit data, as well as direct mode for 16-bit data have been used. Another mode is Indirect.

Indirect mode addressing

In this mode of addressing, the contents of a register are used to point to a memory location. This mode is very useful, as we can load a 16-bit register with the starting memory address of our data, read the data, then increment (i.e. add 1 to) the register pair and access the next item of data.

The instruction family is as follows:-

LD r,(HL)	LoaDs register r with the contents of memory location pointed to by HL. (This in effect means that we load HL with the memory address).

LD (HL),r	LoaDs the memory location pointed to by HL with the contents of register r.

LD A,(BC) LoaDs the accumulator with the contents of memory location pointed to by BC.

LD A,(DE) LoaDs the accumulator with the contents of memory location pointed to by DE.

LD (BC),A LoaDs the memory location pointed to by BC with the contents of accumulator.

LD (DE),A LoaDs the memory location pointed to by DE with the contents of the accumulator.

The above table may appear formidable at first, hopefully not, anyway; all will be explained as the instructions are used in examples.

What we will do now, to illustrate the use of some of the above commands, is to store the ASCII representation of three A's in memory, then print them on the screen using the HL register as a 'pointer'.

To simplify loading the three A's into memory we will use BC as a moving pointer. Thus, if the accumulator contains the code for 'A' (65), then using LD (BC),A and INC BC we can load three A's into consecutive memory locations, starting at the address held in BC, by incrementing BC three times.

Here is a program to do this:-

PROGRAM 3.2 (don't assemble this yet)

```
        ENT
        LD BC,35000             BC=start of data in memory
        LD A,65                 A=65, the ASCII code for 'A'
        LD (BC),A               store first A
        INC BC                  increment pointer
        LD (BC),A               store second A
        INC BC                  increment pointer
        LD (BC),A               store third A
```

Now to write the program to read the data, which starts at 35000, onto the screen.

```
        LD HL,35000             HL=start address
        LD A,(HL)               A=contents of memory location
                                addressed by HL
        CALL 47692              Print contents of A on screen
        INC HL                  Increment pointer
        LD A,(HL)               Reload A
        CALL 47692              Print second character
        INC HL                  Increment pointer
        LD A,(HL)               Reload A
        CALL 47692              Print third character
        RET                     Return.
```

Assemble and run this program now. While this program accomplishes its aim, it is not very efficient. Let's now see how we can improve upon it.

Considering the first section:-

We duplicate the following commands three times, to set up the data.

```
        LD (BC),A
        INC BC
```

Why not set up a loop? If we load an unused register, say E, with 3, then everytime we load a memory location, we decrement E and test it to see if this results in a zero. If so, we end the loop, otherwise we jump back to the beginning of the loop. This method accomplishes the same task with greater elegance and efficiency.

Here is the first section of the modified program:-

PROGRAM 3.3 (Don't assemble this yet)

```
        ENT
        LD BC,35000
        LD A,65
        LD E,3                  load count register
NXT:    LD (BC),A
        INC BC
        DEC E                   decrement count
        JR NZ,NXT:              if not zero jump back to NXT
```

Note the use of the label NXT to simplify the calculation of the jump address.

No great time is saved in typing this program in: imagine though using the previous method to store 200 'A's.

The same techniques can be applied to the second section.

EXERCISE 3.4

Add a second section to the program, using a loop to print out the three A's, then assemble and run it.

EXERCISE 3.5

Write a program to store the alphabet in memory, then read it out of memory onto the screen.

See the solutions chapter for possible answers.

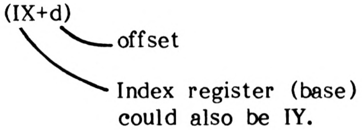
Let's briefly recap upon the addressing modes used so far.

Register -Register	Immediate
LD B,A	LD A,83 or LD BC,300
Direct	Indirect
LD A,(2000)	LD B,(HL)

You will find a full list of the Z80 mnemonics in the appendices. Most of the 8 and 16-bit load groups should be comprehensible by now.

Indexed addressing

In Program 3.3 we used a register pair to access a sequentially stored block of data. While in certain applications this method is very efficient, the Z80 offers an alternative. In addition to the registers already introduced the Z80 contains two 16-bit 'index registers', IX and IY. If you look up the word 'memory' in the index of this book, you will probably start by finding the beginning of the M's, then searching for the word 'memory'. In other words you have treated the start of the M's as a base and then started your search from there. This is directly analogous to Indexed Addressing. The IX or IY register is loaded with a base address. Then, to access a piece of information, an offset is added to this base. The general format for the indexed part of an instruction is as shown below:-

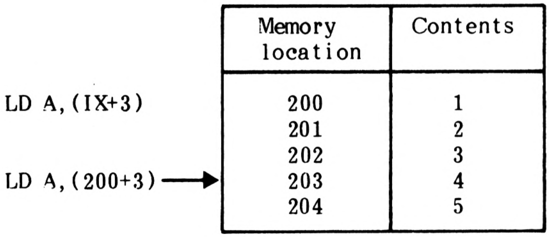


Where d is a number in the range -127 to +128. Here is a sample instruction using the index register IX:

```
        LD A,(IX+3)
```

What will this do?

Suppose IX contains 200. What value will be loaded into A?



Memory location	Contents
200	1
201	2
202	3
203	4
204	5

FIGURE 3.2

The accumulator will therefore contain 4 after this instruction. What offset would you have had to use to load A with 3? Answer: (IX+2)

Instead of using the IX register as a base we could have used IY with the instruction:

```
        LD A,(IY+3)
```

Now let's review the load instructions which use either IX or IY.

Indexed addressing Instructions

There is a shorthand method of describing the operation of an instruction, referred to as its symbolic operation. Here is an example:-

Instruction	Symbolic Operation
LD r,(IX+d)	r ←(IX+d)

This loads register r with the contents of the memory location pointed to by the addition of IX and d. The appendices contain a complete table of Z80 instructions including symbolic operations.

Here are the indexed loading instructions:

Instructions	Symbolic Operation
LD r,(IX+d)	r ← (IX+d)
LD r,(IY+d)	r ← (IY+d)
LD (IX+d),r	(IX+d) ← r
LD (IY+d),r	(IY+d) ← r
LD (IX+d),n	(IX+d) ← n
LD (IY+d),n	(IX+d) ← n

Now let's see how we can load the IX or IY registers themselves. Look up the 16-bit load group table, and see if you can identify what instructions to use.

Here they are; the format should appear familiar:-

Instruction	Symbolic Operation
LD IX,nn	IX ← nn
LD IY,nn	IY ← nn
LD IX,(nn)	IX ← (nn)
LD IY,(nn)	IY ← (nn)
LD (nn),IX	(nn)←IX
LD (nn),IY	(nn)←IY

One point to note: it is not possible to compute the value of 'd', it has to be given absolutely - i.e. as an immediate value.

To illustrate the use of index registers, Program 3.2 has been rewritten.

Note: From now on the ENT instruction will be omitted from all programs: it will be assumed that you will automatically insert it as the first line.

Here is the program:-

PROGRAM 3.4

```
        LD IX,35000          IX=base
        LD A,65
        LD (IX+0),A          first 'A' stored
        LD (IX+1),A          second 'A' stored
        LD (IX+2),A          third 'A' stored
```

The second section, which prints out the contents of memory on the screen, is as follows:-

```
        LD A,(IX+0)
        CALL 47692           print  first character
        LD A,(IX+1)
        CALL 47692           print second character
        LD A,(IX+2)
        CALL 47692           print third  character
        RET
```

Now try it.

Because the Z80 has to add the offset to the base for indexed instructions, it requires more time to execute than an indirect or immediate instruction. Also, because 'd' has to be given as an absolute value, indexed addressing is not much good in a loop, as 'd' cannot be incremented. However, the main advantage of IX and IY lies in their ability to access data, which is, stored in memory relative to some known memory location - i.e. for accessing tables of data, where various items of information are stored using known offsets.

EXERCISE 3.6

Write a program using IX to store five letters, e.g. a name. Then print the third and fourth letters on the screen.

The following table may help.

Memory location	Letter	Code for letter
35000	S	83
35001	U	85
35002	S	83
35003	A	65
35004	N	78

A possible solution is given in the solutions chapter.

Well done! The end of another chapter; all that is left now is a summary of what you have learnt during this chapter. If you feel up to it, try writing your own programs. How about printing a character, erasing it by printing a space over it, then printing it in another location. This could be the basis of a game!

Summary

The following terms should now be understood:-

1. Register Pairs

2. Addressing Modes

   a. Register-Register
   b. Immediate
   c. Direct
   d. Indirect
   e. Indexed

3. M.S.B. and L.S.B.

4. Symbolic Operation

5. Although not specifically mentioned, the following commands will probably make sense (ss is a register pair).

```
INC ss
DEC ss
```

6. You should now be able to understand these three instructions which were mentioned briefly in Chapter 2.

```
DEC (HL)
DEC (IX+d)
DEC (IY+d)
```

Where ss is any one of the following (ignore SP for now)

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

d is a number in the range -126 to +129.

CHAPTER 4

Arithmetic operations

Most real life situations require the manipulation of numbers. So far, we can only increment and decrement the contents of registers or memory locations. The Z80 provides us with some additional arithmetic instructions which although not extensive form the basics for more powerful operations.

This chapter explains these instructions. If you are not familar with the binary or hexadecimal representation of numbers turn to the Appendix 5 and work through it.

Welcome back.

The concepts of binary and hex (hexadecimal) should now be familar. You will see why we use hex soon; it saves typing for one thing!

We can classify the arithmetic instructions into two groups, the 8 and 16 bit. In this chapter we shall be concerned mainly with the 8-bit group, as the 16-bit group naturally follows on.

The most common arithmetic process is adding two numbers. To add two 8-bit numbers the following instruction is used:

ADD A,n	ADD n to the contents of the Accumulator, replacing the contents of the accumulator with the result.

To illustrate this command we will add 65 and 20 giving 85. When the result is printed on the screen we will see a U. (The character corresponding to the ASCII value of 85).

Try the following program.

PROGRAM 4.1

```
        ENT
        LD A,65         Load A with 65
        ADD A,20        ADD 20 to A
        CALL 47962      Print the result on the screen
        RET
```

Lo and behold a U appears on the screen.

The assembler also allows numbers to be represented in hexadecimal. To allow the assembler to distinguish between them, hexadecimal numbers are preceeded by a '&'.

Rewriting Program 4.1 using the hexadecimal representation for 47962, 65 and 20:

PROGRAM 4.2

```
        ENT
        LD A,&41
        ADD A,&14
        CALL &BB5A
        RET
```

Verify that this program is exactly the same as Program 4.1 by running it.

> EXERCISE 4.1
>
> Write a program which will add 200 and 48 together then print the result on the screen.

> EXERCISE 4.2
>
> Write a program that will add &41 and &10 together. Then print the result on the screen. What will be printed?
>
> Possible answers are given in the solutions chapter.

So far all our answers have been less than 255, the maximum number an 8 bit register can hold. What do you think will happen if 150 and 171 are added together? Try the following program. Don't forget ENT.

PROGRAM 4.3

```
        LD A,150         Load A with 150
        ADD A,171        ADD 171 to A
        CALL  47962      Print the result on the screen
        RET
```

What has happened is that the accumulator has overflowed. It reached 255 then when 1 more was added it went back to zero and started again, reaching 65; hence the A on the screen.

What cannot be seen from the program is the contents of the flags register. If observable, it would be seen that as the accumulator overflowed the carry bit was set (1). This fact can be used to add together two numbers whose sum is greater than 255. This process will be illustrated on paper first, then a program will be written to accomplish the same task in assembly language.

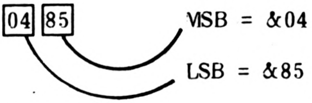
Problem: Add 1157 to itself. i.e: 1157 + 1157 = ?

Convert 1157 to Hex.

```
1157 ÷ 4096 = 0 remainder 1157
1157 ÷ 256  = 4 remainder 133
133  ÷ 16   = 8 remainder 5
5    ÷  1   = 5 remainder 0
```

Thus 1157 = &0485.

&0485 expressed in binary is a 16-bit number. This therefore needs to be split in half to allow the use of 8-bit arithmetic instructions. This is easily accomplished in hex. (Which is why it is used!)



Note the terms MSB and LSB, meaning Least and Most Significant Byte, respectively.

Step 2

To add the two &0485's together we must first add the LSB's then add the MSB's, taking into account any carry generated by the addition of the LSB's.

```
                85                        8   5
               +85                       +8   5
     +carry     0A       + carry          6  10    Decimal
                         + carry          0   A    Hex
```

Thus the result is &0A + a carry.

Step 3

Add the MSB's taking the carry bit into account.

```
     04
   + 04
     08
```

Now add the carry bit:

```
     01 ◄──carry bit
   + 08
     09
```

Step 4

Recombine the new LSB and MSB.

1157 + 1157 = &090A

Now satisfy yourself that &090A equals 2314.

The main point to note is that in all DOUBLE PRECISION work, i.e. using 16-bit numbers, the LSB is always operated upon first, so that the carry (if any) can be taken into account.

Before we can write the program to accomplish this addition we have to be aware of some new instructions:-

AND A	Logical AND of the accumulator.

Only one effect of this operation need to be observed at the moment, which is resetting the carry flag to zero. Why is this necessary? Ans: if the carry flag was unintentionally set the result after the addition would be incorrect due to the inclusion of the incorrect carry in the addition.

An instruction which adds two registers plus the contents of the carry flag is now required.

ADC A,s — ADd the contents of register s plus the Carry flag to the contents of the Accumulator. The result is stored in the accumulator.

Note that the ADC instruction uses a register as one of its operands. An instruction similar to ADD A,n exists using a register in place of the immediate data n.

ADD A,s — ADD the contents of register s to the contents of the Accumulator. The result is stored in the accumulator.

Now on to the program.

PROGRAM 4.4 (don't assemble this yet)

```
LD C,&85          Load C with the 1st LSB.
LD A,&85          Load A with the 2nd LSB.
AND A             Clear the carry flag.
ADD A,C           A = LSB + LSB.
LD (&7000),A      Save new LSB.
LD C,&04          Load C with the 1st MSB.
LD A,&04          Load A with the 2nd MSB.
ADC A,C           A = MSB + MSB + carry.
LD (&7001),A      Save new MSB.
```

Well this program adds the two numbers together. But how can the result be checked? What is now needed is a program to display the answer in a recognizable format. Why can't the contents of the two memory locations be printed on the screen? Nothing recognizable would happen, as the ASCII codes &09 and &0A do not represent characters but control codes. An offset has to be added to both answers to bring them into the ASCII alphabet range (65-122).

As the code for A is 65, this seems a reasonable offset to use. Thus adding the following lines to the previous program from where we left off:

PROGRAM 4.4(a)

```
        LD C,65
        LD A,(&7001)
        ADD A,C
        CALL &BB5A
        LD A,(&7000)
        ADD A,C
        CALL &BB5A
        RET
```

Now assemble and run the whole program. You will see a J and K on the screen (corresponding to &09 + 65 and &04 + 65).

EXERCISE 4.3

Write a program to add together 250 and 600, in hexadecimal, adding 65 to the MSB and LSB. Then print the result on the screen.

Subtraction

SUB	s	SUBtracts the contents of register s from the accumulator. The result is stored in the accumulator.

SBC	A,s	SUBtracts the contents of register s plus the Carry Flag from the Accumulator. The result is stored in the accumulator.

The process of subtracting two 8-bit numbers is left for Exercise 4.4. Try it!

EXERCISE 4.4

Write a program that will subtract 9 from 233. Then print the result on the screen. Note that there is know need to add the offset of 65 to the result. Why?

EXERCISE 4.5

Write a program that will calculate the answer to the following sum:-

(97 + 126) - 153 = ?

Possible answers are given in the solutions chapter.

As seen previously, when adding two numbers that give a result greater than 255, overflow occurs. This overflow causes the carry flag to be set.

The opposite situation occurs when trying to subtract a large number from a smaller one. For example, try the following:-

$$\begin{array}{r} 25 \\ -\ \underline{17} \\ \underline{?} \end{array}$$

It is not possible to directly subtract 7 from 5 so a unit is borrowed from the next column. The first step in the subtraction now becomes:

$$\begin{array}{rl} 5 & +\text{ borrow } = \\ -\ \underline{7} & \\ \underline{?} & \end{array} \qquad \begin{array}{r} 15 \\ -\ \underline{7} \\ \underline{8} \end{array} \qquad \text{Ans1=8}$$

As we have borrowed we have to 'pay back' i.e: subtract one from the next column.

Completing the subtraction.

$$\begin{array}{rl} 2 & -\text{ borrow } = \\ -\ \underline{1} & \\ \underline{?} & \end{array} \qquad \begin{array}{r} 1 \\ -\ \underline{1} \\ \underline{0} \end{array} \qquad \text{Ans2=0}$$

Combining the two answers (Ans1 and Ans2)

0 + 8 = 8

Thus:-

27 - 17 = 8

By using the same process we can perform this subtraction using the Z80 subtract instructions.

Problem: Subtract 2000 from 2224.

Step 1

Convert the two numbers to Hex.

```
2000 ÷ 4096  = 0          remainder 2000
2000 ÷ 256   = 7          remainder 208
208  ÷ 16    = D          remainder 0
0    ÷ 1     = 0

2224 ÷ 4096  = 0          remainder 2224
2224 ÷ 256   = 8          remainder 176
176  ÷ 16    = B          remainder 0
0    ÷ 1     = 0


2000   = &07D0

2224   = &08B0
```

Step 2

Subtract the two LSB's

```
  B0 + borrow =       1B0
- D0                   D0
  ??                   E0

                     0=0-0
                     E=1B-D
```

Step 3

Now the MSB's

```
  08                  08
- 07                - 07 + borrow =  - 08
  ??                  00
```

Combining the two results:

2224-2000 = &00E0

When carrying out this form of subtraction the carry flag acts as a borrow flag, being set when a borrow occurs. Let's now convert this subtraction into Z80 instructions.

PROGRAM 4.5

```
LD C,&D0          C= 1st LSB
LD A,&B0          A= 2nd LSB
AND A             Clear carry flag
SUB C             A= new LSB
LD (&7000),A      Save LSB
LD C,&07          C= 1st MSB
LD A,&08          A= 2nd MSB
SBC A,C           A=A-C-carry
LD (&7001),A      Save MSB
LD A,(&7000)      Recall LSB
CALL &BB5A        Print result
RET
```

Points to note: firstly, as the MSB of the answer is known to be zero there is no need to print it. Secondly, there is no need to add any offset to the answer as it is within the printable range of ASCII characters (65-255).

EXERCISE 4.6

Write a program to perform a 16 bit subtraction using the following instruction (ss is a register pair - i.e. BC or DE in this case).

SBC HL,ss

eg. 4248 - 4008 = ?

(Hint: it is a lot easier than Program 4.5.)

Answer in the solutions chapter.

EXERCISE 4.7

Store two numbers in memory, then add the contents of the accumulator to both. Replace the old values with the new calculated results.

The following tables may help.

Memory location	Contents
35000 35001	10 20

Accumulator = 65

After program has been run

Memory location	Contents
35000 35001	75 85

Use the instruction ADD A,(HL). Verify that the results are as expected by printing the contents of the memory location on the screen.

Answer in the solutions chapter.

EXERCISE 4.8

Draw a line to the point 100,50 on the screen using the line drawing routine. Then add 75 to each coordinate and draw another line, to this point.

The line drawing routine is fully documentated in the appropriate appendix, but briefly:-

Calling Address 48118

Parameters X, Y

X passed in DE
Y passed in HL

A possible answer is given in the solutions chapter.

Well that concludes the basics of addition and subtraction. Although not explicitly mentioned, the following instructions should be self explanatory.

Instruction	Symbolic Operation
ADD HL,ss	HL ← HL+ss
ADC HL,ss	HL ← HL+ss+carry
SBC HL,ss	HL ← HL-ss-carry
ADD IX,pp	IX ← IX+pp
ADD IY,rr	IY ← IY+rr

Where ss is any one of BC,DE,HL or SP
      pp is any one of BC,DE,IX or SP
      rr is any one of BC,DE,IY or SP

You should be able to understand what each instruction accomplishes from the symbolic operation. It is possible to add or subtract 32-bit numbers using the above instructions and the carry flag, using the same method by which 16-bit arithmetic was carried out using 8-bit instructions.

The carry flag Instructions

Before any arithmetic operations, the carry flag has to be reset. This process has been accomplished using the AND A instruction. The carry flag can also be reset by using the two following instructions:

SCF	Set the Carry Flag

CCF	Complement the Carry Flag

To reset the carry flag with these instructions, firstly the carry flag is set using the SCF instruction, then it is complemented by using the CCF instruction. When a binary number is complemented, a zero is replaced by a one; likewise a one is replaced by a zero. The carry flag when set will equal one, so, after complementing it, it will equal zero. The reason AND A was used is that it requires half the time required by SCF and CCF.

Summary

The basics of 8 and 16-bit addition and subtraction should now be understood, including the use of the carry flag.

CHAPTER 5

Binary Coded Decimal and Logical Operators

In addition to numbers being represented in binary, hex and decimal notation, another representation exits. This is given the rather grand name of Binary Coded Decimal. Whereas in binary one number (0-255) is normally stored per byte, in BCD the byte is split into two. The name given for each 4-bit half-byte is a nybble!

For example, 7564 stored in BCD requires 2 bytes (4 digits to encode at 2 digits per byte).

	7	5
Byte 1	0 1 1 1	0 1 0 1

	6	4
Byte 2	0 1 1 0	0 1 0 0

Note that byte 1 is not necessarily stored in memory first. Its location depends upon the storage format used in the program.

It may have crossed your mind that 4 bits are used to store each number (0-9) when 3 bits could uniquely store them. The reason 4 bits are used is that BCD has been around much longer than micros. It was first used on mainframes where it was more efficient to store BCD digits in 4 bits. BCD is used a lot in accountancy programs as well as being very useful for passing information to certain forms of display devices (e.g. the 7-segment displays seen in calculators and digital watches).

Let's now see how two BCD numbers are added together.

```
Decimal      BCD

  8          1000
 +2         +0010
 ——          ————
 10          1010
 ——          ————
```

This is fine but 1010 is 10 in decimal and therefore too large to be stored in one BCD nybble which can only store numbers 0 to 9. Some form of adjustment is required for answers bigger than nine. By adding 6 to any result greater than 9 the correct result is obtained:

Thus:

```
               First   Second
               nybble  nybble


               0000    1010
             + 0000    0110 :- 6 in decimal
               ----    ----
               0001    0000
               ----    ----
```

Giving the correct representation of 10 in BCD; 0001 0000.

It would soon get very annoying if every time we performed some BCD arithmetic we had to check the validity of the result and correct it if necessary. Fortunately the Z80 contains a instruction to do this for us:

DAA   Decimal Adjust the Accumulator

Now on with the programming.

Program 5.1 adds 8 and 2 together to produce the answer in BCD.

PROGRAM 5.1

```
LD A,8
ADD A,2
DAA
ADD A,65
CALL 47962
RET
```

Note the use of the offset (65) to bring the result into the alphabet range.

The result, before the offset is added, will be the BCD representation of 10 (0001 0000). In decimal this binary value corresponds to 16. Thus the character with the ASCII code 16+65 will be printed on the screen(Q).

EXERCISE 5.1

Add 7 and 12 together in BCD then print the answer as a letter on the screen.

EXERCISE 5.2

Subtract &12 from &35 then convert the answer to BCD and print the answer as a letter on the screen.

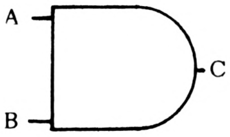
Possible answers are given in the solutions chapter.

In some situations it is necessary to extract the least significant nybble from a byte. This is accomplished by removing the most significant nybble. This can be done with the instruction

AND s	Logical AND of operand s

Where s is any of A,B,C,D,E,H,L,(HL),(IX+d),(IY+d).

Anyone who has come into contact with digital electronics will recognise the following symbol. It is the symbolic representation of an AND gate.



AND Gate

FIGURE 5.1

The AND gate functions as follows; if and only if both inputs A and B are set (1) then the output C will be set. Describing the operation of an AND gate is fairly easy, but as the logic of gates becomes more complex it becomes increasingly difficult to describe clearly. The answer to this problem is to use a truth table. A truth table enables a logic gate's output to be easily and quickly derived from a given set of inputs.

For example here is the AND gate's truth table.

INPUTS		OUTPUT
A	B	C
0	0	0
0	1	0
1	0	0
1	1	1

Truth Table for Logical AND

FIGURE 5.2

Notice C is set only when both A AND B are, hence the name AND.

EXERCISE 5.3

Using Figure 5.2 decide whether C would be set or not for the following inputs

1 A=0 B=1
2 A=1 B=1
3 A=0 B=0

When the Z80 chip performs an AND instruction it operates upon 8 bits at a time. Take a look at this:

Problem: What is the result of ANDing 10101101 with 00001111?

Solution:

```
        10101101
    AND 00001111
        --------
        00001101
        ========
```

What has happened?

The most significant nybble has been 'masked off', i.e. all the bits comprising the MSN have been set to zero whilst those in LSN have been left untouched.

This process is very powerful in that it allows us to extract any portion of a byte using a suitable mask. e.g. To mask off the LSN of 10111011 we would AND it with 11110000.

```
    1010  1101
AND 1111  0000
    ----------
    1101  0000
    ----------
```

Generally any bit position that is required to remain intact is ANDed with one, the rest with zeros. For example, if the result of ANDing an eight-bit number with 00000011 is 2, then we know that the number, whatever it is, is divisible by 2.

EXERCISE 5.4

What mask would have to be used to result in 00000011 from 10101011?

EXERCISE 5.5

What is the result of ANDing 253 with 75?

The answers are given in the solutions chapter.

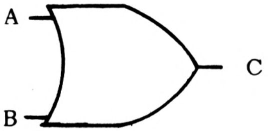
Now to write a program using the AND instruction. Program 5.2 logically ANDs 225 with 254 and then prints the result on the screen.

PROGRAM 5.2

```
LD C,225
LD A,254
AND C
CALL 47962
RET
```

This will produce a 'face' on the screen, the Amstrad's ASCII symbol for 224.

AND is not the only logical operator that the Z80 supports. Let's now investigate the OR gate. The standard symbol for an OR gate is



OR Gate

FIGURE 5.3

The output C is set (1), if either A or B is set (1) or both A and B are set. This is expressed in the following truth table.

INPUTS		OUTPUT
A	B	C
0	0	0
0	1	1
1	0	1
1	1	1

Truth Table for Logical OR

FIGURE 5.4

The Z80 OR instruction is:-

OR s	Logical OR of operand s

Where s is any of A,B,C,D,E,H,L,(HL),(IX+d),(IY+d).

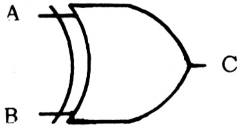
EXERCISE 5.6

What is the result of the following logical operations?

1. 1001 OR 1101 (binary numbers)
2. 250 OR 25
3. (209 OR 20) AND 27

Answers are given in the solutions chapter.

It may seem strange that an OR gate produces an output when **both** inputs are set (1); indeed this can be a problem in certain cases. Thus the 'exclusive OR' gate is used, overcoming this problem. The symbol used to represent an XOR (eXclusive OR) is:-



XOR Gate

FIGURE 5.5

INPUTS		OUTPUTS
A	B	C
0	0	0
0	1	1
1	0	1
1	1	0

XOR Truth Table

FIGURE 5.6

The corresponding Z80 instruction is:

XOR s	eXclusive OR of operand s

Where s is any of A,B,C,D,E,H,L,(HL),(IX+d),(IY+d).

EXERCISE 5.7

What is the result of the following logical operations? Try them on paper first and then verify your answers by writing a program.

1. 1011 XOR 1110100
2. 77 XOR 200
3. (25 OR 255) AND 200

Answers are given in the solutions chapter.

Signed numbers

So far all the numbers dealt with have been positive. In many situations a need arises for negative numbers.

How can negative numbers be represented in binary? The method used allows the Z80 to treat negative numbers in much the same way as it treats positive numbers. This method is given the rather grand name "two's complement". Before investigating two's complement, however, it is necessary to understand the concept of one's complement!

One's complement

When using one's complement notation all positive integers are represented in binary as usual. However, negative numbers are represented by replacing all the 1's in a byte with 0's, and replacing all the 0's with 1's.

For example:

```
+9 = 1001
```

Now replace all the 1's by 0's and all the 0's by 1's, i.e: 1's complement +9:

```
1001      1's complement = 0110
```

Thus in 1's complement notation -9 = 0110

But 0110 also represents +6; this is in fact one of the problems of 1's complement notation.

EXERCISE 5.8

Convert the following numbers to their 1's complement form.

1. 1011
2. 1011101
3. 14

Answers are given in the solutions chapter.

Two's complement

As in 1's complement positive integers are represented normally in binary. Negative numbers are first 1's complemented then one is added to the result. It may seem like a strange way of representing negative numbers at first, but it works. Let's now try adding 7 and -5 using 2's complement for -5.

```
          5 =              0101
    1's complement         1010
    Add 1                + 0001
                           ----
                           1011   = -5
                           ----
```

Now:-

```
          7                0111
          +(-5)            1011
          -                ----
          2                0010   + carry
          -                ----
```

Ignoring the carry bit, the answer is correct. The fact that the carry bit can be disregarded, with the result remaining correct, considerably aids writing simple arithmetic programs dealing with negative numbers.

EXERCISE 5.9

Calculate the answer to each of the following problems using the 2's complement representation for negative numbers.

1. -3 + 10 = ?
2. -1 + 7 = ?
3. -10 + 8 = ?

The answers are given in the solutions chapter.

Using 4 bits, the decimal numbers represented by 2's complement notation are as below:

Decimal	Binary
+7	0111
+6	0110
+5	0101
+4	0100
+3	0011
+2	0010
+1	0001
0	0000
-1	1111
-2	1110
-3	1101
-4	1100
-5	1011
-6	1010
-7	1001
-8	1000

The 16 unique combinations of 1's and 0's which result from 4 bits no longer represent the integers 0-15 but the integers +7 to -8. Similary when using 8 bits the valid range of numbers is to +127 to -128. This can lead to problems when adding together two numbers that result in a number greater than +127.

e.g:

```
  100    01100100
   79    01001111
  ___    ________
  179    10101011
  ___    ________
```

In 2's complement 10101011 represents -85; this is clearly the wrong answer. Overflow is said to have occurred. This is detected by the P/V (Parity/oVerflow) flag and it is up to the programmer how to act on this information. One blunt solution is to declare the result invalid. Overflow will generally occur under the following conditions.

When:-

1. Adding together two large positive or negative numbers.

2. Subtracting a large positive number from a large negative number or subtracting a large negative number from a large positive number.

The Z80 contains instructions which convert the contents of the accumulator into either 1's or 2's complement, as follows:

NEG	NEGates the contents of the accumulator (exactly the same as 2's complement)
CPL	ComPLements the contents of the accumulator (1's complement).

Program 5.3 below calculates the answer to the sum -112 +104.

PROGRAM 5.3

```
LD A,112          A=112
NEG               A=-112
ADD A,104         A=-112+104
CALL &BB5A
RET
```

When run this program will print a little man on the screen, the Amstrad's character for ASCII code 248. The binary representation of 248 is 11111000 which using the 2's complement notation represents -8. The important point to notice here is that the Z80 cannot distinguish between 248 and -8. 2's complement is a concept the programmer uses to represent negative numbers, not the Z80, and as such, care is needed when using it to ensure the expected result is obtained.

EXERCISE 5.10

Use the CPL and INC instruction instead of NEG to calculate the result of the following sum

-20 + 98 = ?

A possible answer is given in the solutions chapter.

That's 'yer lot' for another chapter; now to summarise.

Summary

The following ideas and concepts should now be familar.

1. BCD Arithmetic

2. Logical Operators

    1. AND
    2. OR
    3. XOR
    4. Logical Masks

3. Signed Numbers

    1. 1's Complement
    2. 2's Complement
    3. Overflows

The following instructions should also be recognised.

DAA

CPL

NEG

AND s

OR s

XOR s

CHAPTER 6

Multiplication, Division and the Rotate Group

Almost all real life computer applications require the manipulation of numbers. Whilst some only require basic adding and subtracting, many require multiplication and division. This chapter looks at how these arithmetic functions can be implemented in assembly language.

Binary Multiplication

Before we embark upon Binary Multiplication, let's examine the decimal multiplication process. Take the sum 13x14. We define 13 as the MULTIPLICAND and 14 as the MULTIPLIER and lay out the multiplication as below:-

```
 13  Multiplicand
 14  Multiplier
---
 52
130
---
182  Answer
```

The multiplication is performed by multipling the multiplicand by the rightmost digit of the multiplier and storing this result as the first "partial product". i.e. 13x4=52. Next we multiply the multiplicand by the next digit of the multiplier, resulting in the second partial product. i.e. 1x13=13. This partial product is written down shifted one bit position to the left. The two partial products are now added together resulting in 182, the correct answer.

It is quite possible to use the same method when performing binary multiplication.

For example, to multiply 5x7 in binary:-

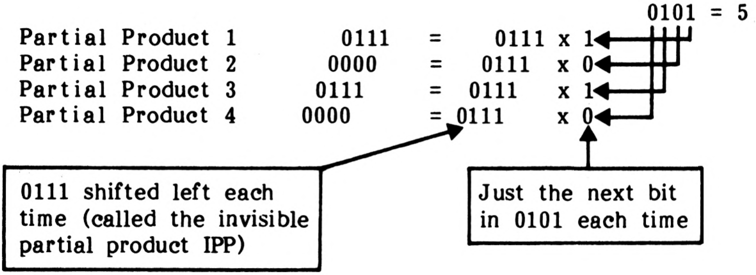
```
          5   = 0101   (working only to 4 bits)
          7   = 0111


                             0 1 1 1      = (7)
                           x 0 1 0 1      = (5)
                          ___________

                                        Adding as we go!
Partial Product 1            0 1 1 1         0 1 1 1
Partial Product 2          0 0 0 0 0         0 1 1 1
Partial Product 3        0 1 1 1 0 0     1 0 0 0 1 1
Partial Product 4      0 0 0 0 0 0 0     1 0 0 0 1 1

                             Ans 100011
```

100011= (1x32)+(0x16)+(0x8)+(0x4)+(1x2)+(1x1)=35

With 1 as the rightmost digit of the multiplier, the partial product has the same pattern of digits as the multiplicand i.e: 0111. Otherwise the partial product is zero, i.e. 0000. Every new partial product is written down shifted one bit position to the left. Thus binary multiplication reduces to successive additions and shifts.

8-Bit Multiplication

In order to perform a multiplication using the Z80 we will use the accumulator to hold the 'running total', register C for the multiplicand and register E for the multiplier. Let's now see how the partial products are formed.



```
                                                0101 = 5
Partial Product 1        0111   =    0111 x 1
Partial Product 2       0000    =   0111  x 0
Partial Product 3      0111     =  0111   x 1
Partial Product 4     0000      = 0111    x 0
```

0111 shifted left each time (called the invisible partial product IPP)

Just the next bit in 0101 each time

The problem of binary multiplication can be expressed in a flowchart as follows:

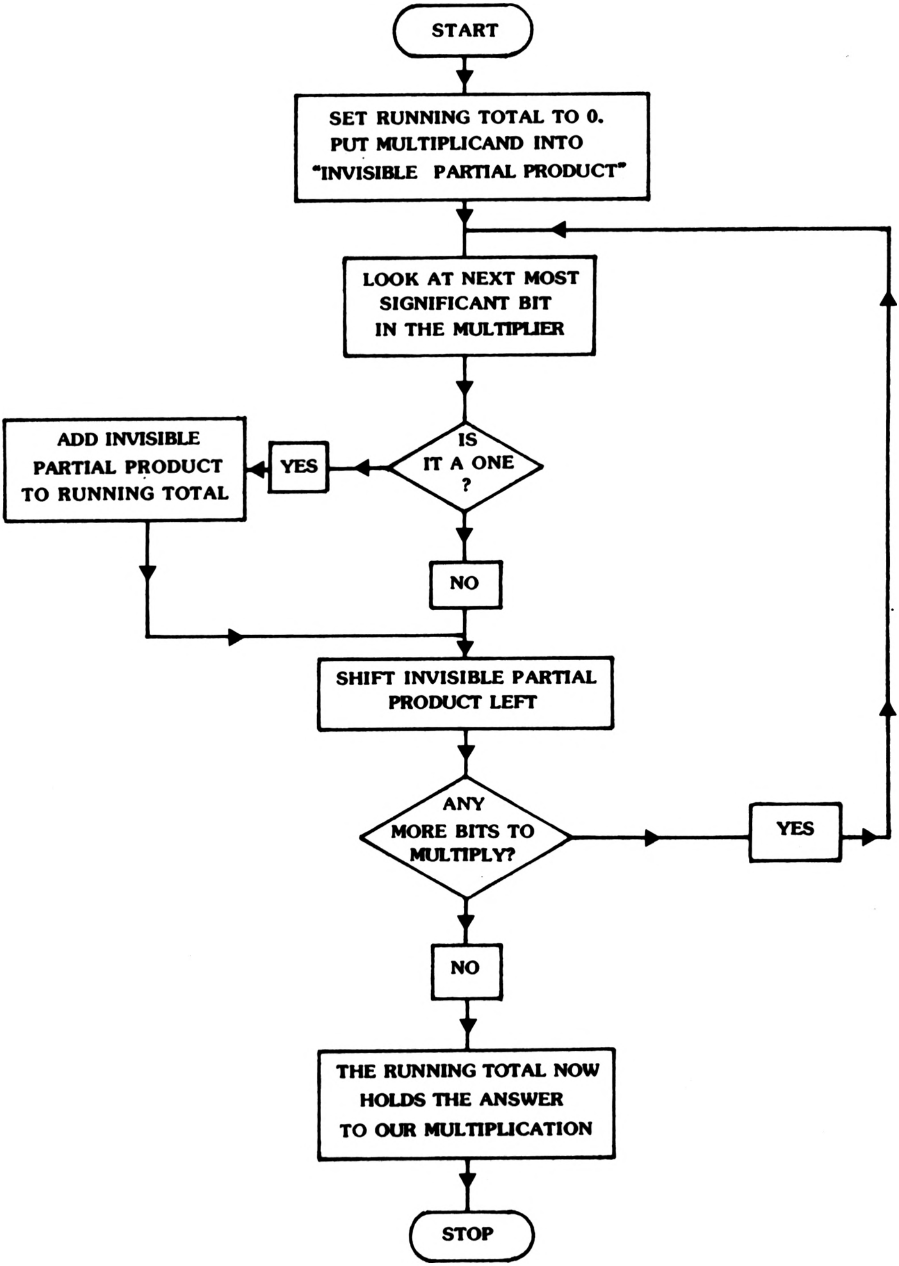
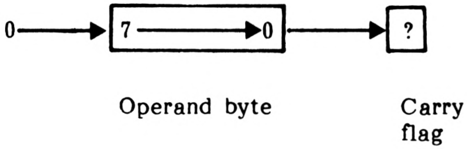

START
SET RUNNING TOTAL TO 0.
PUT MULTIPLICAND INTO
"INVISIBLE PARTIAL PRODUCT"
LOOK AT NEXT MOST
SIGNIFICANT BIT
IN THE MULTIPLIER
IS
IT A ONE
?
YES
ADD INVISIBLE
PARTIAL PRODUCT
TO RUNNING TOTAL
NO
SHIFT INVISIBLE PARTIAL
PRODUCT LEFT
ANY
MORE BITS TO
MULTIPLY?
YES
NO
THE RUNNING TOTAL NOW
HOLDS THE ANSWER
TO OUR MULTIPLICATION
STOP

FIGURE 6.1

All that is required before starting to write the program are the instructions that enable us to shift bits within bytes.

SRL s	Logical Shift Right of operand s

Represented diagramatically:



For example, consider shifting 10110111 right with the instruction SRL.

	Byte								Carry flag
Before	1	0	1	1	0	1	1	1	?
After	0	1	0	1	1	0	1	1	1

Bit 7 has been replaced with a 0, bits 1 to 6 shifted right one position and bit 0 moved into the carry flag.

Now onto the program:

PROGRAM 6.1

```
        LD A,0              Zero Accumulator
        LD C,7              C = Multiplicand
        LD E,5              E = Multiplier
        LD B,4              B = Count (for 4bits)
ADD:    SRL C               Shift C right
        JR NC,NOADD:        If rightmost bit = 0 then
                            jump to NOADD
        ADD A,E             Add IPP to running total in
                            accumulator
NOADD:  SLA E               Shift IPP left
        DEC B               Decrement counter
        JR NZ,ADD:          Jump back to ADD if there are
                            still some more bits to multiply
        ADD A,65            Add offset
        CALL &BB5A          Print character on screen
        RET                 Return
```

An offset of 65 is used resulting in the character with an ASCII code of 100 being printed on the screen, i.e. a lower case d (The answer to 7x5 was 35, 35+65=100)

EXERCISE 6.1

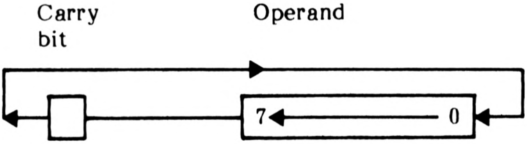
Write a program to multiply 10x9. Then print the result directly on the screen.

A possible answer is given in the solutions chapter.

Program 6.1 can only multiply together two numbers which result in a sum less than 255. The main reason for this limiting factor is that as the invisible partial product is shifted left it will eventually shift completely out of the register. A smiliar problem occured when we were dealing with 16 bit addition and subtraction. The problem was overcome by reloading the high order register with the carry bit when overflow occured. What is now required is an instruction which after a shift instruction on the LSB will shift the MSB including any generated carry. This is accomplished with the following instruction:

RL s	Rotate Left operand s including the carry bit

Represented diagramatically:



The carry flag is loaded into bit 0, bits 1-8 are shifted left one position and the carry flag loaded with bit 7.

To shift a 16 bit register two instructions are thus required. The LSB is shifted normally using the SLA instruction then the MSB is shifted using the RL. For example suppose we wanted to shift left the contents of DE. The following instructions would accomplish this.

```
SLA E
RL D
```

Now to incorporate this new instruction into a program.

Problem Print the result of the following sum on the screen: 7 x 10 = ?

PROGRAM 6.2

```
        LD C,7           C = Multiplicand
        LD E,10          E = Multiplier
        LD D,0           Zero register D
        LD B,8           B = Number of bits
        LD HL,0          Zero HL, it will be used to keep the
                         running total
NXTB:   SRL  C           Shift multiplicand right
        JR NC,NOADD:     If carry=0 jump to NOADD
        ADD HL,DE        Add IPP to running total
NOADD:  SLA E            Shift LSB left
        RL D             Reload carry and shift D left
        DEC B            Decrement counter
        JR NZ,NXTB:      Jump back if any more bits left
        LD A,L           Load A with answer
        CALL &BB5A       Print answer on screen ('F').
        RET
```

Note that although we used a 16 bit register for the result due to the careful selection of the multiplier and multiplicand only the LSB contains a value, allowing us to print it on the screen.

EXERCISE 6.2

Using Program 6.2 calculate the answer to the sum 146x124. Note that this results in a 16 bit answer, thus both H and L will have to be printed on the screen.

A possible answer is given in the solutions chapter.

Program 6.2 used register B as a counter. Everytime the multiplier was shifted left, B was decremented then tested. This operation requires two instructions DEC and JR NZ. These two instructions, can however be replaced by just one, making the program more efficient and elegant.

DJNZ e	Decrement B and if this results in a Non-Zero answer Jump to memory address e.

> EXERCISE 6.3
>
> Replace the DEC B and JR NZ,NXTB with a DJNZ NXTB instruction in Program 6.2, or your answer to Exercise 6.2
>
> An answer is given in the solutions chapter.

Binary division is also possible using a very similar method to that used for multiplication.

Binary Division

Consider the following division:

$$10\overline{)785}$$

Firstly we try to divide 7 by 10; as 10 does not go into 7 the next step is to bring down the 8 and try to divide 78 by 10. This divides seven times with a remainder of 8.

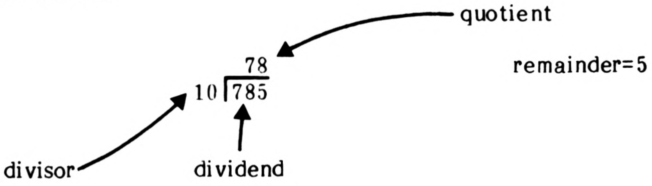
$$\begin{array}{r}7\\10\overline{)785}\end{array}$$

As 10 'doesn't go into' 8, the next step is to bring down the 5 and try to divide 85 by 10. This divides 8 times remainder 5.

$$\begin{array}{r}78\\10\overline{)785}\end{array}\quad \text{remainder } 5$$

Thus the result is 78 remainder 5. This form of division is termed 'integer division' as fractions are not considered.

As with multiplication the various numbers in a division sum have names; for example 'remainder' is already familar. The others are as follows:



Now to consider a 16 bit by 8 bit division.

Problem Print the result of the following division on the screen in a recognisable format.

2765 75 = ?

Binary division is carried out as follows. If without borrowing, the 8 bit divisor can be subtracted from the MSB of the 16 bit dividend, then the relevant bit in the 8 bit answer is set. This process is repeated eight times resulting in an 8 bit quotient and an 8 bit remainder.

PROGRAM 6.3

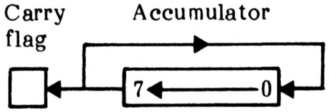
```
        LD HL,2765              HL = Dividend
        LD C,75                 C = Divisor
        LD B,8                  B = Count
NXT:    ADD HL,HL               Shift dividend left
        LD A,H                  A = MSB of dividend
        SUB C                   Subtract divisor
        JR C,NXTB:              If carry jump to NXT
        LD H,A                  Reload dividends MSB
        INC L                   Increment answer
NXTB:   DEC B                   Decrement counter
        JR NZ,NXT:              If non-zero jump to NXT
        LD A,L                  A = answer (quotient)
        CALL &BB5A              Print answer
        LD A,H                  A = Remainder
        CALL &BB5A              Print remainder
        RET
```

Run the program. It will print '$A' on the screen, where the ASCII code for '$' corresponds to the answer and 'A' the remainder. Check that these are correct. One point to note as the Z80 doesn't contain a 16 bit left shift instruction, ADD HL,HL was used. In binary, adding a number to itself is the same as shifting it left one bit position, or multiplying by two. If you compare this with the base 10 case, say, shifting left 19 will give 190 - the same as multipling by 10. What do you think will be the effect of shifting a binary number right one bit position? (Answer: it's the same as dividing the number by 2).

So far only a few of the Z80's shift instructions have been used in general purpose multiplication and division routines. It is sometimes easier and quicker to use a small group of shift or rotate instructions to perform a a specific arithmetic operation. By using a rotate as opposed to a shift instruction the register contents are altered, but no information is lost - i.e. no bits dissapear off the end without trace. Here is an instruction which rotates the contents of the accumulator left:

RLCA	Rotate the contents of the Accumulator Left and load a copy of bit 7 into the Carry flag.

Representing RLCA diagramatically:



Notice that bit 7 is not lost but inserted into bit 0. An instruction to rotate the contents of the accumulator right also exists:

RRCA	Rotate the content of the Accumulator Right and load a copy of bit 0 into the Carry flag.

Let's now use these two instructions.

Problem Multiply 10 by 16, print the result on the screen, divide the result by 2 and print the new result on the screen.

PROGRAM 6.5

```
LD A,10      A=10
RLCA         A=20
RLCA         A=40
RLCA         A=80
RLCA         A=160
CALL &BB5A        Print A ('^')
RRCA         A=80
CALL &BB5A        Print A('P')
RET
```

EXERCISE 6.4

Write a program to calculate and print the results of the following sums in the screen using the rotate and add instructions.

1. 5 x 32 = (160)
2. 254 - 2 = (127)

Possible answers are given in the solutions chapter.

The Z80 supports more rotate and shift instructions than have been used, these are detailed in the appendices under 'The Z80 instruction set'.

Now let's have a look at a set of instructions that allow individual bits within a byte to be set, reset or tested.

The Bit Set, Reset and Test Group

Consider the following instruction:

BIT b,r	Tests the bit at position b in the operand r.

This instruction is used whenever a a particular bit in a byte needs to be tested e.g: in input/output operations.

To illustrate the use of this command, a register will be zeroed. increment until bit 7 is set, the resultant register contents will then be printed on the screen.

PROGRAM 6.6

```
        LD A,0          A=0
NXT:    INC A           Increment A
        BIT 7,A         Test bit 7 of accumulator
        JR Z,NXT:       Is bit 7 set?
        CALL &BB5A      Print contents of A on the
                        screen
        RET
```

This program loads A with zero and repetedly increments it until, eventually, bit 7 is set.Run it; no great effect is noticed as bit 7 being set corresponds to 128 in decimal, the ASCII for a space. Try inserting either a INC A or DEC A instruction just before the CALL &BB5A instruction to print something visible on the screen.

Two further bit instructions exist: one to set and the other to reset an individiual bit within a byte.

SET b,r	SETs the bit at position b in the operand r.

RES b,r	RESets the bit at position b in the operand r.

To illustrate these instructions, consider the ASCII code for 'C', in binary 01000011. If bit 0 is reset, the effect is the same as subtracting one, resulting in the ASCII representation of 'B'.

PROGRAM 6.7

```
        LD A,67         A=ASCII for 'C'
        CALL &BB5A      Print 'C'
        RES 0,A         Reset bit 0
        CALL &BB5A      Print 'B'
        SET 0,A         Set bit 0
        CALL &BB5A      Print 'C'
        RET
```

EXERCISE 6.5

Write a program that loads the accumulator with 255, prints the contents on the screen, resets bit 4, prints the result on the screen, sets bit 4 resets bit 3 and then prints the final result on the screen.

A possible answer is given in the solutions chapter.

Well that concludes another chapter, by now you should have a fair idea as to how to write a fairly complex program. Now to summarise:

Summary

The following ideas and instruction groups should now be reasonably familiar:

1. Binary multiplication and division
2. The shift and rotate group.
3. The bit set, reset and test Group.

CHAPTER 7

The Stack

The stack is a block of memory located from &C000 downwards. It is used for the rapid transfer of data, and is filled downwards from &C000, the next vacant location being recorded by a register called the 'Stack Pointer', or SP. The usual analogy is with a stack of plates, only the top one being accessible as this was the last one put there. However, the stack is filled DOWNWARDS, i.e. from &C000 towards zero, so plates are put in and retrieved from the bottom, antipodean fashion! This mode of filling and emptying the stack is known as 'last in, first out' or LIFO, so the stack is a LIFO store.

One function of the stack is to record addresses during subroutine calls, which the Z80 does automatically. When the Z80 sees an instruction such as CALL &BBBA, it must first of all record where the next instruction in the calling routine (main program) is, so that it can find it again after the subroutine has been executed. It then places the 'BBBA' into the PC (program counter).

Suppose the Z80 is about to execute the following instruction:

```
        CALL &BBBA
```

The Z80 does the following things:

1. Increment PC
2. Fetch command byte ('CALL')
3. Increment PC
4. Fetch first byte of operand (&BA)
5. Increment PC
6. Fetch second byte of operand (&BB)
7. Store MSB of PC on stack
8. SP = SP-1
9. Store LSB of PC on stack
10. SP = SP-1
11. Put &BBBA into the PC
12. Execute subroutine, up to RET
13. Increment PC
14. Fetch command byte ('RET')
15. Load LSB of PC from stack
16. SP = SP+1
17. Load MSB of PC from stack
18. SP = SP+1
19. Increment PC
20. Continue with main program

In this example, the subroutine could well have met further subroutines or 'nested' subroutines, and each time a CALL was executed the return address would have been piled onto the stack. Then as the program RETurned successfully through these subroutines, the return addresses would have been stripped off successively to steer the Z80 back to the original point of departure. The area of memory used for the stack is more than 256 bytes long, so there should be plenty of room.

Fortunately, the operation of the stack in recording addresses when executing subroutines is automatic and so the programmer can allow the Z80 to do the job. However, when using built-in subroutines, the stack does not automatically store register contents but must be programmed to do so. The instructions for doing this are:

Push and Pop

PUSH qq	PUSH register pair qq onto the stack

Where qq is any of AF, BC, DE and HL. PUSH also comes in a form suitable for storing IX and IY:

PUSH IX	PUSH the IX register onto the stack

PUSH IY	PUSH the IY register onto the stack

These instructions copy the contents of the specified registers onto the stack. Having dealt with the subroutine or whatever, to restore the former contents of the registers, the instruction POP is used:

POP qq	Retrieve register contents from the stack

Again, qq is any of AF, BC, DE or HL.

POP IX	Retrieve IX from the stack

POP IY	Retrieve IY from the stack

Whether you are using PUSH or POP, there is no need for you to update the stack pointer - this is taken care of automatically.

When storing register contents on the stack, it is important to remember that the stack is a LIFO structure: you will normally retrieve the registers in the reverse order to that in which you stored them. Also, when writing a program incorporating PUSH and POP, every PUSH should be matched by a POP, otherwise the computer could get mixed up: it might, for instance, think that a number in the stack is a RETurn address, when in fact it is the unPOPed BC register pair. Obviously, this could lead to a system crash. Program 7.1 is an example of using PUSH and POP correctly, to preserve A and HL. It uses two of the computer's built in routines; one is the familiar &BB5A output text routine. The other is at location &BB75 and sets the text cursor position (see Appendix 4 for details). To use it, H must contain the column position and L the row position required. Note that &BB75 corrupts the registers, which is why it is a good idea to store them on the stack!

PROGRAM 7.1

```
        LD HL,&0604       Load H with column 4 and L with row 6
        LD A,&4D          Load A with ASCII 'M'
        PUSH HL           Copy HL onto stack
        PUSH AF           Copy A on stack
        CALL &BB75        Set cursor using HL
        POP AF            Recall A
        CALL &BB5A        Write 'M' at cursor using A
        LD BC,&0101       Put cursor offset in BC
        ADD HL,BC         Add offset to HL
        PUSH AF           Save AF on the stack
        CALL &BB75        Update cursor position (HL)
        POP AF            Recall A
        CALL &BB5A        Write 'M' at new cursor position
        RET               Back to BASIC
```

This should produce two M's, in consecutive diagonal positions. There are a couple of points worth noting in Program 7.1. Firstly, although things are PUSHed onto the stack, the information remains in the relevant registers until they are changed as PUSH only copies registers onto the stack; in the above program, the register contents are changed unpredictably ('corrupted') by the use of the built-in routines. Nevertheless it was still correct to PUSH AF and then CALL a routine which uses the contents of the A register, as this hadn't been changed up until then. Another point to note is that even though only A had to be preserved, AF had to be PUSHed onto the stack, as PUSH only works with register pairs. Similarly for POP.

EXERCISE 7.1

Write a short program to:

a) set the graphics cursor to (0,0)
b) put 100 in DE and 200 in HL
c) save DE and HL on the stack
d) draw a line to (100,200)
e) reset the graphics cursor to (0,0)
f) retrieve DE and HL, but putting the old contents of DE into HL and vice-versa.
g) draw a line to (200,100)

Hint: see Appendix 4; &BBC0 to set the cursor and &BBF6 to draw the lines. A possible answer is given in the solutions chapter.

If you have done the above exercise, you may have noticed in the solution an interesting way of swapping the contents of DE and HL:

```
        PUSH DE
        PUSH HL
            .
            .
            .
        POP DE
        POP HL
```

The registers are POPed in the 'wrong' order.

There is another way of swapping the contents of the register pairs DE and HL:

```
        EX DE,HL            EXchange contents of DE with HL
```

This instruction only works as written above; i.e. EX HL,DE or EX BC,HL etc. will not work.

The Stack Pointer (SP)

To keep track of which items to POP next, or where to place the next item to be PUSHed onto the stack, the Z80 uses a special register called the Stack Pointer, or SP. This normally points to the last location in the stack - i.e. to the last byte within the stack.

There are several instructions that allow the programmer to access the stack pointer, and change it. Using these, it is possible to set up a 'user stack'. After adjusting SP to point to some new memory location, the Z80 will use the new location as the start of a new stack, completely ignoring the old one. Thus, if a new stack is set up within a subroutine, it is important to preserve any RETurn addresses by PUSHing them onto the new stack.

There are three LD instructions which can be used to alter SP:

LD SP,HL	LoaD SP with the contents of HL

LD SP,IX	LoaD SP with the contents of IX

LD SP,IY	LoaD SP with the contents of IY

To find out the current value of SP, the following instruction can be used:

LD (nn),dd	LoaD memory location nn with the contents of register pair dd

dd can be any of BC, DE, HL or in this case, SP.

Just to prove that setting up your own stack will work, try the following program:

PROGRAM 7.2

```
LD A,43              Put '+' in A
PUSH AF              Store A in current stack
CALL &BB5A           Put '+' on screen
LD (&714A),SP        Remember current value of SP
LD HL,&7148          Get ready to:
LD SP,HL             Set up new stack at &7148
LD A,61              Put '=' in A
PUSH AF              Store A in new stack
CALL &BB5A           Put '=' on screen
LD HL,(&714A)        Find original location of SP
LD SP,HL             Go back to original stack
POP AF               Retrieve '+'
CALL &BB5A           Put '+' on screen
RET
```

Program 7.2 will put '+=+' on the screen. If the stack pointer had not been moved to a new location, then the POP AF would have retrieved an '=' instead, and the screen would have displayed '+=='. Just to prove that there is a new stack at &7148, try the following short exercise!

EXERCISE 7.2

Extend the above program so that it retrieves the '=' and displays it and then puts SP back where it came from again to retrieve and display another '+'. The screen should then display '+=+=+'.

Hint: Remember that SP should point to the last location within the stack, so moving it to &7148 will not work. Use &7146.

A solution is given in the solutions chapter.

To conclude this chapter, it is worth mentioning the other commands that can affect the stack. These allow the advanced (and careful!) programmer to mess about with the contents of the stack without necessarily PUSHing or POPing repeatedly to get the SP to point to the desired element in the stack.

First of all, there are three more EXchange instructions:

EX (SP),HL	EXchange contents of HL with the top of the stack.
EX (SP),IX	EXchange contents of IX with the top of the stack.
EX (SP),IY	EXchange contents of IY with the top of the stack.

The other instructions which can affect SP are:

INC ss	INCrement register pair ss

DEC ss	DECrement register pair ss

ss is any of BC, DE, HL or SP.

ADD IX,pp	ADD contents of register pair pp to IX

Where pp is any of BC, DE, IX or SP.

ADD IY,rr	ADD contents of register pair rr to IY

Where rr is any of BC, DE, IY or SP.

Summary

Having read this chapter, you should know about the following:

LIFO
SP
PUSH
POP
EX (SP),HL
EX DE,HL
Which part of the stack SP points to
User stacks
LD (nn),dd

CHAPTER 8

Block moves and compares

The Z80 has several instructions designed to allow whole sections of memory to be handled easily, and without the program having to specifically include the addresses of each individual memory location. These instructions can be divided into two classes. The block moves allow areas of memory to be copied from one place to another. The block compares are used to search a region of memory for some specific data item.

Block moves

The first of the block move instructions to be looked at in this chapter is LDI:

LDI   LoaD memory from memory and Increment data pointers

To use this instruction, HL should contain the address of the memory location that the data is to come from, and DE should contain the address of the memory location that the data is to be copied into. After executing this instruction, the Z80 increments both DE and HL. Also it decrements BC; this makes LDI particularly useful in loops - BC can be used as a loop counter. When BC is decremented to 1, the parity/overflow flag is reset to 0 - at other values of BC, LDI sets this flag to 1, so that it can be tested to end a loop upon being set, using the PO condition. This is described in more detail below; but first the program.

As an example of using LDI, Program 8.1 copies some memory onto the screen. It copies some data from part of the stack, which occupies memory locations &B100 to &BFFF, to the screen, which corresponds to memory locations &C000 to &FFFF inclusive.

PROGRAM 8.1

```
        LD HL,&B992          Load HL with start address of data
        LD DE,&C000          Load DE with destination
        LD BC,&A1            Load BC with amount of data + 1
LOOP:   LDI                  Copy a byte of data
        JP PO,FINISH:        Exit loop if BC = 1
        JP LOOP:             Else continue loop
FINISH: RET
```

This should produce a horizontal multicoloured line across the screen, with a short multicoloured line at opposite ends on either side.

Parity

As stated earlier, the parity/overflow flag is set when it is decremented to 1 rather than zero, so it is necessary for accurate memory copying to load BC with the number of bytes to be copied plus 1. To test the parity/overflow flag, the conditions are PO - 'parity odd' and PE - 'parity even'.

'Parity' refers to the number of bits set in the byte or flag being tested: even parity means that an even number of bits are set and odd parity means that an odd number of bits are set. Thus, when the parity flag is set, it contains an odd number (1) of set bits, and can be tested with the PO condition, as in Program 8.1 above:

```
        JP PO,FINISH:
```

Should you ever need to test the parity of a byte of data, this can be done by loading A with 255, and ANDing it with the byte in question. The parity flag will be set appropriately and can be tested with a statement such as the conditional JP above. Parity tests are usually used in communications systems: one bit of each byte will be set aside for use as 'parity bit' which will be set or reset appropriately for each byte so that all bytes transmitted and received will have identical parity. If any interference garbles the data, the chances are that the parity of some bytes will have changed, and this can be detected.

EXERCISE 8.1

Write a program similar to Program 8.1, to copy &3FFF bytes starting at &B100, and to display them on the screen starting at &C000. Warning: be careful! A mistake might cause the computer to crash, requiring you to reload the assembler. A possible answer is given in the solutions chapter.

The solution to Exercise 8.1 fills the screen with rubbish alright, but it might be nice to see this done more slowly. Try Program 8.2:

PROGRAM 8.2

```
        LD HL,&B100
        LD DE,&C000
        LD BC,&4000
LOOP:   LDI
        JP PO,FINISH:
        LD A,&FF
DELAY:  DEC A                  Time 'wasting' instructions
        JP NZ,DELAY:
        JP LOOP:
FINISH: RET
```

When this is run, the screen fills up line-by-line, the lines at first separated, and then the gaps being filled in.

The reason the screen fills up like that rather than, say, line-by-line with no initial separation between the lines, is because of the way the computer controls the screen. Consecutive bytes of screen-memory (i.e. from &C000 to &FFFF) do not control (or in the jargon are not 'mapped on to') consecutive screen locations on the monitor. Details of how the screen memory is organised are to do with the 'operating system' of the computer rather than machine code or assembly language, and the interested reader is referred elsewhere, e.g. to Amsoft's "The Complete CPC 464 Operating System Firmware Specification", SOFT 158.

Program 8.2 illustrates, indirectly, the main purpose of the LDI block move instruction: you can put extra instructions in between successive operations of the LDI instruction. Program 8.1 illustrates LDI's main drawback: if you don't require any extra instructions between each operation of LDI, you still have to have a jump back to the LDI to get it to repeat. This wastes both memory space and time. The Z80 has a fix for this:

LDIR	LoaD memory from memory, Increment data pointers and Repeat

This instruction acts in exactly the same way as LDI except that it automatically repeats itself until BC has been decremented to 0. Also, it sets the parity/overflow flag to 0 regardless of the current value of BC. This doesn't matter of course, as with automatic repetition there is no need to do any tests to find out whether it has finished or not anyway. Program 8.3 is a rewritten version of Program 8.1, replacing LDI and JP with LDIR.

PROGRAM 8.3

```
        LD HL,&B992
        LD DE,&C000
        LD BC,&A0
        LDIR
        RET
```

Note that in this case, it is not necessary to load BC with the number of bytes to be transferred plus one, as LDIR stops when BC is zero, whereas in Program 8.1 and 8.2 the loop was ended when BC=1. BC need only be loaded with the actual number of bytes to be transferred.

There are two more block move instructions:

LDD	LoaD memory from memory, Decrement data pointers

LDDR	LoaD memory from memory, Decrement data pointers and Repeat

These are the same as LDI and LDIR respectively except that DE and HL are decremented rather than incremented. For example, try Program 8.4, which will fill the screen from the bottom up:

PROGRAM 8.4

```
         LD HL,&BFFF
         LD DE,&FFFF
         LD BC,&4000
LOOP:    LDD
         JP PO,FINISH:
         LD A,&FF
DELAY:   DEC A
         JP NZ,DELAY:
         JP LOOP:
FINISH:  RET
```

Compares

It is often handy in a program to be able to compare two values, and then to make a decision according to the result - as in the BASIC IF...THEN construction. The assembly-language equivalent is

```
CP s    ComPare s with A by subtracting s from A, leaving
        A unchanged
```

In the above, 's' is any one of the following: A, B, C, D, E, H, L, (IX+d), (IY+d), (HL). After subtracting s from A, the result is not kept (i.e. A is not corrupted) although the consequences of the subtraction are recorded by the carry, zero, overflow, sign and half-carry flags. The overflow, sign and half-carry flags are discussed elsewhere in the book. They are used when you wish to compare two's complement numbers.

Since the comparison subtracts s from A, the carry flag will be set if s>A, and reset otherwise. If s=A then the zero flag will be set, otherwise reset. Program 8.5 demonstrates the use of CP.

PROGRAM 8.5

```
        LD A,&21            Put a number into A
        LD B,&40            Put a number into B
        CP B                Compare B with A
        JP C,BIGGER:        If carry set, B>A
        JP Z,EQUAL:         If zero set, B=A
        LD A,&73            Else B<A;Put ASCII 's' in A
        CALL &BB5A          Put on screen
        LD A,&3C            ASCII '<'
        CALL &BB5A          Put on screen
        LD A,&41            ASCII 'A'
        CALL &BB5A          Put on screen
        RET                 Exit program
BIGGER: LD A,&73            ASCII 's'
        CALL &BB5A          Put on screen
        LD A,&3E            ASCII '>'
        CALL &BB5A          Put on screen
        LD A,&41            ASCII 'A'
        CALL &BB5A          Put on screen
        RET                 Exit program

EQUAL:  LD A,&73            ASCII 's'
        CALL &BB5A          Put on screen
        LD A,&3D            ASCII '='
        CALL &BB5A          Put on screen
        LD A,&41            ASCII 'A'
        CALL &BB5A          Put on screen
        RET
```

Try this program out again with values &40 in A and &21 in B, and with &21 in both A and B to check that it works as you might expect it to.

Not only does the Z80 have the above compare instruction, it has some block compare instructions as well. The first of these is CPI

CPI	ComPare A with memory, Increment data pointer

As with LDI, this instruction decrements BC, the 'Byte Counter'. Also, the parity flag is set when BC is decremented to 1, otherwise it is reset. This instruction does not use the carry flag, however, but it does affect the zero flag. This means that CPI can only be used to test for equality, not greater than or less than. Program 8.6 searches memory for a bracket '(' and puts one '(' on the screen for each one it finds in memory.

PROGRAM 8.6

```
         LD A,&28               Put '(' in A
         LD HL,&0               Start at beginning of memory
         LD BC,&0               Search 64k (BC=bytes+1,0-1=FFFF)
LOOP:    CPI
         JP PO,FINISH:          Finish if parity flag set
         CALL Z,FOUND:          Call 'found' if zero set
         JP LOOP:               Loop again
FINISH:  CALL Z,FOUND:          Check zero flag before finishing
         RET                    Back to the assembler
FOUND:   CALL &BB5A             Put '(' on screen
         RET                    RETurn from subroutine
```

One thing to note in this program is that the 'FOUND' subroutine needs to be called from within 'FINISH' because if Z was tested before PO, and a '(' was found, calling BB5A would corrupt the parity flag. If PO is tested first this problem does not arise.

Another block compare instruction is:

CPIR	ComPare A with memory, Increment data pointer and Repeat

This instruction is the same as CPI, except that it automatically continues either until a match is found, or until BC=0. Program 8.6 can be rewritten simply by changing CPIR for CPI. The new program will run faster with CPIR because the conditions do not need to be tested after every comparison.

The last two block comparisons are the same as CPI and CPIR except the data pointer, HL, is decremented rather than incremented:

CPD	ComPare A with memory, Decrement data pointer

CPDR	ComPare A with memory, Decrement data pointer and Repeat

Unlike LDD and LDDR, these instructions do still effect the parity flag.

In case you ever need to know, a table of the effects of comparisons on the testable flags C, S and V is to be found in the appendices. This should be referred to if you ever need to compare two's complement numbers (i.e. numbers greater than 128)

Summary

Having read this chapter, you should now be aware of:

LDI
PE
PO
parity
a delay loop
LDIR
LDD
LDDR
CP s
CPI
CPIR
CPD
CPDR

CHAPTER 9

Special Operations and Interrupts

This chapter is about some of the least useful features of Z80 assembly language, at least from a purely software point of view. Many of the special operations and features described in this chapter are hardly used at all by most programmers. However, it is worth being aware of them. Interrupts, discussed below, are very useful, but detailed discussion of them is beyond the scope of this book as they are really a matter for a hardware manual.

Interrupts

When doing a job that has to be done, no-one likes interruptions until the job is finished. The Z80 is like that too. Whilst executing a piece of program, it has all its registers under control, and all its flags set appropriately.

An interrupt, however, is a subroutine that demands execution when IT is ready, and not the Z80. In other words, an interrupt comes from outside the Z80's direct field of control - either from some external device, or from the keyboard. Thus, flags have to be stored by the Z80 - usually on the stack. Then the Z80 will 'service' the interrupt - i.e. do whatever it is required to do by the interrupt, and then it must restore all the registers and flags and continue with the original program.

Interrupt handling should be allowed for in any program which expects interrupts, particularly if the program does certain jobs which must not be interrupted. If, for intance, another device is sending a stream of data into memory, then a 'hand-shaking' procedure is carried out between the two machines. Quite simply, this is an exchange of messages like: "I am ready to send data, are you ready to receive it?" "Yes." "Here's the data ..... end of data." "Thanks!"

If such an exchange is interrupted, then the data is likely to become garbled, and hence worthless. During such periods when no interrupting is allowable, the program can block most interrupts - not all - to allow a particular process to be completed. Interrupts which can be blocked are called 'maskable' interrupts, and interrupts which cannot be blocked are called 'non-maskable' interrupts, or NMI's.

The instruction to block all maskable interrupts is DI:

DI	DIsable maskable interrupts

Having completed the, preferably short, section of the program which cannot be interrupted, interrupts can be re-enabled using EI

EI	Enable maskable Interrupts

When the Z80 services an interrupt, it does this by means of a machine-code subroutine. In common with other subroutines, they must be terminated with a return statement. For non-maskable interrupts the instruction is:

RETN	RETurn from Non-maskable interrupt

For maskable interrupts it is:

RETI	RETurn from Interrupt

If you think about the above, you may realise that interrupts could be interrupted. There are priorities, however: for instance, a maskable interrupt will not normally be allowed to interrupt a non-maskable interrupt - it will usually just have to wait.

The highest priority interrupt of all is a 'bus request' or BUSRQ. With other maskable and with non-maskable interrupts, the Z80 will at least finish executing the current instruction before dealing with the interrupt. With a BUSRQ, however, the Z80 reacts on the next 'tick' of its internal clock - i.e. on its next cycle - whether it has finished the current instruction or not.

How the Z80 reacts to maskable interrupts depends on the current 'interrupt mode'. In all cases, it first of all disables further (maskable) interrupts, and saves the PC on the stack. It is necessary always to re-enable interrupts (EI) before returning from your servicing routines.

Interrupt Modes For Maskable Interrupts

The default mode is mode 0. It can be set within a program using the instruction:

```
IM 0          Set Interrupt Mode 0
```

Upon receiving and accepting a maskable interrupt in mode 0, the Z80 will expect the external device to give it an instruction on its next clock-cycle via its general input line, the 'data bus'. This instruction is usually a CALL to an interrupt servicing routine that the programmer has placed somewhere in memory. It is also often a ReSTart instruction:

```
RST  n        ReSTart at address n
```

The address n is a single byte address and can only be one of the following: &0, &8, &10, &18, &20, &28, &30 and &38. Because RST is a single byte instruction, it is often used when a fast response to the interrupt is required. The problem with it is that there is only really room at those memory locations for a jump to some other address - especially if there are several interrupt servicing routines, each required by differing devices using different RST addresses.

Both CALL and RST automatically cause the PC to be saved onto the stack. The Z80 disables further maskable interrupts and begins the servicing routine. If the programmer requires the registers and flags to be preserved, then the interrupt servicing routine should begin with something like this:

PROGRAM 9.1

```
        PUSH AF
        PUSH BC
        PUSH DE
        PUSH HL
        PUSH IX
        PUSH IY
```

It should end with something like:

PROGRAM 9.2

```
        POP IY
        POP IX
        POP HL
        POP DE
        POP BC
        POP AF
        EI
        RETI
```

Notice the 'AF'. This is the accumulator and the flags.

The next interrupt mode is mode 1:

```
    IM 1          set Interrupt Mode 1
```

This mode is similar to interrupt mode 0 except that the Z80 automatically executes a RST &38 upon accepting a maskable interrupt. Also, the Z80 will ignore the contents of its data bus in the clock cycle following the interrupt. The final mode is mode 2:

```
    IM 2          set Interrupt Mode 2
```

In this mode, after finishing the current instruction, saving the PC and disabling further maskable interupts, the Z80 will jump to any specified even-numbered memory location - i.e. one in which the least significant bit is a 0. The most significant eight bits of the address should be set in advance in a special register calaled the 'interrupt vector' register, or 'I' register. The least significant eight bits of the address are supplied by the interrupting device. This allows the Z80 to jump to any one of up to 128 addresses held in a table in memory starting at the high-byte address held in the I register.

The Alternate Registers

You are by now well aware that the Z80 has many different registers, in particular, A, B, C, D, E, H, L and the flags. Well, the Z80 has a second set of these particular registers, known as he 'alternate register set', A', B', C', D', E', H', L' and the alternate flags F'.

Although they can be used by the programmer, it is not advisable on the Amstrad. Using the built-in routines will corrupt them, as will any interrupt. The instructions to use them are as follows:

EXX	EXchange register pairs

This instruction swaps the contents of the register pairs BC, DE and HL with their alternates. Note that this means that any following instruction using say DE will now use the new contents of DE, that used to be in DE'.

If the alternate registers B', or C' are used, they must be restored prior to swapping back to the original register set and enabling interrupts. Thus a program to use the alternate registers might look like this:

PROGRAM 9.3

```
DI       Disable interrupts
EXX      Exchange register pairs
PUSH BC  Save new BC
     .
     .
     .
POP BC   Restore BC
EXX      Back to original registers
EI       Enable interrupts
```

Any such program should be kept short, as various automatic operating-system interrupts such as timers, keyboard scanning etc. need to use the alternate registers, and anything other than a short routine could cause problems. This is true whenever interrupts are disabled. (For further details see Soft 158).

Another instruction allows A' and F' to be used:

EX AF,AF'	EXchange contents of AF and AF'

F' must be restored prior to 'unexchanging'. Again, interrupts must be disabled whilst using these registers.

Note that exactly the same instruction is used to unexchange - i.e. EX AF,AF', not EX AF',AF.

More EXchange Instructions

There are two other forms of EX which are worth mentioning:

EX DE,HL	EXchange the contents of DE with the contents of HL

and

EX (SP),HL	EXchange the top of the stack with the contents of HL.

EX (SP),IX	EXchange the top of the stack with the contents of IX

EX (SP),IY	EXchange the top of the stack with the contents of IY

The top byte on the stack goes into L and vice-versa, and the next byte on the stack goes into H (and vice-versa). Similarly with IX and IY.

Halt

HALT	Halt program execution

This instruction stops the Z80 until an interrupt is received. The only thing it does is keep 'refreshing' the memory. As you know, if you switch the computer off, it forgets any program or data which is held in its memory. This is because the memory consists of 'dynamic RAM' or 'Random Access Memory'. If this memory isn't continually updated or 'refreshed' - i.e. if the internal voltages are not continually corrected, then these voltages die away and the data stored disappears with them. The Z80 uses a 'refresh register' to tell it which memory bank to refresh at any one moment.

The INs and OUTs of the Z80

The Z80 has a family of instructions designed to allow inputs and outputs of data from and to external devices. Again, these instructions are hardware orientated, and will only be discussed briefly here.

IN A,(n)	INput contents of port n to the accumulator

A 'port' is a connector to an outside device. Which number refers to which port is determined by hardware - i.e by the actual physical wiring and circuitry of the computer.

The port, as far as the Z80 is concerned, is an 8-bit data store. IN A,(n) does not affect any of the flags.

There is a form of the IN command which allows the contents of a port whose number is held in the C register to be placed in any of A,B,C,D,E,H or L:

IN r,(C)	INput contents of port addressed by C to register r

This instruction resets the add/subtract flag. The carry flag is not affected, but the other flags are altered according to the value INput.

Should they be necessary, the Z80 also has forms of the IN command which parallel LDI, LDIR etc. The first of these is:

INI	INput contents of port addressed by C, decrement B, Increment HL

With this instruction, B can be used as a loop counter if required. Notice, however, that only B is used by INI, rather than BC as in, say, LDI. C should contain the port number, and HL the memory location in which the data is to be placed. INI corrupts the sign, half-carry and parity/overflow flags, and sets the add/subtract flag. The carry flag is not affected. The zero flag is set when B=1.

INIR	INput contents of port addressed by C, decrement B, Increment HL, Repeat until B=0

This is the same as INI except that it automatically repeats until B is decremented to zero. The flags are affected in the same way as with INI except that the zero flag is always set.

IND	INput contents of port addressed by C, decrement B, Decrement HL

INDR	INput content of port addressed by C, decrement B, Decrement HL, Repeat until B=0

These are the same as INI and INIR respectively, except that HL is decremented instead of incremented.

OUT (n),A	OUTput contents of Accumulator to port n

OUT (C),r	OUTput contents of register r to the port addressed by C

The latter of these two differs from IN r,(C) in that it does not affect any of the flags.

OUTI	OUTput to port addressed by C the contents of memory addressed by HL, decrement B, Increment HL

OTIR	OUTput to port addressed by C the contents of memory addressed by HL, decrement B, Increment HL, Repeat until B=0

OUTD	OUTput to port addressed by C the contents of memory addressed by HL, decrement B, Decrement HL

OTDR	OuTput to port addressed by C the contents of memory addressed by HL, decrement B, Decrement HL, Repeat until B=0

These instructions affect the flags in the same way as the corresponding IN instructions.

One slightly useful instruction hasn't been mentioned yet:

NOP

NOP	No OPeration

This tells the Z80 to do nothing for one clock cycle. This can be useful for fine tuning the duration of delay loops. Also, if you are using absolute addresses rather than symbolic labels, it can be used to pad out your program to allow for addressing errors.

Summary

This chapter covered the following instructions:

DI
EI
RETN
RETI
IM0
IM1
IM2
RST n
EXX
OUT (n),A
OUT (C),r
OUTI
OTIR
OUTD
OTDR

EX AF,AF'
EX DE,HL
EX (SP),HL
EX (SP),IX
EX (SP),IY
HALT
IN A,(n)
IN r,(C)
INI
INIR
IND
INDR
NOP

CHAPTER 10

External commands and graphics extensions

This chapter explains how additional commands may be added to those already supported by BASIC. For example, a circle-drawing command will be added to the Amstrad's BASIC. These routines will be written entirely in assembly language and interfaced to the BASIC interpreter using 'external commands'. This chapter makes use of some of the more advanced features of the assembler. It is recommended that you read Appendix 6 first.

External command (RSX's)

All commands in BASIC, e.g. LIST, GOTO, RND etc. are known as 'internal commands'. When a BASIC program is run and the interpreter encounters an internal command it is searched for in ROM (Read Only Memory) whereas, if an 'external command' is encountered, the RAM (Random Access Memory) is searched as well. An external command or Resident System eXtension consists of a string of alphanumeric characters preceded by a vertical bar (Shift and @). The following are all valid external commands.

```
        |CIRCLE     |TRIANGLE     |BOX
```

Try typing in |BOX when in BASIC. The computer will respond with 'Unknown command'. This is because the BASIC interpreter cannot find the command BOX in ROM or RAM. It is quite simple to tell the computer that an external command exists; here's how:-

Step 1

Firstly the computer needs to be informed that some external commands are to be added. This is accomplished by creating a command table of new commands. This command table acts as a jump block to further tables, providing explicit information about the new commands i.e. syntax, location, etc. This information is passed to the operating system, by calling a routine in ROM, after loading the relevant registers with data as well as the address of a four-byte buffer required by the operating system.

Routine's use: Logs an external command onto the operating system (i.e. tells operating system that the command exists.

Calling address: &BCD1.

Entry conditions: BC= The starting memory location of the external command table.
HL= The starting memory location of a four-byte buffer.

This whole process will now be illustrated with an example. The external command table will start at the memory address represented by the label EXCOMT (EXternal COMmand Table). The easiest method of setting aside four bytes for a buffer is to use the 'DEFS' assembler directive. This buffer will be assigned the label BUFF:

Thus:-

```
        BUFF:  DEFS &04
```

Combining these various elements:

PROGRAM 10.1(don't enter this yet)

```
BUFF:   DEFS &04          Set up buffer
        LD BC,EXCOMT:     BC= Start of table
        LD HL,BUFF:       HL= Start of buffer
        CALL &BCD1        Log-on table
        RET               Return to Basic.
```

Don't enter this program yet as although the computer knows the location of the external command table it wouldn't find any new command names when it looked there. Thus the new external command table needs to be filled with the new command names.

Step 2

Further details of the new command table must be held in an additional table. The address of this table is given by the label NENAM (NEw NAMe). Thus the first line of this section is:-

```
        EXCOMT:  DEFW NENAM:
```

The new command names are then added using the JP instructions as follows.

```
        JP BOX:
```

This would assign a jump address for the new command BOX. Assuming that only one external command is going to be added, the second section of the program appears as below.

```
        EXCOMT:  DEFW NENAM:
                 JP BOX:
```

All that is now required is to add the specific details of the new external command's name.

Step 3

The new command name is entered as a string of ASCII characters starting at the memory address pointed to by NENAM. The Amstrad's internal operating system requires that bit 7 be set in the byte representing the last ASCII character of the command name's string. This is most easily accomplished by adding &80 to the ASCII code of the last letter. Therefore &80 has to be added to the ASCII for "X" in the example. Thus this section of the program appears as below.

```
        NENAM:   DEFM    "BO"
                 DEFB    "X"+&80
                 DEFB    &0
```

The &0 is to signify the end of the table.

Combining these three sections together:

PROGRAM 10.2(don't assemble this yet)

```
         ORG 40000
BUFF:    DEFS &04
         LD BC,EXCOMT:
         LD HL,BUFF:
         CALL &BCD1
         RET
EXCOMT:  DEFW NENAM:
         JP BOX:
NENAM:   DEFM  "BO"
         DEFB  "X"+&80
         DEFB &0
```

Note that ORG has been set to 40000. By then setting MEMORY to 39999 hence preventing BASIC from using any memory location above 39999 the external command cannot be corrupted.

When :BOX is typed in, from BASIC, the computer will jump to the memory location addressed by the label BOX - to illustrate the new external command type in Program 10.3 immediately following Program 10.2.

PROGRAM 10.3

```
BOX:    LD A,66
        CALL PRINT:
        LD A,79
        CALL PRINT:
        LD A,88
        CALL PRINT:
        RET
PRINT:  EQU &BB5A
```

Note the use of the memory label PRINT.

Now assemble the whole program; when listed it should appear as below.

```
        ORG 40000
BUFF:   DEFS &04
        LD BC,EXCOMT:
        LD HL,BUFF:
        CALL &BCD1
        RET
EXCOMT: DEFW NENAM:
        JP BOX:
NENAM:  DEFM "BO"
        DEFB "X"+&80
        DEFB &0
BOX:    LD A,66
        CALL PRINT:
        LD A,79
        CALL PRINT:
        LD A,88
        CALL PRINT:
        RET
PRINT:  EQU &BB5A
```

Run the program with a 'CALL 40000' from BASIC. Although nothing visible has happened, the external command BOX has been 'logged on' to the Amstrad's operating system. Exit to BASIC and type in the following.

```
        |BOX
```

BOX appears upon the screen! Thus any assembly-language program addressed with the label BOX can be called with this external command. For example a box could be drawn on the screen, using the inbuilt graphics routines. To specify the size of the box, some parameters have to be passed to the assembly-language program. Consider the following diagram.

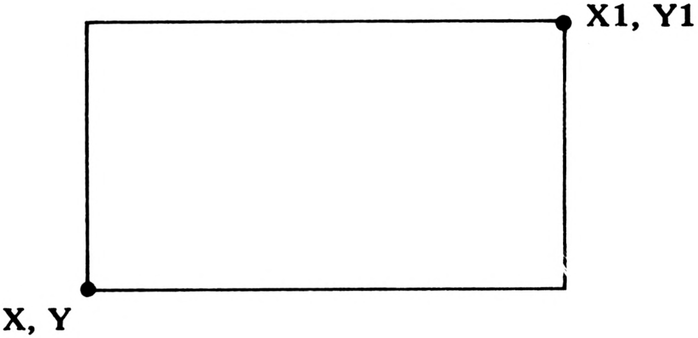


FIGURE 10.1

By defining two diagonally opposing corners, a unique box can be described. It would be useful if, as well as the size, the colour of the box could be changed. Thus five parameters need to be passed to the assembly-language routine. One possible solution would be to poke the data into a block of memory, creating a 'data block'. Although cumbersome, it would work. Fortunately the external command system creates this 'data block' for us. When called, the starting location of the data block is stored in the register pair IX, with the number of data items stored in the accumulator. Consider the following command which would draw a box 100 by 100 units on the screen at 200,100 in ink 2.

```
|BOX,200,100,300,200,2
```

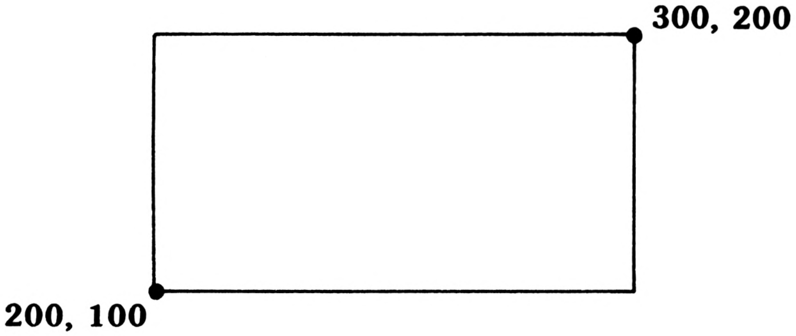


FIGURE 10.2

Before the box's parameters are stored in memory they are converted to hexadecimal.

Step 1

```
100 = &0064
200 = &00C8
300 = &012C
  2 = &0002
```

Step 2

The parameters are stored LSB first followed immediately by the MSB. The last parameter entered is pointed to by IX. Thus the parameters in the example are stored as follows:

Memory location	Contents	
	Hex	Decimal
IX+0	02	2
IX+1	00	
IX+2	00	200
IX+3	C8	
IX+4	01	300
IX+5	2C	
IX+6	00	100
IX+7	64	
IX+8	00	200
IX+9	C8	

Thus by using a suitable offset any one of these parameters may be accessed. Note that as each parameter is stored in two bytes, a parameter can be in the range 0 to 65536.

Now, from the coordinates passed by the external command, four coordinate pairs have to be derived describing the box.

Consider Figure 10.3:

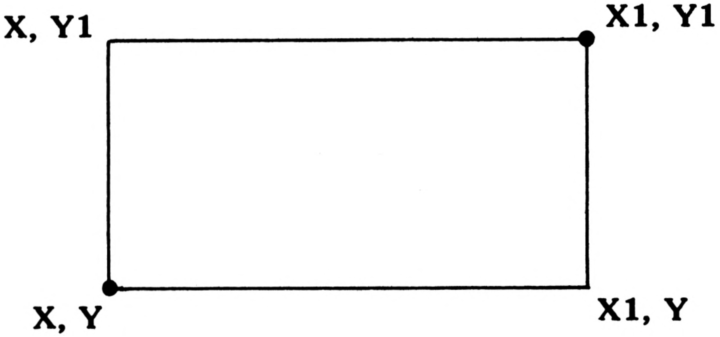


FIGURE 10.3

As the coordinates X,Y and X1,Y1 are passed by the external command the coordinates of the other two corners may be deduced and a box plotted. This is represented by the following flow chart:

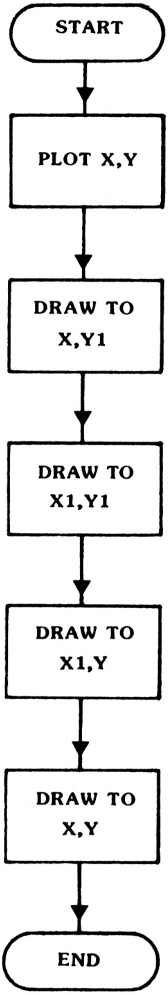


FIGURE 10.4

Expressing this flowchart in a program (note the use of the stack to store the coordinates):

PROGRAM 10.4

```
10      DRAW:    EQU  &BBF6
20      PLOT:    EQU  &BBEA
30      INK:     EQU  &BBDE
40      BOX:     LD   A,(IX+0)
50               CALL INK:
60               LD   D,(IX+8)
70               LD   D,(IX+9)
80               PUSH DE
90               LD   E,(IX+6)
100              LD   D,(IX+7)
110              PUSH DE
120              LD   E,(IX+4)
130              LD   D,(IX+5)
140              PUSH DE
150              LD   E,(IX+2)
160              LD   D,(IX+3)
170              PUSH DE
180              LD   E,(IX+8)
190              LD   D,(IX+9)
200              LD   E,(IX+6)
210              LD   H,(IX+7)
220              PUSH DE
230              CALL PLOT:
240              POP  DE
250              POP  HL
260              PUSH HL
270              CALL DRAW:
280              POP  HL
290              POP  DE
300              PUSH DE
310              CALL DRAW:
320              POP  DE
330              POP  HL
340              PUSH HL
350              CALL DRAW:
360              POP  HL
370              POP  DE
380              CALL DRAW:
390              RET
```

Using labels for memory addresses greatly increases the readability of the program. Now replace the subroutine starting with the label BOX in the previous program with Program 10.4. Assemble and run it. Now exit to BASIC and try Program 10.5, after logging on the command with a 'CALL 40000'.

Note: As the Assembler uses the BASIC lines above 64000 don't type NEW or you will have to reload the assembler again.

PROGRAM 10.5

```
10 MODE 0:CALL 40000
20 X=RND(1)*550
30 Y=RND(1)*350
40 X1=RND(1)*50
50 Y1=RND(1)*50
60 C=RND(1)*15
70 |BOX,X,Y,X+X1,Y+Y1,C
80 GOTO 20
```

Well that concludes the external commands section. The rest of this chapter will explain how additional graphics commands may be added.

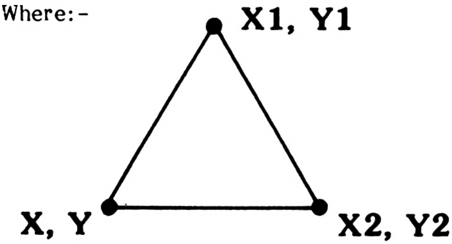
Some Additional Graphics Commands

1. BOX,X,Y,X1,Y1,C
   Draws a box on the screen using PEN C.

2. BOXF,X,Y,X1,Y1,C
   Where the arguments are exactly the same as for BOX. The only difference is that the box is filled in.

3. TRI,X,Y,X1,Y1,X2,Y2,C
   Draws a triangle on the screen.

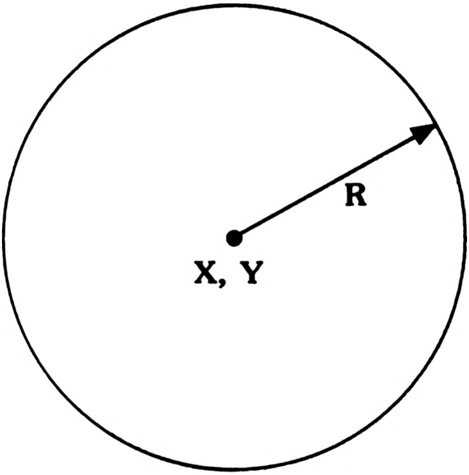
   Where:-



   Note Y must equal Y2

4. CIRCLE,X,Y,R,C

Where:-



The external commands reside in memory starting at memory location 40000. To ensure that they are not corrupted the BASIC memory pointer is moved down to 39999 with the following BASIC command

```
MEMORY 39999
```

Thus it is wise to ensure that the first two lines of any BASIC program using these external commands are as follows:-

```
1 MEMORY 39999
2 CALL 40000
    :
    :
```

The box-fill commands

This command draws a solid box upon the screen; it is drawn as a series of consecutive horizontal lines.

Consider Figure 10.5

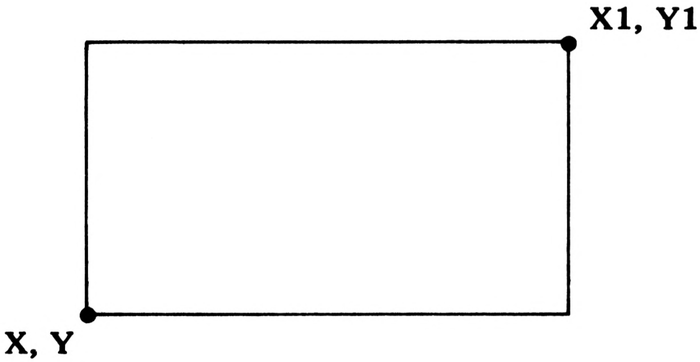


FIGURE 10.5

The algorithm for filling in this box is as follows:-

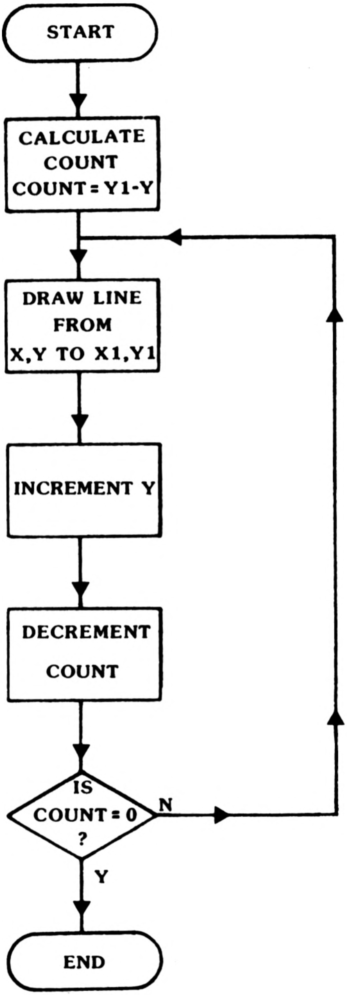


FIGURE 10.6

Expressing this algorithm as a program:

PROGRAM 10.6

```
10    COUNT:   DEFS 4
20    DRAW:    EQU  &BBF6
30    PLOT:    EQU  &BBEA
40    INK:     EQU  &BBDE
50    BOXF:    LD   A,(IX+0)
60             CALL INK:
70             LD   L,(IX+2)
80             LD   H,(IX+3)
90             LD   E,(IX+6)
100            LD   D,(IX+7)
110            AND  A
120            SBC  HL,DE
130            JP   C,END:
140            LD   (COUNT:),HL
150            LD   E,(IX+8)
160            LD   D,(IX+9)
170            LD   L,(IX+6)
180            LD   H,(IX+7)
190            PUSH HL
200            PUSH DE
210            CALL PLOT:
220            LD   E,(IX+4)
230            LD   D,(IX+5)
240            POP  IX
250            POP  IY
260   NXT:     PUSH DE
270            PUSH IY
280            POP  HL
290            CALL DRAW:
300            PUSH IX
310            POP  DE
320            POP  IX
330            INC  IY
340            LD   HL,(COUNT:)
350            PUSH DE
360            LD   DE,1
370            AND  A
380            SBC  HL,DE
390            LD   (COUNT:),HL
400            POP  DE
410            JR   NZ,NXT:
420   END:     RET
```

The assembler tape (side B) contains a file called GRAPHICS-EXT which contains all the additional graphics commands presented in this chapter. To use it load the assembler then load in the file GRAPHICS-EXT and assemble it. If you wish to use the additional commands within your BASIC programs, save a copy of the object code generated by using the file identifier '-b'. e.g. Save it under the name GRAPHICS-B. This file can then be loaded from BASIC with the command LOAD "GRAPHICS-B",40000. However, ensure that the BASIC memory pointer has been moved down first to 39999, using the following command MEMORY 39999. Then to log on the graphics commands type in CALL 40000 from BASIC. All the commands are now available. Study the additional commands program GRAPHICS-DEMO if you are not sure.

The triangle command

This command produces a solid triangle upon the screen. The method used is iterative and is chosen for its simplicity and not efficiency, in that a given procedure is repeated until a condition is met. Here the difference between the two X base-coordinates is used as a count. See Fig 10.7

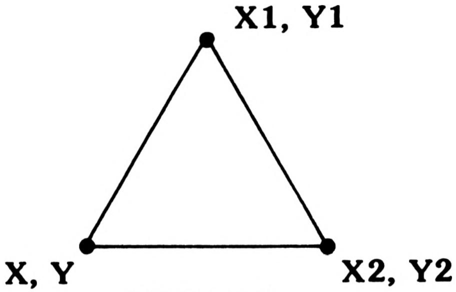


FIGURE 10.7

A series of lines are drawn from the apex (X1,Y1) to the base. This results in the apex pixel being overwritten 'count' number of times. The pixels in the proximity of the apex are also overwritten, but less frequently. This, is a very inefficient algorithm; ideally each pixel in the triangle would be written to once. This may be achieved by using more complex algorithms (e.g. a scan-line algorithm). This is beyond the scope of this book - for the interested reader the following book is recomended:

Fundamentals of Interactive Computer Graphics.
J.D. FOLEY and A. VAN DAM
Addison-Wesley 1982
ISBN 0-201-14468-9.

Here is a listing of the triangle command:

```
10      COUNT:  DEFS 4
20      DRAW:   EQU  &BBF6
30      PLOT:   EQU  &BBEA
40      INK:    EQU  &BBDE
50      TRI:    LD   A,(IX+0)
60              CALL INK:
70              LD   E,(IX+12)
80              LD   D,(IX+13)
90              LD   L,(IX+4)
100             LD   H,(IX+5)
110             AND  A
120             SBC  HL,DE
130             JP   M,END:
140             LD   (COUNT:),HL
150             LD   E,(IX+8)
160             LD   D,(IX+9)
170             LD   L,(IX+6)
180             LD   H,(IX+7)
190             PUSH DE
200             PUSH HL
210             LD   L,(IX+10)
220             LD   H,(IX+11)
230             PUSH HL
240             POP  IY
250             POP  IY
260             LD   E,(IX+12)
270             LD   D,(IX+13)
280             PUSH DE
290             POP  IX
300     NXTT:   POP  HL
310             POP  DE
320             PUSH DE
330             PUSH HL
340             CALL PLOT:
350             PUSH IX
360             POP  DE
370             PUSH IY
380             POP  HL
390             CALL DRAW:
400             INC  IX
410             LD   HL,(COUNT:)
420             LD   DE,1
430             AND  A
440             SBC  HL,DE
450             LD   (COUNT:),HL
460             JR   NZ,NXTT:
470             POP  DE
480             POP  DE
490     END:    RET
```

The circle command

The simplest method of producing a circle on the screen uses the SIN and COS functions of the Amstrad. Consider Figure 10.8:

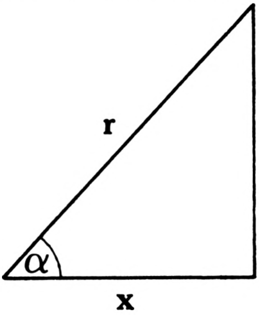


FIGURE 10.8

By Incrementing in the range $0<\alpha<\pi/2$ and plotting a point at the coordinates (X,Y) a quadrant of a circle is obtained. This quadrant can be developed into a full circle. Assuming that the centre of the circle is located at the graphics origin, consider Figure 10.9 for the point X,Y.

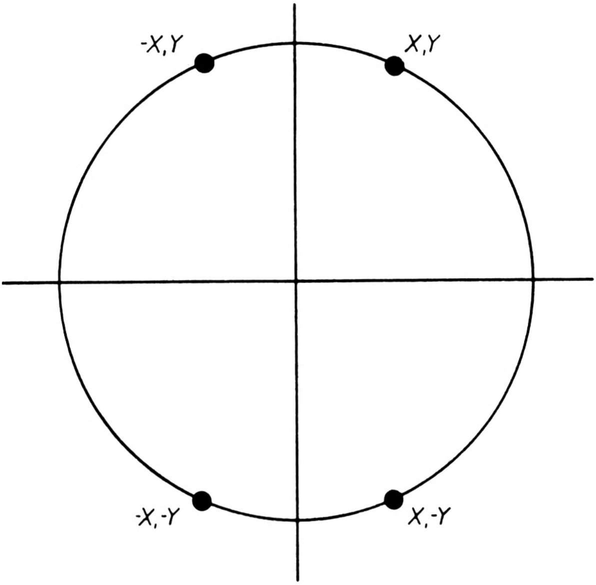
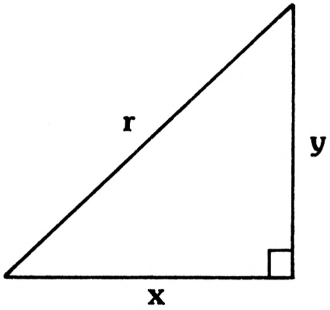


FIGURE 10.9

While this method works, it is grossly inefficient as the functions SIN and COS require considerable computational time hence slowing down the algorithm. A slightly more efficient algorithm can be derived from Pythagoras' Theorem, equation 10.1 below:

Pythagoras' Theorem



$$R^2 = X^2 + Y^2 \qquad \text{Equ 10.1}$$

Rearranging Equation 10.1

$$Y = \pm\sqrt{R^2 - X^2}$$

To draw a quadrant as before, X has to be incremented in the range 0<=X<=R, Y is then found for every value of X, and the pixel at coordinates X,Y set. Try the following BASIC program.

PROGRAM 10.8

```
10 MODE 0
20 DEFINT x,y,r
30 x=0
40 r=100
50 WHILE x<r
60 y=SQR(r*r-x*x)
70 PLOT x,y
80 x=x+1
90 WEND
100 END
```

This program produces a quadrant as shown below in Figure 10.10

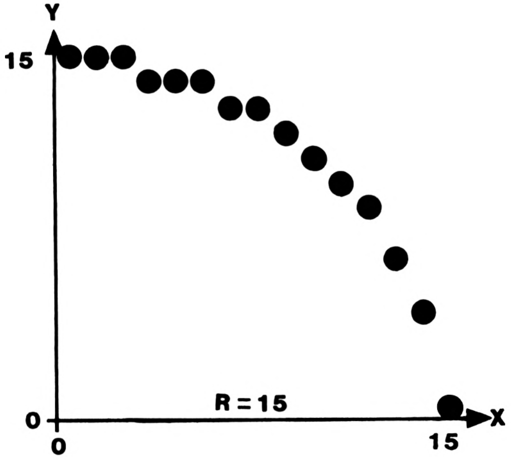


FIGURE 10.10

There are two main limitations to using this method.

1. The quality of the arc as X approaches R leaves a lot to be desired.

2. Although quicker than SIN or COS, finding the square root of a number still requires considerable computational time.

There are various methods by which these problems may be alleviated. In this chapter one possible solution will be looked at. It is based upon Bresenham's algorithm initially developed for mechanical plotters. The algorithm is considerably more efficient than either of the previously mentioned methods as all the arithmetic operations can be accomplished easily using a few additions, subtractions and shifts.

An adaptation of Bresenham's circle drawing algorithm

Instead of incrementing X in the range 0<=X<=R this method uses the range 0<=X<= π /4 thus producing a 45 degree segment. The complete circle is obtained by mirror imaging these calculated points. The heart of the algorithm is a routine which selects the pixel nearest to the true circle at the point in consideration. The distance between the true circle and the selected pixel is called the 'error term' and is derived as follows:-

Using Pythagoras' Theorem

$$r^2 = x^2 + y^2$$

assuming that the pixel is plotted at X,Y then the error term is given by

$$E= (x^2 +y)-r^2$$

By minimising E at each step, the closest approximation to a circle possible on the discrete pixel grid is obtained.

Consider Figure 10.11 which shows the various ways in which the true circle could cut the pixel grid.

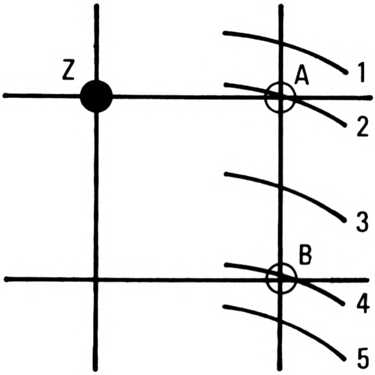


FIGURE 10.11

Assuming that the black pixel Z has just been set, the next pixel to be set could be either A or B. Defining the error term as the difference between the squared distances from the center of circle to either pixel A or B and to the actual circle at this point, the following equations may be derived.

For pixel A

$$E_A =(x_A^2 +y_A^2)-r^2 \qquad \text{Equ} \quad 10.2$$

pixel B

$$E_B =(x_B^2 -y_B^2)-r^2 \qquad \text{Equ} \quad 10.3$$

Thus if $|E_A| >= |E_B|$ pixel B is set, similary if $|E_A| < |E_B|$ then pixel A is set.

Combining equations 10.2 and 10.3 to form the total error term E

$$E_T = E_A + E_B$$

Now if $E_T >= 0$ pixel B is set, otherwise $E_T < 0$ and pixel A is set.

Reconsidering Figure 10.11

Case 1
$E_T < 0$ thus pixel A is set

Case 2
$E_T < 0$ thus pixel A is set

Case 3
$E_T < 0$ thus pixel A is set

Case 4
$E_T >= 0$ thus pixel B is set

Case 5
$E_T >= 0$ thus pixel B is set

As it stands the method works, however it is still necessary to calculate the square and square roots of the data to calculate the error term. It can be shown however by a series of arithmetic operations that the initial error is as follows.

$$E_T = 3-2r \qquad \text{Equ} \quad 10.4$$

The value of E_T changes dynamically throughout the program depending upon the choice of the previous pixel, as follows.

If pixel A is selected as $E_T < 0$ then the new E_T is given by

$$E_{T+1} = E_T + 4x + 6 \qquad \text{Equ} \quad 10.5$$

or if pixel B is selected as $E_T >= 0$ then

$$E_{T+1} = E_T + 4(x-y) + 10 \qquad \text{Equ} \quad 10.7$$

Equation 10.5 requires two adds and two shifts, equation 10.6 two adds, one subtract and two shifts; a considerable improvement upon the previous algorithms. A method of mirror imaging these points is now required to form the complete circle.If the X and Y values can be found for one quadrant then by mirror imaging these coordinates a full circle can be produced. See Figure 10.12

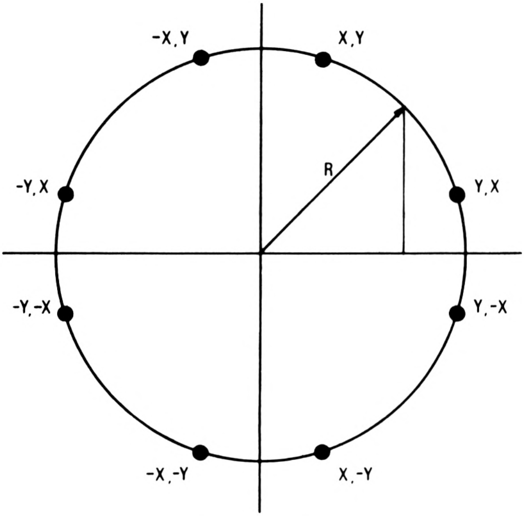


FIGURE 10.12

This algorithm is demonstrated in the following BASIC program.

PROGRAM 10.9

```
10 MODE 1
20 radius=100
30 x=0
40 y=radius
50 ORIGIN 320,200
60 diff=3-2*radius
70 WHILE x<y
80 GOSUB 150
90 IF d<0 THEN d=d+4*x+6: GOTO 120
100 d=d+4*(x-y)+10
110 y=y-1
120 x=x+1
130 WEND
140 END
150 PLOT x,y
160 PLOT y,x
170 PLOT y,-x
180 PLOT x,-y
190 PLOT x,-y
200 PLOT -x,-y
210 PLOT -y,-x
220 PLOT -y,x
230 PLOT -x,y
240 RETURN
```

The program produces a pixel distribution as shown in Figure 10.13, for a radius of 15.

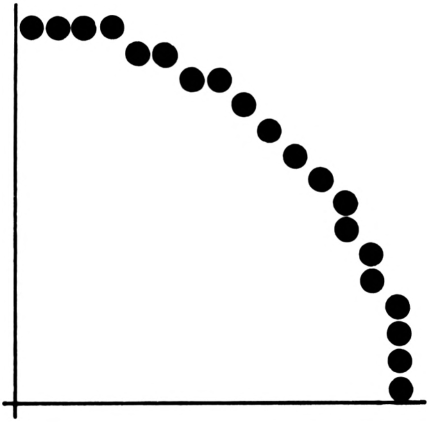


FIGURE 10.13

Here is the algorithm converted to assembly language: To save your fingers, all the external commands are included on the tape under the name GRAPHICS-EXT.

PROGRAM 10.10

```
10      PLOT:    EQU  &BBEA
20      INK:     EQU  &BBDE
30      ORGIN:   EQU  &BBCC
40      ORGET:   EQU  &BBCC
50      DIFF:    DEFS 2
60      RAD:     DEFS 2
70      X:       DEFS 2
80      Y:       DEFS 2
90      XO:      DEFS 2
100     YO:      DEFS 2
110     CIRCLE:  CALL ORGET:
120              LD   (XO:),DE
130              LD   A,(IX+0)
140              CALL INK:
150              LD   (YO:),HL
160              LD   D,(IX+7)
170              LD   E,(IX+6)
```

```
180             LD    H,(IX+5)
190             LD    L,(IX+4)
200             PUSH  HL
210             LD    H,(IX+3)
220             LD    L,(IX+0)
230             LD    (RAD:),HL
240             POP   HL
250             CALL  ORGIN:
260             LD    BC,0000
270             LD    (X:),BC
280             LD    HL,(RAD:)
290             LD    (Y:),HL
300             SLA   L
310             RL    H
320             PUSH  HL
330             POP   DE
340             LD    HL,3
350             XOR   A
360             SBC   HL,DE
370             LD    (DIFF),HL
380     CALC:   LD    HL,(X:)
390             LD    DE,(Y:)
400             PUSH  HL
410             PUSH  DE
420             CALL  MIRIM:
430             POP   DE
440             POP   HL
450             XOR   A
460             SBC   HL,DE
470             JP    P,END2:
480             LD    HL,(DIFF:)
490             LD    BC,0000
500             SBC   HL,BC
510             JP    P,LESS:
520             LD    DE,(X:)
530             SLA   E
540             RL    D
550             SLA   E
560             RL    D
570             LD    HL,6
580             ADD   HL,DE
590             LD    DE,(DIFF:)
600             ADD   HL,DE
610             JP    NXT3:
620     LESS:   LD    HL,(X:)
630             LD    DE,(Y:)
640             XOR   A
650             SBC   HL,DE
660             SLA   L
670             RL    H
```

```
680             SLA   L
690             RL    H
700             LD    DE,10
710             ADD   HL,DE
720             LD    DE,(DIFF:)
730             ADD   HL,DE
740             LD    DE,(Y:)
750             DEC   DE
760             LD    (Y:),DE
770     NXT3:   LD    (DIFF:),HL
780             LD    HL,(X:)
790             INC   HL
800             LD    (X:),HL
810             JP    CALC:
820     MIRIM:  LD    DE,(X:)
830             LD    HL,(Y:)
840             CALL  PLOT:
850             LD    DE,(Y:)
860             LD    HL,(X:)
870             CALL  PLOT:
880             LD    HL,0000
890             LD    BC,(X:)
900             XOR   A
910             SBC   HL,DE
920             PUSH  HL
930             PUSH  HL
940             LD    DE,(Y:)
950             CALL  PLOT:
960             POP   DE
970             LD    HL,(Y:)
980             CALL  PLOT:
990             LD    HL,0000
1000            LD    BC,(Y:)
1010            XOR   A
1020            SBC   HL,BC
1030            PUSH  HL
1040            PUSH  HL
1050            LD    DE,(X:)
1060            CALL  PLOT:
1070            POP   DE
1080            LD    HL,(X:)
1090            CALL  PLOT:
1100            POP   HL
1110            POP   DE
1120            PUSH  HL
1130            PUSH  DE
1140            CALL  PLOT:
```

```
1150            POP  HL
1160            POP  DE
1170            CALL PLOT:
1180            RET
1190  END2:     LD   DE,(XO:)
1200            LD   HL,(YO:)
1210            CALL ORGIN:
1220            RET
```

Well that's it, you have made it. Well done. It is hoped that you have enjoyed reading this book and feel that it was all worthwhile. Happy programming.

APPENDIX 1

The Z80 Instruction Set

Abbreviations used in the following tables:

Registers

r,r' = Refers to any one of the registers A,B,C,D,E,H,L.
dd = Refers to any one of the registers pairs BC,DE,HL,SP
qq = Refers to any one of the register pairs AF,BC,DE,HL
pp = Refers to any one of the register pairs BC,DE,1X,SP
rr = Refers to any one of the register pairs HC,DE,1X,SP
e = Refers to a two's complement offsett
s = Refers to any one of r,n,(HL),(IX+d);(IY+d)
d = Refers to any one of r,(HL),(IX+d),(IY+d) or in a index instruction to a two's complement offset.
H = High Byte: L = Low Byte.
S = Where b=bit 0 7

Addressing Modes

RR = Register - Register
Im = Immediate
IDX = Indexed
D = Direct
In = Indirect

Flags

C = Carry / link flag
Z = Zero flag
S = Sign flag
P/V = Parity or overflow flag
H = Half-carry flag
N = Add/Subtract flag

Key to flag outcomes

?	Flag set depending upon outcome of operation
0	Flag reset
1	Flag set
*	Flag not affected
-	Flag outcome unknown
V	Flag set on overflow
P	Flag set on parity
F	The P/V flag is set to the contents of the interrupt enable flip-flop(IFF)

The P/V Flag

If the operation indicated by a V in the P/V column results in an overflow then the V flag will be set (1) otherwise it will be reset (0). If the operation is used to test the parity, indicated by a P, then the flag will be set (1) if the parity is even else reset (0) on parity odd.

Addressing Modes

LD r,r'	Register,Register
LD r,n	Immediate
LD r,(IX+d)	Indexed
LD r,(nn)	Direct
LD r,(dd)	Indirect

Where

r or r' is a 8-bit register
n is an 8-bit register
d is a 2's complement offset
nn is an 16-bit number
dd is one of the following: BC, DE, HL, SP.

Eight-Bit Load Group										
Mnemonic	Symbolic Operation	No of bytes	No of cycles	Address mode	Flags Affected					
					C	Z	P/V	S	N	H
LD r,r'	r←r'	1	1	RR	*	*	*	*	*	*
LD r,n	r←n	2	2	Im	*	*	*	*	*	*
LD r,(HL)	r←(HL)	1	2	In	*	*	*	*	*	*
LD r,(IX+d)	r←(IX+d)	3	5	IDX	*	*	*	*	*	*
LD r,(IY+d)	r←(IY+d)	3	5	IDX	*	*	*	*	*	*
LD (HL),r	(HL)←r	1	2	In	*	*	*	*	*	*
LD (IX+d),r	(IX+d)←r	3	5	IDX	*	*	*	*	*	*
LD (IY+d),r	(IY+d)←r	3	5	IDX	*	*	*	*	*	*
LD (HL),n	(HL)←n	2	3	In	*	*	*	*	*	*
LD (IX+d),n	(IX+d)←n	4	5	IDX	*	*	*	*	*	*
LD (IY+d),n	(IY+d)←n	4	5	IDX	*	*	*	*	*	*
LD A,(BC)	A←(BC)	1	2	In	*	*	*	*	*	*
LD A,(DE)	A←(DE)	1	2	In	*	*	*	*	*	*
LD A,(nn)	A←(nn)	3	4	D	*	*	*	*	*	*
LD (BC),A	(BC)←A	1	2	In	*	*	*	*	*	*
LD (DE),A	(DE)←A	1	2	In	*	*	*	*	*	*
LD (nn),A	(nn)←A	3	4	D	*	*	*	*	*	*
LD A,I	A←I	2	2	RR	*	?	F	?	0	0
LD A,R	A←R	2	2	RR	*	?	F	?	0	0
LD I,A	I←A	2	2	RR	*	*	*	*	*	*
LD R,A	R←A	2	2	RR	*	*	*	*	*	*

Sixteen-Bit Load Group

Mnemonic	Symbolic Operation	No of Bytes	No of Cycles	Address Mode	Flags Affected C	Z	P/V	S	N	H
LD dd,nn	dd ← nn	3	3	Im	*	*	*	*	*	*
LD IX,nn	IX ← nn	4	4	Im	*	*	*	*	*	*
LD IY,nn	IY ← nn	4	4	Im	*	*	*	*	*	*
LD HL,(nn)	H ← (nn+1) L ← (nn)	3	5	D	*	*	*	*	*	*
LD dd,(nn)	dd ← (nn+1) dd ← (nn)	4	6	D	*	*	*	*	*	*
LD IX,(nn)	IX ← (nn+1) IX ← (nn)	4	6	D	*	*	*	*	*	*
LD IY,(nn)	IY ← (nn+1) IY ← (nn)	4	6	D	*	*	*	*	*	*
LD (nn),HL	(nn+1) ← H (nn) ← L	3	5	D	*	*	*	*	*	*
LD (nn),dd	(nn+1) ← dd (nn) ← dd	4	6	D	*	*	*	*	*	*
LD (nn),IX	(nn+1) ← IX (nn) ← IX	4	6	D	*	*	*	*	*	*
LD (nn),IY	(nn+1) ← IY (nn) ← IY	4	6	D	*	*	*	*	*	*

LD SP,HL	SP←HL	1	1	RR	*	*	*	*	*	*
LD SP,IX	SP←IX	2	2	RR	*	*	*	*	*	*
LD SP,IY	SP←IY	2	2	RR	*	*	*	*	*	*
PUSH qq	(SP-1)←qq (SP-2)←qq	1	3	Im	*	*	*	*	*	*
PUSH IX	(SP-1)←IX (SP-2)←IX	2	4	Im	*	*	*	*	*	*
PUSH IY	(SP-1)←IY (SP-2)←IY	2	4	Im	*	*	*	*	*	*
POP qq	qq←(SP) qq←(SP+1)	1	3	Im	*	*	*	*	*	*
POP IX	IX←(SP) IX←(SP+1)	2	4	Im	*	*	*	*	*	*
POP IY	IY←(SP) IY←(SP+1)	2	4	Im	*	*	*	*	*	*

Exchange Group and Block Transfer and Search Group

Mnemonic	Symbolic Operation	No of Bytes	No of Cycles	Address Mode	Flags Affected					
					C	Z	P/V	S	N	H
EX DE,HL	DE ↔ HL	1	1	RR	*	*	*	*	*	*
EX AF,AF'	AF ↔ AF'	1	1	RR	*	*	*	*	*	*
EXX	$\begin{pmatrix}BC\\DE\\HL\end{pmatrix} \leftrightarrow \begin{pmatrix}BC'\\DE'\\HL'\end{pmatrix}$	1	1	RR	*	*	*	*	*	*
EX (SP),HL	H ↔ (SP+1) L ↔ (SP)	1	5	RR	*	*	*	*	*	*
EX (SP),IX	IX ↔ (SP+1) IX ↔ (SP)	2	6	RR	*	*	*	*	*	*
EX (SP),IY	IY ↔ (SP+1) IY ↔ (SP)	2	6	RR	*	*	*	*	*	*
LDI	(DE) ← (HL) DE ← DE+1 HL ← HL+1 BC ← BC-1	2	4	In	*	*	?	*	0	0
LDIR	(DE) ← (HL) DE ← DE+1 HL ← HL+1 BC ← BC-1 UNTIL BC=O	2	5	In	*	*	0	*	0	0
LDD	(DE) ← (HL) DE ← DE-1 HL ← HL-1 BC ← BC-1	2	4	In	*	*	?	*	0	0

LDDR	(DE)←(HL) DE←DE-1 HL←HL-1 BC←BC-1 UNTIL BC=O	2	5	In	*	*	0	*	0	0
CPI	A-(HL) HL←HL+1 BC←BC-1	2	4	In	*	?	?	?	?	?
CPIR	A-(HL) HL←HL+1 BC←BC-1 UNTIL A=(HL) OR BC=O	2	5	In	*	?	?	?	?	?
CPD	A-(HL) HL←HL-1 BC←BC-1	2	4	In	*	?	?	?	?	?
CPDR	A-(HL) HL←HL-1 BC←BC-1 UNTIL A=(HL) OR BC=O	2	5	In	*	?	?	?	?	?

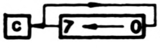
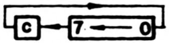
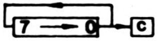
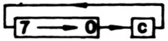
Eight-Bit Arithmetic and Logical Group

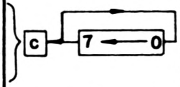
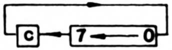
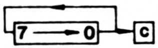
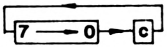
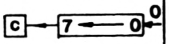
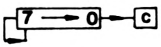
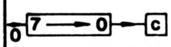
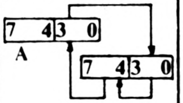
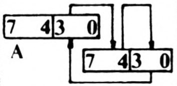
Mnemonic	Symbolic Operation	No of Bytes	No of Cycles	Address mode	Flags affected C	Z	P/V	S	N	H
				Im	?	?	V	?	0	?
ADD A,r	A←A+r	1	1	Im	?	?	V	?	0	?
ADD A,n	A←A+n	2	2	In	?	?	V	?	0	?
ADD A,(HL)	A←A+(HL)	1	2	IDX	?	?	V	?	0	?
ADD A,(IX+d)	A←A+(IX+d)	3	5	IDX	?	?	V	?	0	?
ADD A,(IY+d)	A←A+(IY+d)	3	5	Im	?	?	V	?	0	?
ADC A,s	A←A+s+CY			Im	?	?	V	?	1	?
SUB A,s	A←A-s			Im	?	?	V	?	1	?
SBC A,s	A←A-s-CY			Im	0	?	P	?	0	1
AND s	A←A s			Im	0	?	P	?	0	1
OR s	A←A s			Im	0	?	P	?	0	1
XOR s	A←A.s			Im	?	?	V	?	1	?
CP s	A-s			Im	*	?	V	?	0	?
INC	r-r+1	1	1	In	*	?	V	?	0	?
INC(HL)	(HL)←(HL)+1	1	3	IDX	*	?	V	?	0	?
INC(IX+d)	(IX+d)←(IX+d)+1	3	6	IDX	*	?	V	?	0	?
INC(IY+d)	(IY+d)←(IY+d)+1	3	6	Im	*	?	V	?	1	1
DEC d	d←d-1									

General Purpose Arithmetic and C.P.U. Control Groups

Mnemonic	Symbolic Operation	No of Bytes	No of Cycles	Address Mode	Flags Affected C	 Z	 P/V	 S	 N	 H
DAA	Converts A into packed BCD after an add or subtract on BCD operands	1	1	Im	?	?	P	?	*	?
CPL	A ← $\overline{A}$	1	1	Im	*	*	*	*	1	1
NEG	A ← O-A	2	2	Im	?	?	V	?	1	?
CCF	CY ← $\overline{CY}$	1	1	Im	?	*	*	*	0	*
SCF	CY ← 1	1	1	Im	1	*	*	*	0	0
NOP	No operation.	1	1	Im	*	*	*	*	*	*
HALT	CPU halted	1	1	Im	*	*	*	*	*	*
DI	IFF ← O	1	1	Im	*	*	*	*	*	*
EI	IFF ← 1	1	1	Im	*	*	*	*	*	*
IMO	Set Interrupt mode O	2	2	Im	*	*	*	*	*	*
IM1	Set Interrupt mode 1	2	2	Im	*	*	*	*	*	*
IM2	Set Interrupt mode 2	2	2	Im	*	*	*	*	*	*

Sixteen-Bit Arithmetic Group										
Mnemonic	Symbolic Operation	No of Bytes	No of Cycles	Address mode	Flags Affected C	Z	P/V	S	N	H
ADD HL, dd	HL ← HL+dd	1	3							
ACD HL, dd	HL ← HL+dd+CY	2	4	Im	?	*	*	*	0	-
SBC HL, dd	HL ← HL-dd-CY	2	4	Im	?	?	V	?	0	-
ADD IX, pp	IX ← IX+pp	2	4	Im	?	?	V	?	1	-
ADD IY, rr	IY ← IY+rr	2	4	Im	?	*	*	*	0	-
INC dd	dd ← dd+1	1	1	Im	?	*	*	*	0	-
INC IX	IX ← IX+1	2	2	Im	*	*	*	*	*	*
INC IY	IY ← IY+1	2	2	Im	*	*	*	*	*	*
DEC dd	dd ← dd-1	1	1	Im	*	*	*	*	*	*
DEC IX	IX ← IX-1	2	2	Im	*	*	*	*	*	*
DEC IY	IY ← IY-1	2	2	Im	*	*	*	*	*	*
				Im	*	*	*	*	*	*

Rotate and Shift Group										
Mnemonic	Symbolic Operation	No of Bytes	No of Cycles	Address Mode	Flags affected C	Z	P/V	S	N	H
RLCA		1	1	Im	?	*	*	*	0	0
RLA		1	1	Im	?	*	*	*	0	0
RRCA		1	1	Im	?	*	*	*	0	0
RRA		1	1	Im	?	*	*	*	0	0

RLCr		2	2	Im	?	?	P	?	0	0
RLC(HL)		2	4	Im	?	?	P	?	0	0
RLC(IX+d)		4	6	Im	?	?	P	?	0	0
RLC(IY+d)		4	6	Im	?	?	P	?	0	0
RL s		-	-	Im	?	?	P	?	0	0
RRC s		-	-	Im	?	?	P	?	0	0
RR s		-	-	Im	?	?	P	?	0	0
SLA s		-	-	Im	?	?	P	?	0	0
SRA s		-	-	Im	?	?	P	?	0	0
SRL s		-	-	Im	?	?	P	?	0	0
RLD		2	5	Im	*	?	P	?	0	0
RRD		2	5	Im	*	?	P	?	0	0

Bit set, Reset and Test Group										
Mnemonic	Symbolic Operation	No of Bytes	No of Cycles	Address Mode	Flags Affected					
					C	Z	P/V	S	N	H
Bit 6,r	Z← r	2	2	RR	*	?	-	-	0	1
Bit 6,(HL)	Z←(HL)	2	3	In	*	?	-	-	0	1
Bit 6,(IX+d)	Z←(IX+d)	4	5	IDX	*	?	-	-	0	1
Bit 6,(IY+d)	Z←(IY+d)	4	5	IDX	*	?	-	-	0	1
Set 6,r	r←1	2	2	RR	*	*	*	*	*	*
Set 6,(HL)	(HL)←1	2	4	In	*	*	*	*	*	*
Set 6,(IX+d)	(IX+d)←1	4	6	IDX	*	*	*	*	*	*
Set 6,(IY+d)	(IY+d)←1	4	6	IDX	*	*	*	*	*	*
Res 6,s	s←0	-	-	RR	*	*	*	*	*	*

Jump Group										
Mnemonic	Symbolic Operation	No of Bytes	No of Cycles	Address Mode	Flags Affected					
					C	Z	P/V	S	N	H
JP nn	PC ← nn	3	3	Im	*	*	*	*	*	*
JP cc,nn	If condition cc is true PC ← nn otherwise continue.	3	3	Im	*	*	*	*	*	*
JR e	PC ← PC+e	2	3	Im	*	*	*	*	*	
JR C e	If c=0 continue.	2	2	Im	*	*	*	*	*	
JR NC e	If c=1 continue.	2	2	Im	*	*	*	*	*	
JR Z e	If z=0 continue.	2	2	Im	*	*	*	*	*	
JR NZ e	If z=1 continue.	2	2	Im	*	*	*	*	*	*
JP (HL)	PC ← HL	1	1	In	*	*	*	*	*	*
JP (IX)	PC ← IX	2	2	IDX	*	*	*	*	*	*
JP (IY)	PC ← IY	2	2	IDX	*	*	*	*	*	*
DJNZ e	B ← B-1 if B=0, continue. If B=0 PC ← PC+e	2	3	Im	*	*	*	*	*	*

Call and Return Group

Mnemonic	Symbolic Operation	No of Bytes	No of Cycles	Address Mode	Flags Affected C	Z	P/V	S	N	H
CALL nn	(SP-1) ← PC (SP-2) ← PC PC ← nn	3	5	Im	*	*	*	*	*	*
CALL cc,nn	If condition cc is false continue, otherwise same as CALL nn	3	3	Im	*	*	*	*	*	*
RET	PC ← (SP) PC ← (SP+1)	1	3	Im	*	*	*	*	*	*
RET cc	If condition cc is false continue, otherwise same as RET	1	1	Im	*	*	*	*	*	*
RETI	Return from interrupt	2	4	Im	*	*	*	*	*	*
RETN	Return from non markable interrupt	2	4	Im	*	*	*	*	*	*
RST p	(SP-1) ← PC (SP-2) ← PC PC ← 0 PC ← P	1	3	Im	*	*	*	*	*	*

Input and Output Group

Mnemonic	Symbolic Operation	No of Bytes	No of Cycles	Address Mode	Flags Affected					
					C	Z	P/V	S	N	H
IN A,(n)	A ← (n)	2	3	Im	*	*	*	*	*	*
IN r,(c)	r ← (c)	2	3	In	*	?	P	?	0	?
INI	(HL) ← (c) B ← B-1 HL ← HL+1	2	4	In	*	?	-	-	1	-
INIR	(HL) ← (c) B ← B-1 HL ← HL+1 UNTIL B=0	2	5	In	*	1	-	-	1	-
IND	(HL) ← (c) B ← B-1 HL ← HL-1	2	4	In	*	?	-	-	1	-
INDR	(HL) ← (c) B ← B-1 HL ← HL-1 UNTIL B=0	2	5	In	*	1	-	-	1	-

Input and Output Group (contd.)										
Mnemonic	Symbolic Operation	No of Bytes	No of Cycles	Address Mode	Flags Affected					
					C	Z	P/V	S	N	H
OUT (n),A	(n) ← A	2	3	Im	*	*	*	*	*	*
OUT (c),r	(c) ← r	2	3	In	*	*	*	*	*	*
OUTI	(c) ← (HL) B ← B-1 HL ← HL+1	2	4	In	*	?	-	-	1	-
OTIR	(c) ← (HL) B ← B-1 HL ← HL+1 UNTIL B=0	2	5	In	*	1	-	-	1	-
OUTD	(c) ← (HL) B ← B-1 HL ← HL-1	2	4	In	*	?	-	-	1	-
OTDR	(c) ← (HL) B ← B-1 HL ← HL-1 UNTIL B=0	2	5	In	*	1	-	-	1	-

APPENDIX 2

Effects Of Instructions On The Flags

The Flags

Bit	Flag	
0	C	Carry
1	N	add/subtract
2	P/V	Parity/oVerflow
3	-	-
4	H	Half carry
5	-	-
6	Z	Zero
7	S	Sign

Effects Of Instructions On The Sign Flag

Group	Instruction	Effects
8-bit Loads	LD A,I	Set if I register is negative, else reset
	LD A,R	Set if R register is negative, else reset
Compares	CPI,CPIR, CPD,CPDR	Set if result is negative, else reset
8-bit arithmetic	ADD A,S ADC A,S SUB S SBC A,S AND S OR S XOR S CP S INC S DEC S	Set if result is negative, else reset
General-purpose arithmetic	DAA	Set if msb of A=1, else reset
	NEG	Set if result is negative, else reset
16-bit arithmetic	ADC HL,SS SBC HL,SS	Set of result is negative, else reset
Shifts and rotates	RLC S RL S RRC S RR S SLA S SRA S SRL S	Set if result is negative, else reset
	RLD RRD	Set if A is negative after shift, else reset
Bit	BIT B,S	Corrupted
Inputs and Outputs	IN R,(C)	Set if input data is negative, else reset
	IN,INIR IND,INDR, OUTI,OTIR, OUTD,OTDR	Corrupted

Effects of Instructions On The Zero Flag

Group	Instruction	Effects
8-bit loads	LD A,I LDA,R	Set Z if I register=0, else reset Z Set Z if R register=0, else reset Z
Compares	CPI,CPIR, CPD,CPDR	Set Z if A=(HL), else reset Z
8-bit arithmetic	ADD A,S ADC,A,S SUB S SBC A,S OR S XOR S CP S INC S DEC S	Set Z if result=0, else reset Z
General purpose arithmetic	DAA NEG	Set Z if result=0, else reset Z
16-bit arithmetic	ADC HL,SS SBC HL,SS	Set Z if result=0, else reset Z
Shifts and rotates	RLC S RL RRC S RR S SLA S SRA S SRL S RLD RRD	Set Z if result=0, else reset Z
Bit	BIT B,S	Set Z if specified bit=0, else reset Z
Inputs and Outputs	INR,(C) INI,IND, INIR,INDR OUTI,OUTD OTIR,OTDR	Set Z if input data=0 else reset Z Set Z if B-1=0, else reset Z Set Set Z if B-1=0, else reset Z Set Z

Effects Of Instructions On The Half-Carry And Add/Subtract Flags

Group	Instruction	Half-Carry Effects	Add/Subtract
8-bit loads	LD A,I LD A,R	Reset	Reset
	LDI,LDIR, LDD,LDDR	Reset	Reset
Compares	CPI,CPIR,CPD CPDR	Set if no borrow from bit 4 else reset	Set
8-bit arithmetic	ADD A,S ADC A,S	Set if no carry from bit 3, else reset	Reset
	SUB S SBC A,S	Set if no borrow from Bit 4, else reset	Set
	AND S OR S XOR S	Set	Reset
	CP S	Set if no borrow from bit 4, else reset	Set
	INC S	Set if carry from bit 3, else reset	Reset
	DEC S	Set if no borrow from bit 4, else reset	Set
General purpose arithmetic	DAA	Corrupted	No effect
	CPL	Set	Set
	NEG	Set if no borrow from bit 4, else reset	Set
	CCF	No effect	Reset
	SCF	Reset	Set

Group	Instruction	Half-Carry Effects	Add/Subtract Effects
16-bit arithmetic	ADD HL,SS ADC HL,SS	Set if carry out of bit 11, else reset	Reset
	SBC HL,SS	Set if no borrow from bit 12, else reset	Set
	ADD IX,PP ADD IY,RR	set if carry out of bit 11, else reset	Reset
Shifts and rotates	RLCA RLA RRCA RRA RLC S RL S RRC S RR S SLA S SRA S SRL S RLD RRD	Reset	Reset
Bit	BIT B,S	Set	Reset
Inputs outputs	INR,(C)	Reset	Reset
	INI,INIR, IND,INDR, OUTI,OTIR, OUTD,OTDR	Corrupted	Set

Effects Of Instruction On The Parity/Overflow Flag

Group	Instruction	Effects
8-bit loads	LD A,I LD A,R	Copy of interrupt flip-flop 2
Block instructions	LDI,LDD CPI,CPIR CPD,CPDR	Set if BC<>1, else reset
	LDIR,LDDR	Reset
8-bit arithmetic	ADD A,S ADC A,S SUB S SBC A,S	Set if overflow, else reset
	AND S OR S XOR S	Set if parity even, else reset
	CP S	Set if overflow, else reset
	INC S	Set if operand was 7FH before increment, else reset
	DEC S	Set if operand was 80H before increment, else reset
General-purpose arithmetic	DAA	Set if (A) parity even, else reset
	NEG	Set if (A) was 80H before negate, else reset
16-bit arithmetic	ADC HL,SS SBC HL,SS	Set if overflow, else reset
Shifts and rotates	RLC S RL S RRC S RR S SLA S SRA S SRL S RLD S RRD S	Set if parity even, else reset
Bit	BIT B,S	Corrupt

Group	Instruction	Effect
Inputs and Outputs	IN R,(C)	Set if parity even, else reset
	INI,INIR, IND,INDR, OUTI,OTIR, OUTD,OTDR	Corrupt

Effects Of Instructions On The Carry Flag

Group	Instruction	Carry Effects
8-bit loads	LD A,I LA A,R	No effect
Block instructions	CPI,CPIR, CPD,CPDR LDI,LDIR LDD,LDDR	No effect
8-bit arithmetic	ADD A,S ADC A,S	Set if carry from bit 7, else reset
	SUB S SBC S	Set if no borrow, else reset
	AND S OR S XOR S	Reset
	CP S	Set if no borrow, else reset
General-purpose arithmetic	DAA	Set if bcd carry, else reset
	NEG	Set if A was not 0 before negate, else reset
	CCF	Set if CY was 0 before CCF, else reset
	SCF	Set

Group	Instruction	Carry Effects
16-bit arithmetic	ADD HL,SS ADD LH,SS	Set if carry from bit 15, else reset
	SBC HL,SS	Set if no borrow, else reset
	ADD IX,PP ADD IY,RR	Set if carry from bit 15, else reset
Shifts and rotates	RLCA RLA	Copy A bit 7
	RRCA RRA	Copy A bit 0
	RLC S RL S	Copy bit 7 of operand
	RRC S RR S	Copy bit 0 of operand
	SLA S	Copy bit 7 of operand
	SRA S	Copy bit 0 of operand
	SRL S	Copy bit 0 of operand

APPENDIX 3

The Effects of Compares on the Overflow, Sign and Carry Flags

		Interpretation				
A	s	8-bit	2's comp	V	S	C
30	20	A>s	A>s	0	0	0
20	C0	A<s	A>s	0	0	1
70	C0	A<s	A>s	1	1	1
C0	70	A>s	A<s	1	0	0
C0	20	A>s	A<s	0	1	0
20	30	A<s	A<s	0	1	1

s = One of the following:- A,B,C,D,E,H,L,(HL),(IX+d),(IY+d)

1 = flag set   0 = flag reset

Conditions:

Flag	Set	Reset
V	PO (Parity Odd)	PE (Parity Even)
S	P (Positive)	M (Negative)
C	C (Carry)	NC (No Carry)

In the above, if the sign and overflow flags are the same, then A>s in two's complement. If they are different, then A<s, assuming the zero flag is not set, of course.

APPENDIX 4

Some Built-In Routines

&BB00	Initialise key manager
Entry None	Exit A,B,C,D,E,H,L and Flags corrupt.
Clears keyboard buffer. Sets key repeats to default values. Enables interrupts. Shift and caps lock set to off. Disables break events.	

&BB03	Reset key manager
Entry None	Exit A,B,C,D,E,H,L and Flags corrupt.
Clears keyboard buffer. Sets key repeats to default values. Enables interrupts. Disables break events.	

&BB06	Await character Get character from keyboard buffer. If empty, don't wait.
Entry None	Exit A: The character (ASCII) Carry: True Other flags: Corrupt
Expands single expansion tokens.	

&BB09	Fetch character
Get character from keyboard buffer. If empty, don't wait.	
Entry	Exit
None	A: The character, or corrupt Carry: True if character, else false Other flags: Corrupt
Expands single expansion tokens.	

&BB18	Await character (non expanded)
Get key press from keyboard buffer. Wait if none present.	
Entry	Exit
None	A: The character/token (ASCII) Carry: True Other flags: Corrupt
Does not expand expansion tokens.	

&BB1B	Fetch character (non expanded)
Get key press from keyboard buffer. If none, don't wait	
Entry	Exit
None	A: The character or corrupt Carry: True, if character available. Other flags: Corrupt
Does not expand expansion tokens.	

&BB1E	Key test
See whether specified key or joystick button is pressed.	
Entry	Exit
A: key number (ASCII)	A, HL: Corrupt C: 128 = ctrl pressed 32 = shift pressed 160 = shift & ctrl pressed Carry: False Zero: False if pressed, else true Other flags: Corrupt

&BB21	Keyboard status
See whether shift lock or caps lock is pressed.	
Entry	Exit
None	A: Corrupt L: 0 = shift lock off 255 = lock on H: 0 = caps lock off 255 = caps lock on Flags: Corrupt

&BB24	Joystick Status
Entry	Exit
None	A: Status of joystick 0 H: Status of joystick 0 L: Status of joystick 1

Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Spare	Fire 1	Fire 2	Right	Left	Down	Up Button

Bit 7 = 0; Above bits set means appropriate state applies

&BB39	Set Repeat Key
Enable/disable auto-repeat on specified key	
Entry A: Key number (ASCII) B: 0 = No repeat 255 = Repeat enabled	Exit A,B,C,H,L: Corrupt Flags: Corrupt

&BB3C	Test Repeat
Find repeat status of specified key	
Entry A: Key number (ASCII)	Exit A,HL: Corrupt Zero: False if repeat, else true Carry: False

&BB3F	Repeat Set
Set start up delay and repeat speed	
Entry H: Start up delay L: Repeat speed	Exit A: Corrupt Flags: Corrupt
Delays and repeats are expected in 50ths of a second, e.g. 50 = 1 second.	

&BB4E	Initialise text VDU
Entry None	Exit A,B,C,D,E,H,L: Corrupt Flags: Corrupt
All text streams set to defaults (ink, paper, window, etc). User defined characters lost. Control codes and text indirections set to defaults.	

&BB51	Reset text VDU
Entry None	Exit A,B,C,D,E,H,L: Corrupt Flags: Corrupt
Sets control codes and text indirections to defaults.	

&BB54	Enable text VDU
Allows characters to be placed on the screen.	
Entry None	Exit A: Corrupt Flags: Corrupt
Clears control code buffer.	

&BB57	Disable text VDU
Prevents characters from being placed on the screen.	
Entry None	Exit A: Corrupt Flags: Corrupt
Empties control code buffer.	

&BB5A	Output text
Output character on current stream.	
Entry A: Character to send(ASCII)	Exit

&BB5D	Write text character to screen
Entry A: Character to print	Exit A,B,C,D,E,H,L: Corrupt Flags: Corrupt
Updates cursor position	

&BB60	Read text character from screen
Entry None	Exit A: The character, or zero Carry: True if character, else false Other flags: Corrupt
Cursor position needs to be set first.	

<table>
<tr><td>&BB63</td><td>Set graphic text</td></tr>
<tr><td colspan="2">Enables/disables text writing with graphics VDU</td></tr>
<tr><td>Entry
A: <> 0 enable
= 0 disable</td><td>Exit
A: Corrupt
Flags: Corrupt</td></tr>
<tr><td colspan="2">When enabled, control codes are printed, not executed.</td></tr>
</table>

<table>
<tr><td>&BB66</td><td>Define window</td></tr>
<tr><td colspan="2">Sets size of text window</td></tr>
<tr><td>Entry
H: Column value for one edge
D: Column value of other edge
L: Row value of one edge
E: Row value of other edge</td><td>Exit
A,B,C,D,E,H,L: Corrupt
Flags: Corrupt</td></tr>
<tr><td colspan="2">Lowest of H and D is left column. Lowest of L and E is top row.
Top left of screen = 1,1
Puts cursor at top left of window.</td></tr>
</table>

<table>
<tr><td>&BB6C</td><td>Clear text window</td></tr>
<tr><td>Entry
None</td><td>Exit
A,B,C,D,E,H,L: Corrupt
Flags: Corrupt</td></tr>
<tr><td colspan="2">Puts cursor at top left of window.</td></tr>
</table>

&BB6F	Set text column
Entry	Exit
A: New column	A,H,L: Corrupt Flags: Corrupt
1 = leftmost column within window	

&BB72	Set text row
Entry	Exit
A: New row	A,H,L: Corrupt Flags: Corrupt
1 = Topmost row within window	

&BB78	Find text cursor
Returns current cursor position	
Entry	Exit
None	H: Cursor column L: Cursor row A: Scroll count Flags: Corrupt

<table>
<tr><td>&BB87</td><td>Test cursor position</td></tr>
<tr><td colspan="2">Find out where next character will be printed</td></tr>
<tr><td>Entry
H: Column to check
L: Row to check</td><td>Exit
A: Corrupt
B: = 0 If window would scroll down
= 255 If window would scroll up else corrupt
H: Column at which print would occur
L: Row at which print would occur
Carry = False if window will scroll, else true
Other flags: Corrupt</td></tr>
<tr><td colspan="2">1,1 = top left of window.</td></tr>
</table>

<table>
<tr><td>&BB90</td><td>Set pen colour</td></tr>
<tr><td colspan="2">Sets foreground text colour</td></tr>
<tr><td>Entry
A: Colour number</td><td>Exit
A,H,L: Corrupt
Flags: Corrupt</td></tr>
</table>

<table>
<tr><td>&BB96</td><td>Set paper colour</td></tr>
<tr><td colspan="2">Sets text background colour</td></tr>
<tr><td>Entry
A: Paper colour number</td><td>Exit
A,H,L: Corrupt
Flags: Corrupt</td></tr>
</table>

&BB9C	Swap text colours
Swaps current text foreground and background colours	
Entry	Exit
None	A,H,L: Corrupt Flags: Corrupt

&BBBA	Initialise graphics VDU
Entry	Exit
None	A,B,C,D,E,H,L: Corrupt Flags: Corrupt
Sets graphics VDU indirections to defaults. Resets graphics window to full screen. Does not clear window.	

&BBBD	Reset graphics VDU
Entry	Exit
None	A,B,C,D,E,H,L: Corrupt Flags: Corrupt
Sets graphic VDU indirections to default.	

&BBC0	Graphics absolute move
Entry	Exit
DE: Absolute X coordinate HL: Absolute Y coordinate	A,B,C,D,E,H,L: Corrupt Flags: Corrupt

&BBC3	Graphics relative move
Entry	Exit
DE: Relative X coordinate HL: Relative Y coordinate	A,B,C,D,E,H,L: Corrupt Flags: Corrupt

&BBC6	Find graphics position
Entry	Exit
None	DE: X coordinate HL: Y coordinate A : Corrupt Flags: Corrupt

&BBC9	Set graphics origin
Entry	Exit
DE: X coordinate HL: Y coordinate	A,B,C,D,E,H,L: Corrupt Flags: Corrupt
(0,0) = lower left corner of screen = default graphics origin.	

&BBCC	Find graphics origin
Entry	Exit
None	DE: X coordinate HL: Y coordinate
(0,0) = lower left corner of screen = default graphics origin.	

&BBCF	Set graphics window width
Entry DE: X coordinate of one edge HL: X coordinate of other edge	Exit A,B,C,D,E,H,L: Corrupt Flags: Corrupt
Left edge = smaller of DE and HL (0,0) = lower left corner of screen	

&BBD2	Set graphics window height
Entry DE: Y coordinate of one edge HL: Y coordinate of other edge	Exit A,B,C,D,E,H,L: Corrupt Flags: Corrupt
(0,0) = lower left of screen bottom edge = lower of DE and HL	

&BBDB	Clear graphics window
	Sets window to background colour
Entry None	Exit A,B,C,D,E,H,L: Corrupt Flags: Corrupt

&BBDE	Sets graphics foreground colour
Entry A: Colour number	Exit A: Corrupt Flags: Corrupt

&BBE4	Set graphics background colour
Entry	Exit
A: Colour number	A: Corrupt Flags: Corrupt

&BBE7	Find graphics background colour
Entry	Exit
None	A: Background colour number Flags: Corrupt

&BBEA	Absolute graphics plot
Entry	Exit
DE: X coordinate HL: Y coordinate	A,B,C,D,E,H,L: Corrupt Flags: Corrupt

&BBED	Relative graphics plot
Entry	Exit
DE: Relative X coordinate HL: Relative Y coordinate	A,B,C,D,E,H,L: Corrupt Flags: Corrupt

&BBF0	Absolute graphics test point
Entry	Exit
DE: X coordinate HL: Y coordinate	A: Background colour of point B,C,D,E,H,L: Corrupt Flags: Corrupt

&BBF3	Relative graphics test point
Entry	Exit
DE: Relative X coordinate HL: Relative Y coordinate	A: Background colour of point B,C,D,E,H,L: Corrupt Flags: Corrupt

&BBF6	Absolute graphics line
Entry	Exit
DE: X coordinate of endpoint HL: Y coordinate of endpoint	A,B,C,D,E,H,L: Corrupt Flags: Corrupt

&BBF9	Relative graphics line
Entry	Exit
DE: Relative X coordinate HL: Relative Y coordinate	A,B,C,D,E,H,L: Corrupt Flags: Corrupt

&BBFC	Graphics text
Writes text at current graphics position	
Entry A: Character	Exit A,B,C,D,E,H,L: Corrupt Flags: Corrupt
Top left of character is written at current graphics position. The current graphics position is updated.	

&BC0E	Set Screen Mode
Set screen mode and text and graphics VDU's	
Entry A: Required mode	Exit A,B,C,D,E,H,L: Corrupt Flags: Corrupt

&BC14	Clear screen
Clear the screen memory	
Entry None	Exit A,B,C,D,E,H,L: Corrupt Flags: Corrupt

&BC32	Set ink colour
Entry A: Ink number B: First colour C: Second colour	Exit A,B,C,D,E,H,L: Corrupt Flags: Corrupt
If B and C are different, the ink colour will alternate.	

&BC4D	Scroll line
Move screen up or down 1 line	
Entry To scroll down: B = 0 To scroll up: B <> 0 A: Paper colour of new line	Exit A,B,C,D,E,H,L: Corrupt Flags: Corrupt

&BC59	Set Graphics Plot mode
Entry A:Write mode	Exit A,B,C,D,E,H,L: Corrupt Flags: Corrupt
A = 0: FORCE mode	NEW = INK
A = 1: XOR mode	NEW = INK XOR OLD
A = 2: AND mode	NEW = INK AND OLD
A = 3: OR mode	NEW = INK OR OLD
NEW = FINAL PIXEL SETTING OLD = CURRENT PIXEL SETTING	
INK = INK PLOTTED	

&BCA7	Sound reset
Clear all sound channels and chip	
Entry None	Exit A,B,C,D,E,H,L: Corrupt Flags: Corrupt

&BCAA Sound channel

Add a sound to the channel

Entry	Exit
HL: Address of sound program	HL: Corrupt if sound added : Preserved if channel full Carry: True if sound added : False if channel full A,B,C,D,E,IX: Corrupt Other flags: Corrupt

BYTE 0:	Channels and synchronisation.
BYTE 1:	Amplitude envelope to use
BYTE 2:	Tone envelope to use.
BYTE 3,4:	Tone period.
BYTE 5:	Sound period.
BYTE 6:	Starting amplitude.
BYTE 7,8:	Duration, or number of repetitions.

Configuration of byte 0:

BIT 0:	Channel A
BIT 1:	Channel B
BIT 2:	Channel C
BIT 3:	Synchronise with Channel A
BIT 4:	Synchronise with Channel B
BIT 5:	Synchronise with Channel C
BIT 6:	Sound hold
BIT 7:	Empty queue

&BCAD Sound check

Status of a sound queue

Empty	Exit
A: Test bit	B,C,D,E,H,L: Corrupt Flags: Corrupt

&BCB0 Empty Queue Event

Set up sound to occur when a sound queue becomes empty

Entry	Exit
A: Bit for link (Channel—Event) HL: Address of event	A,B,C,D,E,H,L: Corrupt Flags: Corrupt
Bit 0: Link channel A Bit 1: Link channel B Bit 2: Link channel C	

&BCB3 Sound Release

Release particular held sounds

Entry	Exit
A: Channels to release	A,B,C,D,E,H,L,IX: Corrupt Flags: Corrupt
Bit 0: Release channel A Bit 1: Release channel B Bit 2: Release channel C	

&BCB6 Sound hold

All sounds are stopped immediately

Entry	Exit
None	Carry: True if sound active, else false A,B,C,H,L: Corrupt Other flags: Corrupt

Sounds are automatically restarted by SOUND CHANNEL, SOUND RELEASE and CONTINUE SOUND.

&BCB9 Continue sound

Release all held sounds

Entry	Exit
None	A,B,C,D,E,IX: Corrupt Flags: Corrupt

&BCBC Amplitude envelope

Set up one of the 15 amplitude envelopes

Entry	Exit
A: Envelope number HL: Address of amplitude	HL: Contains data block address plus 16 if envelope correct, else preserved A,B,C: Preserved if envelope invalid, else corrupt D,E: Corrupt Carry: True if envelope correct, else false Other flags: Corrupt

BYTE 0: Number of parts in envelope
BYTES 1,2,3: First part of envelope
BYTES 4,5,6: Second part of envelope
BYTES 7,8,9: Third part of envelope
BYTES 10,11,12: Fourth part of envelope
BYTES 13,14,15: Fifth part of envelope

BCBF Tone envelope

Set up one of the 15 tone envelopes

Entry

A: Envelope number
HL: Address of tone data block

Exit

HL: Contains data block address plus 16 if envelope correct, else preserved
A,B,C: Preserved: envelope invalid
D,E: Corrupt
Carry: True if envelope correct, else false
Other flags: Corrupt

BYTE 0: Number of parts in envelope
BYTES 1,2,3: First part of envelope
BYTES 4,5,6: Second part of envelope
BYTES 7,8,9: Third part of envelope
BYTES 10,11,12: Fourth part of envelope
BYTES 13,14,15: Fifth part of envelope

&BCC2 Amplitude address

Finds address of an amplitude envelope

Entry

A: Envelope number

Exit

A: Corrupt
BC: Length of envelope in bytes if envelope correct, else preserved
HL: Address of envelope if envelope correct, else corrupt
Carry: True if envelope correct, else false
Other flags: Corrupt

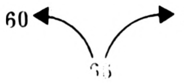
A must be between 1 and 15

<table>
<tr><td colspan="2">&BCC5 Tone address</td></tr>
<tr><td colspan="2">Address of tone envelope</td></tr>
<tr><td>Entry
A: Envelope number</td><td>Exit
A: Corrupt
BC: Length of envelope if envelope correct, else preserved
HL: Address of envelope if envelope correct, else corrupt
Carry: True if envelope correct, else false
Other flags: Corrupt</td></tr>
<tr><td colspan="2">A must be between 1 and 15
For further details on any of these routines see "The Complete CPC 464 Operating System Firmware Specification" (SOFT 158).</td></tr>
</table>

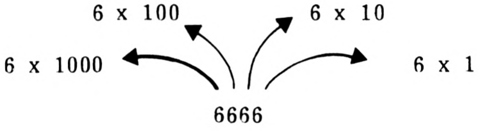
APPENDIX 5

Binary,Binary-Coded and Hexadecimal Notations

Modern counting systems, in general use throughout the world, use the decimal system. This has been developed to count past 10 and also below 1. In this system, the digits to the left in a number are of greater value than those to the right; for instance in the number 66 the first 6 has a value 10 times the second, i.e.

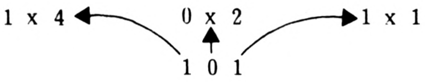


This is extended in larger numbers where digits to the left are successively greater by a multiple of ten, i.e.



A system where the position or place of a digit in a number affects its value is known as a PLACE-VALUE numbering system. In the decimal system, the values of digits increase in multiples of 10 and this is known as the BASE for that system. Other systems use different bases but follow the same pattern as the decimal system, i.e. the place to the left is greater being multiplied by the base.

The computer, being basically electrical in operation, can only recognise two states, on or off, often represented as '1' and '0'. Thus it uses the Binary system - ie. base 2. Any number in binary consists simply of 0's and 1's, or electrically speaking, offs and ons (or electronically, zero volts and some volts). To count past one, the binary system must resort to place-value notation and, as with other systems, the multiplying factor is the base, i.e. 2. Thus, the number 101 in base 2 or binary represents:



i.e. 4+0+1=5. Clearly. the plethora of bases presents a problem when representing numbers. In decimal (base 10), '101' represents one hundred and one while in binary (base 2) '101' represents 5. To overcome this ambiguity, a convention exists when representing numbers so that the base is written to the right of the number, just below the line. Thus, the two numbers discussed above become:

101_{10} = One hundred and one in base ten.

101_{2} = Five in base two.

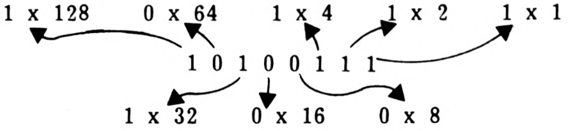
The present-day generation of home computers (1985-style) uses eight bit registers or memories and can, thus, represent numbers up to 11111111 , i.e. 255 in base 10:

128 + 64 + 32 + 16 + 8 + 4 + 2 + 1 $=255_{10}$

1	1	1	1	1	1	1	1	Digit
128	64	32	16	8	4	2	1	Equivalent in base 10

FIGURE A.1

By way of example, let's make another conversion - say, 10100111_{2}.



Thus 10100111 = 1x128 +0x64 +1x32 +0x16 +0x8 +1x4 +1x2 +1x1
= 128+32+4+2+1
= 167_{10}

Just to check your understanding, have a go at the following:

EXERCISE A5.1

Calculate the value of the following binary numbers in base 10:-

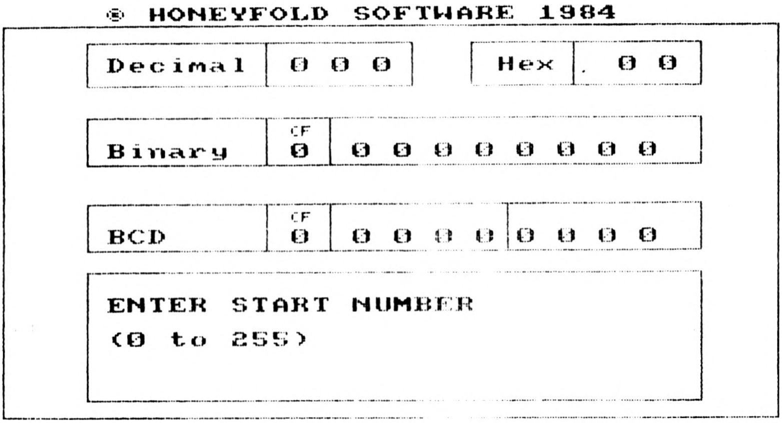
i) 00000011_2
ii) 00000100_2
iii) 10000000_2
iv) 10000011_2
v) 10110111_2
vi) 01110011_2

The answers are in the solutions Chapter.

To make the idea of converting numbers from Decimal to Binary notation clearer try running the program 'BIN/HEX' on the cassette. Type in the following:

```
RUN "BIN/HEX"
```

The program will now automatically be loaded and run by the computer and the following display will appear:



```
© HONEYFOLD SOFTWARE 1984

Decimal   0 0 0        Hex   . 0 0

          CF
Binary    0   0 0 0 0 0 0 0 0

          CF
BCD       0   0 0 0 0 0 0 0 0

ENTER START NUMBER
(0 to 255)
```

Don't worry about the boxes marked 'Hex' or 'BCD', we will be coming to them a little later. What we are interested in, at the moment, are the 'Decimal' and 'Binary' boxes.

As the display indicates, the program is waiting for you to enter a number in the range 0 to 255. Type in '1' and press 'ENTER'. The screen changes to give the following display:

All the boxes have changed to display the value of 1 in the various different notations. The instruction boxes tell you that the current value can be incremented or decremented, using the up-arrow and down-arrow cursor keys respectively.

Press the up-arrow key and as the values increase you will be able to see clearly how binary counts. If you now press 'E' you will be able to enter a new start number. Enter '15' and the binary box should contain '00001111'. Now increment this by one. The left four bytes (the least significant bytes) have all changed to zero and the fifth byte has become one. One way of understanding this is to lay out the addition:

```
   1111   A
 +    1   B
```

On adding the 1 to 1 this gives '2' i.e. 0, carry 1. This carry then produces another '0' + a carry, and so on.

When the register is full i.e. 11111111_2 , the addition of a further 1 will clock the register back to zero and 256 will be lost. However, with the Z80 chip all is not lost as the chip has a carry flag that stores the fact that a carry has occurred. (The carry flag is marked 'CF' on the display.)

To see the carry flag work press E and then start the count at about 250_{10} i.e. enter 250 <RETURN>. It's now not too far to increment up to 255 and 11111111_2 .

Now watch the binary box as you count past 255_{10} to 256_{10} . The binary cycles back to zero + the carry bit. This is a handy feature of the Z80 but must not be relied on as more than a temporary store of the carry. It's just as easily reset to zero as it is set to 1!

The program will also accept numbers that are entered in Binary form. However, to do this the number **MUST** be preceeded by the prefix **'&X'**. Thus to enter the binary number 101010 (42 decimal) type:

```
&X101010
```

and then press ENTER. All the boxes change to show 42 in the various notations.

To see if you really understand Binary notation try converting the following binary numbers to decimal on paper and then use the program to check your answers.

EXERCISE A5.2

i) 11111
ii) 101001
iii) 10111101
iv) 10001000

The answers are in the solutions chapter.

Keep the BIN/HEX program in the computer; you will need it again soon!

While the 0's and 1's are convenient for the computer, they are much less so for the mere human so a compromise is sought. Decimal notation is of little use as, apart from 1_{10} and 1_2 there is no other correspondence. A further idea would be to take the whole eight binary bits as a digit (i.e. up to 255) and use a base of 256! What would you see as the objection to this? That's apart from the idea itself being a bit mind-bending!

Time to think ... The answer comes from an examination of the base 10 case in which ten digits (0 to 9) are needed to represent the ten steps up to 10. In the base 2 system, two digits are needed so base 256 would need 256 digits!

Hexadecimal

A compromise system adopted splits the eight bits up into two parts and represents these separately. Thus, the largest number to be represented is 1111_2 or 15_{10} and this requires, along with the 0, sixteen different symbols. The ones adopted for this job are:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	Decimal number
0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	Symbol

FIGURE A5.2

Using this notation, any eight bit number can be represented by two symbols, one for the most significant four bits and one for the least significant four bits. To avoid the rather long description of these two halves of a byte, they are given the term NYBBLES. Thus a byte consists of two nybbles, a most significant nybble (MSN) and a least significant nybble (LSN) - see Figure A5.3.

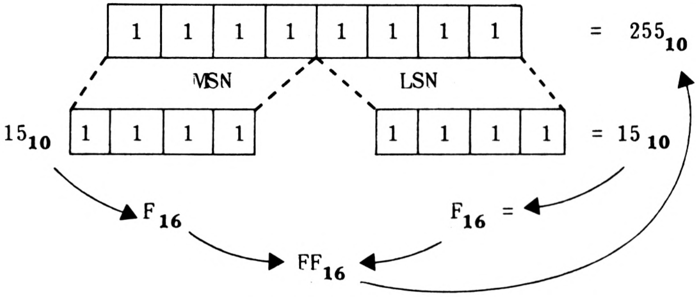


FIGURE A5.3

The system described, which uses sixteen symbols is given the name HEXADECIMAL - usually abbreviated to HEX. Its major advantage, as far as computers are concerned, is that it is compatible with binary. Any eight bit binary number can be represented by two hexadecimal characters.

If you now look at the top right-hand box on the screen that we pretended not to notice earlier, you will see that this counts in HEX. The compatibility between binary and HEX shows wherever a major carry occurs - take for instance 1111_2 , 15_{10} or F_{16} : one increment past this turns the binary ones to zeroes and adds a one to the left, i.e. to 10000_2 or 10_{16} . These major points of correspondence occur at:

Step 1

$1_2 = 1_{16} = 1_{10}$

$0001\ 0000_2 = 10_{16} = 16_{10}$

$0000\ 0001\ 0000\ 0000_2 = 100_{16} = 256_{10}$

$0001\ 0000\ 0000\ 0000_2 = 1000_{16} = 4096_{10}$

As you will have already noticed, the hexadecimal count is present on the DEC/BIN/HEX display and the progress of the count can be followed on this.

Press 'E' and enter the start value of '1' and again, start incrementing the count. Up to 9, the hex characters coincide with the decimal ones and between 10 and 15 the single letters correspond to the decimal numbers. After 15, Hex to decimal conversion becomes a little more tricky, as the use of two numbers together, e.g. FF_{16} =255, once again calls for place-value notation. This time, as the base is 16 the ratio between any place and its neighbour is 16.

The values, in base 10, of the places in hexadecimal are:

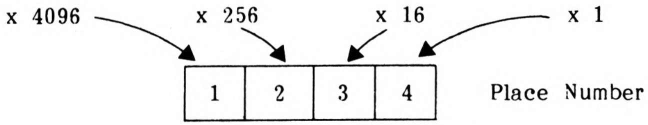


FIGURE A5.4

Using Figures A5.4 and A5.5, the breakdown of E92F in base 16, into 59695 in base 10, is illustated.

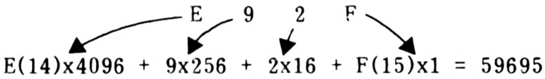


FIGURE A5.5

Now that hex is totally mastered (!) try the following; the first two are explained fully in the solutions chapter.

EXERCISE A5.3

Calculate the value in decimal of the following:-

i) 09_{16}
ii) 13_{16}
iii) $A5_{16}$
iv) AE_{16}

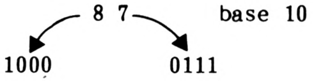
v) $\overset{01}{0A}_{16}$
vi) $1A_{16}$
vii) EA_{16}

Do the above exercise on paper first. The program will also accept numbers input in Hexadecimal but they must be preceeded by the prefix '&'. Thus if you entered the value '&FF' the decimal equilvalent, 255, would be displayed. You can check your answer to Exercise A5.3 using the program or look in the solutions chapter. Keep the program, you will need it in a moment.

Binary-Coded Decimal

As well as decimal, binary and hexadecimal notations, one other system is used in computing - Binary-Coded Decimal. As its name suggests, it is a hybrid form with elements from both binary and decimal. It is commonly used where an output is required in digital format, e.g. a digital clock, or when great precision is required and no bits can be dropped.

In BCD the normal decimal base is retained, i.e. one place is a factor of 10 times its neighbour but each individual digit is represented in binary. Thus the number 87_{10} would be represented:



i.e. BCD = 1000 0111 (or in eight bits
10000111)

FIGURE A5.6

As the largest digit required in decimal notation is 9, only four bits of binary are needed to represent this, i.e. $9_{10} = 1001_2$, thus a BCD digit can be represented by a nybble and two digits by a byte. Figure A5.6 shows this, where 87_{10} is represented in BCD as 10000111_2 . This can give rise to ambiguity in that 10000111_2 in binary is 135_{10} . To overcome this, BCD representations will be given the notation 10000111 (BCD).

Using four bits of binary, it is possible to count up to 15_{10} (i.e. $1111_2 = 15_{10}$) but in BCD the largest digit used is 9, so inevitably BCD is less economical in its use of space. Its largest digit, 9, is 1001_2 and when one is added to this it clocks over to 0000_2 and carries the 1 to the next nybble, i.e.

8_{10}	=	0000 1000	(base 2 BCD)
9_{10}	=	0000 1001	" " "
10_{10}	=	0001 0000	" " "
11_{10}	=	0001 0001	" " "

FIGURE A5.7

The display we are concerned with this time is the one marked BCD. Press 'E' and enter a start value of 1. If you increment through the first 9 counts you will see that up to 9_{10} , binary and BCD are identical. However, as you increment from 9_{10} to 10_{10} keep an eye on the BCD box and you will see the 1 carried over to the most significant nybble. From 10_{10} upwards BCD becomes a true hybrid representing the decimal number in a binary form.

As the number input increases, the uneconomical nature of BCD will become apparent. If you now enter a new start value of 95 and start incrementing you will see the problem as 99_{10} changes to 100_{10} When the count is made, however the accumulator is wrongly changed and contains incorrect values. The program deals with this by displaying the message 'TOO BIG FOR BCD' in the box. If you decrement back down to 99 or less the appropriate BCD value is again displayed but the carry flag remains set. (If you wish, you may reset both carry flags by pressing 'R'. However, the BCD flag will be set again as soon as a value over 99 is encountered). When incrementing from 99_{10} to 100_{10} the BCD generates a carry from its most significant nybble to the carry flag so the Z80 does not lose this but sets the flag.

As mentioned above, this carry is only a short-term expedient and must be picked up at the earliest possible moment if it is not to be lost. The carry is generated on the BCD boxes at 99_{10} while the binary boxes will store up to 256_2 and the decimal of course up to 999_{10} . BCD is less economical but has its uses in particular situations.

As you know all about BCD now, try the following:-

EXERCISE A5.4

Convert the following decimal numbers into BCD:

i) 4
ii) 10
iii) 77
iv) 97
v) 53
vi) 102
vii) 953
viii) 2579

The answers are in the solutions chapter.

EXERCISE A5.5

Convert the following BCD numbers into decimal:-

i) 0000 0001
ii) 0000 1001
iii) 0001 0101
iv) 0010 0000
v) 0100 1001
vi) 0010 0011
vii) 1001 0111
viii) 1000 1000

The answers are in the solutions chapter.

Unfortunately for you, the program does not accept numbers input in BCD notation. However, if you are desperate you could enter each nybble separately.

In the explanation given for the value of places in place-@value notation a simplification was adopted in order to make these explanations clearer for our less mathematically inclined brethren! However, if you wish to see a slightly more mathematical explanation, please read on. Otherwise - END OF APPENDIX 5.

With binary numbers, it was said that the places increase their value in multiples of 2, but the least significant bit of. the binary number was equivalent to the same symbol base 10 (or for that matter base 3, or whatever). In actual fact, the multiplying factor is the base, raised to the power of its place starting with zero at the left. i.e. in binary:

7	6	5	4	3	2	1	0	Place
128	64	32	16	8	4	2	1	Previously stated multiplication factor
2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	Mathematically more precise factor.

Thus the least significant bit is multiplied by 2^0 or 1. (If you are not sure of this try the direct program PRINT 2↑0.) The next bit is multiplied by 2^1, and so on.

This rule holds good for ANY base. Let's apply it for hex, i.e. base 16:

Least significant bit factor = 16^0 = 1

2nd most significant bit factor = 16^1 = 16

3rd most significant bit factor = 16^2 = 256

Most significant bit factor = 16^3 = 4096

APPENDIX 6

The assembler

This appendix explains all of the features of both the assembler and the editor. It is recommended that you read through it and refer to it when necessary throughout the book.

The following editor commands are displayed by the menu.

1 Insert text
2 List
3 Replace line
4 Delete text
5 Assemble
6 Exit to Basic
7 Call Program
8 Save file
9 Fetch file
10 Print file
11 Renumber
12 Convert Base
13 Arithmetic Operation

These commands will be detailed on the following pages.

1. The insert option

This command is used for entering text into the text file. It is possible to specify both the starting line number and the line increment in response to the prompt, "Enter start and increment". The first number will be taken as the starting line number, the second as the line increment. These two numbers must be separated by a dash "-" which is located below the "=" sign on the keyboard. For example the following sequence of commands would cause the assembler to assign a line number of 10 to the first line of text entered and set the second line number to 20 i.e. a line increment of 10. The editor's prompts will be shown in bold **thus.**

> I
> **Enter start and increment?** 10-10 <ENTER>

Note that you will not actually see the 'I' appear on the screen, it is shown here to denote that it was pressed, also <ENTER> will be used in this appendix to represent the pressing of the large blue ENTER key.

If no line increment is specified then 10 will be assumed. To exit the insert mode type in the '@' character as the first character on a line. This will cause the editor to return to the command mode shown by the '>' prompt. For example if the last line in your program was RET in line 120, then '@' would be entered as the first character in line 130.

The screen would appear as below:

```
120 RET <ENTER>
130 @ <ENTER>

Type M for the menu
>
```

2. The list option

This command allows the whole program or a segment of it to be listed to the screen. When selected the prompt "Enter starting line number" appears. To list the whole program just press the ENTER key. Once the listed program has filled a whole screen the listing process will stop until any key is pressed. If you require the listing to start at a particular line number then enter this line number in response to the prompt "Enter starting line number". At any point during the listing pressing the space bar will halt the listing, pressing any key will allow the listing to continue. Also, while listing, pressing the 'M' key will terminate the listing and print the menu on the screen.

3. The replace option

This option is used to replace an existing line with another. Suppose the following program was in the text file

```
10 LD A,65
20 IN C A
30 CALL &BB5A
40 RET
```

Line 20 should be INC A, and would be changed with the following commands.

```
> R
> Enter the line number 20 <ENTER>
20 INC A <ENTER>
```

Note it is not possible to replace more than one line at a time. If it is required to replace a series of lines use the insert option to insert the new text.

4. The delete option

This command enables either a single line or a whole block of text to be deleted from the text file. The syntax for the command is exactly the same as that for the Insert command.

For example:

```
> D
> Enter line number 20 <ENTER>
```

Would delete line 20, whereas:

```
> D
> Enter line number 20-200 <ENTER>
```

Would delete lines 20 to 200 inclusive.

To delete the whole program choose line numbers which are outside the used range. For example, suppose a program uses the line number range 10 to 320. The following sequence of commands would delete the whole program from memory

```
> D
> Enter line number 1-400 <ENTER>
```

5. The assemble option

After entering a program into the text file it is necessary to assemble this source code into machine code. This is accomplished with the assemble option.

After the assemble option has been selected, (by typing A while in the command mode), a menu will appear as below:

1. **No listing**
2. **Listing to screen.**
3. **Listing to printer.**
4. **Both screen and printer.**

1. No listing

   This option will cause no listing to be generated hence it will lead to the fastest assembly. However any errors will be reported.

2. Listing to the screen.

   This option causes the assembly listing to be output to the screen.

3. Listing to printer.

   This is exactly the same as option 2 except that the output is directed to the printer.

4. Both screen and printer.

   The assembly output is directed to both the screen and printer.

If during assembly the assembler encounters any errors it will jump back to the editor after informing you of the error.

6. The exit to basic option

This command jumps out of the assembler program and enters BASIC after clearing the screen with a MODE 1 command. The Assembler and Editor are written in a mixture of BASIC and machine code. The small BASIC 'driver' program uses the BASIC line numbers 64000 and above. Thus as long as the line numbers are kept below 64000 it is possible to have a small BASIC program resident in memory at the same time as the assembler. This is very useful for testing out algorithms before trying them in machine code. The maximum size of the additional BASIC program is about 4K. To re-enter the assembler, saving any program held in the text file, press the decimal point key on the numeric key pad or type in from BASIC 'GOTO 64026'. If you don't need to keep the contents of the text file, i.e. delete any assembly language program in memory, type in from BASIC 'GOTO 64018'.

7. The call program option

This option allows machine-code programs to be run or called from the assembler. To run a machine-code program, the memory location containing the first instruction in the program is called. Note: this option will only work if the program uses the ENT assembler directive as its first line; don't be tempted to use it if your program uses the ORG assembler directive as it may cause the computer to act strangely i.e: crash.

Thus assuming that ENT has been used pressing C while in the command mode will clear the creen and then jump to the machine code program.

To run a program which uses the ORG directive it is necessary to call the beginning of the program from BASIC. Thus suppose the ORG instruction in the program is as follows:-

```
10 ORG 30000
```

This will cause the assembler to store the first byte of the resultant machine code in memory location 30000. Thus to call the program exit to BASIC using X. Now type in the following:-

```
CALL 30000 <ENTER>
```

This will cause the machine-code program starting at memory location 30000 to be run. Ensure that the machine-code program is error free however as any mistake may cause the computer to crash.

As stated previously it is possible to save a copy of the machine code generated by the assembler (an object-code file), this type of file has to be loaded into the computer from BASIC. The load command is exactly the same as that used for loading a BASIC program except that the computer needs to be told where to locate the machine-code file. This information, the load address, is given after the file name, separated by a comma.

For example to load the object code created by the program TEST and saved under the file name TEST-b into memory starting at memory location 30000, the following command would be entered from BASIC

```
LOAD "TEST",30000 <ENTER>
```

Then to run the program

```
CALL 30000 <ENTER>
```

would be used.

The tape commands (S and F)

By using these commands text files can be saved to tape or fetched from tape in a form suitable for the assembler. It is also possible to save the machine code (object code) produced when a text file is assembled. This is very useful as it enables a machine code program to be developed and then run independently of the assembler.

8. The save option

This option is used to either save a copy of the text file to tape or to save a copy of the object code.

Saving the text file

Like BASIC programs, files are assigned a name which can consist of any alphanumeric characters. When the save option is selected you are asked to enter the file name you wish to use in response to the prompt "Enter the file name". Thus to save the contents of the text file under the file name TEST the following commands would be used

```
> S
> Enter the file name? TEST <ENTER>
```

The familiar "Press REC and PLAY then any key:" prompt will appear, at which point you should proceed as if in BASIC: insert a blank cassette, press PLAY and REC then any key. After the file has been saved the command-mode prompt '>' will reappear.

Saving the object code

It is also possible to save a copy of the object code generated by the last assembly of a text file. The object code is the machine code produced by the assembler. Thus an assembly language program can be developed using the assembler and then the resultant machine code saved to tape for use later. (Note: details as to how to run this object-code file are given in the section detailing the 'Call Program' option.)

To save a copy of the object code it is necessary to add a file type-marker to the file name. This marker is a 'b' separated from the file name by a dash. Before the object code can saved the program in the text file must be assembled. Thus to save the object code created by the imaginary file TEST the following file name would be used

```
> S
> Enter the file name? TEST-b <ENTER>
```

9. The fetch option

This option is used to fetch a text file from tape. Note: it cannot be used to reload a copy of the object code created by the assembler and saved with the identifier '-b'. See the 'Call Program' option for this. You are prompted with the "Enter file name" prompt. If you cannot remember the file name the program was saved under just press ENTER in response to the prompt and the first compatible file found on the tape will be loaded. For example the following would be used to fetch the file test

```
> F
> Enter the file name? TEST <ENTER>
Press PLAY then any key: <Press PLAY then ENTER>
```

10. The print option

This option allows the program held in the text file to be output to the printer, if one is present. The syntax for this command is exactly the same as that for the list command, except that P is entered instead of L.

11. The renumber option

This command renumbers all the lines stored in the text file. When selected the prompt "Enter the new line increment" appears; the number entered then will be used as the starting line number and the line increment. Thus to renumber the text file with a line increment of 10 starting with a line number of 10, the following sequence of commands would be used

```
> N
> Enter the new line increment 10 <ENTER>
```

12. The convert base option

While writing assembly language programs it is sometimes useful to be able to convert a number from one base to another. The Convert Base Option allows this, it will accept any number in the range 0-65536. The number to convert can be represented in any one of the three following bases:

1. Decimal.
2. Hexadecimal.
3. Binary.

To distinguish between them, the standard BASIC prefixes are used:

Base	Prefix
Decimal	None
Hexadecimal	&
Binary	&X

The input number is converted to all three bases. Thus to express 275 in decimal, hex and binary the following sequence of commands would be used.

```
> B

Enter number to convert? 275 <ENTER>
```

The screen will then display the number 275 in all three bases. To clear the screen press B again. Note the lower screen display will remain unaffected by all commands except B and O which will clear it.

13. The arithmetic operation option

This command allows two numbers, in any one of the three supported bases, to be either added or subtracted from each other. For example to add 200 to &FF.

```
> 0

First number? 200 <ENTER>
Second number? &FF <ENTER>
A. Add  S. Subtract? A <ENTER>
```

The lower screen will now clear and display the following:

```
Binary = &X11100011
Decimal = 445
Hexadecimal = &1C7
```

As with the Convert Base option, this display will remain until O or B is pressed again.

Assembler directives

As well as accepting the full Z80 instruction set the assembler will accept certain assembler directives. An assembler directive is an instruction to the assembler instead of the Z80 (they are sometimes called pseudo-opcodes for this reason).

Assembly directives considerably increase the legibility and ease of writing programs.

The first directive should be familiar:

ENT Causes the object code to be placed immediately after the source code, in a form suitable for running from the assembler's call program option.

When writing programs that require more storage for data than is available from the registers or easily from the stack, it is necessary to use memory locations to store this data. Ideally these memory locations would be situated in the same vicinity as the program. As the exact length of the program is not known calculating the address of a memory location situated at the end of the program is not easy. The solution to the problem is to use the following assembler directive

DEFS n Reserves n bytes of memory starting at the current value of the location counter.

To allow easy reference to this storage area a label is assigned to this directive. For example to sent up a four byte buffer referenced by the label STORE: the following line would be used

```
STORE:   DEFS 4
```

When the assembler encounters this command it will reserve four bytes of memory and assign to the label 'STORE:' the address of the first byte in this buffer.

For example:

Suppose that an 8 by 8 bit multiplication program produces a 16 bit result in HL; if this result will be needed further on in the program it will have to be kept.

By using the following program the result can be obtained by referencing the label STORE:

```
        ENT
STORE:  DEFS 2
        .
        .
        ..
Multiplication program
        .
        .
        LD (STORE:),HL
        .
        .
```

Thus the contents of the memory location addressed by store contain the result of the multiplication. As well as creating buffers, assembler directives can be used for many other functions. The first to be considered is assigning numeric values to labels.

EQU nn	Assigns the value nn to the preceding label.

This form of directive is used mainly for setting up constants.

For example:

```
        LOOP:   EQU 20
                LD  A,LOOP:
```

When assembled and executed, this would load A with 20.

So far most of the programs have used the ENT directive. This is fine for development and testing purposes but when a machine-code program is to be called from BASIC it has to be stored where BASIC cannot overwrite it. The amount of memory available to BASIC is set by the BASIC command MEMORY. By setting MEMORY to 39999 any memory location above 39999 cannot be used for BASIC program storage: thus, in effect, memory has been reserved for machine-code programs.

Instead of letting the assembler decide where to locate the program it needs to be told. The following directive is used to accomplish this

ORG nn	Set the location counter to nn

This feature of the asembler allows programs to be assembled so as to reside anywhere in memory. Note: the value nn has to be greater than the last memory location used by the editor. If you type in an invalid location the assembler will inform you.

Many programs require specific memory locations to be loaded with data. There are three directives to accomplish this; the first two load numeric data and are as follows

DEFB n	Stores n at the current value of the location counter.

DEFW nn	Stores the LSB of the 2 byte word nn at the current value of the location counter and the MSB at the current value of the loction counter + 1.

The last directive stores the ASCII representation of a string in memory. This is very useful for printing messages on the screen.

DEFM "s"	Set the contents of memory, starting with the current value of the location counter, to the ASCII representation of string "s".

Try the following program:

```
        ENT
MESS:   DEFM "A MESSAGE !"
        LD B,11
        LD HL,MESS:
LOOP:   LD A,(HL)
        CALL &BB5A
        INC HL
        DJNZ LOOP:
        RET
```

Useful heh!

Addition of operands & direct loading of ASCII characters

If by now you have worked through the book and have read chapter 10, it may have struck you that in Chapter 10 &80 was added to an ASCII character held in quotes in a DEFB command. The effect of this line is to load into memory, at the current value of the location counter, the ASCII value of the character contained in the quotes after &80 has been added to it. The effect of adding &80 to a binary number is to set bit 7. This was required in the program as the Amstrad's operating system requires the last character in an external command's name to have bit 7 set.

Thus to load the ASCII representation of A into the Accumulator it is possible to use the following line:

```
        LD A,"A"
```

which would load A with 65. Likewise, to load A with B you could use:

```
        LD A,"A"+1
```

These functions are very useful at times, especially for setting up arrays of data.

Comments

The assembler will allow comment lines to be included in the text file provided that the first character in that line is a semi-colon. Thus a sample program containing comments would appear as below:

```
10         ENT
20 ;Load A with 65
30         LD A,65
40 ;Print A on the screen
50         CALL &BB5A
60         RET
```

Note that comments must be on separate lines from instructions; the two cannot be combined on one line.

GLOSSARY

ADDRESS

The computer has lots of chips in it, in which to remember data. It stores up data in little groups of 8 bits, called bytes. The computer has lots of these bytes in its memory, we call each of the places that holds a byte a memory location. Each memory location has a special number associated with it, just like a house number, this number is called the address of that memory location. It helps the computer find the byte it wants.

ALGORITHM

An algorithm is a set of instructions for doing a particular job. Cooking recipes are algorithms and so are flow charts and computer programs.

ARCHITECTURE

The arrangement or design of the logic in a computer system.

ARITHMETIC AND LOGIC UNIT or ALU

The area in the central processor that does all the arithmetic and logical operations.

ASCII

An abbreviation for American Standard Code for Information Interchange. In the computer, information is stored in the form of binary numbers so, in order to represent a letter in the computer, it was decided that each letter of the alphabet should be given a special number or code. This code is called the ASCII code of the letter. Not all the ASCII codes represent letters, some represent other characters like =,+,!,? or /. Also there are other codes that represent new lines etc. Many computers do not strictly conform to the ASCII standard but most conform to a large extent.

ASSEMBLER

A program which translates assembly language into machine code.

ASSEMBLER DIRECTIVE

These commands tell the assembler something, e.g. ORG. This instructs the assembler to start storing the object code at the given address. See also Assembler.

ASSEMBLY LANGUAGE

A 'low level' computer language, but not as 'low level' as machine code. Assembly language is a language designed to make it easier to enter machine code programs into a computer. A program called an 'assembler' translates assembly language programs ('source code') directly into machine code ('object code').

BASIC

Beginners All Purpose Symbolic Instruction code. A 'high level' programming language.

BIT

One digit, either 0 or 1, in a binary number.

BYTE

A group of 8 bits used to represent a number between 0 and 255.

CENTRAL PROCESSOR UNIT

The part (in this case the Z80 chip) of the computer that does all the executing of machine code programs (every program 'boils down' to machine code of one form or another!).

COMMANDS

The near English equivalents of machine code. We actually write commands when entering assembly language. Thus, ADD A,B is a command; 80 is the machine code equivalent. See also 'mnemonics'. Another word used for command is 'instruction'.

COMPILE

To create the object programs from the source.

COMPILER

A program which converts a whole program written in a 'high level' language such as BASIC into machine code before the program is run. Speeds comparable with assembly language/machine code programs can be achieved. See also 'Assembler', 'Disassembler' and 'Interpreter'.

COMPLEMENT

A 'reverse' form of a binary number in which all the 1's are swapped for 0's and all the 0's for 1's, i.e.

10111100 becomes 01000011

CONTROL CHARACTERS

Control charaters do not display anything but serve to perform some action such as 'new line' <RETURN> or clear screen.

CRASH

If the Z80 is told to execute a program in memory that is incomplete in some respect then it may end up executing what it should not be executing! The Z80 is not very clever and can easily attempt to execute the contents of a buffer, string variable or any other kind of data. When this happens the Z80 gets in a mess, jumping around, executing a bit here and a bit there. It may store things all over the place, generally making a mess of what you have in memory. You will need to reset the computer to bring the Z80 to its senses. (A system crash will not do physical damage to the system. Crash the system for fun if you want!)

CRYSTAL OSCILLATOR

Device based on the vibration of a quartz crystal to produce an oscillating electrical voltage. Used in electronic watches, hence quartz watch, also on timing devices (clocks) for some computer systems.

DATA BUS

A group of electrical pathways used by a computer system to communicate internally.

DOUBLE PRECISION

A rather general term meaning twice as precise. Used in this book to mean mathematics done on numbers 2 bytes long (i.e. ranging from 0 to 65535) rather than one byte long.

DISASSEMBLER

A program which translates machine code into assembly language.

EXECUTE

Used, in the context of this book, to mean the running of a machine-code program.

FLAG

A special location, usually just one bit, that is set to a particular value if some condition was true or to some other value if it was not. It could be altered on the result say, of a compare operation between two numbers, if they were equal it would be set to 1 and if they were not it would be set to 0.

GATE

Generally, an electronic device for performing logical operation. Voltage levels are used to indicate the values TRUE and FALSE, or 1 and 0. (Pneumatic logic gates also exist, but not in this computer!).

INDIRECTION

The process of looking up the address to use in an instruction, when the address is not actually given in the instruction: The instruction contains the address at which the address to use is stored.

INSTRUCTIONS

The near English equivalents of machine code. We actually write instructions when entering assembly language. Thus, ADD A,B is an instruction; 80 is the machine code equivalent. See also 'mnemonics'. Another word used for instruction is 'command'.

INTERFACE

A device to allow two electronic systems, not normally compatable directly, to communicate with each other.

I/0

Abbreviation for Input/Output

INTERPRETER

A program which reads another program (eg. a BASIC program) and converts it instruction by instruction into machine code which is then immediately executed. Note that this is different from an assembler or a compiler which convert whole programs (the 'source code'). Programs in an 'interpreted' language such as BASIC run more slowly than assembly language and compiled programs as each BASIC instruction has to be interpreted every time it is executed, whereas, say, an assembly language program only has to be converted into machine code once before the program is run. See also 'Compiler'.

LOAD

To load a register or the accumulator means to put some number into it. The number can be obtained in various ways.

LOGICAL OPERATOR

Operator or function that only works with the values TRUE and FALSE. The most basic logical operators are AND, OR and NOT. NOT just returns the complement (opposite) of its input.

LOW and HIGH LEVEL LANGUAGES

The 'level' of a computer language refers roughly to its similarity to English or other spoken languages. The higher the level, the closer the language is to a proper spoken language. The lower the level the closer the language is to the inscrutable numbers of machine code. High level languages are characterised by simple commands which can do a lot - i.e. by 'powerful' commands. In approximate order from low to high a selction of computer languages listed here: machine code, assembly language, FORTRAN, BASIC, ALGOL, PASCAL, COBOL, LISP. Some languages such as FORTH, whilst of quite high level, have many features of both low and high level languages and it is difficult to say exactly how high a level they are.

MACHINE CODE

This is the actual numbers that the Z80 microprocessor chip uses. Not merely data, but every instruction which the chip executes is represented as far as the chip is concerned by a machine-code number, and these, controlling internal voltages, are all it can use.

MEMORY LOCATION

The computer has lots of chips in it, in which to remember data. It stores up data in little groups of 8 bits, called bytes. The computer has lots of these bytes in its memory, we call each of the places that holds a byte a memory location. Each memory location has a special number associated with it, just like a house number, this number is called the address of that memory location. It helps the computer find the byte it wants.

MEMORY MAP

A chart showing how the memory is used by the computer, i.e. which memory locations are used for what.

MNEMONICS

Pronounced 'nemoniks', and named after the Greek goddess of memory, Mnemosyne, mnemonics are simply memory aids. It is easier to understand or remember what 'ADD A,B' does than to remember what the Z80 chip will do with the instruction '80'. Hence we tend to use assembly language with its mnemonics rather than enter programs directly in machine code.

NESTING

A structure in a computer program where one loop is put inside another.

NYBBLE

A nybble is a chunk of 4 bits just as a byte is a chunk of 8 bits. For convenience a byte is sometimes considered to consist of two nybbles, the least significant nybble (right-hand one) and the most significant nybble (left-hand one).

OBJECT CODE

The machine code version of an assembly language program. The assembler is said to convert 'source code' into 'object code'.

OBJECT PROGRAM

The object program is the actual machine-code version of an assembly-language program (the source program).

OP CODE

An instruction, usually written in assembly language or in a higher level language, which the computer can follow: eg. ADD, LD, PRINT, etc. Note that the op code is the instruction minus its operands. Thus ADD A,B is an instruction, whilst ADD is the op code.

OPERATING SYSTEM

A built in program which is designed to simplify 'housekeeping' procedures within the computer. It deals with such functions as scanning the keyboard, creating the video display, saving programs to tape, etc.

OPERAND

The operand of an instruction is any number which follows that instruction, such as A and 145 in LD A,145. Some instructions don't need an operand such as RET.

PARAMETER

Most instructions have parameters. For example in the BASIC line 'PRINT a,b'; a and b are the parameters.

PERIPHERAL

A device attached to the main computer system, such as a printer, VDU or light pen.

PIXEL

Short for 'picture element', a pixel is the smallest portion of the screen which the computer can control - i.e. a dot. All pictures, letters etc. which the computer puts on the screen are made up from combinations of pixels.

PROGRAM COUNTER

A special 16 bit register, in the Z80, that points to the next instruction to be executed, so that the Z80 does not lose its place in a program!

RAM

Abbreviation for <R>ead <A>nd <M>odify memory, also sometimes known as <R>andom <A>ccess <M>emory.

REGISTER

A special memory location in a chip, usually in the Central Processor Unit, (CPU) for storing a number, in binary. The registers in the Z80 are either 8 or 16 bits long.

RESET

A flag or a bit is 'reset' when it has the value '0'. A pixel is reset when it is not lit up.

ROM

Abbreviation for <R>ead <O>nly <M>emory. This is memory in which there is a program permantly stored, and cannot be erased by trying to POKE data into it or by means of LD type instructions.

SCREEN (Addresses)

The screen has a special area of the computer's memory reserved for it. This memory area tells the system what information goes where on the screen. If we alter the contents of these locations then the displayed character on the screen will change.

SERVICE ROUTINE

Machine-code routine used for servicing interrupts.

SET

Generally set is used to refer to flags, it means to put a 1 into the flag. A pixel is set when it is lit up.

SOURCE CODE

Assembly language as typed in, and before converion to machine code or 'object code' by the assembler.

SOURCE PROGRAM

The source program consists of the actual assembly mnemonics, as used in this book. It means a program that was created by the user but must first be translated into some other form before execution can take place. An assembly-language program must be changed into machine code in order to execute it.

SYMBOLIC LABEL

This is a name given to a line in a program, so that it can be referred to easily by the programmer. Instead of jumping to a specific memory location within a program, it can jump to the name instead.

VECTOR ADDRESS

The address that must be 'looked up' in an indirection operation is called the vector address.

Z80

Z80 is the type or reference number of the central processing unit CPU used in this computer. The CPU is often referred to as 'THE Z80'.

SOLUTIONS

CHAPTER 1

EXERCISE 1.1

```
ENT
LD A,65
CALL 47962
RET
```

EXERCISE 1.2

```
ENT
LD A,70
CALL 47962
LD A,82
CALL 47962
LD A,69
CALL 47962
LD A,68
CALL 47962
RET
```

CHAPTER 2

EXERCISE 2.1

```
ORG 30000
LD B,10
LD A,65
CALL 47962
DEC B
JR NZ,30004
RET
```

EXERCISE 2.2

The JR NZ instruction is used in preference to the JP NZ instruction because the jump distance is within +129 and -126 bytes, and the JR NZ instruction requires less time to execute. It is therefore more efficient.

EXERCISE 2.3

```
        ORG 30000
        LD C,26
        LD A,65
NXT:    CALL 47962
        INC A
        DEC C
        JR NZ,NXT:
        RET
```

CHAPTER 3

EXERCISE 3.1

Memory location	Contents
:	:
:	:
200	E
201	D
202	L
203	H
:	:
:	:

EXERCISE 3.2

```
        ENT
        LD DE,100
        LD (35000),DE
        LD HL,400
        LD (35002),HL
        LD DE,(35000)
        LD HL,(35002)
        CALL 48118
        RET
```

If you use the machine-code plot routine at 48106, the program is instead:

```
ENT
LD DE,100
LD (35000),DE
LD HL,400
LD (35002),HL
LD DE,0
LD HL,0
CALL 48106
LD DE,(35000)
LD HL,(35002)
CALL 48118
RET
```

EXERCISE 3.3

```
ENT
LD DE,200
LD HL,300
CALL 48118
LD DE,400
LD HL,200
CALL 48118
LD DE,0
LD HL,0
CALL 48118
RET
```

Note: There is no real need, apart from illustration, to load the coordinates into memory. This program is quicker than a program which stores the coordinates in memory then reads them out again.

EXERCISE 3.4

```
        ENT
        LD BC,35000
        LD A,65
        LD E,3
NXT:    LD (BC),A
        INC BC
        DEC E
        JR NZ,NXT:
        LD E,3                  No. of times to loop
        LD BC,35000             Load start location of 'A's
PUT:    LD A,(BC)               Put 'A' in accumulator
        INC BC                  Add 1 to data location
        CALL 47962              Put 'A' on screen
        DEC E                   Decrement loop number
        JR NZ,PUT:              If loop number <> 0, loop again
        RET                     END
```

EXERCISE 3.5

```
        ENT
        LD BC,35000
        LD A,65
        LD E,26                 26 loops required
NXT:    LD (BC),A
        INC BC
        INC A                   Next ASCII letter
        DEC E
        JR NZ,NXT:
        LD BC,35000
        LD E,26                 26 loops required
PUT:    LD A,(BC)
        INC BC
        CALL 47962
        DEC E
        JR NZ,PUT:
        RET
```

EXERCISE 3.6

```
ENT
LD IX,35000
LD A,83
LD (IX+0),A
LD (IX+2),A
INC A
INC A
LD (IX+1),A
LD A,65
LD (IX+3),A
LD A,78
LD (IX+4),A
LD A,(IX+2)
CALL 47962
LD A,(IX+3)
CALL 47962
RET
```

CHAPTER 4

EXERCISE 4.1

```
ENT
LD A,200
ADD A,48
CALL  47962
RET
```

This puts a little man on the screen.

EXERCISE 4.2

```
ENT
LD A,&41
ADD A,&10
CALL &BB5A
RET
```

This places a 'Q' on the screen.

EXERCISE 4.3

```
        ENT
        LD C,&FA            LSB of 250
        LD A,&58            LSB of 600
        AND A
        ADD A,C
        ADD A,65
        LD (&7000),A
        LD C,&00            MSB of 250
        LD A,&2             MSB of 600
        ADC A,C
        ADD A,65
        LD (&7001),A
        LD A,(&7000)
        CALL &BB5A
        LD A,(&7001)
        CALL &BB5A
        RET
```

This produces a shape like a robot's gripping arm.

EXERCISE 4.4

```
        ENT
        LD C,9
        LD A,233
        SUB C
        CALL &BB5A
        RET
```

It is not necessary to subtract 65 from the answer because CHR$(224) is a printable character (a smiling face).

EXERCISE 4.5

```
        ENT
        LD A,126
        ADD A,97
        LD C,153
        SUB C
        CALL &BB5A
        RET
```

This puts 'F' on the screen.

EXERCISE 4.6

```
        ENT
        LD DE,4008
        LD HL,4248
        AND A
        SBC HL,DE
        LD A,L
        CALL &BB5A
        RET
```

This puts an up-arrow on the screen.

EXERCISE 4.7

```
        ENT
        LD HL,35000
        LD A,10
        LD (HL),A
        INC HL
        LD A,20
        LD (HL),A
        LD A,65
        ADD A,(HL)
        LD (HL),A
        DEC HL
        LD A,65
        ADD A,(HL)
        LD (HL),A
        CALL &BB5A
        INC HL
        LD A,(HL)
        CALL &BB5A
        RET
```

This puts 'KU' on the screen.

EXERCISE 4.8

```
ENT
LD DE,100
LD HL,50
CALL  48118
LD D,0
LD E,100
LD A,75
ADD A,E
LD E,A
LD H,0
LD L,50
LD A,75
ADD A,L
LD L,A
CALL 48118
RET
```

CHAPTER 5

EXERCISE 5.1

```
LD A,&7
ADD A,&C
DAA
ADD A,65
CALL &BB5A
RET
```

This will put a 'Z' on the screen.

EXERCISE 5.2

```
LD C,&12
LD A,&35
SUB C
DAA
ADD A,65
CALL &BB5A
RET
```

This will put a 'd' on the screen.

EXERCISE 5.3

1. C=0
2. C=1
3. C=0

EXERCISE 5.4

The required mask would be 00000011.

EXERCISE 5.5

253 AND 75 = 73 (1001001)

EXERCISE 5.6

1. 1001 OR 1101 = 1101
2. 250 OR 25 = 251
3. (209 OR 20) AND 27 = 17

EXERCISE 5.7

1. 1101 XOR 1110100 = 127

```
LD C,11
LD A,116
XOR C
CALL &BB5A
RET
```

This prints a chessboard on the screen.

2. 77 XOR 200 = 133

```
LD C,77
LD A,200
XOR C
CALL &BB5A
RET
```

This prints a vertical bar on the screen.

3. (25 OR 255) AND 200 = 200

```
LD C,25
LD A,200
OR C
LD C,200
AND C
CALL &BB5A
RET
```

This prints two diagonal lines on the screen.

EXERCISE 5.8

1. 0100
2. 0100010
3. 0001

EXERCISE 5.9

1. $1010 \qquad = 10$

   $\underline{1101} \qquad = \underline{-3}$

   $\underline{\underline{0111}} \qquad \underline{\underline{7}}$

2. $1111 \qquad = -1$

   $\underline{0111} \qquad = \underline{7}$

   $\underline{\underline{0110}} \qquad \underline{\underline{6}}$

3. $0110 \qquad = -10$

   $\underline{1000} \qquad = \underline{8}$

   $\underline{\underline{1110}} \qquad \underline{\underline{-2}}$

EXERCISE 5.10

```
LD A,20
CPL
INC A
ADD A,98
CALL &BB5A
RET
```

This prints an 'N' on the screen.

CHAPTER 6

EXERCISE 6.1

The solution is the same as Program 6.1 except for these lines:

```
        LD C,10
        LD E,9
```

The program should put character 155 on the screen, i.e. an upside-down short-tailed T.

Exercise 6.2

```
        LD C,124
        LD E,146
        LD D,0
        LD B,8
        LD HL,0
NXTB:   SRL C
        JR NC,NOADD:
        ADD HL,DE
NOADD:  SLA E
        RL D
        DEC B
        JR NZ,NXTB:
        LD A,H
        CALL &BB5A
        LD A,L
        CALL &BB5A
        RET
```

This gives πF.

EXERCISE 6.3

The easy way to do this is to delete DEC B and replace the line JR NZ,NXTB: with:-

```
        DJNZ NXTB:
```

EXERCISE 6.4

```
1.      LD A,5
        RLCA
        RLCA
        RLCA
        RLCA
        RLCA
        CALL &BB5A
        RET
```

This prints an up arrow head on the screen; the ASCII for 160.

```
2.      LD A,254
        RRCA
        CALL &BB5A
        RET
```

This prints CHR$(127); a chessboard, on the screen.

EXERCISE 6.5

```
        LD A,255
        CALL &BB5A
        RES 4,A
        CALL &BB5A
        SET 4,A
        RES 3,A
        CALL &BB5A
        RET
```

This prints a double headed arrow, a rocket and a pyramid on its side, on the screen.

CHAPTER 7

EXERCISE 7.1

```
        ENT
        LD DE,0
        LD HL,0              Set graphics cursor to (0,0)
        CALL &BBC0
        LD DE,100            Load DE with 100
        LD HL,200            Load HL with 200
        PUSH DE              Save DE on stack
        PUSH HL              Save HL on stack
        CALL &BBF6           Draw line to (100,200)
        LD DE,0
        LD HL,0              Set cursor to (0,0)
        CALL &BBC0
        POP DE               Put old contents of HL in DE
        POP HL               Put old contents of DE in HL
        CALL &BBF6           Draw line to (200,100)
        RET
```

EXERCISE 7.2

```
        ENT
        LD A,43              Put '+' in A
        PUSH AF              Store A on current stack
        CALL &BB5A           Put '+' on screen
        LD (&7148),SP        Remember current value of SP
        LD HL,&7148          Get ready to:
        LD SP,HL             Set up new stack at 30000
        LD A,61              Put '=' in A
        PUSH AF              Store A in new stack
        CALL &BB5A           Put '=' on screen
        LD HL,(&714A)        Find original location of SP
        LD SP,HL             Go back to original stack
        POP AF               Retrieve '+'

        PUSH AF
        CALL &BB5A
        LD HL,&7146          Because A and F are in the new stack
        LD SP,HL
        POP AF
        CALL &BB5A
        LD HL,(&714A)
        LD SP,HL
        POP AF
        CALL &BB5A
        RET
```

CHAPTER 8

EXERCISE 8.1

```
        LD HL,&B100
        LD DE,&C000
        LD BC,&3FFF
LOOP:   LDI
        JP PO,FINISH:
        JP LOOP:
FINISH: RET
```

APPENDIX 5

EXERCISE A5.1

i)	3
ii)	4
iii)	128
iv)	131
v)	183
vi)	115

EXERCISE A5.2

i)	31
ii)	41
iii)	189
iv)	136

EXERCISE A5.3

i)	9
ii)	19
iii)	165
iv)	174
v)	14
vi)	26
vii)	234

EXERCISE A5.4

i) 0000 0100

ii) 0001 0000

iii) 0111 0111

iv) 1001 0111

v) 0101 0011

vi) Too big for two byte BCD representation

vii) Too big for two byte BCD representation

viii) Too big for two byte BCD representation

EXERCISE A5.5

i) 1

ii) 9

iii) 15

iv) 20

v) 49

vi) 23

vii) 97

viii) 88

INDEX

AMSTRAD

ASSEMBLY LANGUAGE COURSE

The Book

This step-by-step text introduces the complete beginner to Z80 programming in the now well proven style that has been described by the critics as "worth its weight in gold". No prior knowledge is assumed and the aim throughout the book is to ensure that the beginner really succeeds. By the end of the book every Z80 class of instruction has been explained in detail. Numerous examples illustrate the points while exercises (along with solutions) test the understanding. Later chapters show how additional commands may be added to BASIC including, for example, a circle drawing routine.

The Associated Software

The software available to accompany this book includes a complete Z80 assember with:

- Symbolic Labels
- Assembler Directives
- Save/Load
- Hard-copy
- Insert/Delete

The assembler allows programs to be written easily in assembly language and these it translates into machine code.

To help understand the mathematical notations used, a binary hexadecimal tutor is included.

Also included is a program demonstrating the use of the additional graphics commands described in the book.

GLENTOP
PUBLISHERS ■ LIMITED

Glentop Publishers Ltd
Standfast House
Bath Place
High Street Barnet
Herts EN5 5XE
Tel:01 441 4130

ISBN 1 85181 112 5

DR WATSON series

AMSTRAD ASSEMBLY LANGUAGE COURSE

HERBERTSON

AMSTRAD

ASSEMBLY LANGUAGE COURSE

Contains a complete course with text and software.

The Book

This step-by-step text introduces the complete beginner to Z80 programming in the now well proven style that has been described by the critics as "worth its weight in gold". No prior knowledge is assumed and the aim throughout the book is to ensure that the beginner really succeeds. By the end of the book every Z80 class of instruction has been explained in detail. Numerous examples illustrate the points while exercises (along with solutions) test the understanding. Later chapters show how additional commands may be added to BASIC including, for example, a circle drawing routine.

The Software

The complete Z80 assembler which is included on tape includes:

- Symbolic Labels
- Assembler Directives
- Save/Load
- Hard-copy
- Insert/Delete

The assembler allows programs to be written easily in assembly language and these it translates into machine code.

To help understand the mathematical notations used, a binary hexadecimal tutor is included.

Also included is a program demonstrated use of additional graphic commands described in the book.

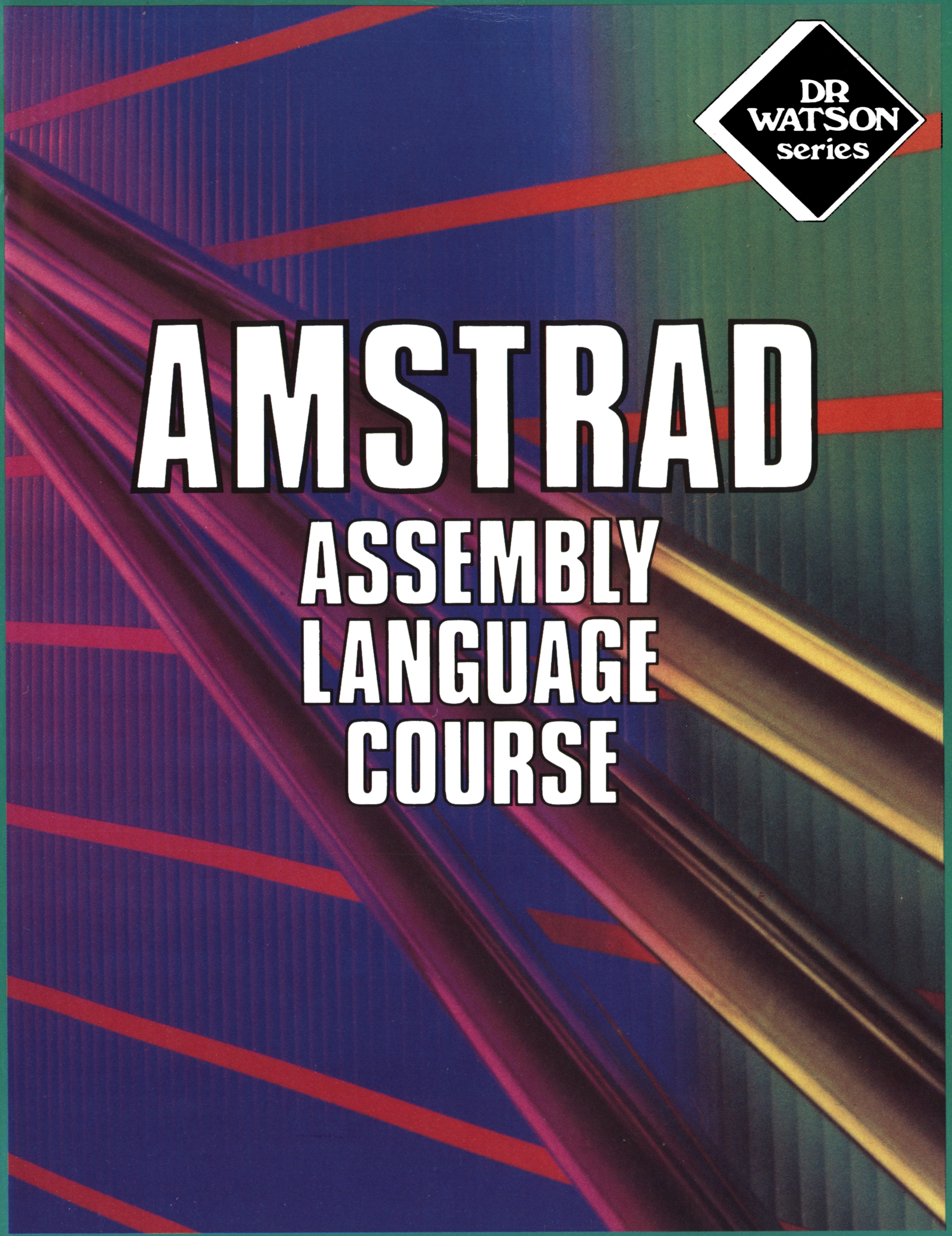
HONEYFOLD

HONEYFOLD SOFTWARE LIMITED
STANDFAST HOUSE, BATH PLACE, BARNET, LONDON.

DR WATSON series

AMSTRAD ASSEMBLY LANGUAGE COURSE

HONEYFOLD



DR WATSON series

AMSTRAD ASSEMBLY LANGUAGE COURSE

© GLENTOP
Publishers Ltd.
1986

AMSTRAD CPC
ASSEMBLER

A

GLENTOP

© GLENTOP
Publishers Ltd.
1986

AMSTRAD CPC BIN/HEX,
GRAPHICS DEMO
See Ch 10 for GRAPHICS-EXT

AMSTRAD
CPC

OCR

MÉMOIRE ÉCRITE
MEMORY ENGRAVED
MEMORIA ESCRITA

ACME https://acpc.me/

[FRA] Ce document a été préservé numériquement à des fins éducatives et d'études, et non commerciales.
[ENG] This document has been digitally preserved for educational and study purposes, not for commercial purposes.
[ESP] Este documento se ha conservado digitalmente con fines educativos y de estudio, no con fines comerciales.