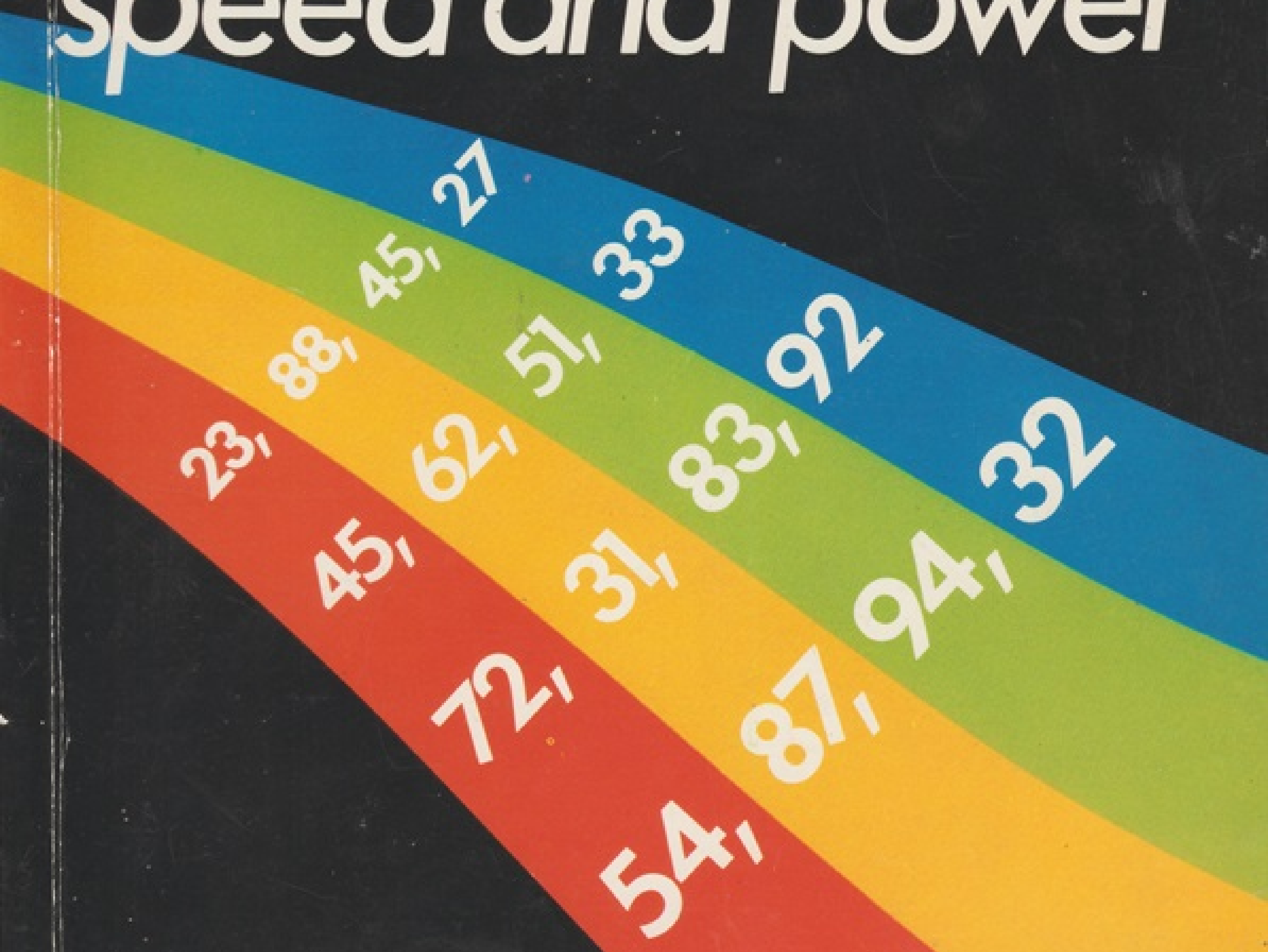
INTRODUCING SPECTRUM MACHINE CODE

How to get more speed and power



IAN SINCLAIR

Introducing Spectrum Machine Code

Introducing Spectrum Machine Code

Ian Sinclair

GRANADA
London Toronto Sydney New York

Granada Publishing Limited – Technical Books Division
Frogmore, St Albans, Herts AL2 2NF
and
36 Golden Square, London W1R 4AH
515 Madison Avenue, New York, NY 10022, USA
117 York Street, Sydney, NSW 2000, Australia
100 Skyway Avenue, Rexdale, Ontario, Canada M9W 3A6
61 Beach Road, Auckland, New Zealand

British Library Cataloguing in Publication Data
Sinclair, Ian
Introducing Spectrum machine code.
1. Sinclair ZX Spectrum (Computer)—Programming
2. Assembly language
I. Title
001.64'25 QA76.8.S62/

ISBN 0-246-12082-7

First Published in Great Britain 1983 by Granada Publishing Ltd

Typeset by V & M Graphics Ltd, Aylesbury, Bucks
Printed in Great Britain by Mackays of Chatham, Kent

Granada ®
Granada Publishing ®

Contents

Preface

Many computer users are content to program in BASIC for all of their computing lives. A large number of others are eager to find out more about computing and their computer than the use of BASIC can provide them. Few, however, seem to make much progress to the use of machine code, and I think this is so because so many books which deal with machine code seem to assume that the reader is already familiar with the ideas and jargon words of machine code. Also, these books tend to treat machine code as a study in itself, leaving the reader with little clue to the application of machine code to his or her own computer.

This book has two main aims. One is to introduce the Spectrum owner to some of the details of how the Spectrum works, so allowing for more effective programming even without delving into machine code. The second aim is to introduce the speed and power of machine code by means of simple examples. I must emphasise the word 'introduce'. No single book can tell all about machine code, and all I can claim is to give you, the reader, enough information to get started. Getting started means being able to write short machine code routines, understand such routines printed in magazines, and generally make more effective use of your Spectrum. It also means that you will be able to make effective use of books on machine code programming such as those which are listed in Appendix A – these are the books which are your entry to much more advanced work. From there, complete mastery of machine code programming is just a short step.

Together, understanding the operating system of the computer and having the ability to work in machine code can open up an entirely new world of computing to you. Understanding the operating system allows you to do things like renumbering program lines, changing PRINT instructions to LPRINT with one command, altering the key-press BEEP, or printing out a list of all variables. Writing machine code allows you also to take complete control over the computer system so that you can carry out tasks like reading ZX-81 tapes, driving serial printers from the cassette port, speeding up actions like denary/hex conversion or screen graphics. I must emphasise that this book does not consist of programs – it consists of explanations, because it is only by 'doing your own thing' that you will ever learn effective machine code programming.

No workman can operate without good tools. The tools that I have used are the Spectrum itself, a Trophy CR100 cassette recorder, a Philips 14CT3005 television receiver and the ZX printer. The software tools were the 'dpas' Disassembler from Campbell Software, which allowed me to investigate the Spectrum operating system, and, latterly, the ULTRAVIOLET Assembler from ACS Software, which allows a much faster and easier coding of programs written in the intermediate assembly language.

Finally I must thank some of the many people who made this book possible – Richard Miles of Granada Publishing for encouragement, Bill Nichols and Jane Boothroyd of Sinclair Research for the loan of a 16K Spectrum, my wife for tolerance, and Miss Leake for hospitality. Add to that list the names of Hector Berlioz and Jean Sibelius for serenity, and you have the whole of this writer's back-up force. Perhaps I should also mention that I have no connections with Clive Sinclair nor with Sinclair Research. The operating system of the Spectrum is a matter of copyright, and the addresses in the system which are not mentioned in the manual but which are supplied in this book were not supplied by Sinclair Research nor approved by them.

Ian Sinclair

Chapter One
ROM, RAM, Bytes and Bits

One of the discouraging things about digging below the surface of BASIC is the number of jargon words that you encounter. The writers of many books on computing seem to assume that the reader has an electronics background, so that it's not surprising that readers with such a background slip into the jargon quite easily. I shall assume that you, the reader, have no such background knowledge, and that all I can ask you to call upon is some experience of computing in BASIC with a Spectrum, preferably by writing your own BASIC programs. We shall start at the correct place, at the beginning. Since I don't want to interrupt explanations by having to include mathematical or other technical details, I have referred to these, where relevant, in the Appendices, so that you can take them or leave them according to your feelings about them.

In the beginning, then, there is memory. A unit of memory, as far as we are concerned, is just an electrical circuit that acts like a switch. You walk into a room, switch a light on, and you never think of it as remarkable that the light stays on until you switch it off. You never tell your friends, in reverent tones, that the light circuit contains a memory, and yet each memory unit of a computer is no more than a very small variety of switch which can be turned on or off and which will then stay that way until it is used again. One unit of memory like this is called a *bit* – the name is a contraction of *b*inary dig*it*.

Now let's stick with the idea of a switch, because it is so useful. Suppose that we wanted to signal with electrical circuits and switches. We could use a circuit like the one in Fig. 1.1. When the switch is on, the light is also on, and we take this as meaning YES. Turn the switch off, and the light goes out; we could take this to mean NO. You could use any other two meanings that you wanted, so long as there are only two. Things improve if you use two switches, two lights, two lines, as in Fig. 1.2. Now four different combinations are possible: (a) both off (b) A on, B off (c) B on, A off,

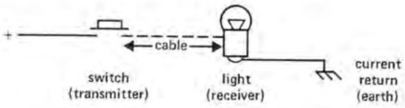
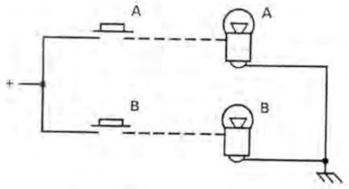


Fig. 1.1. A single-line switch and bulb signal system.



A	B
off	off
off	on
on	off
on	on

Fig. 1.2. Two-line signalling – four possible signals can now be sent.

or (d) both on. This means that we could signal four different meanings. Using one line gives two possible meanings; using two lines gives four meanings (2 × 2 = 4), and if you feel inclined to work them all out you will find that using three lines will permit eight different combinations of signals and therefore eight different meanings. Since 8 is 2 × 2 × 2, it should not be a surprise to learn that eight lines would allow you 2 × 2 × 2 × 2 × 2 × 2 × 2 × 2 = 256 different meanings to be communicated by means of signals. For N lines, the number of possible signals is 2^N, in fact. Any collection of eight switches, each of which can be on or off can be set into 256 different arrangements. It's up to us to make some sense of how we use these signals.

One particularly useful way is called binary code. Binary code is a way of writing numbers using only two digits, Ø and 1. The zero is often shown crossed to avoid confusion with the letter O. We can think of zero as meaning 'switch off', and 1 as meaning 'switch on', so that 256 different numbers could be signalled using eight switches, by thinking of the switch positions as digits, Ø for off, 1 for on. This group of eight is called a byte, and this is why the number 256 is encountered so much in computing. Why a group of eight? It just happened: the early calculators were able to work with four bits at a time, and the step up to eight bits has lasted for quite a long while. Sixteen-bit machines are still not very common.

The way the bits in a byte are arranged so as to indicate a number is along the same lines as we use to indicate numbers normally. When you write a number such as 256, the 6 means six units, the 5 is written to the immediate left of the 6 and means the number of tens, and the two is written one more place to the left, and is the number of hundreds. These *places* indicate the importance or *significance* of a digit (see Fig. 1.3). The 6 in 256 is called the 'least significant digit', the 2 is the 'most significant digit'. Change the 6 to 7, and the change is one part in 256. Change the 2 to 3 and the change is one hundred parts in 256 – much more important.

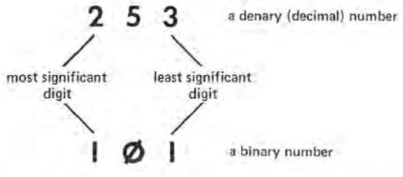


Fig. 1.3. Significance of digits. Our numbering system, unlike the old Roman system, uses the place of a digit to indicate its significance or importance.

Having looked at bits and bytes briefly, it's time to go back for a moment to the idea of memory as a set of switches. As it happens, we need two types of memory. One type must be permanent, like mechanical switches or fixed connections, because it is used to hold number-coded instructions that operate the computer. This is the type of memory which is called ROM, the letters meaning Read-Only Memory. The ROM is the most important part of your computer because it contains the instructions which make the computer carry out all of its actions. When you write a program for yourself, you store another set of number-coded instructions in a part of memory that you will want to be able to use over and over again. This is a different type of memory which can be 'written' or 'read', and if we were logical about it we would call it RWM, standing for read-write memory. Unfortunately, we're not very logical about it, and we call it RAM (Random-Access Memory), which was a name used in the very early days of computing to distinguish this type of memory from one which operated in a different way. We're stuck with the name RAM now, so we'll have to make the best of it!

All done by numbers

Now let's get back to the bytes. We saw that a byte, which is a group of eight bits, can consist of any of 256 different arrangements of these bits, and that the most useful way of using these arrangements is to make each one represent a number in what is called binary code. The numbers are Ø to 255 (*not* 1 to 256, because we need a code for zero), and each byte of the 16,384 bytes of RAM in your Spectrum 16K can store a number in this range.

Numbers by themselves are not of much use, and we wouldn't find a computer particularly useful if it could deal only with numbers between Ø and 255, so we make use of these numbers as *codes*. Just as your Spectrum uses each key to do several different actions, each number code can be used to mean several different things. If you have worked with some BASIC programming, you will know that each letter of the alphabet and each of the digits Ø to 9, and each of the punctuation marks is coded as a number between 32 (which is the space) and 127 (which is the copyright sign on Spectrum). That leaves us with a large number of code numbers to use for other purposes such as graphics characters, and Spectrum, like all other small computers, uses most of the numbers also as codes for actions. When, for example, you press the key which is marked PRINT, what is placed in the RAM memory of your Spectrum is not the sequence of ASCII number codes for PRINT, which would be 80,82,73,78,84, needing five bytes; but one single byte, 245. This single byte is called a 'token', and it can be used by the computer in two ways. One is to locate the actual characters which make up the word PRINT. These are stored in ASCII number-code form in the ROM, because they won't be changed (you don't want the word PERPLEXING to appear when you press the PRINT key!) and so that they don't take up space in your RAM. The other use of the token is to locate a set of instructions, also in coded form in the ROM, which will cause the action of printing (on the screen) to be carried out. These codes are the ones that we refer to as 'machine code', because they directly control what the machine does.

Practical interlude

As an aid to digestion, try a short program. This one (Fig. 1.4) is designed to reveal these 'keywords' that are stored in the ROM, and it makes use of the BASIC instruction, PEEK. PEEK has to be

```
1Ø PRINT 15Ø;"       ";: FOR n = 15Ø TO 516
2Ø LET k = PEEK n
3Ø IF k < = 127 THEN PRINT CHR$ k;
4Ø IF k > = 128 THEN PRINT CHR$ (k - 128):
   PRINT n;"      ";
5Ø NEXT n
```

Fig. 1.4. A BASIC program to PEEK at words stored in the ROM.

followed by a number or a number variable, and it means: 'find the byte stored at this address number'. The groups of eight units of memory in your Spectrum are numbers from zero upwards, one number for each byte whether it is ROM or RAM, and because this is so much like the numbering of houses in a road, we refer to the numbers as *addresses*. The action of PEEK is to find out what number, which must be between Ø and 255, is stored at each address, and the Spectrum automatically converts the binary-coded numbers into ordinary (decimal, or more correctly *denary*) form. By using CHRS, we can print the character whose code is the number we have PEEKed at. So far, so good. The program allocates 'n' as an address number, and then checks that PEEK n is less than 128 – in other words, that it is a character in ASCII code. If it is, it is printed.

Now the reason that we need the check is that the last character in each set of words or word is stored with a different coding. The number that is PEEKed for the last character is 128 + the ASCII code, rather than just the ASCII code. For example, the first three locations which the program PEEKs at, with addresses 15Ø, 151, and 152, contain the numbers 82, 79 and 196. The number 82 is the ASCII code for R, 79 is N, and 196—128 = 68, which is the ASCII code for D, so this is where the word RND is stored. Why fiddle the

D? The reason is that the Spectrum designers did not want to waste memory, so instead of having another byte to separate RND from the next word, which is INKEY$, they used this method of indicating to the computer where each word ends. When the Spectrum reads these codes one by one, it is programmed to stop reading when it comes to the one whose code number is greater than 128. We have made use of the same system in line 4Ø of the program of Fig. 1.4 to print the correct letter (by subtracting 128 before using CHR$), and to print a space before moving on to the next letter.

Now for the next revelation. Take a look at the table of keywords that starts on page 186 in your Spectrum manual. Notice that they are in the same order as they are stored in the memory. By keeping them in order like this, with their token codes (starting with 165) also in order, it's easy for the computer to find one code if it is fed with the address of the start of the list – which is what it has to do each time you press a key.

Spectrum analysis

Now take a look at a diagram of the Spectrum, Fig. 1.5. It's quite a simple diagram because I've omitted all the detail, but it's enough to give us a clue about what's going on. This is the type of diagram that we call a 'block diagram', because each unit is drawn as a block, with no details about what may be inside. Block diagrams are like large-scale maps, which show us main routes between towns but don't show side roads or town streets. A block diagram is enough to show us the main paths for signals in the computer, without the sort of confusing detail that you need if you want to show exactly what electrical connections are made.

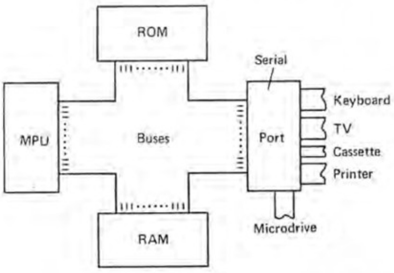


Fig. 1.5. A block diagram Spectrum. The connection marked *Buses* consists of a large number of connecting links which join all of the units of the system.

Two of the blocks have already been introduced to you, ROM and RAM. ROM is the memory that can't be changed; it contains all of the essential instructions, along with keywords and token numbers, that are needed to make the computer work. The RAM is used to contain your programs and a lot more besides, but we'll go into that later.

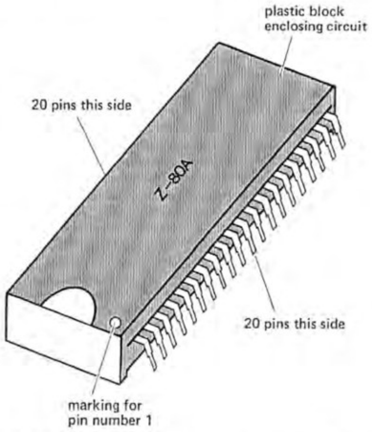


Fig. 1.6. The Z-80A MPU. The actual working portion is smaller than a fingernail, and the larger plastic case (52 mm long, 14 mm wide) makes it easier to work with.

The block marked MPU is a particularly important one. MPU means Microprocessor Unit (although some block diagrams use the letters CPU, meaning Central Processing Unit), and that's the main 'doing' unit in the system. Unit is a well-chosen name in this case, because the MPU is just a single plug-in chunk, one of these silicon chips you read about, encased in a slab of black plastic, and provided with 40 connecting pins arranged in two rows of 20, as Fig.

1.6 indicates. There are several types of MPU made by different manufacturers, and the type which the Spectrum uses is called Z-80A. It is almost identical to the type called Z-80; the only difference is that the Z-80A can be operated more quickly if required.

What does the MPU do? The answer is practically everything, and yet the actions that the MPU can carry out are remarkably few and simple. The MPU can load a byte, meaning that a byte stored in the memory can be copied into another store inside the MPU. The MPU can also store a byte, meaning that a byte stored in the MPU can be copied into any address in the memory. These two actions (Fig. 1.7) are the ones that the MPU spends most of its working life in carrying out, and by combining them we can copy a byte from any one address to any other. You don't think that's very useful? That copying action, you see, is just what goes on when you press the 'j' key and see the letter 'j' appear on the screen. The MPU treats the keyboard as one piece of memory and the screen as another, and shifts bytes from one to the other as you type. That's a considerable simplification, but it will do for now.

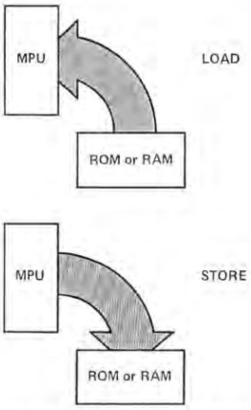


Fig. 1.7. Loading and storing. Loading means signalling to the MPU from memory, so that the digits of a byte are copied into the MPU. Storing is the opposite process.

Loading and storing are two very important actions of the MPU, but there are several others. One set of actions is the arithmetic set. Contary to what you might expect, these consist of addition and subtraction only, and of no more than two-byte numbers either. How does the computer carry out arithmetic with larger numbers, numbers with fractions, how does it carry out multiplication, division, raising to powers, logarithms, sines and cosines? The answer is by machine code programming that is contained in the ROM. If these programs were not there, you would have to write your own, and a BASIC program for carrying out multiplication, using only addition, would be long and tedious, not a pretty sight.

There's also the logic set. MPU logic is, like all MPU actions, simple and obeys rigorous rules. Logic actions compare the bits of two bytes, and produce an 'answer' which depends on these bit values (Ø or 1) and on the logic rule that is being used. The three logic rules are called AND, OR, and XOR, and Fig. 1.8 shows how they are applied.

Another set of actions is called the jump set. A jump means a change of address, rather like the action of a GOTO in BASIC, and it's the way in which the MPU carries out its decision steps. Just as you can program in BASIC:

```
1ØØ IF a = 36 THEN GOTO 1Ø5Ø
```

so the MPU can be made to carry out an instruction at an entirely different address from the normal one, which would be the next address number. The MPU is a programmed device, meaning that it carries out each action as a result of being fed with an instruction byte which has been stored in the memory. Normally when the MPU is fed with an instruction from an address somewhere (usually in ROM), it carries out the instruction and then 'reads' the instruction byte that is stored in the next address up. A jump instruction would prevent this from happening, and would instead cause the MPU to read another address, one that was specified in the jump instruction. This jump action can be made to depend on some previous action, such as a zero, or positive, or negative answer to a subtraction, addition or comparison.

That isn't a great list, but the actions which I've omitted are not very important, nor very different from the ones in the list. What I want to emphasise is that the magical microprocessor isn't such a very smart device. What makes it so vital to the computer is that it can be made to carry out its actions very quickly, and each action is completely controlled by programming, sending it electrical signals.

```
AND

The result of ANDing two bits will be 1 if both
bits are 1, Ø otherwise:

    1 AND 1 = 1  { 1 AND Ø = Ø }  Ø AND Ø = Ø
                 { Ø AND 1 = Ø }

For two bytes, corresponding bits are ANDed

                    1Ø11Ø111
           AND      ØØØØ1111
                    ________
                    ØØØØØ111
                         ~~~
                        only
                      these bits
                     exist in both
                        bytes.

OR

The result of ORing two bits will be 1 if either
or both bits is 1, Ø otherwise:

       1 OR 1 = 1  { 1 OR Ø = 1 }  Ø OR Ø = Ø
                   { Ø OR 1 = 1 }

For two bytes, corresponding bits are ORed

                    1Ø11Ø111
           OR       ØØØØ1111
                    ________
                    1Ø111111
                     ↑
                    only
                  bit which
                   is Ø in
                    both.

XOR (Exclusive-OR)

Like OR, but result is zero if the bits are identical

     1 XOR 1 = Ø   { 1 XOR Ø = 1 }  Ø XOR Ø = Ø
                   { Ø XOR 1 = 1 }

                    1Ø11Ø111
           XOR      ØØØØ1111
                    ________
                    1Ø111ØØØ
                    ~~~~~~~~
                  if two bits
                 are identical
                  the result
                   is zero.
```

Fig. 1.8. The rules for the three logic actions, AND, OR and XOR.

These signals are sent to eight pins, called the data pins, of the MPU and, as you will have realised, these eight pins correspond to the eight bits of a binary-coded byte. Each byte of memory will therefore be able to affect the microprocessor by sharing its electrical signals with the MPU. Descriptions in words like this take too long to write more than once, so we speak of reading and writing, always from the point of view of the MPU. Reading means that a byte of memory is connected to the MPU so that each 1 bit causes a 1 signal on a data pin, and each Ø bit causes a Ø signal on the corresponding data pin. Just as reading a paper or listening to a tape doesn't destroy what is written or recorded there, reading a memory doesn't change the memory in any way, and nothing is taken out. The opposite process of writing does, however, change memory. Like recording a tape, writing obliterates whatever existed there before, so that when the MPU writes a byte to an address in the memory, whatever was stored at that address previously is there no more and has been replaced by the new byte. This is why it is so easy to write new BASIC lines replacing old ones at the same line number.

Pick a number, any number ...

Do you really write programs in BASIC? It might sound like a silly question, but it's a serious one. The actual work of a program is done by coded instructions to the MPU and so far you don't write any of these. All you do is to select from a menu of choices that we call the BASIC keywords, and arrange them in the order that you hope will produce the correct results. Our choice is limited to the keywords that are designed into the computer in the ROM. We can't alter the ROM, and if we want to carry out an action that is not provided for in the ROM, we must either try to make it work by combining BASIC commands, or operate directly with machine code on the MPU. It's like the difference between talking of a 'motorised vehicle with a capacity for transporting more than eight persons', and a 'bus'. When you have to carry out actions with only a limited number of commands, the result can be clumsy, especially if each command is a collection of other commands. Direct action is quick, but it can be difficult. The 'direct-action' that I'm talking about is machine code, and a lot of this book will be devoted to understanding this 'language' which is difficult just because it's simple!

Take a situation to illustrate this paradox. Suppose you want a wall built. You could ask a builder. Just tell him that you want a wall built across the back garden, and then sit back and wait. This is like using BASIC with a command-word for 'build a wall'. There's a lot of work to be done, but you don't have to worry about the details.

Think of an option. Suppose you had a robot which could carry out instructions mindlessly but incredibly quickly. You couldn't tell it to 'build a wall', because these instructions are beyond its understanding. You have to tell it in detail, such as: 'Stretch a line from a point 85 feet from the kitchen edge of the house and against the fence, to one 87 feet from the lounge end of the house and touching the opposite fence. Mix three bags of sand and two of cement with four barrow-loads of pebbles. Mix water in until a pail filled with the mixture will just empty when held upside down. Fill the trench with the mixture ...'. The instructions are very detailed – they have to be for the brainless robot – but they will be carried out faultlessly and quickly. If you've forgotten anything, it won't be done, no matter how ludicrous is seems. Forget to specify how much mortar, what mixture and where to place it, and your bricks will be put up without mortar. Forget to specify the height of the wall as a number of layers of bricks, and the robot will keep piling one layer on top of another, in the style of the Sorcerer's Apprentice, until someone sneezes and the huge wall falls down.

The parallel with programming is remarkably close. One keyword in BASIC is like the 'build a wall' instruction to the builder – it will cause a lot of work to be done, but not necessarily as fast as you would like. If you can be bothered with specifying the detail, machine code is a lot faster because you are giving your instructions to an incredibly fast but mindless machine, the microprocessor. We can stretch the similarity further. If you said to your imaginary builder – 'repair the car' – he might be either unwilling or unable, but a set of the correct detailed instructions to the robot would ensure that this task also was carried out. Machine code can be used to make your computer carry out actions that simply are not provided for in BASIC, though it's fair to say that many modern computers allow a much greater range of commands than early models, and this aspect of machine code is not as important now as it was then.

One last look at the block diagram is needed before we start on the inner workings of the Spectrum. The block which is marked Port covers a lot of circuits that are contained in one single chip, along with others. A PORT, in computing language, means something that is used to pass information, one byte at a time, into or out from the microprocessor system – the MPU, RAM, ROM parts. The reason for having a separate section to handle this is that inputs and outputs are important but *slow* actions. By using a port, we can let the microprocessor choose when it wants to read an input or write an output. For example, just imagine a BASIC program in which every line contained an INPUT. It would run very slowly, because it will hang up and wait for you to press a key, followed by ENTER, on each line. If the program contained just one INPUT on the first line, it could then run uninterrupted by the keyboard until the end of the program. You will find, for example, that if you put the computer into an endless loop with:

```
1Ø GOTO 1Ø
```

then no single key will have any effect on the computer. The computer still checks the port to see what's there each time it carries out the instruction, though, because if you interrupt the program by pressing CAPS SHIFT and SPACE together then you will break out of the loop. The use of the PORT, however, is a method of letting the computer get on with its work with only this type of interruption permitted.

In addition, there is no output from the computer except where we have commanded a PRINT or LPRINT, or when a program has come to an end. Once again we don't see anything new on the screen while the microprocessor is getting on with its work. The port isolates the section of the computer that deals with the screen, keeping whatever is there in place while the microprocessor deals with subsequent instructions. Without this isolation, your program would run much more slowly, and a lot of gibberish would appear on the screen each time an action took place. This is illustrated in the program shown in Fig. 1.9.

```
1Ø FOR n = 16384 TO 22528
2Ø POKE n, RND*255
3Ø NEXT n
```

Fig. 1.9. A program which POKEs on to the screen bytes generated at random.

We have now looked at all of the important sections that make up the heart of your Spectrum. I've used some terms loosely – purists will object to the way I've used the word 'port', for example – but there's no quarrelling with the actions that are carried out. What we have to do now is to look at how the computer is organised to make use of the MPU, ROM, RAM and PORTs to be programmed in BASIC and to run a BASIC program. This looks like a good place to start another chapter!

Chapter Two
Digging inside Spectrum

Don't take the title too literally, you don't need to open the case! What I mean is that in this chapter we are going to look at how the Spectrum is organised to load and run BASIC programs, and we shall in the course of this discover the meanings of some of the numbers and the cryptic titles that occur in Chapter 25 of the Spectrum manual.

Let's start with a simplified version of the action of the whole system - simplified in the sense of omitting a lot of detail that would just be confusing at this stage. The ROM of your Spectrum consists of a large number of short programs - *subroutines* - which are written in machine code. There will be at least one of these machine code subroutines for each keyword of BASIC, and some of the keywords may require the use of many subroutines. When you switch on the Spectrum at first, the piece of machine code program that is carried out is called the 'initialisation' section. This is a long piece of program, but because machine code is fast, carrying out instructions at the rate of several hundreds of thousands per second, you see very little of it – the only evidence on the screen is the black rectangular pattern that appears just before the Sinclair Research copyright notice. In this brief time, however, the size of the RAM has been checked (in case you added some extra chips last night). It has been cleared of any unwanted bytes, a process that is necessary because of the effects of switching a memory off and then on again. When a RAM memory is switched off, all the units of memory revert to the ∅ setting, as you might expect. When you switch on again, however, there is no guarantee that they will all stay that way. In fact, roughly half of them go to the '1' setting, completely at random, so that if you were to read each byte of memory at the instant when the computer was switched on, you would find that each byte of memory stored a number between ∅ and 255, quite at random, with no rhyme or reason to the numbers. This kind of thing is called *garbage*, and one of the tasks of the initialisation program is to replace each of these garbage bytes by a Ø, by deliberately writing a Ø into each unit of the RAM memory in turn. As a result, if you switch on and PEEK at RAM memory address above about 239ØØ you will find that the content of each byte is zero.

Initialisation consists of a lot more than this, however. Of the 16K of RAM in the smaller Spectrum, a large chunk is used by the operating system, meaning that the machine code routines use the RAM to store quantities that may have to be altered at various times as the program is used. Addresses from 16384 up to 23755 are used in this way, and that's more than 7K of memory (1K = 1024 bytes) out of your 16K before you have typed a single character of BASIC. In addition, once you start to enter a BASIC program, more RAM has to be set aside, this time at higher memory addresses for storing quantities that are needed to make your program run. Every time you 'declare a variable', for example, by using a line such as:

```
LET n = 2Ø   or   LET a$ = "Smith"
```

you cause several bytes of memory to be taken up by entries which store the variable name, n or a$ or whatever you used, and the quantity or the characters (2Ø or Smith). The piece of memory that is used for these purposes is called the Variable List Table (VLT), and it is kept immediately above the space used by your program. Add one line to your program, and the VLT has to be shifted up to make more space. Delete a line, and the whole VLT list moves down to lower memory addresses.

This is a type of behaviour that has to be controlled rather carefully because the computer must at all times keep a note of where this piece of memory is located. This is done by an entry in one of the pieces of RAM reserved for such an 'index'. Since the principle is used very extensively, we might as well examine this particular example in detail.

The important addresses that the computer must keep a track of are stored between 23552 and 23733, and the starting address for the variable list table is stored at the addresses 23627 and 23628. Now the addresses that we want to store will also consist of five-figure numbers like these, and we know already that one byte of memory can hold a number which is between Ø and 255 inclusive. If you want to store numbers that are greater than 255, then you need at least one more byte, and the scheme that computers use to store address numbers is to take one more byte to store the number of complete 256's in the number that is to be stored – using a scale of 256 rather than a scale of two or ten. If we had two bytes stored which read, in order, 1Ø,1; this would mean 1Ø + 256*1 = 266. If the bytes were, in order, 24,32, this would represent the number 24 + 256*32 = 8216. Note that the order of storing the numbers is low byte, then high byte. To find the address number that is stored in the addresses 23627 and 23628, then, we need to use the formula PEEK 23627 + 256*PEEK 23628. The result of this is an address – and it's the address of the starting point for the variable list table.

Try typing into your Spectrum:

```
1ØØ PRINT PEEK 23627 + 256* PEEK 23628
```

and run this. Note the number that is printed. Now add some lines such as 1Ø LET n = 12 and 2Ø LET a$ = "Smith" and RUN again. You will get a larger number printed, because the start of the variable list table has been moved up to a higher memory address to make room for the additional lines of BASIC. Add still more lines of BASIC and the start of the VLT has to move even higher. Delete some lines, and the VLT can move lower again. When the memory addresses that are allocated for some purpose or other are stored in this way, we say that they are 'dynamically allocated'. That's also why the manual warns you not to alter the address of the VLT (by a POKE command), because this would cause the computer to lose track of some of its variables. Try it – use the command:

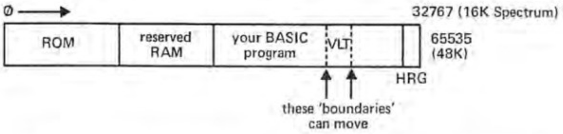


Fig. 2.1. The position of the variable list table (VLT) in the memory is not fixed – it is placed just beyond the BASIC program, and will move up or down as the BASIC lines are added to or deleted.

```
POKE 23627,2Ø3:POKE 23628,92
```

Now try to RUN and watch the chaos. Switch off and on again if your Spectrum hangs up and refuses to respond to keys. What you have done is to mislead the fast but brainless microprocessor that runs your Spectrum.

This, incidentally, is an introduction to the POKE command. The first number following POKE is the address whose byte you want to change, the second is the byte you want to put into that address. Later on, we'll look at the POKE process in more detail, but for the moment, note that the number which is POKEd into memory in the example above is 23755 – where a BASIC program starts rather than where the VLT should start.

Now try something very much more constructive when you have regained control. We shall now take a look at what the Variable List Table contains. As you might expect, it has to contain the 'name' of the variable and its value, but the Spectrum does some coding of values so that it can find and use the values when it needs to.

To start with, the first byte in the variable list table is always a number greater than 4Ø denary. The reason for this is so that the computer can detect the *last* line of a program. Program line numbers are restricted to the range of Ø to 9999 denary, and since 9999 would be coded by the two bytes 15 (low byte) and 39 (high byte), the high byte of the number can never exceed 39. Since the high byte of the line number is always the *first* byte of a line, the computer can check this byte, and if it is 4Ø or more, this will signal the end of the BASIC program.

This, however, means that some care is needed to ensure that the first byte of an entry in the VLT is a number greater than 4Ø, which is easy enough, because the ASCII code for a valid variable name must be a number greater than 4Ø. The other point is to code these variable names so that the computer can distinguish the different types of variables from each other. This is done by using only five bits of the first byte for the variable name letter code, and using the top three bits for a coding for the variable type (number, string, array, etc.). This needs some more detailed explanation.

For a simple number variable which consists of one letter only, the first byte of its VLT entry is just the ASCII code for the name. The manual shows this in rather a confusing form – with 96 subtracted from the ASCII code and then added on again. It ensures that each entry starts with the bits Ø11, which is the coding for a number variable whose name is a single letter. The value for the number is then contained in the next five bytes after the letter code. Unless you are curious or of mathematical inclination, it doesn't do to enquire too closely as to how the value of a number which is fractional or negative is coded. Details are shown in Appendix B if you really want to know, but if we stick to integers, meaning positive whole numbers less than 65535 as far as Spectrum is concerned, then for an integer less than 256, the number is stored as a single byte, the third byte following the coded name. If the integer lies between 256 and 65535, then two bytes are used – the third byte holds the less significant part of the number and the fourth byte holds the more significant part (the number of 256's). The fact that integers still need five bytes for storage is one of the factors which makes Spectrum BASIC so much slower than the BASIC of machines which can treat integers differently.

```
1Ø LET a = 1Ø
2Ø LET b = 11
3Ø LET c = 12
4Ø FOR n = Ø TO 3Ø
5Ø PRINT PEEK (( PEEK 23627 + 256* PEEK 23628)
   + n) ;"      ";
6Ø NEXT n
```

Fig. 2.2. A program to investigate the storage of integers.

Fig. 2.2 shows a simple program which allows you to investigate the storage of some integers, and displays the results on the screen. If the name of the number variable consists of more than one letter, another type of coding is needed, because otherwise the computer would still read the five bytes following the first character as if they were the value. The scheme that is used in this case is to add 64 to the ASCII value of the first letter of the name, then store each subsequent character in normal ASCII code form until the last character, which has 128 added to the ASCII code. This is done so that the computer can recognise the end of the name and then count out the next five bytes as the bytes used to hold the value. Fig. 2.3 shows an example of this method that you can try for yourself.

A quick look at a variable which is not an integer is instructive, and Fig. 2.4 shows such an example. What is of interest now is not the coding of the letter variable name, but the fact that all five bytes are used for coding the value. This indicates (see Appendix B for

```
1Ø LET julie = 23
2Ø FOR n = Ø TO 17
3Ø PRINT PEEK (( PEEK 23627 + 256* PEEK 23628)
   + n) ;"      ";
4Ø NEXT n
```

Fig. 2.3. How a long number variable name is stored.

```
1Ø LET a = .5
2Ø LET b = .25
3Ø LET c = .125
4Ø FOR n = Ø TO 3Ø
5Ø PRINT PEEK (( PEEK 23627 + 256* PEEK 23628)
   + n)
6Ø NEXT n
```

Fig. 2.4. Non-integer variables. The examples have been chosen to give reasonably simple results.

details) that the value which is stored is never exact, so that when you have a program that uses fractions, there will be some 'rounding' errors. These will seldom show on the screen, because the value that is shown on the screen is not of as many decimal places as the stored value (it has been rounded up), but it can cause trouble in BASIC statements which look for identical values. Try the program of Fig. 2.5 to see what I mean – the fact that the computer does not recognise the numbers as being equal is due to the small differences in the stored values which do not appear on the screen. It's like saying that 1.00000007 is not equal to 1.00000008 – if we print only the first six places after the decimal point, then both numbers appear on the screen as 1.000000, and the computer is detecting a difference of one part in 100 million! No-one needs accuracy like this, and it causes trouble only when you have equality statements in programs. A good way of dealing with the problem is always to equate quantities which have been rounded, as for example, by statements like:

```
IF INT(1ØE4*a) = INT(1ØE4*b) THEN GOTO ...
```

which will perform the GOTO if the numbers are equal to an accuracy of four decimal places – good enough for all but the most exacting purposes.

```
1Ø LET n = 1/1ØØØ
2Ø LET p = 1Ø/1ØØØØ
3Ø PRINT "n = ";n;" and p =  ";p: IF n = p
   THEN PRINT "Equal"
4Ø IF n<>p THEN PRINT "Not equal"
```

Fig. 2.5. The computer may not have the same idea about equality as you do! This is caused by 'rounding errors'.

```
1Ø LET a = -12
2Ø LET b = -176
3Ø LET c = -255
4Ø FOR n = Ø TO 17
5Ø PRINT PEEK ((PEEK 23627 + 256*PEEK 23628)
   + n);"      ";
6Ø NEXT n
```

Fig. 2.6. How negative integers are represented in the VLT.

```
1Ø LET a$ = "Sinclair"
2Ø FOR n = Ø TO 17
3Ø PRINT PEEK ((PEEK 23627 + 256*PEEK 23628)
   + n);"      ";
4Ø NEXT n
```

Fig. 2.7. Storing a string variable.

Fig. 2.6 shows what happens when the VLT stores a negative integer – once again you don't need to be able to follow the coding exactly, but note that three bytes are used. Turning from number variables, Fig. 2.7 shows how a string variable is stored. The variable name is coded as the ASCII value minus 32, and the string sign, $, is not included in the coding. Spectrum permits only single letter names for string variables, so the complications of extra letters do not arise in this case. The string name is followed by two bytes which store the number of characters in the string so that the computer can be instructed how many bytes to read. Since two bytes are used to store the string length (most computers use only one, which speeds up string handling), Spectrum permits very long strings at the expense of a byte which is wasted if strings of normal length are being handled. The length bytes are followed by the characters of the string using normal ASCII codes.

Appendix C shows the coding that is used for arrays or numbers or characters (string arrays). This is rather more complicated, and the subject is touched on again in more detail in Chapter 9. While we are on the subject of storing variables, however, we can explain the

```
1Ø FOR n = Ø TO 17
2Ø PRINT PEEK ((PEEK 23627 + 256*PEEK 23628)
   + n);"      ";
3Ø NEXT n
```

Fig. 2.8. How the values for a loop are stored.

numbers that you will have noticed on the screen following the VLT for the numbers and strings that we have been looking at. These are the numbers associated with the FOR ... NEXT loop that was used to print out the values. Fig. 2.8 shows a program that consists only of such a FOR ... NEXT loop, arranged to print out its own VLT. The variable name that is used for the variable of the loop, in this example n, is stored as the ASCII code plus 128, and its value is stored in the next five bytes in the usual way. This value will change on each pass through the loop, so that the value which you see printed on the screen is the value that happened to exist at the time of printing. Since this value is the fourth character printed (with n starting at Ø) then the value is 3 at this time, but will change to 4 after the NEXT, and so on. The next set of five bytes is reserved for the final value (18 in the example) and the third set of five bytes is for the STEP value, which will be 1 unless we have specified something else. Two bytes near the end of the block specify the number of the 'looping line', the line number to which the loop returns if the loop has not ended, and the last byte specifies the statement number within that line, in case the FOR ... NEXT loop has started inside a multi-statement line like:

```
1Ø LET a = 2 : LET b = 3 : FOR n = 1 TO 6
```

A total of 19 bytes are used; another reason for the comparatively slow running of Spectrum BASIC. Many machines permit statements like:

```
FOR N% = 1 TO 16 STEP a%
```

which specifies integers stored in two bytes each; such loops can run very much faster and take much less memory. Spectrum does not permit such options.

Program entry

We've seen now that two sections of the RAM are used in ways that are controlled by the computer. The lower area of RAM is reserved for use by the system, meaning that it will be reserved whether you type a BASIC program into Spectrum or not. The upper section of the RAM is reserved mainly for use with a BASIC program, and the less call your BASIC program makes upon it, the more memory will be left for you to program with! It's time now to take a closer look at what goes on when you enter a program.

As you know, the computer shows the line you are typing on the bottom lines of the screen, before you use ENTER. You can delete parts of this line or lines, because the whole section is not put into the normal final resting place in memory, which starts at 23755. Instead, it is stored in a temporary space, a piece of memory that is called a 'buffer'. This is another piece of dynamically allocated memory which has to shift up each time another line is added to the BASIC program. That, incidentally, is why it can take a noticeable time between pressing ENTER and seeing the 'K' cursor appear again on the bottom line.

Suppose, for example, you typed:

```
1Ø LET n = 12
```

Until you press ENTER, this is treated as a temporary entry only, and is placed in the buffer space. If you had omitted the line number, then the line would *never* occupy any other space, but with the line number present, the computer is programmed to place this line into the main program memory space when ENTER is pressed. Where is the start of this program space stored? It doesn't usually change, but nevertheless its starting address is stored at 23635, 23636, so that we can find the address of the first byte of the program by using:

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

and this number is usually 23755. If you alter this address by POKEing different numbers into 23635 and 23636, then your computer will not recognise that there is a program stored, because it can't be listed or run unless the correct starting number is present. Furthermore, if the new address you have put into these two bytes is well above the first part, you could write a second program in this other piece of memory, list it and run it. By restoring the original address bytes into 23635 and 23636, you could then list, run and work with the first program, ignoring the second. You would have to be sure, however, that each program carried its own variable list table, and that the variable list table address was correct before running a program.

Back to normal programming, though. What does this line of program look like when it is stored in the program space? We can find out by using a short piece of BASIC command, assuming that you have the line 1Ø still in the memory:

```
FOR n = 23755 TO 23785: PRINT PEEK n;" ";:NEXT n
```

This is a direct command so that it will not get mixed up with the line that we want to investigate.

Fig. 2.9 shows what you can expect to see – a string of numbers, of which the key is the ASCII code for n, which is 11∅. We can pick this out in the listing, and knowing that 11∅ means 'n', we can deduce that 241 is the code for LET (there is a list of the codes for keywords in the manual), and we should know that 61 is the ASCII code for the equality sign. We can also see that the number 12 is coded as two bytes of ASCII code, 49 for 1, 5∅ for 2. That codes LET n = 12 into 5 bytes, but there are 4 bytes ahead of LET and a large number of them after, with a 13 at the end of the string. The end byte is 13, the code that is used for the ENTER key, and the bytes between the 14 and the 13 are the coded form of entry which will be used in the variable list table: 14 is used as a signal that what follows is in correct number code form, as distinct from ASCII form. The four bytes that precede the LET code are of more interest to us at the moment, however, because they illustrate yet another way in which the computer uses the RAM to keep a track of what is going on. The first two bytes are the line number. Unlike all other two-byte numbers that the computer uses, these are in conventional (to you) order, with the high byte (the number of 256's) first, and the low byte second. This is done, as was explained earlier, so as to allow the computer to sense the end of a program by looking for a byte greater than 4∅. Your line 1∅ should give rise to bytes ∅, 1∅, but if you had started with a line 1∅∅∅, then the numbers would have been 232,3, because 3 * 256 + 232 equals 1∅∅∅.

```
∅    1∅    12    ∅    241    11∅    61    49    5∅
14    ∅    ∅    12    ∅    ∅    13    234
∅    ∅    222    92    ∅    ∅    ∅    233    92
∅    ∅    ∅    1    ∅
```

Fig. 2.9. The appearance of a line of BASIC in memory – this BASIC program allows you to look at how it is stored.

What of the next two bytes? If you typed the line just as I have shown it, then the next two bytes will be 12,∅. This is in the more usual low then high order of bytes, and the number twelve is the total number of bytes of codes in the line, including the text (LET n = 12), the VLT entry of the code for n and five bytes of value, and the 13 byte at the end of the line, but excluding the four bytes at the start. Why is this length number needed? The reason is simple, like the action of the microprocessor. If the computer keeps a count of the bytes as it goes along, then the number that is stored in its address counter after it has read these two 'length-of-line' bytes will be the address of the first useful byte (LET in our example) in the program line. When this number has added to it the number stored in these two 'length-of-line' bytes, the result is the address of the start of the next set of the four bytes that give the line number and length for the next line. This is how the computer gets from one line to the next, how the correct number of bytes can be transferred to the 'bottom-line' of the screen (a different part of memory) for editing, and how lines can be re-sorted into order.

Every time you type a new line of BASIC and ENTER it, then, you use up five bytes of memory for the line number, the length number, and the ENTER (code 13) byte. This is called the line overhead, and the interesting feature about it is the use of two bytes for the line length. Most computers use only one byte here, permitting lines of no more than 255 characters, and since most lines are much shorter than 255 characters, this byte is usually zero. There is a wasted byte in each line, therefore, unless you are in the habit of typing quite impossibly long lines. You can overcome some of the line overhead problems by making full use of multistatement lines, because the line number and length bytes are used only when a new line number is started, not when you use the colon to separate statements. Try this one:

```
1∅ LET a = 12 : LET g$ = "Sinclair"
```

and then look at the line by using:

```
FOR n = 23755 TO 23785: PRINT PEEK n;" ";:NEXT
```

to look at the bytes that have been stored. The result is shown in Fig. 2.10. Now try POKE 23756,20, and then LIST your program line.

```
∅    1∅    27    ∅    241    97    61    49    5∅
14    ∅    ∅    12    ∅    ∅    58    241
97    36    61    34    83    1∅5    11∅    99
1∅8    97    1∅5    114    34    13
```

Fig. 2.10. The result of storing a multi-statement line.

You have changed the line number to 20 simply by changing the line number byte! Now try some more POKEing. Type and ENTER:

```
POKE 23759,234 : POKE 23760,13:POKE 23752,2 :
  POKE 23761,238
```

and then LIST your program. It now appears to be:

```
2∅ REM
```

There is no trace of the old program, though bytes from it are still in the memory. This illustrates very dramatically what POKE, which places bytes directly into the memory, can do. We'll look at the syntax of the POKE command later.

Running a program

When you have typed a BASIC program and entered each line into memory, the arrangement will be as we have seen, with the line number bytes, line length bytes, the keyword tokens and ASCII codes that make up the line, and VLT entry codes, and the code 13 that signals the end of the line. As each line is entered, which includes transferring it from the buffer address to its more permanent resting place, it is checked for syntax, and the usual error warnings will be delivered if such an error is detected. The ZX family of computers is almost unique in carrying out this syntax check before the line is entered, and this checking can save much tedious work later. Many computers will accept syntax errors quite happily, and will report them only when you try to RUN the program, which is rather late to find out.

Now, however, we want to look at the way in which the coded BASIC lines in the memory are treated by the computer when you press the RUN key and follow it with ENTER. Before you get to this stage, remember, the computer has stored a lot of information about your program in the reserved portions of its RAM. All of the variables, for example, will be ready to place into the VLT, whose starting address is calculated and stored ready. The start of the first line of BASIC is held in addresses 23635, 23636, and the end of the BASIC program bytes is where the VLT starts, as we have seen.

Now computers work to quite inflexible rules. The first byte of a line, after the line number and the line length bytes which are really just part of the computer's housekeeping system, is always a command word token. The Spectrum ensures that the first byte in each line is a command token by its entry system, which checks for this and will not permit a line to be entered if this is incorrect. This avoids having to check this when the program is RUN, as other computers have to do. What happens then depends on what the instruction happens to be. If it is a simple assignation, like LET n = 12, then an entry is copied into the variable list table from the part of the line that starts with the code number 14.

A complex assignation, like LET y = 2*n needs some more action. The code token for LET calls up a machine-code subroutine so that the name y is entered into the variable list table, but this routine has to be interrupted to carry out the multiplication routine, 2*n. This uses another set of machine code subroutines which first search through the variable list table to find the name n, extract its coded value, carry out the multiplication, and then return to the task of assignation, putting the answer that has been worked out into five bytes of the variable list table that follows the variable name entry.

These two simple examples illustrate two important actions that take place during a RUN (RUN-time actions), making entries into the variable list table, and reading entries that have already been made. Most of the simple BASIC actions are of these types. For example, the command PRINT a$ will start with the token for PRINT, which calls up a PRINT subroutine (one of the longest and most complex in the ROM). This subroutine will in turn call up another one which will search for the entry for a$ in the variable list table and which will, once a$ is found, store the number of characters in a$ and the address of the first character. Now that the machine has some clue as to where to find the ASCII codes corresponding to a$, and how many of them are to be printed, it can complete the PRINT routine. This is quite complicated because of the design of modern computers like Spectrum. At one time, the PRINT routine only had to copy the bytes of the characters from their places in the program or VLT memory into another set of memory addresses called the video memory. This was a physically separate chunk of memory, so that a computer like the 16K TRS-80 actually had 16K available for the user – the 1K that served the screen display was separate and not counted in this total. The convention that is used nowadays is to use a lot of the RAM for what the Spectrum manual calls a 'display file' – another form of video memory. This is part of the 16K or RAM which is used to store the bytes that dictate the shapes of characters and, on the Spectrum, each character needs eight bytes of RAM. These bytes, however, are not stored consecutively in the memory. Of the eight bytes per character, each is stored 256 places beyond the previous one. You can check this for yourself by using the simple BASIC program of

Fig. 2.11, which POKEs bytes into the memory locations that are used for the first character location on the screen.

The bytes which are used to make up the characters that are printed on the keyboard are stored in the ROM and, as you might expect by now, the starting address of this set of bytes, the character set, is kept in the reserved RAM. The address stored in 236Ø6, 236Ø7 is 256 less than the address of the start of this character set (so that the program which uses it can run in a loop which starts by adding 256 to the address), which is at 15616. The character set is in the ROM, but since its starting address is stored in the RAM, we could change this address to point to a new set of characters which we can store in RAM – this is what we do when we create the user-defined characters, in fact. The PRINT routine makes use of these stored character bytes, and sends copies to the correct addresses in the display file at the start of RAM, so that the TV circuits can then generate the signals which form the shape of the character of your receiver. At the same time, the 'housekeeping' routines swing into action to ensure that the next character will be either in the next space along, or placed wherever it is commanded by TAB or AT, or put into the start of the next new line.

```
1Ø FOR n = Ø TO 7
2Ø POKE 16384 + 256 * n, 68
3Ø NEXT n
```

Fig. 2.11. POKEing to a screen location. This requires values to be POKEd to memory positions 256 bytes apart.

It would be possible to fill volumes by examining each action of the Spectrum in detail, but there isn't much point as far as we are concerned here, because they all follow pretty much the same general pattern. A command word is represented in the memory by a token which at the RUN time is used to call up a subroutine, which will in turn make use of the variable list table and other subroutines to carry out its work. Sometimes a subroutine may have to 'hang up' – for example, a routine that carries out a PRINT n*k can't get on with printing until the result of n*k has been calculated. The computer will have to provide for this, and its provision takes the form of using RAM for temporary storage, so that a number of bytes are reserved for this purpose.

Finally, how does the computer call up the correct subroutine? It's rather simple – when you know. The tokens, like the reserved words, are stored in order in the ROM. Looking for a token means starting at the first one, and checking each in turn, keeping a count. When the matching token is found, the computer will have a count byte (in reserved RAM) of how many places down the table the token was situated. This count number is then added to another address number (stored in ROM), and the result is where the address of the correct subroutine is stored! In practice it's not quite so simple, but the principle is the same – finding an item on one list gives a count number that is then used to find an address on another list.

Chapter Three

The MPU

In this chapter, we'll get to grips with the Z-80A microprocessor of the Spectrum. The microprocessor, or MPU is, you remember, the 'doing' part of the computer as distinct from the storing part (memory) or the input/output part (the PORT), so that what the microprocessor does will decide what the computer as a whole does.

The MPU is itself a set of memory stores, but with a lot of organisation added. By means of circuits aptly named gates, the way in which bytes are transferred between pieces of memory inside the MPU can be changed and controlled, and it is these actions which constitute the addition, subtraction, logic and other actions of the MPU. Each action is programmed. Nothing will happen unless an instruction byte is present in the form of a 1 or a ∅ signal at each of eight terminals, and these bytes are used to control the gates inside the MPU. What makes the system so useful is that because the program instructions are in the form of electrical signals on eight lines, these signals can be changed very rapidly. The speed is decided by another electrical circuit called a 'clock-pulse generator' (or 'clock' for short). The Z-80A can work with a 4 MHz clock, meaning that the clock can be operated at a rate of 4 million pulses per second. The microprocessor will then carry out its internal operations at this rate, but since each complete action may require several internal operations, the rate of carrying out complete machine code instructions is rather less than this – typically a rate of half a million instructions per second. The clock speed that is used by Spectrum is 3.5 MHz–3½ million clock pulses per second.

Hardware and software

The electrical parts of the computer are what we call *hardware*. Changing the design would be hard work, snipping contacts here, soldering new ones there, making it into a mass of spaghetti that the designer would hardly recognise. The program which makes the assembly of hardware work like a computer is a form of *software*. It's a lot more easily altered, because in the case of the Spectrum it is all contained in one ROM chip; change this chip and you have a rather different computer! A few computers in the 'home' category have very small amounts of ROM; practically all of their software is read in from tape or disc and can be changed very easily. This way, if you tire of BASIC you can load in another language and use that instead. Most of us, however, are likely to stay with BASIC in ROM, where it is safe from the effects of ill-judged POKE commands!

As far as the MPU is concerned, software is a collection of bytes, the collection that we call machine code. A program written in machine code is just to our eyes a list of numbers, each one having a value between ∅ and 255. Some of these numbers may be instruction bytes, which cause the MPU to do something. Others may be the data bytes, which are numbers to add or store, or which may be ASCII codes. The MPU can't tell which is which, and it is entirely up to the programmer to make sure that everything is done correctly by putting the numbers in the correct order.

The correct order, as far as the MPU is concerned, is quite simple. The first byte that is fed to the MPU after switching on or after completing an instruction is treated as an instruction byte. Now some instructions consist of one byte only, and others need to have two or more bytes of further instruction or data following them. If you think of RND and TAB in BASIC, you'll remember that we can use RND by itself (though it can be multiplied by a number), but TAB must be preceded by PRINT and followed by a number in brackets. When the MPU receives an instruction byte, the only way that it can be instructed about what comes next is by the coding of that instruction byte. Instruction bytes therefore come in four types: (a) the ones that are complete in themselves, (b) the ones that need one extra byte of data, (c) the ones that need two extra bytes of data, and (d) some that need an extra instruction byte which may in turn be followed by data. Each instruction byte carries coding that the MPU can use to determine what comes next.

The snag is that the programmer must get it right. 100% right is just about good enough! Feed a microprocessor with an instruction byte when it expects a data byte, or with a data byte when it expects an instruction byte, and you have trouble in a big way. Trouble can mean an endless loop, which causes the screen to stop displaying any new information, the keys to have no effect (even BREAK), so that you have to switch off and start all over again. Another possibility is that the machine swings into its switch-on routine, clearing the memory and showing the copyright notice. In either case, you will often lose any stored program (it will always be lost if you have had to switch off) and the moral is that you should always save the program on cassette or microdrive before you try it out!

What I want to stress at this point is that machine code programming is tedious. It isn't difficult – you are drawing up a set of simple instructions for a simple machine – but it's often difficult for you to remember how much detail is needed. When you program in BASIC, the machine's error messages will help to keep you right and sort out mistakes, but when you write machine code you are on your own, and you have to sort out your own mistakes, though using an assembler helps considerably. Since the best way of learning about machine code is to write and use machine code, and learn from your inevitable mistakes, I'll devote the rest of this chapter to methods of writing machine code numbers before you even learn what the numbers mean and what commands are available, or how to use them. We'll start with ways of writing the numbers that constitute the bytes of a machine code program.

Binary, denary and hex

A machine code program consists of a set of number codes. Since each number is a way of representing the 1's and ∅'s in a byte of memory, it will consist of numbers between ∅ and 255 when we write it in our normal scale-of-ten (denary scale). The program isn't of any use until it can be stored in the memory of the Spectrum, because the microprocessor is a fast device, and the only way of feeding it with bytes as fast as it can cope with them is by storing the bytes in ROM or RAM and letting the microprocessor help itself to them in order. You can't possibly type numbers fast enough to satisfy the microprocessor, and even methods like tape or disc just aren't fast enough.

Getting bytes into memory, then, is an essential part of making a machine code program useful, and we shall look at methods in more detail in Chapter 5. At one time, simple and short programs were put into the memory of simple microprocessor systems by the most primitive possible method – having eight switches which could each be set to give a ∅ or a 1 output, and a button which caused the memory to read the binary number that had been set up on the switches. In addition, some method of addressing the memory was needed, and this could be provided by the microprocessor itself, as we shall see later.

Programming like this is just too tedious, however, and working with binary numbers is a process which can cause endless mistakes. Since every binary number is a set of ∅'s and 1's, after reading and entering a few dozen of them you start to make mistakes, exchanging ∅'s and 1's, repeating numbers, and all the other possibilities. The obvious step is to make use of the computer itself to place the numbers in the memory, and an equally obvious step is to use a more convenient number scale.

Just what is the most convenient number scale is a matter that depends on how you enter the numbers and how much machine code programming you do. A computer like Spectrum contains subroutines that convert binary numbers into a form that allows it to print denary numbers on the screen automatically, and also carry out the reverse transformation. When you use PEEK, therefore, the address that you use is in denary, and the result of the PEEK will be a denary number between ∅ and 255. When you use POKE, you can write both the address number and the byte number in denary. It makes sense, therefore, to work with the number system that we are used to and which the Spectrum uses for its own PRINT and input commands.

Serious machine code programmers, however, find this just too inflexible a system. By far the best way of entering machine code programs is to write them in what is called *assembly language*, which uses commands that are shortened words. Programs called *assemblers* can be written which will convert these commands into the correct binary codes. So that the programmer need never be bothered with the actual binary codes, many assemblers will show the codes on the screen in a form called hexadecimal, or hex. These assemblers will also require numbers to be typed in using hex code.

Hex codes

Hexadecimal means scale of sixteen, and the reason that it is used so extensively is that it is naturally suited to representing binary bytes. Four bits of binary digits, half a byte, will represent numbers which lie between ∅ and 15 on our familiar scale, which is the range of just one hex digit (Fig. 3.1). This means that a byte can be represented by

Hex	Denary
Ø	Ø
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	8
9	9
A	1Ø
B	11
C	12
D	13
E	14
F	15
then	
1Ø	16
11	17
to	
2Ø	32
21	33
etc.	

Fig. 3.1. Hex and denary digits.

two hex digits, and a two-byte address by four hex digits; it is, in addition, much easier to relate the form of a hex number to the pattern of bits in a byte than is possible when you use denary scale. In addition, the number codes that are used as the instruction bytes for microprocessors are generally written in terms of the hex scale and we can see the patterns of organisation in the hex numbers much more clearly, for example, when a set of related commands all start with the same hex digit.

At the time of writing this, an assembler had just become available for the Spectrum, and could be used with denary numbers, though it displayed the results of its actions in hex. Disassemblers such as the Campbell Software dpas, and the ACS Software INFRARED, show the codes and data numbers in hex, and it is fairly certain that as you progress with machine code you will find that you need to know about hex, even if you avoid using it. You can write your machine code programs for Spectrum entirely in denary numbers, but you will certainly find that a knowledge of hex will be necessary if you use a different machine. You will certainly find that the more advanced books on machine code programming, some of which are listed in Appendix A, assume that you are fluent in the use of hex.

The hex scale

The hexadecimal scale consists of sixteen digits, starting conventionally with Ø and ascending equally conventionally up as far as 9. The next figure up is not 1Ø, however, because this would mean sixteen (one sixteen and no units), and since we aren't provided with symbols for digits beyond nine, we have to make use of the letters A to F. The number that we write as 1Ø in denary (one ten, no units) is written as ØA in hex, eleven as ØB, twelve as ØC, and so on up to fifteen, which is ØF. The zero doesn't have to be written, but programmers get into the habit of writing a data byte with two digits and an address with four, even if fewer digits are needed. The next number after ØF is 1Ø, sixteen, and the scale then repeats to 1F, thirty-one, which is followed by 2Ø. The maximum size of byte, 255, is in hex terms FF. When we write hex, it is customary to write an 'H' after the code, so that there is no possibility of confusing the hex numbers with denary numbers. A number such as 16 could be sixteen (denary) or twenty-two (one sixteen and six units), but there is no doubt about 16H.

Now the great value of hex is how closely it corresponds to binary code. If you look at the hex-binary table of Fig. 3.2, you can see that 9 is 1ØØ1 in binary and F is 1111. The hex number 9FH is just 1ØØ11111 in binary – you simply write down the binary digits that correspond to the hex digits. The conversion in the opposite direction is just as easy – group the binary digits into fours, starting at the least significant bit (right hand side), and then convert each group into its corresponding hex digit. Fig. 3.3 shows examples of conversion in each direction, so that you can see how easy it is.

There are a lot of differences between computers in the way that they handle hex numbers. Some, like Spectrum make no provision at all for the use of hex, assuming that anyone who wants to carry out machine code programming in hex will use an assembler that deals in hex. Others, like the BBC Microcomputer, have a built-in hex translater; the BBC machine even has a built-in assembler.

Assuming for the moment that you do not use an assembler to create your machine code programs, what do you do? The answer is

Hex	Binary
Ø	ØØØØ
1	ØØØ1
2	ØØ1Ø
3	ØØ11
4	Ø1ØØ
5	Ø1Ø1
6	Ø11Ø
7	Ø111
8	1ØØØ
9	1ØØ1
A	1Ø1Ø
B	1Ø11
C	11ØØ
D	11Ø1
E	111Ø
F	1111

Fig. 3.2. Hex and binary numbers.

that you design your program in assembly language, which is by far the easiest way to design machine code programs, and then convert into denary code instead of typing the instructions into the assembler program. Converting means looking up, in a set of tables called the *instruction set*, the number which represents each instruction. Instruction sets that are provided by the manufacturers of microprocessor chips or by computer manufacturers are practically always in hex, but the Spectrum manual lists the codes in denary as well, though in numerical order rather than the alphabetical order that we need. Just to assist you, a complete list of Z-80 codes in alphabetical order has been included in this book in Appendix F, using denary, hex and binary forms. Don't refer to it at present – it'll put you off!

Most machine code programs, however, use data bytes as well as instruction bytes, and those are often shown in hex, though it is just as easy to show them in denary. Because of this, it's useful to be able to convert between denary and hex. If you are using the Campbell Software dpas Disassembler, you will then be able to make use of the denary-hex converter program which is part of the package. Otherwise, you will have to convert 'by hand' or use a simple conversion program each time you are working with these codes.

Conversion from hex to denary is illustrated in Fig. 3.4. The method for a one-byte number is very simple – take the denary value of the more significant digit, multiply by 16, and add the value of the other digit. Double-byte numbers such as address numbers are more tedious. If the number is a full four-digit one, the value of the most significant digit is multiplied by 16 × 16 × 16, which is 4Ø96 (all in denary). This value is noted, and the value of the next digit along is multiplied by 256, and also noted. The next digit value is then multiplied by 16, the result noted, and the least significant digit value is also written down. Finally, all of these values are added to get the final denary figure. Examples are shown in Fig. 3.4.

```
               Conversion: Hex to Binary

Example: 2CH ............. 2H is ØØ1Ø binary
                           CH is 11ØØ binary

So 2CH is ØØ1Ø11ØØ binary         (data byte)


Example: 4A7FH ........... 4H is Ø1ØØ binary
                           AH is 1Ø1Ø binary
                           7H is Ø111 binary
                           FH is 1111 binary

So 4A7FH is Ø1ØØ1Ø1ØØ1111111 binary (an address)


               Conversion: Binary to Hex

Example: Ø11Ø1Ø11 ............. Ø11Ø is 6H
                                1Ø11 is BH

So Ø11Ø1Ø11 is 6BH


Example: 1Ø11Ø1ØØ1ØØ1Ø ... note that this is
not a complete number of bytes.

   Group into fours, starting with lsb:
               ØØ1Ø is 2H
               1ØØ1 is 9H
               11Ø1 is DH and
    the remaining 1Ø is 2, making 2D92H
```

Fig. 3.3. Converting between hex and binary.

```
                 Hex to Denary Conversion

(a) Single bytes.
    Example: convert 3DH to denary.  The
value is 3 * 16 + 13 (denary D) which is
61 denary.
    Example: convert A8 to denary.  The value
is 1Ø * 16 + 8, which is 168 denary.

(b) Double bytes.
    Example: convert 2CA5 to denary.  The
first digit gives 2 * 4Ø96 = 8192.  The second
digit gives 12 * 256 = 3Ø72, and the third
digit gives 16 * 1Ø = 16Ø.  The last digit is
5, and adding 8192 + 3Ø72 + 16Ø + 5 gives 11429.
    Example: convert F3DBH to denary.
         1st digit .. 15 * 4Ø96 = 6144Ø
         2nd digit ... 3 * 256  =   768
         3rd digit .. 13 * 16   =   2Ø8
         4th digit .. 11        =    11
         Sum .................. = 62427
```

Fig. 3.4. Hex to denary conversion, single or double byte.

Denary to hex is not so simple. A single byte number is dealt with by division by 16, which will give a whole number part (the integer part) and a fraction. The whole number part is converted into a hex digit, and the fractional part by itself is multiplied by 16, and the result converted into another hex digit. For address numbers, division by 16 will result in an integer part which is greater than 16. Convert the fractional part into a hex digit by multiplying the fraction by 16, and then treat the integer part in the same way again, dividing by 16, writing down the integer, and converting the fractional part into a hex digit. Continue until the integer part is less than 16, and then write down its hex value. This method finds the digits in order starting at the least significant digit. Fig. 3.5 shows some examples of the conversion.

```
                 Denary to Hex Conversion

(a)  Single bytes - numbers less than 256
denary.
     Example: convert 153 to hex.
153/16 = 9.5625 so 9 is upper digit, lower
digit is Ø.5625 * 16 = 9, so that number is
99H.
     Example: convert 58 to hex.
58/16 = 3.625, so 3 is upper digit, lower
digit is Ø.625 * 16 = 1Ø, A in hex.  So that
number is 3AH.

(b)  Double bytes - number between 256 and 65535.
     Example: convert 23815 to hex.
23815/16 = 1488.4375.  Ø.4375 * 16 = 7, lowest
digit.
1488/16 = 93, Ø remainder - Ø is next digit.
93/16 = 5.8125.  Ø.8125 * 16 = 13, hex D.  Last
digit is 5, so that the complete number is 5DØ7H.
```

Fig. 3.5. Denary to hex conversion, single or double bytes.

It's simpler to use programs for the conversion. Denary to hex can be done very simply by successive divisions by 4Ø96, 256, and 16, converting the whole number part of the answer to hex digits each time. The conversion is done by using ASCII codes, because when your Spectrum prints a number in hex, it must be printed as a string, because it includes letters as well as number digits. As it happens, there is a fairly simple relationship between the ASCII codes for digits Ø to 9, and the numbers themselves. If you add 48 to the number, then you have the ASCII code for that digit, which can be printed as a string character. A slight complication arises when we get to ten, because in hex this is A, and the ASCII code for A is 65, which is 55 greater than 1Ø. The conversion program must therefore add 48 to a digit of 9 or less and 55 to a digit of ten to fifteen to make the correct conversion to hex code. This isn't difficult for a BASIC program, and an example of denary to hex conversion program is shown in Fig. 3.6.

Hex to denary can be done in much the same way, converting the ASCII code for each digit into the number digit itself, multiplying for the correct place factor (16,256,4Ø96), and then adding. The conversion can use a loop for both the multiplication and the addition, to obtain the denary number. Once again, the BASIC program is fairly simple (see Fig. 3.7) which is why the '£5 per published letter' pages of magazines are always choked with denary-hex conversion programs for each new computer.

Throughout this book, we shall use mainly denary codes, with a few hex values thrown in where they are needed. It's only fair to point out, though, that there are parts of machine code programming

```
1Ø  CLS: PRINT "Please type denary number_":
    INPUT d
2Ø  IF d>65535 THEN PRINT " Too large - limit
    is 65535": PAUSE 5Ø : GOTO 1Ø
3Ø  IF d<1 THEN PRINT "No numbers less than
    unity, please": PAUSE 5Ø: GOTO 1Ø
4Ø  IF d<>INT d THEN PRINT "No fractions,
    please": PAUSE 5Ø: GOTO 1Ø
5Ø  LET f = 4Ø96: LET h$ = ""
6Ø  LET y = INT (d/f)
7Ø  GO SUB 2ØØ
8Ø  LET d = d - y * f : LET f = INT (f/16)
9Ø  IF f<1 THEN GOTO 11Ø
1ØØ GOTO 6Ø
11Ø FOR n = 1 TO 3: IF h$ (1) = "Ø" THEN LET h$
    = h$ (2 TO )
12Ø NEXT n
13Ø PRINT "Hex number is "; h$ + "H"
14Ø GOTO 9999
2ØØ IF y<= 9 THEN LET h$ = h$ + CHR$  (y + 48)
21Ø IF y>9 THEN LET h$ = h$ + CHR$ (y + 55)
22Ø RETURN
```

Fig. 3.6. A BASIC program for denary to hex conversion.

which do need some working at in terms of mastering arithmetic methods. The most important of these is the way of representing negative numbers.

Negative numbers

We represent negative numbers in denary by the use of a negative sign, so that we can write values like +15 and −15, using the same digits but with the signs + and − showing whether the values are positive or negative. Microprocessors make no provision for the use of these signs, so that binary code has to use one bit from a byte to represent the sign, and the bit that is always chosen is the most significant bit (left hand side). The convention that is followed is that if the most significant bit is a 1, then the sign of the byte is negative, and if the most significant bit is a Ø, then the sign is positive. It's a simple convention, and as it happens a particularly useful one, but it does have disadvantages for human operators. One of these disadvantages is that the digits for a negative number are *not* the same as those of its positive counterpart: +5 in binary is ØØØØØ1Ø1 but −5 is 11111Ø11, which doesn't look like the same number. A second disadvantage is that using one bit as a sign bit leaves fewer bits for representing the number value. If we use the highest bit of a single byte number as a sign bit, then the remaining 7 bits can represent numbers only as high as +127. We can, of course, also represent a negative number down to −128, so that we haven't lost anything from the range of numbers that we can represent. If we use two bytes, then losing one bit to use as a sign bit means that the range available is −32768 to +32767 in denary. Using five bytes for each number, as Spectrum does, means that a single bit used for sign purposes does not greatly affect the range of numbers that we can use.

```
1Ø  CLS: LET y = 1 : LET d = Ø : PRINT "Please
    type hex number" : INPUT h$
2Ø  IF LEN h$>4 THEN PRINT "Too long - maximum
    of four "'" characters please": PAUSE 1ØØ :
    GOTO 1Ø
3Ø  LET p$ = h$  (LEN h$): LET h$ = h$ (1 TO
    (LEN h$ - 1))
5Ø  GO SUB 2ØØ : IF LEN h$>Ø THEN GOTO 3Ø
6Ø  PRINT "Denary number is "; d
1ØØ GOTO 9999
2ØØ LET a = CODE p$
21Ø IF a<48 OR a>1Ø2 THEN GO SUB 3ØØ
22Ø IF a<65 AND a>57 THEN GO SUB 3ØØ
225 IF a<= 97 AND a > 7Ø THEN GO SUB 3ØØ
23Ø IF a<= 57 THEN LET q = a - 48
24Ø IF a>= 65 THEN LET q = a - 55
25Ø IF a>= 97 THEN LET q = a - 87
26Ø LET d = d + q * y : LET y = y * 16
27Ø RETURN
3ØØ PRINT "Bad hex ... please try again ":
    PAUSE 1ØØ: RETURN
```

Fig. 3.7. A BASIC program for hex to denary conversion.

The third disadvantage is that human readers cannot distinguish between a single byte number which is negative, and one which is written with no regard for sign. The short answer is that the human operator doesn't need to worry – the microprocessor will use the number in the same way no matter how we happen to think of it and, in this book, you will soon see how the conversion is made and used and some practise will make you completely confident with the methods.

Most of our applications of negative numbers call for single byte numbers to be used, so we shall look at conversions to negative form for just this range. The binary number conversions are the basis of all the others, so we shall look at them first, though we may not use them to any great extent. To start with, we can't form a negative version of a single-byte number whose denary value is more than +127, or negative value is less than 128, because such numbers require more than one byte. The positive version of the number is written in eight-bit form, and this may mean padding it out with some Ø's on the left hand side. If the most significant bit is not zero, the number won't convert.

Each bit is then inverted. This means writing a 1 in place of each Ø and a Ø in place of each 1, as illustrated in Fig. 3.8. The resulting number, called the complement, has 1 added to it, and this gives the negative form or 'two's complement' as it is also known. The illustration in Fig. 3.8 shows the process in action, and bears out the point that the binary number in negative sign form is nothing like its positive counterpart. Note the process of binary addition, where 1+Ø = 1, 1+1 = Ø and carry 1, and 1+1+ carry 1 = 1 and carry 1.

What it amounts to in denary terms is that a single byte number ranging between 128 and 255 is negative, and numbers of 127 and less are positive. To find the equivalent value of a negative number in denary terms, simply subtract the value of the number from 256 (Fig. 3.9 shows examples). This is by far the easiest way to handle these negative numbers in the only point where we have to, which is in jump-relative instructions, dealt with later.

```
Binary number ØØ11Ø11Ø      Denary 54
inverted .... 11ØØ1ØØ1
add 1 ....... 11ØØ1Ø1Ø      Denary -54


denary number  -5
In binary this is 1Ø1, and in eight-bit binary
                  is ØØØØØ1Ø1
Inverted, this is ... 11111Ø1Ø
Add 1 ............... 11111Ø11  which is byte
for -5
```

Fig. 3.8. Forming the two's complement (negative form) of a binary number.

```
Denary number -5   Equivalent byte, in denary,
is 256 - 5 = 251
Denary number -8   Equivalent byte, in denary,
is 256 - 8 = 248
```

Fig. 3.9. The denary equivalent of single-byte negative binary numbers.

The hex equivalent of negative binary numbers can be found from either the binary or the denary versions. Some microprocessor actions will treat any number greater than 127 denary as a negative number (7FH), causing the equivalent of a subtraction when it is added to any other number. Other microprocessor actions will treat all numbers as unsigned, meaning that no importance will be attached to the sign bit which will be counted as a number of 128's. The only problem arises when you are trying to find out what the microprocessor has done. In general, when you read in an instruction set that a number is treated as 'signed', this indicates that the most significant bit will be treated as a sign bit. When the number is said to be 'unsigned', it will be treated simply as a binary number, with a byte able to represent positive numbers between Ø and 255.

Chapter Four
Z-80 Details

Registers - PC and accumulator

A microprocessor consists of sets of memories, of a rather different type compared with ROM or RAM, which are called *registers*. These registers are connected to each other and to the pins on the body of the MPU by the circuits called gates. In this chapter, we shall look at some of the most important registers in the Z-80 (identical to the Z-80A and Z-80B) and how they are used. A good starting point is the register called the PC (or *Program Counter*).

The PC is a sixteen-bit register which can store a full-sized address number, up to FFFFH or 65535 denary. Its purpose is to count instruction bytes (not programs!), so that the number that is contained in the register will be incremented (increased by 1) automatically each time an instruction byte is read. This is a completely automatic action which doesn't need any instructions from the programmer, because it's built into the action of the Z-80. The PC register will start at a count of zero when the Z-80 is first switched on, so this is where the first instruction of the ROM will start.

The usefulness of the PC is that it is the method by which memory is addressed. When the PC contains an address, the electrical signals corresponding to the Ø's and 1's of that address appear on a set of connections, collectively called the address bus, which link the microprocessor to all of the memory, RAM and ROM. The number which is stored in the PC register therefore selects one byte in the memory, the byte which has that address. At the start of an instruction, the microprocessor will send out a signal called the *read signal* on another line, which will cause the memory to connect its stored bits to another set of lines, the *data bus*. The signals on the data bus therefore correspond to the pattern of Ø's and 1's stored in the byte of memory that has been selected by the address in the PC.

Each time the number in the PC changes, another byte of memory is selected, so this is the way that the microprocessor can keep itself fed with bytes, incrementing the number in the PC each time a byte has been read.

There are other ways in which the PC number can be changed, but for the moment we'll pass over that and look at another register, the accumulator. The accumulator is the main 'doing' register of the MPU, meaning that you would normally fill it by copying a byte from memory (loading the accumulator), or use it to write to memory (loading memory from the accumulator). The memory byte which is used in each case will be the one whose address is stored in the PC at the time.

As the name suggests, the accumulator also holds the result of operations. If you have a number byte stored in the accumulator, you can add another number to it, and the result will be stored back in the accumulator. It's as if you had a number variable called Total, and you wrote the BASIC line:

```
LET Total = Total + extra
```

where extra is a number that you add to Total. The difference, and it's an important difference, is that the accumulator can't store a number greater than 255 because it is a single-byte register.

The importance of the accumulator is that there are more instructions which affect or use this register than any of the many other registers in the Z-80. When a byte is read into the MPU, it is generally read into the accumulator. When arithmetic is done it is normally done in the accumulator. When a byte is stored in memory it is usually stored from the accumulator. Unlike earlier designs of microprocessors, the Z-80 has a large number of registers which can be used in much the same way as the accumulator, but none of them has the same range of possible operations.

Addressing methods

When we program in BASIC, we don't have to worry about memory addresses - these are taken care of by the operating system in the ROM. When a variable is allocated a value in a BASIC program as, for example, by a line like:

```
LET n = 12
```

we never have to worry about where the number 12 is stored.

Similarly, when we write:

    20 LET k = n

we don't have to worry about where the value of n was stored so as to copy it into k. Remembering our comparison with wall building, we can expect that when we carry out machine code programming we shall have to specify each number that we use or alternatively the address at which the number is stored. This latter is referred to as the addressing method. What makes it particularly important is that a different code number is needed for each different addressing method used for each command. This means that each command exists in a number of versions, one code for each possible addressing method. A list of all the possible Z-80 addressing methods at this stage is rather a baffling document, and for that reason has been consigned to Appendix D. What we shall do here is to look at examples of some addressing methods and the way that we indicate them in assembly language.

Assembly language

Trying to write down machine-code directly as a set of numbers is a very difficult and error-prone activity. The most useful way of starting to write a program is to write it as a set of steps in what is called assembly language (or assembler language), which is a set of abbreviated words called *mnemonics*, and numbers which will be data or address numbers. The numbers can be in hex or in denary. Each line of an assembly language program indicates one microprocessor action, and this set of brief instructions is later 'assembled' into machine code, hence the name.

The aim of each line of an assembly language program is to show the action and the data or address that is needed to carry out that action, just as when we program a TAB in BASIC we need to complete it with a number. The part of the assembly language that specifies what is done is called the *operator*, and the part which specifies what the action is done to or on is called the *operand*. A few instructions need no operand, and we'll look at some of them later, but for the most part, the operand for Z-80 instructions consists of two parts, one which specifies a register and another which specifies a data byte or an address.

An example makes this easier to follow. Suppose we look at the assembly language line:

    LD A,12

The operator is LD, a shortened version of LOAD, which is used for each transfer or copying of a byte from one address to another or one register to anywhere else. A is the first part of the operand, and the abbreviation means *Accumulator*. Its placing immediately after the operator means that this is a destination for a byte, the place where the byte will end up at the completion of the instruction. The other part of the operand, following the comma, is the number 12. Generally, when the number is written in this way it means denary 12 rather than hex 12, which would be typed as 12H.

The whole line, then, should have the effect of placing the number 12 into the accumulator register of the Z-80. It is the equivalent in machine code terms of the BASIC command:

    LET a = 12

if we could imagine that the number variable 'a' existed as a hardware circuit inside the MPU, rather than as an entry in the memory which we know it is.

A command such as LD A,12 is said to use *immediate addressing*, because the byte which is loaded into the accumulator is placed in the memory address which immediately follows the operator code. There is one single byte code for the LD A part of the line, and this byte is 62 (3EH), so that the sequence 62,12 in memory will represent the entire command LD A,12. It's a lot easier to remember what LD A,12 means than to interpret 62,12, however, which is why we use assembly language as much as possible.

Immediate addressing can be convenient, but it ties you down to the use of a definite number, like programming in BASIC:

    LET n = 4*12 + 3

rather than

    LET n = a*b + c

In the first example, n can never be anything but 51, and we might just as well have written:

    LET n = 51

The second example is very much more flexible, and the value of n depends on what we choose for the variables a, b and c. When a machine code program is held in RAM, then the numbers which are loaded by this immediate addressing method can be changed if we want them changed, but when the program is held in ROM no change is possible, and that's just one reason for needing other addressing methods. One of these other methods is *direct addressing*.

Direct addressing uses a complete two-byte address in the operand. This creates a lot of work for the Z-80, because when it has read the code for the operator and the first part of the operand (one byte of code) it will then have to read the next two bytes immediately following the instruction byte, place these two bytes of address in the PC, read in the data byte from this address, deal with it, and then restore the PC address to its correct value (Fig. 4.1). A direct operation is therefore slow, and needs a lot of bytes of memory to store all the bytes that are needed.

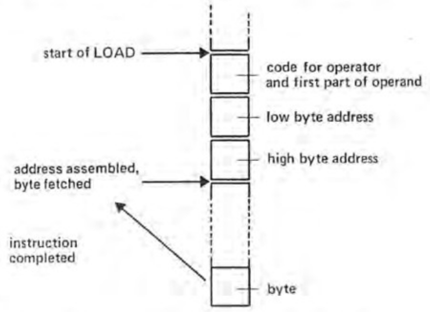

start of LOAD
code for operator and first part of operand
low byte address
high byte address
address assembled, byte fetched
instruction completed
byte

Fig. 4.1. How the bytes of a direct-addressed command are stored in the memory.

Suppose, for example, that we have the instruction:

LD A,(7FFFH)

which appears in lists as LD A,(NN), where NN means a full address. In assembly language, the operator is LD, load, and the destination for the byte is the accumulator A. The source of the byte is the address 7FFFH (denary 32767), and the assembly language uses the brackets to mean 'contents of'. It's a way of reminding yourself that what is put into the A register is not 7FFFH (which wouldn't fit) but *the byte which is stored at this address*. The effect of the complete instruction, then, is to place a copy of the byte stored at address 7FFFH into the accumulator. When the instruction has been completed, the address 7FFFH will still hold its own copy of the byte because reading a memory does not change the content of memory in any way.

The way in which operands are written in assembly language follows the order destination, source, so that if we write:

LD (7FFFH),A

then this means that the byte stored in the accumulator is copied to the address 7FFFH. Note the use of the brackets again. Some types of microprocessors use the word ST (store) for this type of action, but the Z-80 uses only the order of writing the quantities and symbols.

Indirect addressing

Immediate addressing and direct addressing (also called extended addressing) are useful, but the Z-80 also permits a very handy form of what is called *indirect addressing*. Indirect addressing means going to an address, an address pair in fact, to find another address, and then using this second address to find or send the data byte. It's rather like going to a travel agency to find the address of a hotel in which you can take a room (or eat a bite?). The form of indirect addressing which the Z-80 uses is called register-indirect, and it makes use of some of the other registers in pairs.

The use of eight-bit registers in pairs to hold a full sixteen-bit address is a special feature of the Z-80, and one which helps to make it such a popular microprocessor with designers of computers that are intended for business and other serious uses. There are three sets of such registers which are labelled as HL, BC and DE respectively and, of these, the HL pair are the ones used most frequently for this purpose. All of these registers can be used singly, incidentally, just as if they were spare accumulators, but with a more limited range of actions.

We can load a complete address into the HL pair of registers by using a command which is written in assembly language in the form:

LD HL,32767 or LD HL,7FFFH

This means that the high byte of the address, 7FH in the hex version, will be held in the H register (H for high), and the low byte FFH will be stored in the L register (L for low). The names, as you can see, have been picked deliberately to remind you of which byte is stored where. We can then load the accumulator (or another register) with the byte which is stored at this address 7FFFH by using the command:

```
LD A,(HL)
```

or we can store a byte which is in the accumulator to the address 7FFFH by using:

```
LD (HL),A
```

This is another example of how the order of writing the operands is used to indicate which is source and which is destination; the brackets have also been used in their usual meaning of 'contents of'.

Now you might think that this is just a rather long-winded way of writing the command LD A,(7FFFH), but there is an important difference. Once an address number has been placed in HL, we can increment or decrement that address with a single-byte (no operand) instruction. If we have loaded HL with the address 7FFFH, then the instruction:

```
DEC HL
```

will cause the address number stored in HL to become 7FFEH (denary from 32767 to 32766), so that if we use:

```
LD A,(HL)
```

again, the accumulator is loaded from address 7FFEH(32766) rather than from 7FFFH(32767). If you are familiar with the idea of a loop in BASIC, then you can see how instructions like this can be used in a loop to allow a different address to be used on each pass through the loop, decrementing HL on each pass round the loop as in this example or, of course, incrementing if this is what is needed.

PC-relative addressing

PC-relative addressing is one of the simplest and most primitive methods of obtaining an address in the PC. The operand contains a byte called the displacement, which is added to the address which is stored in the PC of the microprocessor, and the resulting number is then the address which is used for the load, store or whatever is being carried out. Early types of microprocessors used PC-relative addressing for practically all of their actions, but the Z-80 uses it only for one small set of instructions – the jump-relative (JR) instructions.

A jump-relative means a transfer to a new address using a PC-relative addressing method. The complete JR instruction consists of two bytes, the operator JR and the operand, which consists of a condition and the displacement. The instruction needs some care and experience, though, because the displacement byte is treated as a signed byte, meaning that if the most significant bit of the byte is 1 (denary value 128 or more) then the byte is treated as a negative number, and the address which is obtained when this byte is added will be *lower* than the address which existed in the PC before the JR instruction. In terms of denary numbers, then, if the byte is 127 or less, there will be a jump forward; if the byte is 128 to 255, there will be a jump backward. When the jump of address has occurred, the PC will then resume normal action from that address and will not return to its previous address unless by the action of incrementing (if the jump was back) or by another jump (if the jump was forward).

When we use PC-relative addressing, we will have to work out the value of the displacement byte. When an assembler program is used to convert the assembly language into code, the displacement byte will be calculated by the assembler program, but when you assemble 'by hand', then you will have to calculate it for yourself. The calculation is like this:

(1) Write down the address at which the operator byte of the JR instruction will be placed. This is the source address.
(2) Write down the address to which the program must jump, which is the destination address.
(3) Subtract source address from destination address, and then subtract two from this answer. What you now have is the displacement in denary terms. If it is positive, use it directly. If it is a negative number, subtract the value from 256, and use the result as the displacement.

Why subtract 2? It's because the displacement is always calculated from the operator address, but the jump can't take place until the operand containing the displacement has been read (which means that the PC will have incremented), and in addition, the PC will increment automatically at the end of the instruction. By subtracting 2, we allow for these two incrementing actions, and obtain the correct displacement byte. Fig. 4.2 shows some examples of positive

```
Source address 32542
Destination address 32565  Difference is 23
                           Subtract 2 to get 21
                           21 is displacement
                           number

Source address 32533
Destination address 32504  Difference is -29
                           Subtract 2 to get -31
                           256 - 31 = 225 is
                           displacement number
```

Fig. 4.2. Positive and negative displacements for a program-counter relative addressed instruction.

and negative displacements. It's often easier, if you haven't decided on address numbers, just to count bytes between source and destination. Note that PC-relative addresses cannot cause a jump to more than 127 places forward, or 128 places back from the PC address of the operator, because this is the complete range of a signed byte.

The other Z-80 registers

We've mentioned a number of the Z-80 registers already. The PC is the addressing register, which keeps a count of the address of each instruction byte and data byte of the program – it's the 'where-are-we-now' register. When a machine code program is run, the address of the first byte of the program must be placed into the PC, and the microprocessor will then take over quite automatically, so we need some method of placing an address into the PC. We'll look at the Spectrum methods of doing this later, but generally we leave the PC alone once the program has started.

The accumulator is the register in which most of the work is done, and we'll look at some of the accumulator operations in detail in the next chapter. There are six other single-byte registers, labelled as B,C,D,E,H and L which can be used rather as we use the accumulator itself, though none of them offers quite such a large range of actions. In addition, as we have seen, these registers can be grouped as BC,DE and HL to store complete 16-bit numbers, such as addresses. A limited number of 16-bit arithmetic operations are also possible in the HL register pair.

This does not exhaust the register count of the Z-80. There are two registers which we seldom, if ever, use – the interrupt (I) and the refresh (R) registers whose uses are rather specialised. Another single byte register, however, the *status* or *flag register*, is of very great importance despite the fact that we cannot store or load it directly. The status or flag register is a way of keeping a score of results. When an arithmetic operation like addition or subtraction, or a logic operation like AND, OR or XOR, is carried out, the result in the accumulator may be a number that is positive, negative or zero. This 'status' of positive, negative or zero is indicated by the state (∅ or 1) of bits in the *status register*. The status register is not particularly well named, because it's not really a register which can store a number, but just a collection of single bit stores with no connection between them. Some books call this the *flag register* because this is such a descriptive name – a flag is raised each time one of these results is available, and stays raised until a new result appears.

7	6	5	4	3	2	1	∅	bit position
S	Z	X	H	X	P/V	N	C	

Carry flag—set if there is a carry or borrow in arithmetic
N-Flag—set for subtract operation
Z-Flag—set by zero result of some operations
S-Sign flag—set if result is negative

C—carry flag
N—add/subtract flag
P/V—parity/overflow flag
H—half-carry flag
Z—zero flag
S—sign flag
Z—not used

(Other flags have rather specialised uses)

Fig. 4.3. The Z-80 status register. Bits 3 and 5 are not used (they may be at either ∅ or 1 permanently).

Fig. 4.3 shows the layout of the Z-80 status register. The bits that we will be most interested in are the S,Z and C flags, bits 7,6 and ∅ respectively. The S-flag is 1 when the byte in the accumulator is negative after an arithmetic/logic operation; the flag value will be ∅ if the accumulator byte is positive or zero. The Z-flag is 1 if the accumulator contains zero after any of the operations that affect the flags but it will be ∅ otherwise. The C flag is 1 if there has been a carry resulting from an addition, or if a borrow is needed in a subtraction. Any register which can be used for the operations that affect the flags will also be signalled by the flags, so that if you are working with the C registers and a subtraction causes a zero to appear, this will be signalled by the Z-flag in the status register just as if the action had taken place in the accumulator.

The status register can't normally be affected by the programmer, because it exists to signal precisely the results of actions. The programmer very seldom knows directly what is stored in the status register, and its importance lies in the fact that it controls jumps. To make a comparison with BASIC again, suppose we programmed in BASIC:

```
1ØØ IF a = Ø THEN GOTO 3ØØ
```

where a 'jump' to line 3ØØ is carried out if the value assigned to variable a happens to be zero. We may not know at that instant what the value of a will be, only that the jump will take place when a is zero. The machine code version of this, in assembly language, is:

```
JR Z,Disp
```

where JR is the jump-relative operator, Z is the part of the operand which indicates what flag is used as a condition for the jump, and Disp means the single-byte displacement number which will direct the jump, if it is taken, to the correct address. When this action is carried out by the microprocessor, the single byte of code that represents the JR Z part of the command causes the microprocessor to check the status of the Z-flag in the status register. If the previous arithmetic/logic action left the accumulator (or whatever register was being used) at zero, then the Z-flag will be set (meaning it will be a 1 bit), and the displacement byte will be added to the PC address to cause the jump to be carried out. If the Z-flag is not set (reset, at Ø), then the displacement is ignored, just as the instructions following THEN in BASIC are ignored when the condition is not met. In the BASIC example, if variable a is not zero, the program does not GOTO 3ØØ. In the machine code example, the next address held in the PC will be the one (Fig. 4.4) which follows the address of the displacement byte rather than the PC + displacement address that we would find if the condition was met by having the accumulator (or other register) zero.

```
Address - JR Z, disp

Z-flag set - jump to new address given by address
of JR instruction + displacement + 2

Z-flag not set - ignore displacement and move to
instruction following JR
```

Fig. 4.4. A conditional jump instruction.

One peculiarity of the way in which the Z-80 uses its status register is that only certain actions, particularly arithmetic and logic actions, actually affect flags. Load and store operations do not affect flags, so if you have previously used 6502 machine code, you will have to adjust your thinking! For example, suppose we have a piece of assembly language that reads:

```
LD A,(HL)
DEC A
LD A,(DE)
JR Z,Disp
```

If the DEC A (decrement the accumulator) action, which is one that can affect flags, causes the contents of the accumulator to become zero, then the jump at the JR Z,Disp stage will be carried out, even though the accumulator has been reloaded from DE and probably does not contain a zero byte any longer. This is very much a Z-80 peculiarity, and one that can at times be very useful.

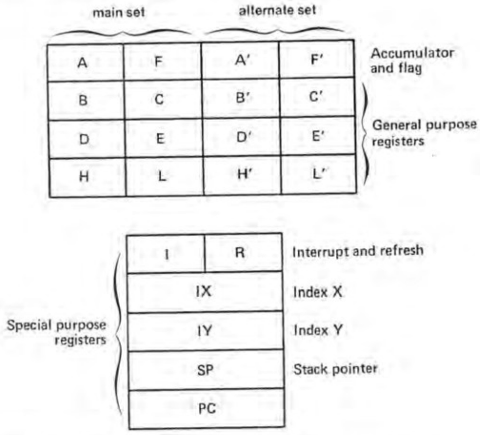
Index registers

The Z-80 also contains two index registers. These are 16-bit registers, so that they can hold a full 16-bit address number, and they are labelled as IX and IY. The reason for the word 'index' is that these registers can be used in a form of addressing that is similar to the action of a book index. An address called the *base address* or *page address* is held in an index register, and any address up to 127 numbers higher or 128 numbers lower than this base address can be used by adding a displacement byte to the index address. In assembly language, this is written as (IX + d) or (IY + d), where d means the single byte displacement. We can use commands such as

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

in this way, meaning that the accumulator will be loaded from the address which is equal to the base address stored in IX plus the displacement. The IX register would have to have been loaded earlier in the program. We shall not elaborate on this very brief description, because the IX and IY registers are used very extensively in the operating system of the Spectrum, and the manual cautions against their use in machine code programs.

Fig. 4.5 shows a 'map' of the Z-80 registers to give an overall view of what is available. As well as the main set of A,F,B,C,D,E,H, and L registers, there is also an 'alternate set' which can be used. The alternate registers are labelled as A',F',B',C',D',E',H', and L'. These can be selected to be used in place of the main set by register exchange commands – you can't use the main and the alternate registers at the same time.



The main and alternate registers can be exchanged by instructions such as EX AF, A F' and EXX.

Fig. 4.5. The Z-80 registers. The alternate set cannot be used at the same time as the main set.

Chapter Five

Register Actions

Accumulator actions

Since the accumulator is the main single-byte register, we can list its actions, and describe them in detail, knowing that the description will also hold true for any of the other single-byte registers that can be used in the same way. Of all the single-byte register actions, simple transfers of bytes are by far the most important. We don't, for example, carry out any form of arithmetic on ASCII code numbers, so that the main actions that these require of the microprocessor are fetching and storing – loading the accumulator from one memory address and storing the byte back at another address. The systems in which the Z-80 is used do not permit a byte to be copied directly from one memory address to another, so that the rather clumsy-looking method of loading from one address and storing to another is used almost exclusively.

The next most important group of actions is the arithmetic and logic group, which contain addition, subtraction, AND, OR and XOR and NOT. We can add two others to this group: SHIFT, which causes all the bits of a byte to move along one place (you have to specify in the command whether you want left shift or right shift, among other details), and ROTATE, which connects the ends of the register before carrying out a shift (see Fig. 5.1).

The effect of the main shift and rotate commands, with their assembly language mnemonics, is shown in Fig. 5.2. A shift always results in the register losing one of its stored bits, the one at the end which is shifted out, and gaining a zero at the other end. A rotation, by contrast, keeps the bits stored in the register, but changes the position of the bits in the byte, though preserving the relative order. These instructions are used mainly for selecting half-a-byte (one hex digit) for arithmetic purposes, or for sending one bit at a time out from, or reading one bit at a time into the register as is needed in cassette recording or replaying operations. Fig. 5.3 summarises the arithmetic and logic group of commands, and their assembly language mnemonics.

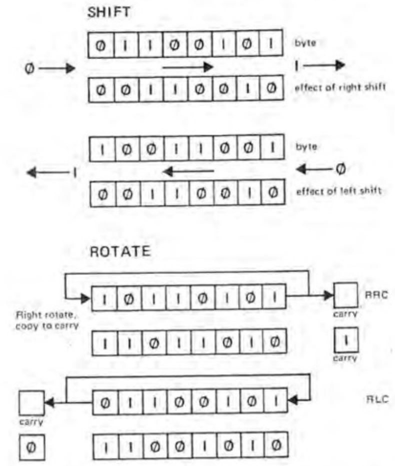


Fig. 5.1. The effect of SHIFT and ROTATE commands. This has been simplified – the Z-80 SHIFT and ROTATE commands often involve the bit called the carry flag as well as the register contents.

A third group is comprised of increment, decrement, and comparison. Increment and decrement mean adding and subtracting, respectively, the number one; and when they are applied to a single register, mean that the number stored in that register will be incremented or decremented. When one of the double registers is used, then INC and DEC can be used in two ways. The command INC HL, for example, will increment the number that is stored in the HL register pair, just as INC A will increment the number stored in the A register. The command INC (HL) will, however, increment the byte which is stored in the address held in the HL pair, without any effect on the address itself, so that this single instruction can replace the set:

```
LD A,(HL)   ; put byte into accumulator
INC A       ; increment the byte
LD (HL),A   ; put it back at the same address
```

SLA Left shift, with MSB shifted to carry flag
SRA Right shift, MSB not changed, LSB to carry flag
SRL Right shift, zero to MSB, LSB to carry flag
RLD and RRD not used to any extent
RLCA Left rotation of accumulator, bit 7 copied to carry flag
RLA Left rotation of accumulator, including carry
RRCA Right rotation of accumulator, LSB copied to carry
RRA Right rotation of accumulator, including carry
RLC Left rotation of register, MSB copied to carry
RL Left rotation of register, including carry
RRC Right rotation of register, LSB copied to carry
RR Right rotation of register, including carry

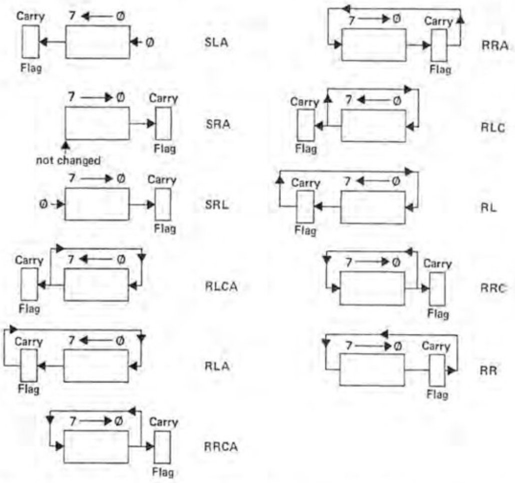


Fig. 5.2. The Z-80 main shift and rotate commands, with their effects and the mnemonic codes.

ADD A, r	Add the byte in the register r to the byte in the accumulator. Store result in accumulator, with any carry used to set the carry flag of status register.
ADC A, r	Add the byte in the register to the byte in the accumulator, and add in any carry, as indicated by the carry flag. Store result in the accumulator, and if there is another carry, then set the carry flag, otherwise reset the flag.
SUB r	Subtract the byte from the byte in the accumulator, and store the result back in the accumulator. Set the carry flag if there has been a borrow.
SBC A, r	Subtract the byte in a register, and the carry bit also, from the byte in the accumulator. Store the result back into the accumulator, and set the carry flag if there has been another borrow.
AND r	AND the byte represented by r with the byte in the accumulator, store the result in the accumulator. Affects S and Z flags mainly.
OR r	OR the byte with the byte in the accumulator, and store the result in the accumulator. Affects S and Z flags mainly.
XOR r	XOR the byte with the byte in the accumulator, and store the result in the accumulator. Affects S and Z flags mainly.
ADD HL, rr	Add contents of register pair to contents of HL pair. Result stored in HL pair. Set carry flag if there is a carry from the MSB of H.

Fig. 5.3. The arithmetic and logic group of instructions – samples only!

ADC HL, rr	Add with carry bit the byte in a register pair to the bytes in HL. Result stored in HL. Carry flag set if there is a carry out.
SBC HL, rr	Subtract the contents of the register, and the carry bit, from the contents of HL. Store result in HL, set carry if a borrow takes place.

NOTE r has been used to indicate a register, such as B, C, D, E, etc., or an immediate-loaded byte, or the contents of an address or an address held in a register pair, such as (HL). Not all addressing methods will be available for each command.

Fig. 5.3 Continued

Note the way that comments are put into the assembly language statements, treating the semicolon like a REM in BASIC – anything following the semicolon is a comment and not part of the instruction.

When a single register is incremented or decremented, the result will affect the flags in the status register, so that the S-flag will be set (to 1) if the result is negative, reset (to Ø) if the result is positive or zero, and the Z-flag will be set (to 1) if the result is zero. When the byte whose address is held by a register pair is incremented or decremented using INC (HL) or DEC (HL), the same applies; flags will be set or reset according to the result of the action. When a double register pair is incremented or decremented, however, by such instructions as INC HL or DEC HL, then the status register is *not* affected. This is a quirk of the instruction set of the Z-80 which is often rather a nuisance, but there isn't much we can do about it. Ways of programming so as to set flags will be dealt with later.

CP, the mnemonic for compare, is a particularly useful form of arithmetic instruction. CP has to be followed by a byte, a register, or some source of a byte such as (HL) or (address), and it compares the byte that is fetched with the byte that already exists in the accumulator. Comparing is rather like subtraction, but there is one vital difference. If, for example, we had the byte 5Ø in the accumulator, then SUB 4Ø would have the effect of leaving 1Ø in the accumulator, SUB 5∅ would leave zero in the accumulator, and set the zero flag in the status register. CP 4∅ would do nothing very noticeable, and CP 5∅ would set the zero flag, but neither of these CP instructions would alter the value of the byte in the accumulator. CP allows us to find out what a subtraction would do without altering the accumulator, and we can make use of it to set flags for jumps which can then make use of the unchanged byte in the accumulator.

Finally, we have the test and jump set of instructions. There are two groups, the absolute jumps and the relative jumps; the absolute jumps are written in assembly language as JP and the relative jumps as JR. Each of these can be unconditional, meaning that the jump will take place no matter what flags are set or reset in the status register. Alternatively, they can be conditional, meaning that the jump will take place only if some selected flag in the status register is set or reset according to the programmer's choice. It's rather like the difference between GOTO 3∅∅ and IF A = ∅ THEN GOTO 3∅∅. When we write conditional jumps in assembly language, we have to specify what the condition is, because each different condition means a different operating code, for example:

JR Z,disp

meaning jump to the new address if the zero flag is set, or:

JR NZ,disp

meaning jump to the new address if the zero flag is NOT set.

The complete list of available jumps is illustrated in Fig. 5.4. Some of them are very seldom used in the type of programs that you are likely to want to write for the Spectrum. The JP instructions need to be followed by a complete two-byte address, the JR instructions by a single-byte displacement number.

Interacting with Spectrum

The time has come to start some practical machine code programming with your Spectrum. This is not simply a matter of typing the assembly language lines, because unless you have loaded an assembler program, the Spectrum will simply ignore these commands. What you have to do is to find the machine code bytes that correspond to the assembly language instructions, POKE them into the memory of the Spectrum, place the address of the first byte

```
JP addr     Jump to the address given.
JP c,addr   Jump to the address given if the
            condition c is met.
JR d        Jump relative to PC address,
            using displacement d.
JR c, d     Jump relative to PC address, using
            displacement d if condition c is
            met.

Conditions for JP                  Conditions for JR
NZ - not zero                      NZ - not zero
Z - zero                           Z - zero
NC - no carry                      C - carry
C - carry set                      NC - no carry
PO - parity odd
PE - parity even
P - sign positive
M - sign negative

Note: There is a JP version, JP (HL) which
takes a jump to the address held in the HL
registers.
```

Fig. 5.4. The jump commands, both absolute and relative, unconditional and conditional.

into the PC register of the Z-80A in your Spectrum, and watch it all happen. It sounds simple, but there is quite a lot to think about and a number of precautions to take. To start with, Spectrum uses quite a lot of the RAM, as we have seen in the first few chapters of this book, for its own 'housekeeping' operations. If we simply POKE a number of bytes into memory without taking a few precautions, the chances are that they will either replace bytes that the Spectrum needs to use, or they will be replaced by bytes that the Spectrum places in these memory addresses as it forms its variable list table and all the other items that are put into memory when a BASIC program is run.

We can prepare a section of memory for machine code in two ways. One is to type a REM statement as the first line of a BASIC program, and fill it with spaces – as many spaces as there will be bytes of the machine code program. This was the method used for the ZX-81, which had no other provision for machine code. The Spectrum allows a memory space to be reserved by using the BASIC word CLEAR, however. If you are using a 16K Spectrum, in which the last usable RAM address is 32767 (denary), then the RAM from 326∅∅ to 32767 is normally reserved for user-defined graphics. You can reserve more space for your own machine code bytes just under this area by using CLEAR. CLEAR 325ØØ, for example, will cause the RAM between 325ØØ and 326ØØ to be reserved for your machine code programs, and if you do not use the defined graphics in your program, there is nothing to prevent you from using the space between 326ØØ and 32767 as well.

The next problem is how to place the address of the start of your machine code program into the PC of the Z-80A. Once again, there is a BASIC instruction which copes with this, USR. When USR is followed by an address, the computer places that address into the PC of the Z-80A (and into the BC register pair as well), so that your program will run. By way of a bonus, when the Spectrum returns to normal operation, it will make a number available to you. This number will be the number stored in the BC registers, and you will see it printed on the screen if you started your program by:

```
PRINT USR address
```

or you can assign it to a variable by using:

```
LET x = USR address
```

or ensure that nothing is printed or assigned by using:

```
RANDOMIZE USR address
```

Note that USR cannot be used by itself – it has to be used after a 'do-something' statement. You may not wish or need to have a number returned to BASIC, but it's very useful. If your program does not make any use of the BC registers, then what you get back is the address number that followed USR when the program was called.

The next thing is to ensure that the machine code program will stop in an orderly way. Nothing you have done so far will indicate to the Spectrum where the end of the machine code occurs, so that it will continue to read bytes after the end of your program until it encounters some byte that will cause a 'crash'. You can prevent this by making the last byte of each program a 'return-from-subroutine' byte, code 2Ø1 denary, C9H, which will automatically cause a return to BASIC when the program has been started by a USR address command.

There is one additional point that we don't have to worry about at the moment. When you use a machine code program, you are running this program on the same Z-80A microprocessor as is used for all the actions of the Spectrum. If we make use of the Z-80A registers, we must be quite certain that we are not destroying information that is needed for Spectrum. For example, if, at the instant that your machine code program started, the Z-80A contained in its BC registers the address of the start of the table of reserved words (looking for USR, perhaps), then it wouldn't be a good idea to replace this with something else. When a machine code program is called into action by using the USR instruction, however, this problem is taken care of for you. The contents of most of the registers are placed in a part of reserved RAM which is often called 'the stack'. This, incidentally, is another reason for being careful as to how you place your machine code in the memory – if you wipe out any of the stack the Spectrum will quite certainly not like it. When the RET instructions are carried out, the register values are restored equally automatically, and normal BASIC action can resume. If you call a machine code program into action by any other method, as is possible using some assembler programs, then you will have to do this salvage operation for yourself. The assembler language mnemonic for saving register contents is PUSH, and the registers are pushed in pairs, AF,BC,DE,HL, and so on in 16-bit sets. To get them back, POP instructions for the same registers are used, and the registers have to be restored in the correct order – last in, first out, like NEXTs after FORs. If you have used:

```
PUSH AF
PUSH BC
```

at the start of your program, then you need:

```
POP BC
POP AF
```

at the end. If you use:

```
~~POP AF~~ PUSH [illegible]
POP BC
```

then you will have interchanged the contents of AF and BC! This is sometimes done deliberately (it's one way of changing the status register contents, for example), but it's not a technique for the beginner.

For the moment, then we'll forget about PUSH and POP because the problem simply doesn't arise when we use USR as a way of starting the program. There is one exception, however. Spectrum makes a lot of use of the IX and IY registers, and these do not appear to be saved when a USR instruction is carried out. It may be that they can be used safely if they are PUSHed before using and POPped after, but until you have had some experience in the use of Spectrum with machine codes, it's better not to make use of them.

Practical programs

With these preliminaries out of the way, we can start on some programs which are very simple, but which are intended to familiarise you with the way in which programs are placed in the memory of the Spectrum, and with the use of assembly language and machine code.

We'll take the simplest possible example first – a program which places a byte into memory. In assembly language, it reads:

```
ORG 325ØØ       ;start address
LD A,85         ;load 85 immediate
LD (3251Ø),A    ;put into 3251Ø
RET             ;return to BASIC
```

The first line contains a mnemonic, ORG, which you haven't seen before and which isn't actually part of the Z-80 set. It's short for ORIGIN, and it's a reminder to you that this is the address of the first byte of the program, the first byte of reserved memory. We have chosen an address here which reserves much more room than we need, but for the sake of an illustration it's as good as any other, and it leaves plenty of room for longer programs. When you program with an assembler, this statement can be typed in (see Chapter 8), and the assembler will then automatically start allocating address numbers or even putting the bytes of code into memory at this specified address. Some assemblers do this indirectly, creating the machine code on tape or disc rather than directly into memory.

The next step in the programming is to write down the codes. The codes must be looked up, taking care to select the correct mnemonics. The code for LD A,N, which is the way in which the immediate load is written in lists, is 3EH or 62 denary, so this is the first byte of the program, because there is no code corresponding to the ORG 325ØØ statement. We can start a table of address and byte numbers with this entry:

```
325ØØ   62
```

and then move on. The LD A,N command has to be followed by the other operand byte, the byte that you want to place in the A register, which is 85 denary. Therefore 85 is the byte that has to be stored in the next address, making the table look like:

```
325ØØ   62
325Ø1   85
```

The next byte we need is the instruction code for LD(Addr),A, and this is 5Ø denary, 32H. It goes down into the table, and then it has to be followed by the address that we want to use, which is 3251Ø. The snag here is that this address has to be converted into two-byte form, and the bytes written in the correct order, lower byte first. You'll find it simpler if you have a calculator handy, or Spectrum switched on! Using a calculator, find first 3251Ø ÷ 256 – this gives 126.99218. Write down the 126, which is the higher byte number, and label it HB. Now carry out the calculator steps (the circles round the symbols indicate that each of them is on a calculator key):

126 ⊗ 256 ⊜ (+/-) ⊕ 32510 ⊜

What you are doing is multiplying 126 by 256 to get 32256, and then subtracting this from 3251Ø to get 254, which is the lower byte. You can now write the address 3251Ø in low, high order as 254,126 into your table, which now appears as:

```
325ØØ   62
325Ø1   85
325Ø2   5Ø
325Ø3   254
325Ø4   126
```

There's one last entry at address 325Ø5, the return byte 2Ø1. If you are unhappy about the way of finding the bytes corresponding to the address number, use the Spectrum program shown in Fig. 5.5.

The next step is to place this short program into the memory. We have to start by clearing a space, using CLEAR 325ØØ, and then POKEing the bytes in one by one. Since Spectrum, unlike the ZX-81, can use READ ... DATA, the POKE operation is most simply done using a loop, and since we know how many bytes we have, we can use the loop to read the bytes and place them into the correct address, starting with 325ØØ. The program reads:

```
1∅ CLEAR 325∅∅
2∅ FOR n = ∅ TO 5: READ b
3∅ POKE 325∅∅ + n,b
4∅ NEXT n
1∅∅ DATA 62,85,5∅,254,126,2∅1
```

Note that we've used n = ∅ TO 5, rather than 1 TO 6, so that we start at 325∅∅ rather than 325∅1. When you count the bytes, start your count ∅,1,2 rather than 1,2,3 ... and all will come right.

```
1∅  PRINT "Please type address - use ∅ to"'"stop":
    INPUT d
2∅  IF d = ∅ THEN GOTO 9999
3∅  IF d > 65535 THEN PRINT "Too large - exceeds
    65535": PAUSE 1∅∅ : GOTO 1∅
4∅  LET msb = INT (d/256) : LET lsb = INT (d -
    256 * msb)
5∅  PRINT "Address bytes are "; FLASH 1; lsb;
    ","; msb
6∅  PRINT ' : GOTO 1∅
```

Fig. 5.5. A BASIC program for calculating the two bytes of an address number.

When you RUN this BASIC program, nothing appears to happen. All that it has done is to place these bytes into memory, and it certainly has *not* caused the machine code program to run. Before we do this, we need some way of checking that the machine code actually does something! Type PRINT PEEK 3251∅, and ENTER, and you should see the result ∅ appear, unless for some reason something has been stored at 3251∅ by an earlier program. If you have just switched on, the result should certainly be zero.

Now type PRINT USR 325∅∅ and ENTER. The number that appears now is 325∅∅. It's just an echo of what you requested, because the machine code program does not use the BC registers, and these are loaded with the USR address number before your program starts running, as we commented earlier. The Spectrum has, however, carried out its action. If you now type: PRINT PEEK 3251∅, you will find the number 85 printed. This has been placed into the memory at address 3251∅ by your short section of machine code program.

It's no more than the command POKE 3251∅,85 would do from BASIC, but it's a start, and the main thing is to get used to how machine code programs are written, placed into memory and run. If you now press NEW and ENTER, you will see the screen clear after the black rectangle appears, but your machine code program is NOT cleared, though the BASIC program that put it into place will have gone. If you type PRINT PEEK 3251∅ you will see that the result is still 85, and if you clear this piece of memory by typing:

POKE 3251∅,∅

followed by ENTER, of course, then you can check by another PEEK that this memory has been cleared. Your program still exists, however, and can be used again to load the number 85 into the address 3251∅. This time, try:

LET x = USR 325∅∅ (and ENTER).

Nothing appears on the screen as a result of this form of command (though variable x is allocated to 325∅∅, and PRINT x will reveal this), and you will find when you use PEEK 3251∅ that the value stored here is 85 once again. If you want to clear the memory completely, you can type

CLEAR 32767 (for a 16K machine)

and ENTER, then NEW (ENTER). This will remove everything, and if you forgetfully type PRINT USR 325∅∅ and ENTER now, some number may be printed on the screen, but the computer will 'lock-up', no key having any effect, or possibly go into its NEW routine. If you get a lock-up, which is very common when a machine code program goes wrong, then all you have to do is to switch off and on again. Your program will, of course, be lost when you do this, but see Chapter 7 for how to save machine code on tape.

We very seldom need to clear the memory out in this drastic way. If you have reserved spaced for a machine code program by typing CLEAR 325∅∅, then you can write as many programs as you like in this space. If you use the same addresses again, then the most recent program will replace any previous ones, but as long as each program uses a 2∅1 code to return to BASIC, the old program bytes that are still stored at other locations will not affect the new program. One thing to watch, however, is how you use memory for storage, thinking back to the way in which we stored the byte 85 in address 3251∅. If the address that you use for storage in this way is inside the range of addresses that you use for your program, then you will have to write your BASIC program so that some bytes are POKEd before this address and some after but none in. Don't assume that because you didn't place anything there, there is nothing in the address! You can make your BASIC program POKE a zero into the space if you want, but you must not place any of the machine code instructions in that address. You will have to be equally careful that you don't let the MPU read this piece of memory as if it contained an instruction, so it will have to be jumped around. Fig. 5.6 shows how this can be done, but a much safer way when you are starting with machine code is to make all your use of memory for storage of quantities like this in addresses that are not within the range of addresses used by the program. Any address which is beyond the RET or a final JP or JR back will be suitable. If necessary, don't fill in details of address numbers until you can be sure how long the program is going to be, or use addresses that you can be sure will be well clear of the program addresses, like 32599 for a program that uses only 325øø to 3255ø.

```
      LD A, (HL)        ; load accumulator
      JR 1              ; jump over byte
      (data byte)       ; data byte used in program
addr: LD C, A           ; another load
      LD (ADDR), C      ; put into data
```

Fig. 5.6. Jumping over a data byte. This must be done if a data byte is stored within a program, because it must not be read by the MPU as if it were an instruction.

Now try another example. We'll be more bold this time, and come back with something in the BC registers. This, remember, is simply a convenience of the Spectrum operating system. It's not a feature of the Z-80A, but since we're dealing with the machine code programming of the Spectrum it seems daft not to make use of the features that it provides. The assembly language version is shown in Fig. 5.7. There are no addresses shown this time, but note that the RL A command (rotate left accumulator) consists of two bytes, 203 and 23, so that your BASIC program takes the form:

```
1ø CLEAR 325øø
2ø FOR n = ø TO 7: READ b
3ø POKE 325øø + n,b
4ø NEXT n
1øø DATA 62,85,2ø3,23,6,ø,79,2ø1
```

As before, when this is RUN, nothing noticeable happens because it simply places bytes into memory. When you use:

```
PRINT USR 325øø
```

```
LD A, 85        ; 85 into accumulator
RL A            ; rotate left
LD B, Ø         ; zero B
LD C, A         ; load in result
RET             ; return
```

Fig. 5.7. An assembly language program which will return with a byte in the BC registers.

and ENTER, however, the number 17ø appears on the screen, and this demands some explanation. What the program has done is to load the number 85, which in binary is ø1ø1ø1ø1 into the accumulator. A left rotation of this, Fig. 5.8, gives the number 1ø1ø1ø1ø in binary, and this in denary is 17ø. By loading register B with zero, and carrying out LD C,A so that the single byte number 17ø is copied from A to C, you end up with 17ø in the BC register when the program returns. As Spectrum will always return with the number in the BC register pair (unless RANDOMIZE 325øø is used), the PRINT part of the USR call will ensure that this number is printed. Had you used LET x = USR 325øø, then the value allocated to x would have been 17ø.

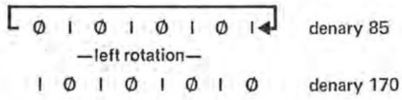


Fig. 5.8. Explaining the number which appears.

Now try this. List your BASIC program, and delete the numbers 6,ø from the DATA list, being careful not to leave a space with two lots of commas. The DATA list will now read:

```
1øø DATA 62,85,2ø3,23,79,2ø1
```

and you will have to alter line 2ø to read FOR n= ø TO 5 in place of the old count of 7. Now RUN to place the codes into memory, and use PRINT USR 325øø again. This time you get the number 32426 appearing. Why?

The answer is simple. The Spectrum operating system places the address that follows USR into the BC register pair. The address 325øø has a high byte of 126 and a low byte of 244 (so that 325øø = 126× 256+ 244), so that when USR 325øø is called up, the content of the B register is 126, and the content of the C register is 244. The revised program has omitted the LD B,∅ step, so that register B will still have 126 in it when the program ends with register C holding 17∅. This gives the denary number 126 × 256 + 17∅, which is 32426, in the BC register when the program returns. Automatic procedures like this can be very useful, but you have to keep your wits about you, because unless you do something to change numbers, they stay stored!

Chapter Six
Byting Deeper

The simple programs that we looked at in Chapter 5 don't do much, though they form very useful practice in converting assembly language into code bytes, putting them into the machine and running the resulting program. In this chapter, we shall look at how simple machine code programs are designed – in other words, how to get to the assembly language version, because this is by far the most difficult part of the process for the beginner to machine code programming.

The difficulty, curiously enough, doesn't arise because machine code is difficult but because it is simple. Because machine code is so very simple, you need to use a large number of instruction steps to achieve anything useful, and when a program contains a large number of steps, it's more difficult to plan. The most difficult part of the planning is breaking down what you want to do into a set of steps that can be tackled by assembly language instructions, and for this part of the planning, flowcharts are by far the most useful method of finding your way around. I never feel that flowcharts are ideally suited for planning BASIC programs, but for machine code, they can be very useful indeed.

Flowcharts

Flowcharts are to programs as block diagrams are to hardware – they show what is being done (or attempted!) without going into any more detail than is needed. A flowchart consists of a set of shapes, each one meaning some type of action. Fig. 6.1 shows some of the most important flowchart shapes for our purposes (taken from the British Standard set), the terminator (start or stop), input/output, process (action) and decision steps. Inside these shapes we write the action that we want, but without details.

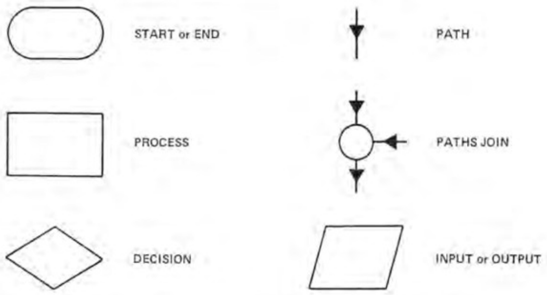


Fig. 6.1. Flowchart symbols – a small selection from the BS range.

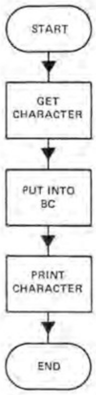


Fig. 6.2. A flowchart for the 'get character' program.

An example is always the best way of showing how a flowchart is used. Suppose you want a machine code program that takes the ASCII code for a key that has been pressed, and prints the character corresponding to that code on the screen. A flowchart for this action is shown in Fig. 6.2. The first 'terminator' is START, because every program has to start somewhere, and this leads to the first 'action' block, which is labelled 'get character'. This describes what we want to do – how we do it is something we don't know yet. After getting the character, the next 'action' block is: 'put into BC', because that is what we need if the value of the ASCII code is to be returned to the BASIC routine. Following that, the 'print character' block is something we can do in BASIC (it's not so straightforward in machine code). The END terminator then reminds us that the program ends here – it's not an endless loop.

This is a very simple flowchart, with no decision steps or loops, but it is enough to illustrate what we mean. Note that the descriptions are fairly general ones, so don't ever put assembly language lines into the action boxes of a flowchart, for example, because that is just downright confusing. Strictly speaking, I shouldn't have used the 'put into BC' box, but this is so essential to getting a number back into BASIC that it really needs to be mentioned as a reminder. A flowchart should show anyone who looks at it what is going on; it shouldn't be something that only the designer can understand and which serves mainly to confuse everyone else. A lot of flowcharts, alas, are constructed after the program has been written in the hope that they will serve to make the action of the program clear. You wouldn't do that, would you?

Once we have a flowchart, we can check that it will actually do what we want by going over it very carefully. In the example of Fig. 6.2, the parts labelled 'get character' and 'put into BC' are going to be done using machine code, so we'll concentrate on them for now.

Getting the ASCII code for a character looks tricky at first. A lot of computers put the ASCII code into the accumulator during the subroutines of reading the keyboard, and then store the value temporarily in RAM. Looking at the description of the Spectrum system variables in Chapter 25 of the manual reveals that address 2356Ø is used for this purpose – the code for the last key pressed is stored here. If we load A from this address, we should then have in the accumulator the ASCII code for the last key that was pressed; step 1 completed. The next step is already familiar – we want Ø in the B register, and to copy the byte in the A register into the C register. This is necessary because we can't load the C register directly from a memory address – that's one of the restrictions which makes the A register more useful for a lot of purposes than any other single-byte register. Having loaded BC, we can then return to BASIC, and make use of the code in BC in our BASIC program. If we call the program by using LET x = USR 325ØØ, then we can print CHR$ x to display the character.

That's half of the problem overcome. The next part is how to get the byte into address 2356Ø using BASIC. If we use INPUT for this stage, then the byte in 2356Ø will always be 13 – because that's the code for the ENTER key that you have to press to enter your input! Perhaps INKEY$ will be more useful – we can set up a loop so that pressing a key will leave the loop and call the machine code program when the value of that key is in address 2356Ø.

We can start by designing the assembly language program, which might read:

```
LD B,Ø          ;clear B
LD A,(2356Ø)    ;get ASCII code
LD C,A          ;put into C
RET             ;back to BASIC
```

which looks as if it should do what is wanted. To call it and use it, we can have:

```
2ØØ LET k$ = INKEY$ : IF k$ = "" THEN GOTO 2ØØ
21Ø LET x = USR 325ØØ
22Ø PRINT CHR$ x
```

Now we can complete the operation by looking up the codes and writing the part of the BASIC program that will POKE them into memory:

```
1Ø CLEAR 325ØØ
2Ø FOR n = Ø TO 6 : READ b
3Ø POKE 325ØØ + n, b
4Ø NEXT n
1ØØ DATA 6,Ø,58,8,92,79,2Ø1
```

Add this to your lines 2ØØ to 22Ø and we should be all set.

Sure enough, when this runs, we can press a key and we will find the character printed on the screen. Once again, it's not terribly impressive because it's no more than we would get if we programmed in BASIC:

```
2ØØ LET k$ = INKEY$ : IF k$ = "" THEN GOTO 2ØØ
21Ø PRINT k$
```

The aim, however, is to get experience of how machine code programs can be written and can interact with the Spectrum BASIC, not to start writing original works of genius.

Now let's take a look at some ways of cutting out some flab from this program. To start with, we find that we don't need two lines for 21Ø, 22Ø because we can substitute the single line:

```
PRINT CHR$ USR 325ØØ
```

If it looks odd, think of it as PRINT the character whose code is obtained from the USR 325ØØ subroutine. Now the next bit we have to tackle is the INKEY$. Do we need to use BASIC here? The answer is that we can create our own INKEY$ loop, using the program shown in Fig. 6.3. The machine code section simply places the byte at address 2356Ø into the BC register as before, but the BASIC section then tests this, and if the byte is less than 32 (the space byte), then the loop repeats until a byte greater than 32 is found. This doesn't completely release us from BASIC, because we are still using line 1ØØ to make the test. Could we test the byte and loop back within the machine code program? The answer *for the moment* is no, because to get a new byte into 2356Ø, the Spectrum operating system must be running, and it can't run while our own program is continually looping round to test 2356Ø in machine code! If we want to test key values, then we have to use other ways, as we shall see later.

```
        LD A, (2356Ø)     ; get last character
        LD B,Ø            ; clear B
        LD C, A           ; load in result
        RET               ; back to BASIC

1Ø   CLEAR 325ØØ
2Ø   FOR n = Ø TO 6; READ b
3Ø   POKE 325ØØ + n, b: NEXT n
5Ø   DATA 58, 8, 92, 6, Ø, 79, 2Ø1
1ØØ  LET x = USR 325ØØ: IF x<= 32 THEN GO TO 1ØØ
11Ø  PRINT CHR$ x
```

Fig. 6.3. A get-character program. The BASIC part is needed to place the character into memory.

Let's look at another aspect of input for the moment. Suppose we wanted automatic upper-case (capital letter) shift, so that when we pressed the h key we would get H printed. Looking at the ASCII codes reveals that the lower-case letter codes use numbers which are 32 greater than the corresponding upper-case codes, so we might try subtracting 32 from these lower-case codes to get the upper-case letters. What we need to be careful about though is what happens if we already have typed an upper-case letter. We don't want this change to be made when the ASCII code is 96 or less, because this is the code for a (A is 97 − 32 = 65).

A flowchart for a simple scheme is shown in Fig. 6.4. We want to get the code, and compare it with 97. If it is equal to 97 or greater, we shall subtract 32, but if the code is less than 97, we shall leave it unchanged. Either way, we then load the code into BC and that's the end of the machine code section.

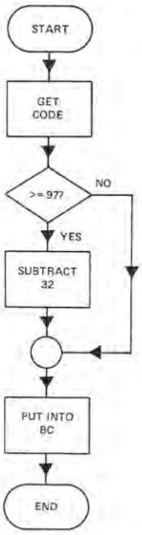


Fig. 6.4. A flowchart for the conversion to upper-case program.

Now this introduces a decision step which we haven't used in assembly language before, so we'll take the next few stages slowly. Fig. 6.5 shows the assembly language version of the flowchart. The character code is obtained as before by loading the accumulator from address 2356Ø, but the decision step needs two instructions. The first one is CP 97, meaning compare whatever number is in the accumulator with 97. As we saw earlier, it's like subtraction, but without the content of the accumulator being affected. If the number in the accumulator is 97 or greater, then it will be possible to subtract 32 from it, but if the byte in the accumulator is less than 97, the subtraction will require a 'borrow', which is indicated by the carry flag of the status register being set to 1. It will be Ø if the byte in the accumulator is 97 or greater. If the carry flag is set by the CP step, then, we do not need to subtract 32 from the code number. If the carry flag is not set, then we do have to subtract 32. This part is done by a decision step, a jump. In the assembly language version, the destination of the jump is indicated by using a word, out. This is a label word, which is a convenient way of indicating on an assembly language program where a jump is to go, because we have no addresses written down, and it would be awkward to calculate displacement bytes at this stage. The destination is confirmed by having the same word placed in front of the step to which the jump must go, the LD B,Ø step. If the number in the accumulator is 97 or more, however, the carry flag will not be set, the jump will not occur, and we need to subtract 32 from the byte in the accumulator. This is done by the next step in line, SUB 32, which is then followed by the

```
     LD A, (2356Ø)       ; get character
     CP 97               ; compare
     JR C, Out           ; out if less than 97
     SUB 32              ; otherwise subtract 32
Out: LD B, Ø             ; zero B
     LD C, A             ; transfer answer
     RET                 ; back
```

Fig. 6.5. The assembly language version of the conversion program.

```
1Ø  CLEAR 325ØØ
2Ø  FOR n = Ø TO 12
3Ø  READ b: POKE 325ØØ + n, b
4Ø  NEXT n
5Ø  DATA 58, 8, 92, 254, 97, 56, 2, 214, 32,
    6, Ø, 79, 2Ø1
1ØØ LET k$ = INKEY$: IF k$ = "" OR CODE k$ =
    13 THEN GOTO 1ØØ
11Ø LET x = USR 325ØØ
12Ø PRINT CHR$ x;
13Ø IF INKEY$<>"" THEN GOTO 13Ø
14Ø GOTO 1ØØ
```

Fig. 6.6. The coded version of the conversion program.

LD B,Ø command. A close scrutiny of the assembly language program shows that it should do what we want it to do; it certainly follows the actions of the flowchart.

The next step is coding. This is straightforward enough apart from the jump step. The difference between the JR address and the LD B,Ø address is 4 bytes, so we should get the correct displacement byte, following the rule given in Chapter 5, by subtracting 2 to get a displacement of 2. The result is the program shown in Fig. 6.6.

There is an unexpected line 13Ø in this program. The program will work perfectly if the lines after 12Ø are omitted, but will print just one letter on each run when a key is pressed. It is much more useful when it is used in a loop, but if you simply add GOTO 1ØØ, you will find curious effects, letters repeating, or unwanted spaces. This is because the buffer memory which is used for the INKEY$ function is not always cleared in time, and line 13Ø ensures that it is by holding a loop until INKEY$ is a blank. You'll find that this program gives you an upper-case letter each time you press a letter key.

Once again, it's not a masterpiece (what happens when you press a number key?). It could have been done completely in BASIC by using some of the reserved RAM addresses – if you try the program of Fig. 6.7, then you will see that most of the keys will give the correct upper-case characters though others give the query sign (which means that the number stored at that address has no ASCII code). Once again, what we are doing is something that can be done with BASIC, but the aim is to learn machine code methods, and if we stuck to examples of things which could not be done in BASIC, we would end up with machine code examples which would be just too difficult to follow at this stage.

```
1Ø  LET k$ = INKEY$: IF k$ = "" OR CODE k$ =
    13 THEN GOTO 1Ø
2Ø  LET x = CODE k$
3Ø  IF x >= 97 THEN LET x = x - 32
4Ø  PRINT CHR$ x;
5Ø  IF INKEY$ <> "" THEN GOTO 5Ø
6Ø  GOTO 1Ø
```

Fig. 6.7. The conversion program in BASIC.

In any case, we have now used a decision step in a machine code program as well as in flowchart and assembly language form. It's another step on the way for us, and some more practice in loading and running machine code programs. Perhaps we can now try something more ambitious, and at the same time probe the secrets of the Spectrum rather more deeply. Earlier on, we found that there were certain difficulties attached to trying to simulate the action of INKEY$ without using BASIC. Well, now is the time to show the way. There is a machine code routine in the ROM for reading the keyboard, and if we can find it, then we may be able to make use of it in our programs. Finding the routine with the aid of the Campbell disassembler wasn't difficult. I looked for a piece of program that loaded the address 2356Ø, and having found this, traced it back to find where it started. This led me to the address Ø2BFH, 7Ø3 in denary, as the start of a routine which read the electrical signals from the keyboard and converted them into ASCII codes. A preliminary try-out showed me that the routine placed 255 in the accumulator if no key was pressed, and the ASCII code for the key if a key was pressed.

Now the new feature for you at this stage is how you can make use of a Spectrum ROM routine. A subroutine in BASIC is called by using GOSUB, and the return is forced by using the BASIC command RETURN at the end of the subroutine. In assembly language, a subroutine is called by CALL, followed by the address of the start of the routine. The return is forced by the RET instruction which we have already used. CALL and RET have to be used together, and the reason that we were able to use RET at the end of our routines to return to BASIC is that the CALL part of the process has been done by the USR routine.

We can get a byte into the accumulator from the keyboard by using CALL 7Ø3 as an instruction. The CALL code is 2Ø5, and the address which is called must be written in the usual low-byte, high-byte form (you must follow CALL by two bytes of address, even if one is zero). The address 7Ø3 is coded as 191,2 , so that the set of numbers 2Ø5,191,2 will carry out the action of CALL 7Ø3.

Fig. 6.8 illustrates the flowchart we shall need for this program. the byte is fetched from the keyboard, using the CALL, and tested for equality to 255, because this is the number that the ROM routine will put into the accumulator if no key is pressed (it's the number that means 'false'). If the byte equals 255, then the call is repeated, but if a key has been pressed, then the value in the accumulator will be the ASCII code for that key. This value in the accumulator can then be passed to the BC registers as before.

Fig. 6.9 shows a BASIC program that POKEs the code into memory and uses it. Just for a change, I've illustrated a quicker way of using the POKE and USR commands. By having LET j = 325ØØ

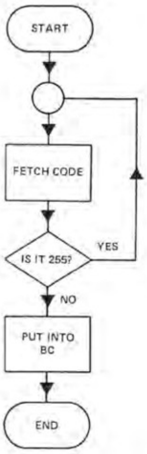


Fig. 6.8. The flowchart from a key-reading program.

```
1Ø  CLEAR 325ØØ
2Ø  LET j = 325ØØ
3Ø  FOR n = Ø TO 1Ø
4Ø  READ b: POKE j + n, b: NEXT n
1ØØ DATA 2Ø5, 191, 2, 254, 255, 4Ø, 249, 6, Ø,
    79, 2Ø1
2ØØ PRINT CHR$ USR j

Assembly language:      Back:  CALL Ø2BFH
                               CP 255
                               JR Z, Back
                               LD B, Ø
                               LD C, A
                               RET
```

Fig. 6.9. The BASIC program which POKEs the code and makes use of the machine code routine.

early in the program, we can save memory by using j in place of 325ØØ. Since this saves a conversion from ASCII to binary each time 325ØØ is used, it saves time. The program does in machine code what INKEY$ does in BASIC; it waits for a key to be pressed, and then returns with the value of the ASCII code in the accumulator.

This is the first use that we have made of a routine in the ROM, and it's not always something that we can do easily. It's not necessarily easy to find ROM routines unless you have a disassembler, lots of time, and some confidence in reading assembly language code – it was fortunate that the INKEY$ routine was so easy to spot. There are, however, complete commented disassemblies of the ZX-81 ROM available, so it's just a matter of time before we find some similar treatment for Spectrum (some of the routines are almost identical, though not always in identical addresses). When this is done, you will be able to look up the routines and use them for yourself, though you must be careful to make sure that your registers are loaded correctly before calling the routines.

It's time now to try something for yourself. Can you combine the routine we have just examined, the machine code equivalent of INKEY$, along with the idea of subracting 32 if the byte is 97 or more, to get a routine which will put all printing into upper-case?

Loops

The loop action in BASIC that you use most of all on the Spectrum is the FOR ... NEXT loop. This uses a 'counter' variable to keep a score of how many times you have used the loop, and compares the value of the counter with the limits you have set on each pass through the loop. Now the action of a FOR ... NEXT loop can be simulated without using FOR ... NEXT, as is shown in Fig. 6.10, using an IF ... THEN line to make the decision as to whether or not to keep looping back. Most types of microprocessors can tackle

```
1Ø LET count = Ø: LET end = 1Ø
2Ø PRINT "Action "; count
3Ø LET count = count + 1
4Ø IF  count < = end THEN GOTO 2Ø
5Ø PRINT "Finished"
```

Fig. 6.10. How the action of a FOR ... NEXT loop can be simulated in BASIC.

looping in a similar way, using one register or a memory address to keep a count of what is going on. The Z-80 family, however, has a set of ready-made loop instructions, and these can save a considerable amount of programming.

The Z-80 equivalent of FOR ... NEXT is written in assembly language as DJNZ, short for decrement and jump if not zero. Register B is used as a counter, and each time the DJNZ code (1ØH = 16 denary) is encountered, the number in the B register is decremented and the register is tested, using the flags in the status register, to see if the decrement action has resulted in the number reaching zero. If it has not been decremented to zero, then a loop back is taken , using a displacement byte in the same way as a JR instruction. If the register content has reached zero, the instruction following the DJNZ one is carried out.

Fig. 6.11 illustrates this in a program which simply loads B with 255, the maximum possible size of a single byte (DJNZ treats this as being *unsigned*). The byte which causes the jump back is −2 (254), so that the jump is simply to the start of the DJNZ instruction again. The program lets you see how fast this count is performed compared to a FOR ... NEXT loop using the same numbers. In fact, the machine code count is very much faster that it appears, because of the time that is needed for the BASIC part of the program to respond in lines 1ØØ and 11Ø – you see it return by the Ø appearing on the screen because this is the number in BC.

The machine code count in this last example is so fast that we can't time it, and the time that is taken is practically all due to the BASIC part of the machine code program, so it's interesting to compare a much longer count. DJNZ can be used only for a single-byte count: there is no provision for a DJNZ count in a double register, so that this is a good opportunity to look at a method that is used for counts of more than one byte. We start by loading the BC register pair, in this example, with the largest number that we can get in two bytes, FFFFH, which is 65535 denary. Counting starts by decrementing BC, but because decrementing a pair of registers does not, on the Z-80, cause any flags to be affected, we have to test for the end of the count in a different way. The count will be complete when both registers of the BC pair have reached zero, and the standard method of checking this is to load one of the registers into A, and the OR in with the other one. If either of the registers contains a 1 bit anywhere in its byte, the OR will produce a 1 in the same place in the result, and the zero flag will not be set. As a result, the program loops back at the JR NZ stage – the jump is three address numbers back, which is −5 denary, 251 as a displacement byte. Fig. 6.12 shows the program and the denary numbers for the machine code.

```
Back: LD B, 255
      DJNZ Back
      LD C, B
      RET

1Ø   CLEAR 325ØØ: LET j = 325ØØ
2Ø   FOR n = Ø TO 5: READ b
3Ø   POKE j + n, b: NEXT n
5Ø   DATA 6, 255, 16, 254, 72, 2Ø1
1ØØ INPUT "Press any key ..."; a$
11Ø PRINT USR j: PRINT "Done"
12Ø PRINT "Now try a BASIC count ..."
13Ø INPUT "Press any key ..."; a$
14Ø FOR q = 255 TO Ø STEP -1: NEXT q
15Ø PRINT "Done!"
```

Fig. 6.11. A countdown program in machine code, using DJNZ, with the BASIC version to compare speeds. The result is misleading!

```
       LD BC, FFFFH        1, 255, 255
Count: DEC BC              11
       LD A, B             12Ø
       OR C                177
       JR NZ, Count        32, 251
       RET                 2Ø1
```

Fig. 6.12. Assembly language version of a countdown of a much larger number stored in a register pair. Note the method used to check the end of the count.

```
1Ø   CLEAR 325ØØ: LET j = 325ØØ
2Ø   FOR n = Ø TO 8: READ b
3Ø   POKE j + n, b: NEXT n
5Ø   DATA 1, 255, 255, 11, 12Ø, 177, 32, 251, 2Ø1
1ØØ INPUT "Press any key ..."; a$
11Ø PRINT USR j: PRINT "Done"
12Ø PRINT "Now try a BASIC count ... "
13Ø INPUT "Press any key ..."; a$
14Ø FOR q = 65535 TO Ø STEP -1: NEXT q
15Ø PRINT "Done at last !!"
```

Fig. 6.13. A program which allows you to compare counting speeds of BASIC and machine code over the same range of numbers.

Now if you substitute this for the previous steps, as shown in Fig. 6.13, you will find that the machine code count takes about half a second. The BASIC count of the same number, however, takes several minutes, and this is a more accurate picture of how much faster machine code instructions can be. We can, incidentally, make use of this sort of loop for timing. If the crystal of the Spectrum is running at a speed of 3.5 MHz, meaning 3½ million pulses per second, then by counting the number of pulses that are needed for a program we can find how long it takes. Appendix E shows some of the more important times, and Fig. 6.14 shows how these are applied to our loop. The sum of pulses in the loop between DEC BC and JR NZ is 26 pulses, so that to go round the loop 65535 times will need about 1,7Ø3,91Ø pulses. We can add to this, if we like, the time needed for the LD BC and the RET instructions; strictly speaking we should count the time for JR NZ for only 65534 loops, and use a shorter time for the last run, which doesn't loop back. These, however, are small corrections to a large number. By this reckoning, we should have finished the loop in a time of (1703910)/(3500000) seconds, about 0.48 seconds, which seems to correspond with experience.

Operation	Time	Comment
DEC BC	6	Same as for INC
LD A, B	4	
OR C	4	As for ADD C
JR NZ	12	In each loop
TOTAL	26 clock cycles	In loop

Fig. 6.14. The times for the loop instructions added together.

Counting can produce time delays which, since the clock rate is controlled by a crystal of quartz (like digital watches), are very precise. These time delays are used to a considerable extent in the Spectrum ROM – for the TV display, the BEEP routine, cassette input and output, and the printer routines to name the most obvious examples.

Chapter Seven
INS and OUTS

Storing and reloading

Up to this point, we have generated machine code programs as BASIC routines which POKEd the numbers into memory. This is easy to carry out, and there is no reason why the whole program should not be saved as a BASIC program. If you use the Campbell disassembler, you will find that you can use it to place code in memory, using hex rather than denary, and this can be faster than writing READ ... DATA loops in BASIC. You then have the problem of how to save such a program on tape, when there is no BASIC program that has generated it. The same problem arises when you have used the ASC ULTRAVIOLET assembler to produce machine code which must be recorded and relocated. Even if you have used a BASIC program to place the bytes in memory, you may want to record the machine code separately so that you can place it into the Spectrum without having to load in the BASIC program that performs the POKE routines.

Happily, Spectrum, unlike its predecessors, provides for saving and loading machine code directly though, as you might expect, you have to be much more specific about what you want. The syntax is summarised in Chapter 30 of the Spectrum manual, but some practice with one of our programs will give you more confidence in the use of this set of commands.

Let's use the program that we finished with in Chapter 6, which stores 9 bytes starting at address 325ØØ. The syntax of a SAVE command for this code will be:

```
SAVE "Count" CODE 325ØØ,9
```

You can choose your own name for the program of course. When you press ENTER you will get the usual message about starting the recorder and pressing any key. The program saves in much the same way as a BASIC program, so keep a note of where it is on the tape so that you can wind back to the start again.

To make a fair test, you will then have to switch off the computer (perhaps you might like to save the BASIC POKE program first, just in case!) so that the entire memory is cleared when you switch on again. Switching off and then on again re-allocates all of the memory, so before you try to load the program again, you will have to type CLEAR 325ØØ so as to reserve space for your program. Having done this, type:

```
LOAD "Count" CODE 325ØØ,9
```

and when you press ENTER and start the recorder on PLAY, your code will be loaded in again, starting at address 325ØØ. You will see the message:

```
Bytes: Count
```

to remind you that this is a machine code program called Count. When the bytes have completely loaded you will get the usual Ø OK, Ø:1 message at the foot of the screen when loading is complete.

There are a number of variations on the LOAD part of this set of commands. Using the full version above keeps a careful check for errors, so that if there are more bytes recorded on the tape than you provided for in the length number, you will get an error report. If you have forgotten or don't know how many bytes there are, you can use:

```
LOAD "Count" CODE 325ØØ
```

This will start the loading at address 325ØØ, and load in as many bytes as are recorded on the tape. If you run out of memory, too bad – you'll have to try loading at a different address. Not all programs will still work when they are loaded at a different address, but this particular one will, so with the program loaded at 325ØØ, try re-loading with:

```
LOAD "Count" CODE 325Ø6
```

This will carry out the loading, starting this time at 325Ø6, and replacing any bytes that existed previously at these addresses – there will be some left over from the program that was loaded in at 325ØØ. You can now call this program up to check it by using:

```
PRINT USR 325Ø6
```

which will return with zero after a short delay.

What you must not on any account do now is to use PRINT USR 325ØØ again. You might get away with it (in this example you will because of the way the program happens to work), but generally the result will be a locked out computer, with no keys having any effect. Switch off and start again.

If you know neither the starting address nor the length of a machine code program on tape, then Spectrum can still cope. By using the form:

```
LOAD "Count" CODE
```

with no numbers, the Spectrum will load the bytes into the position in the memory that they occupied when they were recorded, which is 325ØØ onwards in our example. This is always a safe method provided that you have cleared enough memory space.

Coming back to the idea of loading machine code into a part of the memory which is not the same as was used originally, this is possible only if the machine code is *position independent*. Position independent code is code which uses no full addresses that are within the program or within the range of addresses to which the program will be shifted. For example, suppose you have a program which starts at 325ØØ and ends at 32599. If in the program there is an instruction such as JP 3251Ø or CALL 32577, then these address numbers will be written into the program as code. If you then tried to load this program at addresses 32ØØØ to 32Ø99 (assuming that you had cleared enough memory), you would find that it did not run and you would get a lock-up. Why? Because you still have instructions JP 3251Ø or CALL 32577 which refer to addresses where by now there may be completely different codes, or only zeros stored. Code like this can't be relocated simply, and a similar thing applies if you had a call to 32ØØØ in a program that was placed at 325ØØ; if you relocated this program to 32ØØØ, you would overwrite the section of code which you wanted to call. Relocating a program of this type is not possible unless all of these addresses are changed (see Chapter 8 for relocating programs created with the ULTRAVIOLET assembler). If, however, your program had all its jumps in the JR form and all calls to the operating system which, being in ROM, is always at a fixed address that cannot be overwritten, then you could place such code anywhere in the RAM and run it. This type of code is 'relocatable', and the LOAD "Name" CODE start address type of command can be used to place it wherever you want it, assuming that you have cleared memory for it.

There is, incidentally, a VERIFY routine for machine code so that you can check if a valuable program has recorded correctly. This is particularly useful if the code has not been generated from a BASIC program using a READ ... DATA and POKE routine.

Berthing at a port

A *port*, as we saw in Chapter 1, is a circuit which is used to send signals into or out from the microprocessor system of MPU, ROM and RAM. As far as the MPU is concerned, a port is just another address which can be used rather like memory, but with a more restricted instruction set. The simplest Z-80 port instructions are those using the IN A,(port) and OUT (port),A assembly language instructions, and Spectrum BASIC provides commands in BASIC which perform port action, using the words IN and OUT. As you might expect,,the machine code version is faster and requires a much more detailed specification.

Before we start to look at the use of the ports in machine code, it's useful to look at how they can be used in BASIC. The port commands IN and OUT act rather like a PEEK and POKE respectively, and have to be followed by an address, which is *not* necessarily the same as an address in memory. There are eight addresses, for example, that are used to connect the keyboard switches to the microprocessor system, and these addresses, each of which deals with a half-row of keys, are listed in Fig. 7.1. They are also shown in Chapter 23 of the Spectrum manual. Suppose we pick one to try it. Fig. 7.2 shows a simple routine which uses port 65Ø22 (an address which does not correspond with any memory address on the 16K Spectrum). This port handles the keys A to G on the second row of the keyboard, and if none of these keys is pressed, then the result is 255 when you use IN 65Ø22. We have to use this like an INKEY$ loop, then, by programming:

```
2Ø LET x = IN 65Ø22: IF x = 255 THEN GOTO 2Ø
```

Port Address	Keys (left to right)
65278	CAPS SHIFT to V
65Ø22	A to G
6451Ø	Q to T
63486	1 to 5
61438	Ø to 6 (right to left order from now on)
57342	P to Y (misprint in the manual!)
4915Ø	ENTER to H
32766	SPACE to B

Fig. 7.1. Port addresses for keys – note that each port handles half a row of keys.

```
1Ø  PRINT "Press a or g"
2Ø  LET x = IN 65Ø22: IF x = 255 THEN GOTO 2Ø
3Ø  IF x = 254 THEN PRINT "That was a": GOTO 9999
4Ø  IF x = 239 THEN PRINT "That was g": GOTO 9999
5Ø  PRINT "You cheated!"
```

Fig. 7.2. A BASIC routine which makes use of the key ports.

Code (Denary, Hex and Binary)			Key position in half-row
254	FE	1111111Ø	1st
253	FD	111111Ø1	2nd
251	FB	11111Ø11	3rd
247	F7	1111Ø111	4th
239	EF	111Ø1111	5th

Note how much better the pattern is shown by the binary version of the code.

Fig. 7.3. The codes that are read in from the keys at each port.

The numbers that are read in from the various keys are shown in Fig. 7.3: note that these same numbers are read in even if the shift keys are pressed at the same time. We'll return to that point later, but for the moment, we can use the port in the simple key-press program of Fig. 7.2. This is a faster way of checking which key has been pressed than lines such as:

```
1ØØ LET a$ = INKEY$ : IF a$ = "" OR CODE a$ = 13 THEN
    GOTO 1ØØ
11Ø IF a$ = "g" THEN PRINT "That was - g"
```

and so on. We can take it a stage further by checking two ports so that we look for two keys being pressed, which is how Spectrum manages to generate different codes when the letter and shift keys are pressed at the same time. The keys that are accessed by different ports cause the same sequence of numbers, 254, 253, 251, 247, and 239 to be generated, so that we can use a program such as that of Fig. 7.4 to check for two keys being pressed. This is rather elementary BASIC, and we could program it rather more neatly but, no matter, it does what we want.

```
1Ø  PRINT "Press g and t"
2Ø  LET x = IN 65Ø22: LET y = IN 6451Ø
3Ø  IF x = 255 OR y = 255 THEN GOTO 2Ø
4Ø  IF x = 239 AND y = 239 THEN PRINT "That's
    it!": GOTO 9999
5Ø  PRINT "Cheat!"
```

Fig. 7.4. A BASIC program to check for two keys being pressed at the same time.

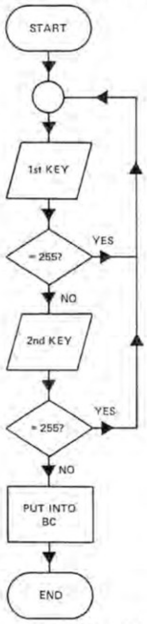


Fig. 7.5. A flowchart for an assembly-language version of the two-key program.

How would this work in assembly language? We have to start by writing a flowchart, and Fig. 7.5 shows a suggestion. The problem that we are tackling is one of reading two ports, and only when both report a key-press do we proceed – this will be when neither port comes in with 255. In the flowchart, the first port is tested, and if it reads 255, the program loops back to test this port again. If the reading is less than 255, indicating that a key in this half-row is pressed, then the second port is tried. If this one reads 255, meaning that no key is pressed in the second half-row, the program loops back to the start again. If we only looped back to the second test, this would be testing for two keys pressed in sequence rather than at the same time – not what we want, though it can be useful (you could make a program do one thing by typing GO and another by typing NO, for example). When both keys are pressed in the different half-rows, we put the resulting numbers into B and C (one byte in each) so that they can be passed back to BASIC.

It all looks very reasonable, but if we use the commands IN A,(port), then the assembly language requires a port address of one byte only, and the addresses in the manual consist of two bytes. If we attempt to load the accumulator from these addresses, rather than from a port, we find that they do not exist on the 16K Spectrum – there is no memory there. The clue lies in the type of port input command we need to use. The Spectrum uses a different type of port command for its keyboard reading routines, one which makes use of a two-byte port address in the BC register pair rather than the single-byte method of IN A,(port). When the port input command IN A,(C) is used, the number that is held in the BC pair is used as an address, the lower half in the C register being used as the port

```
Strt: LD BC, 65Ø22         1,254,253
      IN A, (C)            237,12Ø
      CP 255               254,255
      JR Z, Strt           4Ø,247
      LD D, A              87
      LD B, 251            6,251
      IN A, (C)            237,12Ø
      CP 255               254,255
      JR Z, Strt           4Ø,239
      LD C, A              79
      LD B, D              66
      RET                  2Ø1
```

Fig. 7.6. An assembly language program for detecting two keys. This uses the IN A,(C) instruction.

address. At the same time, the half address on the B register is sent out on the address lines, and this is how the Spectrum addresses its keyboard ports. The same scheme was used by the ZX-81 as you might expect.

To make our flowchart work, then, we must place suitable addresses in the BC register pair, and the addresses are just those shown as port addresses in the manual, though you will have to split them into the usual two-byte form to load them into BC. Fig. 7.6 shows the assembly language version of the two-key program, taking for the sake of example the ports that we used in the BASIC version. The QWERT group of keys uses the address that codes as 254,253 (low,high order), and the ASDFG group uses 254,251, so that we load BC with 253,254 (high-low order) initially, we need only change B to detect the second half row of keys. Note that we have to save the first value we find in register D, because A will be reloaded with that value, when the second key is detected. At the end of the program, the values are put into BC so that they can be returned for inspection. We would normally, of course, make some use of these values. Fig. 7.7 shows the numbers that are returned, which are identical to the numbers that would be returned by a BASIC version of the program.

```
10  CLEAR 32500: LET j = 32500
20  FOR n = 0 TO 20: READ b
30  POKE j + n, b: NEXT n
40  PRINT USR j
100 DATA 1, 254, 253, 237, 120, 254, 255, 40,
    247, 87, 6, 251, 237, 120, 254, 255, 40, 239,
    79, 66, 201
```

```
Keys pressed      Number        Analysis (B, C)

   g and t        61423            239,239
   a and q        65278            254,254
   w and s        65021            253,253
   e and d        64507            251,251
   f and r        63479            247,247

The g and t example has been shown first
because it was used previously.
```

Fig. 7.7. The BASIC POKE program for the assembly language program, and the results of pressing keys.

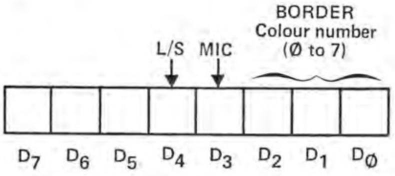


Fig. 7.8. How the Port 254 bits are allocated.

Let's try a port output now. Port 254 uses three bits to control the border colour, one for the MIC socket, and one for the loudspeaker. The byte is divided as shown in Fig. 7.8, with the lowest three bits (called DØ, D1 and D2) forming the number for border colour, D3 sending out one bit at a time to the MIC socket, and D4 sending out one bit at a time to the loudspeaker.

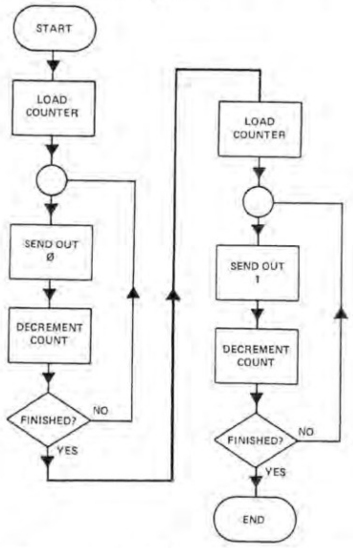


Fig. 7.9. A flowchart for sending bits out from the loudspeaker port.

We can try a short assembly language program to send something out to the loudspeaker bit of this port and observe the effect. The flowchart for this action is shown in Fig. 7.9, and the assembly language version with its numbers to POKE is shown in Fig. 7.10. When we put this into memory (Fig. 7.11) and run it, the most obvious thing that happens is that the border turns black! A little thought shows that this is logical, because we are using the numbers ∅ and 16 to put out to the port in our program. ∅ denary is ∅∅∅∅∅∅∅∅ in binary, and 16 denary is ∅∅∅1∅∅∅∅ binary. In each case, the three lowest bits are zero, corresponding to the number ∅, if we think of these three bits only, and this presumably means border zero, which is black. Suppose we used the number 4 denary in these bits, which would mean 1∅∅ for the last three bits. This means loading the accumulator with 4 and 16+ 4=2∅ in place of ∅ and 16; so we can alter the data bytes accordingly. Yes, this gives us a green border, so we've discovered how to control the border colour in machine code!

That's not what we were aiming at, however, and if you listen very carefully when you run the program each time, you will notice that when you press ENTER to run, you will hear a double click. One of these is the click you normally get when you press a key, but the other one is the result of your program. It has sent a ∅ and then a 1 to the loudspeaker, causing one click.

```
       LD B, 255       6,255
       LD a, ∅         62,∅
outl  :OUT (port),A   211,254
       DJNZ, outl      16,252
       LD B, 255       6,255
       LD A, 16        62, 16
out2  :OUT (port),A   211,254
       DJNZ, out2      16,252
       RET             2∅1
```

Fig. 7.10. The program in assembly language form.

```
1∅   CLEAR 325∅∅: LET j = 325∅∅
2∅   FOR n = ∅ TO 16: READ b
3∅   POKE j + n, b: NEXT n
5∅   DATA 6, 255, 62, ∅, 211, 254, 16, 252, 6,
     255, 62, 16, 211, 254, 16, 252, 2∅1
1∅∅  PRINT USR j
```

Fig. 7.11. The program in POKE form.

The next exercise is to prolong this into something more worth listening to. This means repeating the click often enough and fast enough to make a sound that will be more noticeable, and that in turn means another loop. Once again, we have to use a flowchart to see what we need to do, and Fig. 7.12 illustrates this one. It's a short flowchart, because the whole of the 'click' routine that we used in Fig. 7.9 has been shown as just one 'action' block rather than being drawn out in detail. The problem now is how to set about programming this action.

There are several ways, and it's instructive to look at more than one. Since we have another count to perform, it would be very useful if we could use DJNZ again. This, however, needs some care because

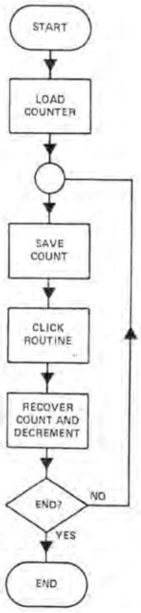


Fig. 7.12. The flowchart for a 'buzz' program.

we have used DJNZ twice in the click routine, and it always ends up with the B register counted down to zero. If we put a value into B to use as another counter, it will be overruled by the DJNZ commands in the click routines. What we can do, however, is to load B, PUSH this on to the stack, run the click routine, POP BC back, do the DJNZ and if the stored value has not decremented back to zero, jump back to the PUSH operation. Remember that the stack is a piece of RAM which is used by the microprocessor as a temporary store for register contents, so that the value we place there will be unchanged until we recover it and use it. If we carry out a PUSH BC before the click routine, then the number that was in the B register before the PUSH will be put back into the B register when we POP BC again.

The alternative method is to abandon the use of DJNZ for the number of click routines, and use a spare register or register pair for a count. A single register can be loaded with a value, and decremented after the click routine, then tested to make a jump back if it has reached zero. If a register pair is used, then remember that decrementing a register pair does not cause flags to be affected. Whichever method of counting the number of clicks is used, the click routine could also be put into the form of a subroutine which is called when wanted.

We'll illustrate the use of the DJNZ method, with the new commands added as shown in Fig. 7.13. We can do this with a little bit of editing of the code that we used for the click program, and we'll change the accumulator values to 7 and 23 at the same time so that the border colour stays white.

```
              LD B, 255
       Back : PUSH BC
              ... click routine ...
              POP BC
              DJNZ, Back
              RET
```

Fig. 7.13. How DJNZ can be used for two different counts. The contents of the BC registers are saved on the stack.

```
1Ø  CLEAR 325ØØ: LET j = 325ØØ
2Ø  FOR n = Ø TO 22: READ  b
3Ø  POKE j + n, b: NEXT n
5Ø  DATA 6, 255, 197, 6, 255, 62, 7, 211, 254,
    16, 252, 6, 255, 62, 23, 211, 254, 16,
    252, 193, 16, 236, 2Ø1
1ØØ PRINT USR j
```

Fig. 7.14. The buzz program in POKE form.

The result is shown in Fig. 7.14. It produces a buzz which illustrates well that we got the methods correct, and we can now try altering some items. For example, if we change the 255 byte in the click routine to 128, run it, and then try 5Ø and run that we can see (or hear) how the note varies with the time delay – shorter delays give higher pitch notes.

We can confirm that we need both the 1 and the Ø bits sent to the speaker by altering the routine so that it sends only one bit (try Ø or 16). You will see the border affected, but no sound. Fig. 7.15 shows the routine – remember that the displacement number for the overall DJNZ step has to be changed each time you change this number of

```
1Ø  CLEAR 325ØØ: LET j = 325ØØ
2Ø  FOR n = Ø TO 14: READ b
3Ø  POKE j + n, b: NEXT n
5Ø  DATA 6, 255, 197, 6, 255, 62, Ø, 211,
    254, 16, 252, 193, 16, 244, 2Ø1
1ØØ PRINT USR j
```

Fig. 7.15. Checking that there is no buzz if only one value of bit is sent to the loudspeaker.

```
        LD B, 255        6,255
loop:   PUSH BC          197
        LD A, B          12Ø
exit:   OUT (port), A    211,25Ø
        DJNZ, exit       16,252
        POP BC           193
        DJNZ, loop       16,247
        RET              2Ø1
```

```
1Ø  CLEAR 325ØØ: LET j = 325ØØ
2Ø  FOR n = Ø TO 11: READ b
3Ø  POKE j + n, b: NEXT n
5Ø  DATA 6, 255, 197, 12Ø, 211, 25Ø, 16, 252,
    193, 16, 247, 2Ø1
1ØØ PRINT USR j
```

Fig. 7.16. Putting numbers out from the port. Note the colour and sound effects when this runs, but disconnect your printer!

bytes in the routine. The small loops in the click routine do not need changing.

You can create some interesting visual effects if you alter the border colour bytes each time you run the click part of the routine. The simplest way of doing this is just to load the contents of the B register that is used for the overall count into the accumulator to send to the port. Since this number changes each time, the result will be different border colours. Fig. 7.16 shows some suggestions.

Since the same port which controls border colour and the loudspeaker also controls the MIC socket, we can use it to send signals out to the cassette recorder, and it would be possible to write a routine that would transmit signals in standard serial form for a printer (not the ZX-printer) or other serial device. Serial means one bit at a time, and some method has to be used to disinguish one byte from another when the bits are sent out singly, so a standardised system is used. The standard that practically everyone uses is called RS232, and it consists of a pattern of eleven bits per byte sent. The pattern comprises a 'start' bit, which is 1, followed by the eight bits of the byte that is being transmitted, and followed by two 'stop' bits, both zeros (Fig. 7.17). Messages are sent in ASCII code, and the time between each bit is held constant, with various standard values being used. The normal way of expressing this is not as the time between bits but as the number of bits per second, or *Baud rate* (this isn't a strict definition of Baud rate, but it will do for now). Rates of 110 Baud were used for the old mechanical teleprinters, and this corresponded to 10 characters per second. Modern serial printers can accept signals at rates which range from 11∅ Baud to 96∅∅ Baud (though they can't print at these higher rates, only accept signals into a buffer memory). Fig. 7.18 shows the standard Baud rates.

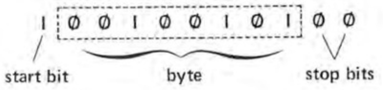


Fig. 7.17. Using start and stop bits in serial transmission.

Writing an RS232 routine for the cassette port is not exactly the kind of job that is recommended for the beginner, but you can see at this stage how the essential parts of it must work. The start bit is sent out by loading a 15 into the accumulator and sending it out to port 254. The reason for using 15 is that it sets bit D3 to 1, and keeps bits D2 to D∅ at 1 so that the border does not change colour. A timing loop must be used to control for how long this signal is sent out, then each bit from the selected byte had to be loaded into the accumulator and sent out. How do we do this? One method is to clear the carry flag (an OR A will do this), and then rotate the byte in the accumulator left. If a 1 is rotated out, the carry flag will be set; if a ∅ is rotated out, the carry flag will be reset, so that this flag can be tested and used to call routines which put out the numbers 15 or 7 according to the bit. The delay routine will be used between each of these port output operations, and when all eight bits have been sent out (you will have to count them!), then two stop signals (1) will be sent, again using the same time delay.

```
75        11∅       15∅       3∅∅       12∅∅
24∅∅        48∅∅        96∅∅        192∅∅
```

Fig. 7.18. Standard Baud rates.

This is a comparatively simple scheme, and it ignores a convention called parity, which is used to check for signal corruption. Parity means making the number of 1's in a byte always either even or odd, according to choice. If you work with even parity, the number of 1's in a byte will always be even, and this is possible because RS232 is used for ASCII codes which consist of only 7 bits. The eighth bit (the most significant bit) can then be set or reset to make the total even if even parity is wanted, or odd if odd parity is wanted. There is a bit in the status register which can be used to check parity, so that elaborate programming is not needed. For driving a serial printer, the parity operation can often be ignored, and many circuits can work with ∅ and + 5 V signals in place of the -12 V and + 12 V signals that RS232 strictly should use.

That's the easy bit, though. The difficult bit is that the Spectrum stores its listings in compressed form, using the tokens in place of keywords, so we can't send out the bytes simply by reading through the memory. Once again, it's possible if you can find the routines in the ROM which are used for listing a program, and there is also the possibility of intercepting the signals at the printer port – but this is not beginners' work!

Chapter Eight

Debugging and More Programming

Debugging delights

Now that you have been exposed to the delights of machine code programming, the time seems ripe to talk about debugging. A bug is a picturesque name for a fault in a program, and debugging is the process of removing such flaws. There's probably a name for the programmer who put the flaws in place, but our task at present is to find how we can best deal with bugs.

The first part is prevention. Check your flowchart carefully to make sure that it really describes what you want to do, and then check your assembly language program equally carefully to make sure that it also does what you expect of it. Then check that the bytes that you are placing into memory correspond to the assembly language instructions and data. If you do all this you will eliminate quite a number of bugs before they start to crawl out of the woodwork. Don't feel that you are a failure if you don't get them all out – unless a program is very simple, there's a very good chance that there will be a bug in it somewhere, and it happens to all of us.

If you use an assembler, one very potent source of bugs disappears almost instantly. Human frailty means that the process of converting assembly language instructions into bytes in memory by looking up tables is an error-prone business, and by letting the computer tackle this, a lot of bugs can be prevented. I shall describe briefly the use of the ULTRAVIOLET© assembler program later in this chapter. At the time of writing the ULTRAVIOLET was the only assembler available for Spectrum, though another was being developed which relied on the use of the Microdrive. It is highly likely that by the time this book appears, a choice of assemblers will be available, and if machine code really catches your imagination and you want to branch out into much more advanced work than this book can cater for, then an assembler should be a priority item.

If, however, you want to use machine code only as a way of carrying out minor tasks that are impossible or too slow for BASIC, then the POKE-to-memory method that we have used throughout this book may be perfectly adequate. It does mean, however, that there will be bugs lurking in every corner of the code. The main cause of these bugs is tedium. Converting an assembly language program into denary bytes is a tedious job, and all tedious jobs generate mistakes (ever seen a 'Friday' car?). Faulty conversion of hex to denary is one possibility, just writing down the wrong figure is another which is surprisingly common. One very potent source of trouble is in JR or DJNZ displacement values. You may get the number wrong, or you may start with them correct and then forget that if you add or delete code between the jump start and its finish, you will also need to alter the displacement. This again is a problem which is solved when an assembler is used. An incorrect jump will almost always cause the computer to lock-up or go into its NEW routine, and unless you recorded your source routine (the BASIC program which POKEd the memory or which holds the assembler instructions), or the machine code itself, then you have lost what might have been a fair amount of effort. Another form of incorrect jump, of course, which is more difficult to spot, is doing the opposite of what you intended, such as JR Z in place of JR NZ. Careful thought about what the jump will do for various bytes should eliminate this one.

All of these problems can be overcome by meticulous checking, and it pays to be extra careful about JR displacements, and about the initial contents of registers. A very common fault is to make use of registers as if you could assume that they contained zero at the start of the program. You can never assume this – it's much safer to assume that each register will contain some value that will drive the program bananas if it is used! What do you do, however, if you have checked everything in sight, and the program completely refuses to do what you expect?

There's no simple answer to that one. It may be that your flowchart doesn't do what you expected it to, and if you didn't draw one, then you've got what you deserved. It may be that you are making use of a Spectrum ROM routine, and it doesn't operate in the way that you expect – until we have a complete breakdown of the ROM, we simply have to use trial and error. All I can do here is to give you some hints about removing the bugs from problem programs that seem to be reasonably well constructed but which don't work according to plan.

The first golden rule is never to try anything new in the middle of a large program. Ideally, your machine code program will be made up from subroutines, each of which you have thoroughly tested before assembling into a program. In real life, this is not so easy, because Spectrum cannot MERGE machine code routines, but it is still possible to load recorded code into sections of memory that are adjacent to each other, or to MERGE data lines of BASIC programs, so that more bytes can be POKEd, which is a much safer method. If you keep well-tried subroutines on tape, preferably as BASIC POKE programs, then you can save a lot of programming effort by combining them, remembering that if any of the subroutines are non-relocatable, you will have to alter address bytes within the program. Once again, users of an assembler have the best of all worlds, because if the assembly language instructions are stored within REM statements, they can be MERGEd and edited to your heart's content before being assembled.

Even if you do not use a subroutine library on tape, it helps to keep a note of routines. *Personal Computer World* runs a series called SUBSET, which prints several general-purpose machine code subroutines each month, and most of these are Z-80 routines, reflecting the importance of this microprocessor type. Even if you don't use the routines, the way in which they are documented should give you some ideas about how you are going to keep a record of your own routines and I personally think that this feature is worth my annual subscription by itself. If you are going to use a new routine in a program it makes great sense to try it out on its own first, so that you can be sure of (a) what you need to have in the registers before the program is called, and (b) what you will have in the registers after it has run.

This type of planning should eliminate bugs in a big way, but if you are still faced with a program which you don't want to start pulling apart, routine by routine, the best method of dealing with it, assuming that no good monitor program is available, is to insert breaks. A break as far as a Spectrum machine code program is concerned, is the RET instruction, coded 2Ø1. When this is encountered, the contents of the BC register are handed to the Spectrum operating system, and normal service is resumed. Now if you want to find where the problem lies in a program, the way of using a break is to pick a spot where you want to check a value. This value might be in one of the other registers, so you will have to add an instruction before the RET which will place the contents of that register or registers into the BC pair. The break bytes can then be edited into the POKE part of the program in the DATA lines, and the program assembled and tested. When the program reaches the breakpoint, it will dump a value into the BC registers and then return to BASIC leaving you to chew over the meaning of the number that appears. If the program appears to be working well as far as this breakpoint, then remove it and put another breakpoint in at a later stage. By repeating this process, you should eventually be able to find where the fault lies, and this is usually all the clue you need to find what the fault is.

The most awkward faults are faulty loops, because they almost invariably cause a lock-up, and on the Spectrum there is no way out of this. Some machines have a 'hardware reset', a button which can be pushed to restore the machine to normal operation even if it has become locked into a machine code loop. This is not available on the standard Spectrum but it does appear on other Z-80 based machines. It is likely, therefore, to be offered by one of the many independent hardware suppliers, and it would make life much easier for assembly language programmers.

One fault which can so often cause such an endless loop lock-up is a loop back to the wrong position. For example, if we had a program such as:

```
      LD B, 255
Back: OUT (Port),A
      DJNZ Back
```

assembled 'by hand', we may have made the DJNZ instruction loop back to the LD B,255 instruction rather than to the OUT instruction. This will result in the B register being topped up to 255 each time the loop is executed, so that the DJNZ can never decrement B to zero, and so the loop is endless. A mistake like this is easily spotted in assembly language, because the position of the label name is easily checked, but it can be very difficult to find when you can only see the machine code bytes. Once again, taking care over loops is the only answer, and the method that has been shown in this book of writing the machine code bytes against the assembly language instructions is a very good way out of the problem.

Monitors

I mentioned monitors briefly a few pages back. It is unfortunate that the word 'monitor' has come to be used for two different items that are both associated with computers. The TV engineering definition of a monitor is of a TV-type display which accepts TV signals directly, rather than by conversion to a transmitter-type signal as is used by Spectrum. The other meaning of monitor is a program that checks (monitors) every action of the machine, and this is the type of monitor I mean in this section – we'll keep the name video-monitor for the display.

Monitors have developed over the years. In the early days of home computers, the most that could be expected of a monitor was that it would display a section of memory in hex, change memory contents to other values one byte at a time, shift a set of bytes from one starting address to another, and possibly insert breakpoints.

A few computers in days gone by had machine code monitors built in, sometimes confusingly referred to as 'front-panels', a name used in the dinosaur days of computers. Nowadays, monitors are available as programs on tape or disc which can greatly extend the usefulness of the machine for checking machine code. Some recent monitors, for example, allow a machine code program, whether in RAM or in ROM, to be run one step at a time, and will display register and memory contents (of memory locations affected by the step) at each step. A monitor of this type, of which the best known example is the Mumford STEP-80 for the TRS-80, is to a machine code programmer what a power drill is to the handyman – you start to wonder if life could continue without it.

In the early days of Spectrum, when this book is being written, there is not a great deal of choice of monitors for the Spectrum. I have made use of the monitor facilities of the Campbell Disassembler, which is one of the first useful programs of this type specifically for the Spectrum. Undoubtedly, there will be many more by the time this book is published, so that machine code debugging on the Spectrum should eventually become relatively painless.

Using an assembler

Though at the time of writing only one assembler was available for the Spectrum, some description of how an assembler is used is necessary in any book dealing with assembly language and machine code – it would be like describing motor maintenance with no mention of spanners if I missed this out. Assemblers are so useful for machine code programming that it can't be long before there are as many assemblers available for the Spectrum as for well-established machines like the TRS-80.

An assembler is a program, usually in machine code, which has to be loaded like any other program. This is a fairly fast operation, even when cassettes are used. Once the assembler is loaded, it may present you with a menu of options. Typical of these would be entry or editing of assembly language, saving or loading of a program written in assembly language (known as source code) or one written in machine code (known as object code), assembly of source code into machine code, and movement of an editing 'pointer' which allows lines to be selectively edited. A session of code writing would start by the selection of the writing option.

Each assembler has its own peculiarities, often reflecting peculiarities of the machine on which it is used, but most Z-80 assemblers follow the standards of assembly language that were laid down by Zilog, the designers of the Z-80. Rather than deal with assemblers in general, however, it's probably more useful at this point to illustrate this section with reference to the first assembler that became available for the Spectrum, the ACS ULTRAVIOLET.

The ULTRAVIOLET assembler

The ULTRAVIOLET assembler accepts address and data in denary, with no hex needed at the writing stage, so that it is easy to use with the denary numbers that are shown in the Spectrum manual. All of the Z-80 instructions are correctly assembled, and code can be placed into memory directly (with some precautions) and if necessary relocated. Full listings of the assembly language and the code, with the curious mixture of denary for addresses and hex for codes can be displayed or sent to the ZX printer, which is particularly useful if long programs are being developed.

The style of the ULTRAVIOLET assembler, however, is very similar to that of assemblers for the ZX-81, and it does not take full advantage of the ability of Spectrum to make memory space for machine-code in high memory addresses. The program also requires the assembly language statements to be written inside BASIC REM lines, as was used for the ZX-81, and the code is also stored in a REM line, the first line of the program. Assembled code can, however, be written in a form that will allow it to be recorded and correctly relocated to high memory when it is reloaded, with all addresses corrected. Despite the disadvantages of being modified from a ZX-81 program (which was inevitable because to write an assembler from scratch is an awesome task), it is a useful piece of software which at the time of writing looks like being the only assembler available for some time to come.

ULTRAVIOLET, like its matching disassembler, INFRARED (I wonder if they'll make a monitor called X-RAY?), is available in 16K or 48K versions, of which I have used only the 16K version. This is on a cassette which, once I had adjusted the head angle of my cassette recorder slightly, loaded easily. The system uses a loading section, titled 'uv loader', a main piece of machine code titled 'uv 16K', and a display section ('uv 16K') which shows a few instructions – detailed instructions come on a printed sheet. When the display is complete, pressing any key will produce the switch-on display, with the copyright notice showing. This, though alarming, is perfectly normal for this program – it means that the program is in place in protected memory and ready for use.

Writing the program

The assembly language program has to be written using lower case for the assembly language instructions, and in a BASIC REM line. The first REM line should contain enough spaces, or any other characters, to leave room for the machine code which will be placed in it. Since BASIC starts at 23755, and the first four bytes of the first line are line number and length number, with the fifth byte the REM token, the machine code bytes will start at 2376Ø. The second line of the program must contain the word 'go' following REM – this is the 'start assembly' command. The third line must contain the origin address, written as 'org', which will normally be 2376Ø. Any attempt to assemble code into high memory, such as 325ØØ, for example, is likely to overwrite the code of the assembler itself and cause a crash. This can be avoided by using a modification of the 'org' statement, which places the relocation address in brackets after the 'org' address. For example, using org 2376Ø(325ØØ) will cause code to be assembled at 3276Ø, but in a form which will run correctly when the bytes of code are shifted to 325ØØ (this can mean that it would not run at 2376Ø!). The relocation can then be performed by saving the code and then loading it into the higher address. You are not restricted to a single 'org' statement, and the assembler can be used to produce a number of machine code sections which are located at different addresses.

Following the 'org' statement, the assembly language program can be typed in, following the rules of the ACS system. Each line must start with a REM, following which the assembly language can be typed using *lower case letters*, as used in the Spectrum manual, for the instructions. Label names *must start with a capital* (upper-case) letter, and can be of any length. It's best, however, to stick to short names to save memory, particularly on a 16K machine which does not have much memory left when the ULTRAVIOLET assembler has been loaded in – the length of the code is about 5000 bytes. The *close-bracket* and + characters must not be used in a label name, otherwise there are no restrictions. The separating mark between a label and the assembly language following it is a *semicolon*.

Comments have to be preceded by an exclamation mark rather than by the semicolon which is normally used in Z-80 assembly language work. Up to 32 characters of comment can be printed on a line; if more are needed, then a new REM line must be used. Several assembly language statements can be written in a single assembly language line following a single REM – it appears that unless your program is very large, you could place all of it in one line! The statements must, however, be separated by semicolons. This is essential because if a colon is used as it would be in BASIC, the Spectrum operating system will then treat the next key as signifying a keyword rather than a letter.

The end of the assembly language is signalled by another REM line which contains the word 'finish'. The BASIC program can then be checked and edited in the usual way, and when you are satisfied that it looks correct, assembly can start. This, for the 16K version, means typing RANDOMIZE USR 275ØØ – the figure for a 48K version is 6ØØØØ – which if all is well will produce the program code on the screen. The display shows the assembly language and the generated machine code, with addresses in denary and code in hex, so that if you want to copy the bytes to use in a BASIC POKE program, you will have to convert them for yourself. The assembler runs twice, causing the screen to seem to flicker when the program is a short one. The reason is that on the first run through, the assembler may find label names to which it can't allocate an address, because the addresses for these labels come later. They are noted, and on the second run through all such 'forward references' are correctly allocated. A short program will occupy less than one screen, and if a printout is needed, the COPY key of Spectrum has to be used. When the assembled code requires more than one screen, assembly will stop at the bottom of a screen, and will not continue until a key is pressed. On the second run, when a screen is filled, pressing the 'p' key will cause that screen to be printed. This is necessary because Spectrum facilities like the COPY key cannot be used while the assembler program is operating.

Error messages

Errors are very well detected and pointed out. If there is a fault in 'go', 'org', or 'finish' statements, an error message will be printed instead of starting assembly. If the fault is in the assembly language syntax, such as writing lda,12 in place of ld a,12, then assembly will stop at the faulty line, and the error message will show in detail which statement in the BASIC REM program is faulty. There will also be a Spectrum operating system error message, which will be "B integer out of range", or "2 variable not found" or "Q parameter error", but these messages are less important.

Useful features

As well as permitting the 'org' instruction (strictly speaking, this is a *pseudo-instruction*, meaning that it does not assemble into code), the ULTRAVIOLET permits four others of this type. One is equ, which can specify an address for a label name in advance. For example, the line:

```
30 REM equ 23560 Loop
```

will ensure that the address 23560 is filled in wherever the word Loop is used as an operand. This is very useful if a program has to be written in several versions (such as for 16K or 48K, for example), as all the addresses can be altered just by changing a few equ statements. Another use suggested by the program writers of ULTRAVIOLET is in linking separate sections of code so that they will run as one, as the 16K version of ULTRAVIOLET does.

Other useful pseudo-instructions are defb, defw and defs, which allow a form of POKE operation. Using defb will place the single byte number that followed defb into memory at the address to which the statement is assembled; defw allows two bytes to be placed into memory, low byte first in this fashion. The defs pseudo-instruction is one of the most useful of this group, because it allows a string of ASCII codes to be stored by specifying the letters. For example, defs Sinclair will store the eight ASCII codes for the name Sinclair.

Other features and summary

It is possible to write code that will change addresses or data by itself – self-modifying code. By labelling a load instruction with a label word, for example, the bytes following the load instruction (byte to be loaded or address bytes) can be changed by a later piece of code which loads to the address label + 1 or label + 2. This is rather more

```
10  REM

20  REM go
30  REM org 23760
40  REM xor a
50  REM ld de, Addr
60  REM call 3082
70  REM set 5, (iy + 2)
80  REM call 5588
90  REM ret
100 REM Addr; defb 128
105 REM defs, Push any key ...
110 REM defb 174
120 REM finish
```

Fig. 8.1. How assembly language instructions are written in the ULTRAVIOLET assembler. This program can be saved by normal BASIC methods. It is assembled by calling the assembler – for the 16K version this means typing RAND USR 27500.

```
org 23760
23760  AF                xor a
23761  11   DF   5C      ld de, Addr
23764  CD   0A   0C      call 3082
23767  FD   CB   02   EE set 5, (iy + 2)
23771  DC   D4   15      call 5588
23774  C9                ret
Addr
defb 128
defs Push any key
defb 174
```

Fig. 8.2. The appearance of the assembled code on the screen. The program prints a message and waits for a key to be pressed.

advanced programming than we can cover in this book, but it is a very useful feature of the assembler.

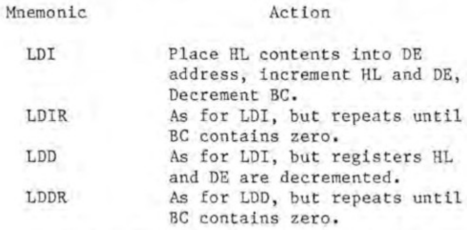
Once its features, which seem odd to anyone who has used other assemblers, have been mastered ULTRAVIOLET is a useful program. It is not, however, in the same class of assembler as the superb ZEN, which is used on TRS-80, Nascom, and Sharp machines. ULTRAVIOLET will, however, be a program with which ZX-81 machine code programmers will feel at home, and it is a useful tool, particularly for the 48K Spectrum, which has much more memory to play with. The programs in this book were written before ULTRAVIOLET became available but the demonstration program in Figs 8.1 and 8.2 gives some idea of how ULTRAVIOLET can be used, and the type of printout that it produces. Another assembler, from Abersoft, uses the Microdrives, and was not available when this book was written.

Chapter Nine
Last Round-up

One of the problems of writing a book about machine language programming for beginners is knowing where to stop. Volumes could be written about Spectrum programming, and still leave room for more, so that any finishing point has to be rather arbitrary. My aim has been to introduce machine code programming of Spectrum in such a way that the reader can then progress to much more specialised books. This chapter is concerned with tying up loose ends, mentioning a few more instructions, and illustrating how to make use of some more features of the Spectrum.

We'll start with another set of Z-80 instructions, the block-shift set. Block-shift is a good summary of what these instructions are about – the ability to move code from one part of memory to another simply by setting up registers with memory addresses and using a single assembly language instruction. A list of these codes, with a brief explanation of what each one does is shown in Fig. 9.1 , and since they are so similar to each other in operation, we'll single one of them, LDIR, out for demonstration.

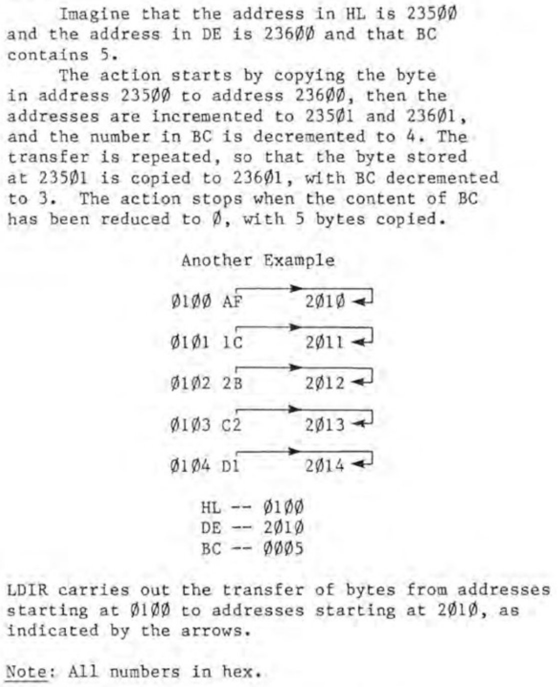
LDIR is a shortened version of Load, Decrement, Increment



Mnemonic	Action
LDI	Place HL contents into DE address, increment HL and DE, Decrement BC.
LDIR	As for LDI, but repeats until BC contains zero.
LDD	As for LDI, but registers HL and DE are decremented.
LDDR	As for LDD, but repeats until BC contains zero.

Fig. 9.1. The block-shift commands of the Z-80.

Register. It uses the double registers HL, DE, and BC, with HL holding the address of the start of a 'source' block of memory, DE the address of the start of a 'destination' block, and BC holding the number of bytes of code that are to be transferred from the source block to the destination block. When the LDIR instruction is executed, a byte is copied from the HL address to the DE address, the BC count number is *decremented*, and both the HL and the DE address numbers are *incremented*. This continues until the number in BC has reached zero. The action is illustrated in Fig. 9.2.



Imagine that the address in HL is 235ØØ and the address in DE is 236ØØ and that BC contains 5.

The action starts by copying the byte in address 235ØØ to address 236ØØ, then the addresses are incremented to 235Ø1 and 236Ø1, and the number in BC is decremented to 4. The transfer is repeated, so that the byte stored at 235Ø1 is copied to 236Ø1, with BC decremented to 3. The action stops when the content of BC has been reduced to Ø, with 5 bytes copied.

Another Example

```
Ø1ØØ AF      2Ø1Ø
Ø1Ø1 1C      2Ø11
Ø1Ø2 2B      2Ø12
Ø1Ø3 C2      2Ø13
Ø1Ø4 D1      2Ø14

     HL -- Ø1ØØ
     DE -- 2Ø1Ø
     BC -- ØØØ5
```

LDIR carries out the transfer of bytes from addresses starting at Ø1ØØ to addresses starting at 2Ø1Ø, as indicated by the arrows.

Note: All numbers in hex.

Fig. 9.2. The action of LDIR.

```
VIDEO   EQU   16384
LENGTH  EQU   6143
START:  LD HL, VIDEO
        LD DE, VIDEO + 1
        LD BC, LENGTH
        LD (HL), 68
        LDIR
        RET

1Ø  CLEAR 325ØØ: LET j = 325ØØ
2Ø  FOR n = Ø TO 13 : READ b
3Ø  POKE j + n, b : NEXT n
4Ø  LET z = USR j
1ØØ DATA 33, Ø, 64, 17, 1, 64, 1,
    255, 23, 54, 68, 237, 176, 2Ø1
```

Fig. 9.3. Using LDIR to produce a screen pattern. Try this for speed!

Fig. 9.3 illustrates this program instruction at work in an application which produces a pattern on the Spectrum screen. The number which is loaded into the HL register pair is the start of the display file of Spectrum, so that the loading to these addresses will cause something to appear on the screen. The DE registers are loaded with the next address up from the one held in HL, and BC is loaded with the number of bytes to transfer – a number which is the difference between the end and the start addresses of the display file, less one. The first address, the one held in HL, is then loaded with the byte 68 denary, which corresponds to the binary number Ø1ØØØ1ØØ. When the LDIR action starts, the byte 68 which is held in address 16384 is copied into 16385. The BC register then decrements, the HL address increments to 16385 and the DE address increments to 16386. Next time round, the byte which has been placed into 16385 is copied into 16386, and this 'bucket-chain' process continues until the whole of this section of memory has been filled with the byte 68. Convert this one into bytes for yourself, and watch it run – it's a powerful demonstration of how fast machine code can be for purposes like this. This is why machine code is preferred for writing fast-action games. The screen can be cleared and the action called up each time you want it by typing LET x = USR j , or RANDOMIZE USR j. If you use PRINT USR J, then the number Ø will appear at the top left hand side of the screen which rather spoils the effect. If you feel that a finer grid would look better, try the number 85 in place of 68; the clue to the pattern is the appearance of the 1's and Ø's in the binary version of the number, because each 1 corresponds to a dark dot, each Ø to white. It's amusing to write a BASIC version of this program, POKEing the number into each memory address of the display file in turn. Try it, and see how long it takes by comparison.

Passing values on

We have seen how the operating system of Spectrum is arranged so that values which are stored in the BC registers at the end of a machine code routine can be passed back to BASIC at the RET command. The Spectrum manual says nothing, however, about passing values *to* the machine code routine. Other machines which feature USR allow values to be passed as the number which follows USR, but the Spectrum uses this number to locate the address of the machine code. This does have the advantage that it's easy to find out what various routines in the ROM do – just type RANDOMIZE USR address, with the routine address in place, press ENTER and watch what happens!

One rather neat method is to pass a value which is stored as a BASIC variable. If we have a variable n, just to take an example, it will have an entry in the variable list table which looks like the example in Fig. 9.4. Provided that we stick to integers (positive whole numbers), then the value in the variable list table can be passed easily to a machine code subroutine provided that the machine code subroutine can locate it. This is made possible by the fact that the address of the start of the variable list table is held in RAM at 23627. Let's take a look at a simple (though rather limited) program.

```
Variable list table entry for LET n = 5

11Ø      Ø      Ø      5      Ø      Ø
start                                end
```

Fig. 9.4. VLT entry for variable n – a reminder.

The search routine must pick up the bytes in 23627 (low byte) and 23628 (high byte), and assemble these two into an address which is the address of the variable list table. If this address is put into the HL register pair, we can then use one of the 'search-and-report'

Mnemonic	Action
CPI	Compare byte stored in HL address with byte in A. If the two match, the Z-flag will be set, otherwise not. The HL address is then incremented, and the BC count is decremented.
CPIR	As for CPI, but if no match is found the action repeats until BC has been decremented to zero.
CPD	As for CPI, but after the comparison, the HL register is decremented.
CPDR	Automatic version of CPD.

Fig. 9.5. The block-search commands of the Z-80.

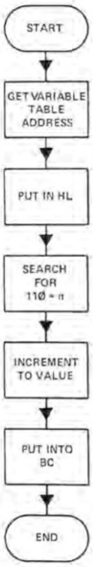


Fig. 9.6. The flowchart for finding a variable name.

commands of the Z-80 to find the variable name 'n' which, for a simple number variable, is coded as its ASCII code 11Ø. The search-and-report command is CPIR, which is a block-search command, and a summary of this group of commands is illustrated in Fig. 9.5. CPIR will search through a specified block of memory byte by byte, looking for a matching byte. The address of the start of the block of memory is put into HL, the byte that you want to find in A, and the maximum number of bytes that you want to search through in BC. Using the CPIR code (which is a two-byte code) then causes the byte at the address held in HL to be compared with the byte in the accumulator, and if no match is found, the HL register pair is incremented, the BC pair decremented, and the search continues until either a match is found, or the BC register contains zero.

Fig. 9.6 shows the flowchart for this program. We are not going to make any use of the variable value in this example, because at the moment we are more interested in how to find it rather than what to do with it! The key parts of the flowchart are getting the address of the variable list table, searching for the address at which the variable name is located, and then shifting to the address where the bytes of the value are stored. If we confine ourselves to integers, positive numbers less than 65535, the number will be stored as two bytes only, and these can then be loaded into the BC registers, so that we can print the number on the screen as confirmation that we have found it.

The CPIR instruction will leave the value of the HL address incremented, so that it has gone one step beyond the address of the variable name when the name of the variable has been found. Only two more increment steps will therefore be needed to get to the address of the low byte of the value.

Fig. 9.7 shows the assembly language version of this flowchart. The HL register pair is loaded with the 'pointer' address in RAM where the low byte of the variable list table is stored. This byte is loaded into the E register, and the address is then incremented so that HL contains 23628. The byte stored here, the high byte of the address, is loaded into the D register. This leaves DE holding the address of the variable list table, with the bytes in the correct order, high byte in D, low byte in E. We shall, after one more instruction, then use a command which I haven't introduced earlier – EX DE,HL. This does what you might expect; it swaps the numbers in DE and HL, so that HL will now hold the address of the start of the variable list table, and DE will hold the address 23628, which doesn't interest us any longer. This swap command is used extensively when addresses are being assembled, because we always try to work with important addresses in HL rather than in the other register pairs.

```
LD HL, 23627    ; pointer to variable table
LD E, (HL)      ; get lower byte
INC HL          ; bump up address
LD D, (HL)      ; get higher byte
LD BC, FFFFH    ; maximum count
EX DE, HL       ; get start of table address
                  into HL
LD A, 11Ø       ; byte to recognise
CPIR            ; look for it!
INC HL          ; move to ...
INC HL          ; value, twice
LD C, (HL)      ; low byte to C
INC HL          ; point to high byte
LD B, (HL)      ; high byte to B
RET             ; back to BASIC
```

Fig. 9.7. The assembly language program for finding the variable name.

In this example, BC has been loaded with the maximum possible count number. In practice, it might be wiser to limit this to a smaller quantity to avoid long searches for non-existent variable names. A further refinement is to print an error message if the variable is not found, which means if BC is decremented to zero with no match found. Error messages can be called by putting the error message code number into the memory following a RST 8 command. For the sake of example here, however, the maximum length for BC has been used – it also stops the program from growing beyond the bounds of a beginner's exercise! Remember – if you should change the value in BC – that BC must still be loaded with two bytes. So, if you want to search 1ØØ bytes only, you must use 1,1ØØ,Ø , where 1 is the LD BC,NN code, and the two following are the bytes of the number.

We now use CPIR to search for a byte equal to 11Ø. This makes the program rather vulnerable, because if the variable list table contains a lot of entries preceding the entry for n, and one of these happens to be one which contains the number 11Ø (such as LET s= 11Ø, or, less obviously, LET s= 28165, which is 5+ 256*11Ø), then the address of this value will be found rather than the address of the value of n. This can be avoided by making the program considerably more elaborate, but we're trying to illustrate principles here, not to show off. If you want a clue, a good one is to look at the most significant three bytes of the first variable name, and use this to jump directly to the next one, and so on. This can't be done using CPIR, which is why I want to illustrate the simple method.

Once the address has been found, remembering that HL now contains an address one higher than the address of the variable name, we can increment HL twice to find the low byte of the value of n. This is put into C, HL is incremented again, and the next byte, which is the high byte, is put into B. On return, then, BC will contain the value that we are looking for.

```
10   CLEAR 32500
20   LET n = 240
30   GO SUB 1000
40   PRINT USR j
50   GOTO 9999
1000 LET j = 32500
1010 FOR x = 0 TO 19: READ b
1020 POKE j + x, b: NEXT x
1030 RETURN
1040 DATA 33, 75, 92, 94, 35, 86, 1, 255,
     255, 235, 62, 110, 237, 177, 35, 35,
     78, 35, 70, 201
```

Fig. 9.8. The BASIC POKE version of the variable finder.

Fig. 9.8 shows a BASIC program which contains the code and illustrates the routine in action. At line 1Ø, the assignment LET n = 24Ø places this value into the variable list table, and the USR routine which is called at line 3Ø will find this value. Values between Ø and 65535 can be used for n, but fractions and negative numbers have to be avoided.

The usefulness of this method does not end with passing a single number variable value. Since the program finds the address of a variable name in the list table, it can be used, with a few modifications, to find the names of string variables, number arrays or string arrays, providing you remember how these names are coded (see Chapter 2). For example, if a number array is to be used, the variable name is followed by a count number. This can be loaded into BC, and a loop set up to find the values and transfer them into RAM addresses held in register pair DE. Looking at the analysis of the variable array in Fig. 9.9, you can see that the length of the array is stored in two bytes, and is followed by a number of dimensions, and then two bytes for the size of the first dimension. If we get the address into HL of the first byte after the dimension size, this is the start of the array, three bytes on from where counting starts (subtract 3 from BC). The numbers can then be read by the procedure shown in the flowchart of Fig. 9.6.

The array is called 'a', and the values are 1 to 10, so that a(1) = 1, a(2) = 2, ... a(10) = 10.

The variable list table entry reads:

$$\underset{\text{'a'}}{129}\quad \underbrace{53\quad \text{Ø}}_{\text{length}}\quad \underbrace{1}_{\text{dim}}\quad \underbrace{1\text{Ø}\quad \text{Ø}}_{\text{1st dim}}\quad \underbrace{\text{Ø}\quad \text{Ø}\quad 1\quad \text{Ø}\quad \text{Ø}}_{\text{1st value}}$$

$$\underbrace{\text{Ø}\quad \text{Ø}\quad 2\quad \text{Ø}\quad \text{Ø}}_{\text{2nd value}}\ \ldots\ldots\ldots\ \underbrace{\text{Ø}\quad \text{Ø}\quad 1\text{Ø}\quad \text{Ø}\quad \text{Ø}}_{\text{last value}}$$

Dim means dimension of array, how many groups of figures are contained in the backets; for example, a(4) is a single dimension array, b(2,3) is a two dimension array, and so on.

The length is the total number of bytes following the two length bytes - that is from the 1st dimension byte to the last item.

Fig. 9.9. The typical VLT entry for a number array, when the number has one dimension only.

By incrementing DE each time a number is read, the values can be put into consecutive addresses and can be used in routines, such as sorting routines, which are beyond the scope of this book.

String and string arrays entries in the variable list table can also be accessed in this same way, providing that you remember the methods that are used for coding the variable names, and the following values.

Horses for courses

A lot of books on machine code programming devote a lot of space to numerical routines – additions, subtractions, multiplications and divisions. In general, this is rather a waste of effort, because these routines exist in the Spectrum ROM, and can just as easily be called up using BASIC. The fact that they run slower in BASIC is very seldom a real disadvantage. Any program that requires a lot of arithmetic is therefore better written in BASIC, unless it would run so painfully slowly that it would be unusable – some of the 'spreadsheet' financial programs fall into this class. Where machine code really comes into its own is when you want high speed, perhaps for graphics work, or for actions which don't exist normally, like sending serial printer bytes out from the cassette port. At some stage during the development of a program, you will have to decide whether it should be written entirely in BASIC, or mainly in BASIC with some machine code sections, or entirely in machine code. Unless you have an assembler, I would strongly advise you not to tackle any program entirely in machine code unless it is fairly short.

Many programs, then, can best be tackled by interleaving bits of machine code with other sections of BASIC. The USR address method of accessing machine code that the Spectrum uses is a recognition of this, so it's wise to take advantage of this provision. For many programs that make use of graphics, only those parts which require rapid movement, or very fast filling of the screen (or part of it) need to be written in machine code. Several blocks of machine code can be held in the RAM, providing enough space has been reserved; and each can be called by putting its starting address following the USR command, allowing you the choice of placing as many machine code sections as you want in with your BASIC steps. Passing variables to the machine code routine (another method will be illustrated later) means that you can attempt actions like starting a screen-fill routine at a place on the screen that can be specified by the value of a variable, or moving a pattern at a speed that is decided by another variable values (because it is used as a count variable in a DJNZ delay loop).

The important thing to avoid is writing machine code for its own sake. Machine code can become hypnotically fascinating, and it is very tempting to embark on very large programs simply to convince yourself that you can master this language. It's a temptation that has to be avoided unless you have time and the inclination to do it. If your object is simply to produce efficient programs, then a mixture of BASIC and machine code is probably a better bet. If you want to spend your time generating fast-action games programs, then it might be more profitable to look at alternatives to BASIC, such as FORTH, rather than working directly in machine code. Choose the right horse for your course, and you will be on to a winner.

Last fling

We're nearly at the end of this book, which means that you can hang up your 'L' plates as far as machine code is concerned, and start looking at some of the books that are mentioned in Appendix A. Before we part, though, how about one last useful utility. This one is to find any line number address, that is, the starting address in memory for any BASIC line. It's useful, because the techniques that are used for this one are also usable in a lot of other utilities,

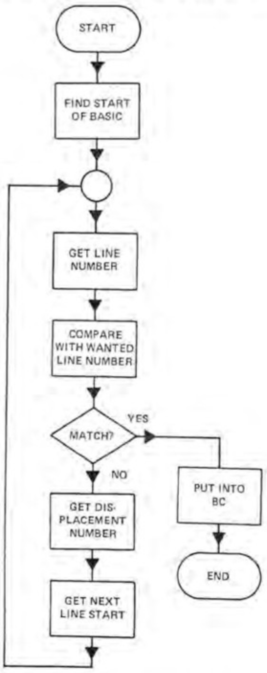


Fig. 9.10. Finding the address of the start of a line in a BASIC program – the flowchart.

particularly in those which renumber BASIC lines. It's also by far the most elaborate program that we shall look at in this book, making me wish that I had been able to get hold of an assembler rather earlier!

As a 'front end', I have used the utility we have already developed – the one which places into BC the value of an integer whose name is 'n'. This is certainly not the only way of putting this value into place. Later on we shall look at a much simpler method, but I want to show how one section of subroutine can be joined to another to make a longer program. The variable value finder, remember, puts the value of the integer variable into the BC register pair. Since line numbers are always positive integers, this will do very nicely, thank you, and we'll use the whole routine as it is. The next stage, then, is to design a program which will go through the lines one by one, checking the line numbers, until it finds line number bytes that match the bytes in BC. For the moment, we won't worry about what the program does if it doesn't find the numbers!

The flowchart is shown in Fig. 9.10. The first item is to find the starting address of the BASIC program. This is done by placing the address taken from reserved RAM into the HL pair. When we have the start address, we know that the first two bytes of code will be the first line number, in unorthodox high-byte, low-byte order. We can therefore extract these and compare them with the bytes we have stored in BC, which are the line number bytes handed on by the variable finder part of the program. We can't compare both at once, so this has to be a two-step business; the flowchart doesn't have to go into such detail. If the bytes match, then the address bytes from HL are put into the BC register pair, and the program returns to BASIC so that the address in the BC pair will be printed. If there is no match, then we have to read the next two bytes of the line. These form a 'displacement' number which will give us the address of the end of that line. The start of the next line is one address number higher, and once we have incremented the address, we can return to repeat the whole checking process.

The flowchart in this example contains no fine details, and the assembly language is not quite such a good match with the flowchart as earlier examples were. When you are unsure of how to carry out any single flowchart item, it's a good idea to draw a more detailed flowchart *just for that item* – it's always time well spent. As an illustration, look at Fig. 9.11, which shows details of the COMPARE and MATCH stages of the flowchart. The reason for this is that two comparisons are needed. One is a comparison of the

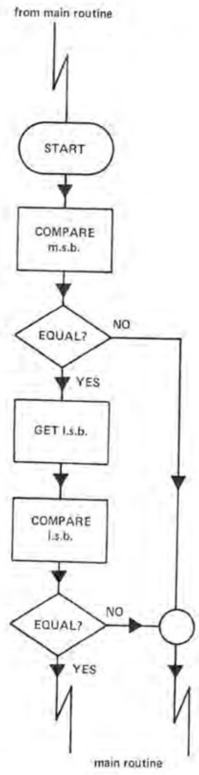


Fig. 9.11. A more detailed flowchart to show the comparison stages.

most significant byte of the line number with the byte in the B register; the other is a comparison of the least significant byte of the line number with the byte in the C register. If either fails, then you have to go to the GET DISPLACEMENT routine; if both succeed, then the two address bytes are put into BC and you return to BASIC.

```
     (At the start, BC contains the line number
          found by the previous routine)

      LD HL, 23635 ; get pointer      33, 83, 92
      LD E, (HL)   ; get low byte     94
      INC HL       : bump up pointer  35
      LD D, (HL)   ; get high byte    86
      EX DE, HL    ; start of BASIC   235
Comp: LD A, (HL)   ; msb of line      126
      CP B         ; equal?           184
      INC HL       ; bump pointer     35
      JR NZ, Next  ; not equal        32, 8
      LD A, (HL)   ; lsb              126
      CP C         ; compare          185
      JR NZ, Next  ; no               32, 4
      DEC HL       ; found match      43
      LD C, L      ; put into C       77
      LD B, H      ; and B            68
      RET          ; and back         2Ø1
Next: INC HL       ; displacement lsb 35
      LD E, (HL)   ; into E           94
      INC HL       ; bump pointer     35
      LD D, (HL)   ; msb to D         86
      ADD HL, DE   ; end of line      25
      INC HL       ; start of line    35
      JR Comp      ; try again        24, 235
```

Fig. 9.12 The assembly language version of the line finder. Remember that this is preceded by the variable finder!

Fig. 9.12 shows the assembly language version fully commented and with the denary byte numbers added alongside. The 'pointer' address is 23635, and by taking the bytes from 23635 and 23636, we can collect the address of the start of BASIC into the DE register – this is essentially the same type of programming as we used earlier in the variable finder. By exchanging DE and HL, we can then get the starting address into HL. The HL pair, remember, is more suitable for this purpose than the other two because it has a wider range of commands available, including the block search and the compare commands, though we don't use these commands here. Programmers tend to use HL for the most important source addresses, DE for destination addresses and for assembling address numbers to pass to HL, and BC for counting.

The first two bytes, beginning at the start-of-BASIC address, are the two bytes of the first line number. By loading A from the HL address, then, we get the msb of the line number, and we can compare it with the msb of the wanted line, which is in the B register. Before we act on this comparison, though, we increment the address number in HL, since this saves complications later. This is one of the operations which does affect flags in the status register, so that we can follow it with JR NZ , using the flags that were affected by the comparison to jump to a 'find-next-line' routine if the bytes were not equal.

Very often, the msb's will be equal. They will be equal, for example, if the BASIC program has no line numbers greater than 255 and you have not asked for a number greater than 255. We then have to test the lsb's. This is done in the same way, using CP C, and then JR NZ to jump to the 'find-next-line' routine if these numbers are unequals. If a match is found, then the HL address has to be decremented again to point to the start of the line, and the bytes in H and L are transferred to B and C (note that there is no convenient EX BC,HL command) so that they can be returned to the BASIC program.

If no match has been found, the routine which starts at the label 'next' will find the starting address of the next line. The HL value is incremented, so that the address is now that of the lsb of the displacement, and this number is put into register E. HL is incremented again, and the byte at this new address is put into register D. By adding DE to HL and incrementing again (because DE + HL gives the address of the end of the line), we get the start address for the next line, and we can then repeat the comparison by jumping to the label word 'comp'.

This is about the length of program at which you start to want to use an assembler! It's rather tedious to assemble 'by hand', and more tedious to sort out, so that you can try it and decide for yourself by trying some modifications if this is the life for you. Before we end, however, I would like to point out one drastic simplification.

This concerns getting the line number into BC. In the program as it is used at present, we have made use of the variable finder routine, but this has snags. One of the main snags is that it forces us to have 'n' as the first variable in the list table, in case the same number appears anywhere else in the table. Ideally, we would like GOSUB 1ØØØ to appear in the first line of the program, assembling the code into memory once only and thereafter using it. This would greatly speed up the program because, as it is shown in Fig. 9.13, it carries out the POKE to memory each time the program runs, which wastes time. If, however, we run the subroutine once only, we shall place

```
5    CLEAR 325ØØ
1Ø   INPUT "Line number, please - "; n: LET N
     = INT n
2Ø   GO SUB 1ØØØ
3Ø   LET x = USR j: PRINT "Line is at - "; x
4Ø   GOTO 9999
1ØØØ LET j = 325ØØ
1Ø1Ø FOR x = Ø TO 46: READ b
1Ø2Ø POKE j + x, b: NEXT x
1Ø3Ø RETURN
1Ø4Ø DATA 33, 75, 92, 94, 35, 86, 1, 255,
     255, 235, 62, 11Ø, 237, 177, 35, 35, 78,
     35, 7Ø
1Ø5Ø DATA 33, 83, 92, 94, 35, 86, 235, 126,
     184, 35, 32, 8, 126, 185, 32, 4
1Ø6Ø DATA 43, 77, 68, 2Ø1, 35, 94, 35, 86,
     25, 35, 24, 235
```

Fig. 9.13. The program put into BASIC POKE form. It has the disadvantage of being slow, because the machine code is POKEd in at each run.

some entries ahead of the entry for n in the variable list table, and this is risky.

Fig. 9.14 shows another method. By eliminating the variable finding routine, we remove quite a lot of machine code, so that there is less to POKE, and by rearranging the BASIC program, we can speed up the action. The program can be put into high reserved memory for tape loading if desired – or it can be put in by a subroutine as before – but in either case, ahead of the INPUT statement in the BASIC portion. The machine code now consists of the LD BC,address (3 bytes) followed by the rest of the line-finder routine, unchanged. Since the routine is fixed at address 325ØØ in this example, the numbers that are loaded into BC at the start of this section are located at 325Ø1 and 325Ø2. If the line number is less than 256, then it can be placed directly into 325Ø1, because we have put values of 1Ø and Ø into the machine code. If the line number is greater than 255, then it has to be split into high-byte and low-byte and these bytes POKEd into the appropriate addresses.

Using this technique of placing a variable value, you can investigate the lines of a BASIC program. Start by the direct command CLEAR 325ØØ. Then load in the machine code on tape. Load in your BASIC program, the one you want to analyse, and then add, with high line numbers, the lines shown in Fig. 9.14 above the program that you want to investigate. By executing a GOTO (start of the line-finder BASIC program), you can then run the BASIC part of the line-finder, which will call the machine code as needed, POKEing the selected line number into memory, activating the machine code, and printing the result.

The machine code used in the line finder, but with no number variable finding routine. The first instruction is LD BC, 1Ø (code 1, 1Ø, Ø), so that BC is loaded with line number 1Ø in default, that is, if no other line number is passed to it. The BASIC program loads in the machine code, and then the line number is POKEd into the memory, so that its value is put into BC when the machine code is called. This allows the finder part of the routine to work in a loop, so that line 9Ø can be changed to GOTO 4Ø if needed.

```
1Ø   CLEAR 325ØØ: LET j = 325ØØ
2Ø   FOR n = Ø TO 3Ø: READ b
3Ø   POKE j + n, b: NEXT n
4Ø   INPUT "Line number, please - ";n: LET n
     = INT n
5Ø   IF n < 256 THEN POKE 325Ø1, n
6Ø   IF n>= 256 THEN LET m = INT (n/256): POKE
     325Ø2, m: POKE 325Ø1, n - 256 * m
7Ø   LET x = USR j
8Ø   PRINT "Line no. is - "; x
9Ø   GOTO 9999
1Ø5Ø DATA 1, 1Ø, Ø, 33, 83, 92, 94, 35, 86,
     235, 126, 184, 35, 32, 8, 126, 185, 32,
     4
1Ø6Ø DATA 43, 77, 68, 2Ø1, 35, 94, 35
     86, 25, 35, 24, 235
```

Fig. 9.14. A faster line-finder. The line number is POKEd directly into the machine code instead of using the variable finder. This allows the machine code to be placed once, and then used each time round.

Perhaps you might now like to design a BASIC line-finder program using PEEK and POKE, to see how long it takes to do the same task – another proof of the value of machine code. After that, you can start looking for problems to solve – you are a beginner no longer!

Appendix A
Books and Magazines

Computing is a rapidly changing business, with new ideas continually being aired, and it is only by extensive reading that you can hope to keep up with developments. One essential is to join a user group, because members of such groups rapidly explore all aspects of the performance of their chosen computer, and spread their knowledge around. The second point is to make use of magazines. *Personal Computer World*, as has been mentioned, runs a series which is concerned with machine code subroutines, and you will also find much that is useful in magazines such as *Computing Today*, *Electronics and Computing Monthly* and *Your Computer*.

The following books will also be helpful as you emerge from beginner's status:

Z-80 Assembly Language Programming by Lance A. Leventhal - a comprehensive text of Z-80 assembly language, both a reference book and a guide to style. This is a must if you are going to take Z-80 programming seriously.

Z-80 CPU Instruction Set, published by SGS-ATES - a complete list of all Z-80 instructions, codes, and effects. Not easy to follow, and sometimes frustrating because of the way it is organised, this is nevertheless a book I would not want to be without.

You may like to know that Alan Tootill, the author of the SUBSET series in PCW, is writing a book with David Barrow on Z-80 machine code which will be published by Granada Publishing in 1983. It is called *Z-80 Machine Code for Humans*. The authors' intention is to show how machine code can be fun!

Appendix B
Floating-point Numbers

The coding of integers has been dealt with in the text, and this appendix, which is for the strong in heart (and maths) only, deals with the coding of non-integers, or 'floating-point' numbers. A denary number can be represented in 'scientific' or 'standard' form as $N \times 10^n$, where N is a number less than 10 (and which can contain a fraction) and n is an integer, positive or negative. In this form, the number 10200 becomes 1.02×10^4, and the number 0.000345 becomes 3.45×10^{-4}. N is called the mantissa, and n is called the exponent when this scheme is used. Its advantage is that it makes very large or very small numbers easier to deal with.

The binary form of this notation is used for floating-point numbers. The mantissa uses four bytes, and the exponent uses one byte, but the digits that are placed in these bytes are not simply the digits of the numbers. To start with, the first byte of the number is the exponent to which has been added 128 (8ØH). The next four bytes are used for the mantissa, which is a binary fraction whose denary value lies between 0.5 and 1, never actually reaching 1. The first bit of this mantissa is always 1, so that it can be used as a sign bit. If it is replaced by a zero then this signals a positive number, but leaving it as a 1 signals a negative number. The exponent will be 128 or more for a number of 1 or more, and less than 128 for a fraction with no whole-number part.

The value of the mantissa hinges on how a binary fraction is written. Just as we use the places in a binary number which is a whole number to represent positive powers of two (1,2,4,8,16,32,64...), so we can use places after the binary (not decimal!) point to represent negative powers of two, 0.5,0.25,0.125,0.0625..., corresponding to $\frac{1}{2}$, $\frac{1}{4}$, $\frac{1}{8}$, $\frac{1}{16}$, etc. The binary fraction .101001 is therefore 0.5 + 0.125 + 0.015625 ($\frac{1}{2} + \frac{1}{8} + \frac{1}{16}$) = 0.640625.

A number is converted into this form by dividing it by the nearest (larger) power of two. For example, the denary number 1.64 is larger than 2^0 (which is 1), but smaller than 2^1 (which is 2), so we divide it by 2, and write it as:

0.82 (D) × 2^1

The 2^1 gives us an exponent of 1, and 128 is added to this, giving 129 as the first byte of the code. 0.82 then has to be converted to a binary fraction. This is done by comparing it to each fraction in the binary series (0.5,0.25,0.125, etc.) and subtracting only when the binary fraction is less than the number. Starting with 0.82, for example, subtracting 0.5 gives 0.32 remainder, and a 1 in the first place of the mantissa, as must always happen if the conversion is correctly carried out. The remainder of 0.32 is larger than the next binary fraction of 0.25, so another 1 bit is placed in the mantissa, and 0.07 is left. We can't subtract 0.125 from this, so a ∅ is placed in the mantissa, but 0.0625 will subtract, placing another 1 in the mantissa and leaving 0.0075. This process has to be continued until either a subtraction leaves no remainder, or until all 32 bits (four bytes) of the mantissa have been allocated. The conversion is actually carried out for 33 bits, and if the 33rd bit is a 1, then the 32nd bit is rounded up to 1 if it is zero. It is because the process of creating a binary fraction is *never exact* except for numbers which are negative powers of two, like 0.5, 0.125, etc., that floating-point numbers in computers and calculators are never exact.

When you place a variable value in the list table, then you make the computer carry out all of this work, but at least it has to be done only once, when the line is entered. If you use numbers in a program, like:

```
5∅ LET n = N*2.1416
```

then the conversion of the number to floating-point form from a string of ASCII codes has to be done in the program, and will slow the program down. This is why you are advised always to make such numbers into variables early in the program.

Appendix C
Coding of Arrays

Array of numbers

The first byte is the ASCII code + 32.
The next two bytes are the length of all the entries from the following byte to the end of the array.
The next byte is the number of dimensions.
The next two bytes form the first array dimension.
The other dimensions then follow, two bytes each.
The elements (values) then follow, five bytes each, in order. All the numbers whose first subscript is 1 are first in order, then these with a first subscript of 2, and so on, so that a typical order would be a(1,1),a(1,2),a(1,3),a(2,1)a(2,2),a(2,3) for a 2,3 dimensioned array.

String arrays

Strictly speaking, these are arrays of characters, with one dimension being the number of characters in each element.
The first byte is ASCII code + 96.
The next two bytes record the number of bytes to the end of the array.
The next byte is the number of dimensions.
The dimension bytes follow, two bytes each. There must be at least two dimensions for a string array.
The elements then follow, but only one byte is needed for each, because the elements are each ASCII character codes.

Appendix D

Z-80 Addressing Methods

There are some differences in the names used for the Z-80 addressing methods. The book by Leventhal, for example, uses the word 'implied' quite differently from its use here, and in a different sense from that used by some other authors.

Implied addressing: the address is implied by the instruction, and no reference to memory is needed. Example: RET.

Immediate addressing: the address of the data is the memory address following the address of the instruction byte.

Direct addressing: the full (two byte) address of the memory is contained in the two bytes following the instruction byte.

Register indirect: the address for the data is contained in a register pair, usually HL.

Indexed addressing: the address is found by adding a displacement byte (part of the code) to a 'base address' contained in an index register.

Program relative addressing: the address for data is obtained by adding a signed displacement byte to the program counter address.

Stack addressing: the byte is stored on the stack and can be removed by a POP command, placed back by a PUSH.

Zero page addressing: the RST instruction is a special form of subroutine call which uses addresses ØØH to 38H in sets of eight. If it is used extensively by the operating system of Spectrum, and is not available for other purposes.

Appendix E

Times for Operations

This table shows the time in terms of clock pulses for a few instructions. For the Spectrum clock operating at 3.5 MHz, the clock pulse time is 0.2857142 μs. The microsecond (μs) is one millionth of a second.

Operation	Time	Comments
LD A,r	4	load register A from any other
LD r,n	7	immediate load, any register
LD r,(HL)	7	indirect load, any register
LD A, (NN)	13	direct addressing
LD RR,NN	10	double register load
LD HL,(NN)	16	HL from two memory bytes
PUSH RR	11	push register pair
POP RR	10	pop register pair
ADD A,r	4	addition
INC r	4	increment register
INC(HL)	11	increment content of HL
RLA	4	rotate accumulator
JP NN	10	jump to address NN
JR disp	12	jump relative, unconditional
JR condition	12	condition not met
	7	condition met
CALL NN	17	call subroutine
RET	10	return

The block shift and compare instructions have not been shown here, because the time taken depends on how many times the instructions repeat. Full details of these timings will be found in the SGS-ATES book mentioned in Appendix A.

Appendix F

697 Z-80 Operating Codes, with Mnemonics, Hex Codes, Denary Codes and Binary Codes

The following list of all 697 Z-80 operating codes, with mnemonics, hex codes, denary codes, and binary codes consists of four columns. Where a code consists of more than one byte, the bytes have been arranged vertically, to avoid confusion. Where a data byte, displacement byte, or address has to be inserted, this is shown by using 00 for a data or displacement byte, or 0000 for an address. Customarily, this is shown on such lists by using N for a single byte of NN for an address (hence the position of these items in the otherwise alphabetical listing), but because a hex to denary and binary conversion program was used to produce the lists, letters such as N could not be used in the program.

The very considerable labour of producing such a list means that inevitably some mistakes are made during compiling the list. I have eliminated these are far as I know, but I shall be grateful if any errors are brought to my notice.

ADC A,(HL)	8E	142	10001110
ADC A,(IX+00)	DD	221	11011101
	8E	142	10001110
	00	0	00000000
ADC A,(IY+00)	FD	253	11111101
	8E	142	10001110
	00	0	00000000
ADC A,A	8F	143	10001111
ADC A,B	88	136	10001000
ADC A,C	89	137	10001001
ADC A,D	8A	138	10001010
ADC A,00	CE	206	11001110
	00	0	00000000
ADC A,E	8B	139	10001011
ADC A,H	8C	140	10001100
ADC A,L	8D	141	10001101
ADC HL,BC	ED	237	11101101
	4A	74	01001010
ADC HL,DE	ED	237	11101101
	5A	90	01011010
ADC HL,HL	ED	237	11101101
	6A	106	01101010
ADC HL,SP	ED	237	11101101
	7A	122	01111010
ADD A,(HL)	86	134	10000110
ADD A,(IX+00)	DD	221	11011101
	86	134	10000110
	00	0	00000000
ADD A,(IY+00)	FD	253	11111101
	86	134	10000110
	C0	0	00000000
ADD A,A	87	135	10000111
ADD A,B	80	128	10000000
ADD A,C	81	129	10000001
ADD A,D	82	130	10000010
ADD A,00	C6	198	11000110
	00	0	00000000
ADD A,E	83	131	10000011
ADD A,H	84	132	10000100
ADD A,L	85	133	10000101
ADD HL,BC	09	9	00001001
ADD HL,DE	19	25	00011001
ADD HL,HL	29	41	00101001
ADD HL,SP	39	57	00111001
ADD IX,BC	DD	221	11011101
	09	9	00001001
ADD IX,DE	DD	221	11011101
	19	25	00011001
ADD IX,IX	DD	221	11011101
	29	41	00101001
ADD IX,SP	DD	221	11011101
	39	57	00111001
ADD IY,BC	FD	253	11111101
	09	9	00001001
ADD IY,DE	FD	253	11111101
	19	25	00011001
ADD IY,IY	FD	253	11111101
	29	41	00101001
ADD IY,SP	FD	253	11111101
	39	57	00111001
AND (HL)	A6	166	10100110
AND (IX+00)	DD	221	11011101
	A6	166	10100110
	00	0	00000000
AND (IY+00)	FD	253	11111101
	A6	166	10100110
	00	0	00000000
AND A	A7	167	10100111
AND B	A0	160	10100000
AND C	A1	161	10100001
AND D	A2	162	10100010
AND 00	E6	230	11100110
	00	0	00000000
AND E	A3	163	10100011
AND H	A4	164	10100100
AND L	A5	165	10100101
BIT 0,(HL)	CB	203	11001011
	46	70	01000110
BIT 0,(IX+00)	DD	221	11011101
	CB	203	11001011
	00	0	00000000
	46	70	01000110
BIT 0,(IY+00)	FD	253	11111101
	CB	203	11001011
	00	0	00000000
	46	70	01000110
BIT 0,A	CB	203	11001011
	47	71	01000111
BIT 0,B	CB	203	11001011
	40	64	01000000
BIT 0,C	CB	203	11001011
	41	65	01000001
BIT 0,D	CB	203	11001011
	42	66	01000010
BIT 0,E	CB	203	11001011
	43	67	01000011
BIT 0,H	CB	203	11001011
	44	68	01000100
BIT 0,L	CB	203	11001011
	45	69	01000101
BIT 1,(HL)	CB	203	11001011
	4E	78	01001110
BIT 1,(IX+00)	DD	221	11011101
	CB	203	11001011
	00	0	00000000
	4E	78	01001110
BIT 1,(IY+00)	FD	253	11111101
	CB	203	11001011
	00	0	00000000
	4E	78	01001110
BIT 1,A	CB	203	11001011
	4F	79	01001111
BIT 1,B	CB	203	11001011
	48	72	01001000
BIT 1,C	CB	203	11001011
	49	73	01001001
BIT 1,D	CB	203	11001011
	4A	74	01001010
BIT 1,E	CB	203	11001011
	4B	75	01001011
BIT 1,H	CB	203	11001011
	4C	76	01001100
BIT 1,L	CB	203	11001011
	4D	77	01001101
BIT 2,(HL)	CB	203	11001011
	56	86	01010110
BIT 2,(IX+00)	DD	221	11011101
	CB	203	11001011
	00	0	00000000
	56	86	01010110

BIT 2,(IY+00)	FD	253	11111101
	CB	203	11001011
	00	0	00000000
	56	86	01010110
BIT 2,A	CB	203	11001011
	57	87	01010111
BIT 2,B	CB	203	11001011
	50	80	01010000
BIT 2,C	CB	203	11001011
	51	81	01010001
BIT 2,D	CB	203	11001011
	52	82	01010010
BIT 2,E	CB	203	11001011
	53	83	01010011
BIT 2,H	CB	203	11001011
	54	84	01010100
BIT 2,L	CB	203	11001011
	55	85	01010101
BIT 3,(HL)	CB	203	11001011
	5E	94	01011110
BIT 3,(IX+00)	DD	221	11011101
	CB	203	11001011
	00	0	00000000
	5E	94	01011110
BIT 3,(IY+00)	FD	253	11111101
	CB	203	11001011
	00	0	00000000
	5E	94	01011110
BIT 3,A	CB	203	11001011
	5F	95	01011111
BIT 3,B	CB	203	11001011
	58	88	01011000
BIT 3,C	CB	203	11001011
	59	89	01011001
BIT 3,D	CB	203	11001011
	5A	90	01011010
BIT 3,E	CB	203	11001011
	5B	91	01011011
BIT 3,H	CB	203	11001011
	5C	92	01011100
BIT 3,L	CB	203	11001011
	5D	93	01011101
BIT 4,(HL)	CB	203	11001011
	66	102	01100110
BIT 4,(IX+00)	DD	221	11011101
	CB	203	11001011
	00	0	00000000
	66	102	01100110
BIT 4,(IY+00)	FD	253	11111101
	CB	203	11001011
	00	0	00000000
	66	102	01100110
BIT 4,A	CB	203	11001011
	67	103	01100111
BIT 4,B	CB	203	11001011
	60	96	01100000
BIT 4,C	CB	203	11001011
	61	97	01100001
BIT 4,D	CB	203	11001011
	62	98	01100010
BIT 4,E	CB	203	11001011
	63	99	01100011
BIT 4,H	CB	203	11001011
	64	100	01100100
BIT 4,L	CB	203	11001011
	65	101	01100101
BIT 5,(HL)	CB	203	11001011
	6E	110	01101110
BIT 5,(IX+00)	DD	221	11011101
	CB	203	11001011
	00	0	00000000
	6E	110	01101110
BIT 5,(IY+00)	FD	253	11111101
	CB	203	11001011
	00	0	00000000
	6E	110	01101110
BIT 5,A	CB	203	11001011
	6F	111	01101111
BIT 5,B	CB	203	11001011
	68	104	01101000
BIT 5,C	CB	203	11001011
	69	105	01101001
BIT 5,D	CB	203	11001011
	6A	106	01101010
BIT 5,E	CB	203	11001011
	6B	107	01101011
BIT 5,H	CB	203	11001011
	6C	108	01101100
BIT 5,L	CB	203	11001011
	6D	109	01101101
BIT 6,(HL)	CB	203	11001011
	76	118	01110110
BIT 6,(IX+00)	DD	221	11011101
	CB	203	11001011
	00	0	00000000
	76	118	01110110
BIT 6,(IY+00)	FD	253	11111101
	CB	203	11001011
	00	0	00000000
	76	118	01110110
BIT 6,A	CB	203	11001011
	77	119	01110111
BIT 6,B	CB	203	11001011
	70	112	01110000
BIT 6,C	CB	203	11001011
	71	113	01110001
BIT 6,D	CB	203	11001011
	72	114	01110010
BIT 6,E	CB	203	11001011
	73	115	01110011
BIT 6,H	CB	203	11001011
	74	116	01110100
BIT 6,L	CB	203	11001011
	75	117	01110101
BIT 7,(HL)	CB	203	11001011
	7E	126	01111110
BIT 7,(IX+00)	DD	221	11011101
	CB	203	11001011
	00	0	00000000
	7E	126	01111110
BIT 7,(IY+00)	FD	253	11111101
	CB	203	11001011
	00	0	00000000
	7E	126	01111110
BIT 7,A	CB	203	11001011
	7F	127	01111111
BIT 7,B	CB	203	11001011
	78	120	01111000
BIT 7,C	CB	203	11001011
	79	121	01111001
BIT 7,D	CB	203	11001011
	7A	122	01111010
BIT 7,E	CB	203	11001011
	7B	123	01111011
BIT 7,H	CB	203	11001011
	7C	124	01111100
BIT 7,L	CB	203	11001011
	7D	125	01111101
CALL 00	CD	205	11001101
	00	0	00000000
CALL C,00	DC	220	11011100
	00	0	00000000
CALL M,00	FC	252	11111100
	00	0	00000000
CALL NC,00	D4	212	11010100
	00	0	00000000
CALL P,00	F4	244	11110100
	00	0	00000000
CALL PE,00	EC	236	11101100
	00	0	00000000
CALL PO,00	E4	228	11100100
	00	0	00000000
CALL Z,00	CC	204	11001100
	00	0	00000000
CCF	3F	63	00111111
CP(HL)	BE	190	10111110
CP(IX+0)	DD	221	11011101
	BE	190	10111110
	00	0	00000000
CP(IY+0)	FD	253	11111101
	BE	190	10111110
	00	0	00000000
CP A	BF	191	10111111
CP B	B8	184	10111000
CP C	B9	185	10111001
CP D	BA	186	10111010
CP 00	FE	254	11111110
	00	0	00000000
CP E	BB	187	10111011
CP H	BC	188	10111100
CP L	BD	189	10111101
CPD	ED	237	11101101
	A9	169	10101001
CPDR	ED	237	11101101
	B9	185	10111001
CPI	ED	237	11101101
	A1	161	10100001
CPIR	ED	237	11101101
	B1	177	10110001
CPL	2F	47	00101111
DAA	27	39	00100111
DEC(HL)	35	53	00110101
DEC(IX+0)	DD	221	11011101
	35	53	00110101
	00	0	00000000
DEC(IY+0)	FD	253	11111101
	35	53	00110101
	00	0	00000000
DEC A	3D	61	00111101
DEC B	05	5	00000101
DEC BC	0B	11	00001011
DEC C	0D	13	00001101
DEC D	15	21	00010101
DEC DE	1B	27	00011011
DEC E	1D	29	00011101
DEC H	25	37	00100101
DEC HL	2B	43	00101011
DEC IX	DD	221	11011101
	2B	43	00101011
DEC IY	FD	253	11111101
	2B	43	00101011
DEC L	2D	45	00101101
DEC SP	3B	59	00111011
DI	F3	243	11110011
DJNZ,00	10	16	00010000
	00	0	00000000
EI	FB	251	11111011
EX	8	8	10001000
EX(SP),HL	E3	227	11100011
EX(SP)IX	DD	221	11011101
	E3	227	11100011
EX(SP)IY	FD	253	11111101
	E3	227	11100011
EX AF,AF'	08	8	00001000
EX DE,HL	EB	235	11101011
EXX	D9	217	11011001
HLT	76	118	01110110
IM 0	ED	237	11101101
	46	70	01000110
IM 1	ED	237	11101101
	56	86	01010110
IM 2	ED	237	11101101
	5E	94	01011110
IN A,(C)	ED	237	11101101
	78	120	01111000
IN A,PORT	DB	219	11011011
	00	0	00000000
IN B,(C)	ED	237	11101101
	40	64	01000000
IN C,(C)	ED	237	11101101
	48	72	01001000
IN D,(C)	ED	237	11101101
	50	80	01010000
IN E,(C)	ED	237	11101101
	58	88	01011000
IN H,(C)	ED	237	11101101
	60	96	01100000
IN L,(C)	ED	237	11101101
	68	104	01101000
INC(HL)	34	52	00110100
INC(IX+00)	DD	221	11011101
	34	52	00110100
	00	0	00000000
INC(IY+00)	DD	221	11011101
	34	52	00110100
	00	0	00000000
INC A	3C	60	00111100
INC B	04	4	00000100
INC BC	03	3	00000011
INC C	0C	12	00001100
INC D	14	20	00010100
INC DE	13	19	00010011
INC E	1C	28	00011100
INC H	24	36	00100100
INC HL	23	35	00100011
INC IX	DD	221	11011101
	23	35	00100011

INC IY	FD	253	11111101
	23	35	00100011
INC L	2C	44	00101100
INC SP	33	51	00110011
IND	ED	237	11101101
	AA	170	10101010
INDR	ED	237	11101101
	BA	186	10111010
INI	ED	237	11101101
	A2	162	10100010
INIR	ED	237	11101101
	B2	178	10110010
JP(HL)	E9	233	11101001
JP(IX)	DD	221	11011101
	E9	233	11101001
JP(IY)	FD	253	11111101
	E9	233	11101001
J 0000	C3	195	11000011
	00	0	00000000
	00	0	00000000
JP C,0000	DA	218	11011010
	00	0	00000000
	00	0	00000000
JP M,0000	FA	250	11111010
	00	0	00000000
	00	0	00000000
JP NC,0000	D2	210	11010010
	00	0	00000000
	00	0	00000000
JP NZ,0000	C2	194	11000010
	00	0	00000000
	00	0	00000000
JP P,0000	F2	242	11110010
	00	0	00000000
	00	0	00000000
JP PE,0000	EA	234	11101010
	00	0	00000000
	00	0	00000000
JP PO,0000	E2	226	11100010
	00	0	00000000
	00	0	00000000
JP Z,0000	CA	202	11001010
	00	0	00000000
	00	0	00000000
JR C,00	38	56	00111000
	00	0	00000000
JR 00	18	24	00011000
	00	0	00000000
JR NC,00	30	48	00110000
	00	0	00000000
JR NZ,00	20	32	00100000
	00	0	00000000
JR Z,00	28	40	00101000
	00	0	00000000
LD(0000),A	32	50	00110010
	00	0	00000000
	00	0	00000000
LD(0000),BC	ED	237	11101101
	43	67	01000011
	00	0	00000000
	00	0	00000000
LD(0000),DE	ED	237	11101101
	53	83	01010011
	00	0	00000000
	00	0	00000000

LD(0000),HL	ED	237	11101101
	63	99	01100011
	00	0	00000000
	00	0	00000000
LD(0000),HL	22	34	00100010
LD(0000),IX	DD	221	11011101
	22	34	00100010
	00	0	00000000
	00	0	00000000
LD(0000),IY	FD	253	11111101
	22	34	00100010
	00	0	00000000
	00	0	00000000
LD(0000),SP	ED	237	11101101
	73	115	01110011
	00	0	00000000
	00	0	00000000
LD(BC),A	02	2	00000010
LD(DE),A	12	18	00010010
LD(HL),A	77	119	01110111
LD(HL),B	70	112	01110000
LD(HL),C	71	113	01110001
LD(HL),D	72	114	01110010
LD(HL),00	36	54	00110110
	00	0	00000000
LD(HL),E	73	115	01110011
LD(HL),H	74	116	01110100
LD(HL),L	75	117	01110101
LD(IX+00),A	DD	221	11011101
	77	119	01110111
	00	0	00000000
LD(IX+00),B	DD	221	11011101
	70	112	01110000
	00	0	00000000
LD(IX+00),C	DD	221	11011101
	71	113	01110001
	00	0	00000000
LD(IX+00),00	DD	221	11011101
	36	54	00110110
	00	0	00000000
	00	0	00000000
LD(IX+00),E	DD	221	11011101
	73	115	01110011
	00	0	00000000
LD(IX+00),H	DD	221	11011101
	74	116	01110100
	00	0	00000000
LD(IX+00),L	DD	221	11011101
	75	117	01110101
	00	0	00000000
LD(IY+00),A	FD	253	11111101
	77	119	01110111
	00	0	00000000
LD(IY+00),B	FD	253	11111101
	70	112	01110000
	00	0	00000000
LD(IY+00),C	FD	253	11111101
	71	113	01110001
	00	0	00000000
LD(IY+00),D	FD	253	11111101
	72	114	01110010
	00	0	00000000
LD(IY+00),00	FD	253	11111101
	36	54	00110110
	00	0	00000000
	00	0	00000000

LD(IY+00),E	FD	253	11111101
	73	115	01110011
	00	0	00000000
LD(IY+00),H	FD	253	11111101
	74	116	01110100
	00	0	00000000
LD(IY+00),L	FD	253	11111101
	75	117	01110101
	00	0	00000000
LD A,(0000)	3A	58	00111010
	00	0	00000000
	00	0	00000000
LD A,(BC)	0A	10	00001010
LD A,(DE)	1A	26	00011010
LD A,(HL)	7E	126	01111110
LD A,(IY+00)	DD	221	11011101
	7E	126	01111110
	00	0	00000000
LD A,(IX+00)	FD	253	11111101
	7E	126	01111110
	00	0	00000000
LD A,A	7F	127	01111111
LD A,B	78	120	01111000
LD A,C	79	121	01111001
LD A,D	7A	122	01111010
LD A,00	3E	62	00111110
	00	0	00000000
LD A,E	7B	123	01111011
LD A,H	7C	124	01111100
LD A,I	ED	237	11101101
	57	87	01010111
LD A,L	7D	125	01111101
LD A,R	ED	237	11101101
	5F	95	01011111
LD B,(HL)	46	70	01000110
LD B,(IX+00)	DD	221	11011101
	46	70	01000110
	00	0	00000000
LD B,(IY+00)	FD	253	11111101
	46	70	01000110
	00	0	00000000
LD B,A	47	71	01000111
LD B,B	40	64	01000000
LD B,C	41	65	01000001
LD B,D	42	66	01000010
LD B,00	06	6	00000110
	00	0	00000000
LD B,E	43	67	01000011
LD B,H	44	68	01000100
LD B,L	45	69	01000101
LD BC,(0000)	ED	237	11101101
	4B	75	01001011
	00	0	00000000
	00	0	00000000
LD BC,0000	01	1	00000001
	00	0	00000000
	00	0	00000000
LD C,(HL)	4E	78	01001110
LD C,(IX+00)	DD	221	11011101
	4E	78	01001110
	00	0	00000000
LD C,(IY+00)	FD	253	11111101
	4E	78	01001110
	00	0	00000000

LD C,A	4F	79	01001111
LD C,B	48	72	01001000
LD C,C	49	73	01001001
LD C,D	4A	74	01001010
LD C,00	0E	14	00001110
	00	0	00000000
LD C,E	4B	75	01001011
LD C,H	4C	76	01001100
LD C,L	4D	77	01001101
LD D,(HL)	56	86	01010110
LD D,(IX+00)	DD	221	11011101
	56	86	01010110
	00	0	00000000
LD D,(IY+00)	FD	253	11111101
	56	86	01010110
	00	0	00000000
LD D,A	57	87	01010111
LD D,B	50	80	01010000
LD D,C	51	81	01010001
LD D,D	52	82	01010010
LD D,00	16	22	00010110
	00	0	00000000
LD D,E	53	83	01010011
LD D,H	54	84	01010100
LD D,L	55	85	01010101
LD DE,(0000)	ED	237	11101101
	5B	91	01011011
	00	0	00000000
	00	0	00000000
LD DE,0000	11	17	00010001
	00	0	00000000
	00	0	00000000
LD E,(HL)	5E	94	01011110
LD E,(IX+00)	DD	221	11011101
	5E	94	01011110
	00	0	00000000
LD E,(IY+00	FD	253	11111101
	5E	94	01011110
	00	0	00000000
LD E,A	5F	95	01011111
LD E,B	58	88	01011000
LD E,C	59	89	01011001
LD E,D	5A	90	01011010
LD E,00	1E	30	00011110
	00	0	00000000
LD E,E	5B	91	01011011
LD E,H	5C	92	01011100
LD E,L	5D	93	01011101
LD H,(HL)	66	102	01100110
LD H,(IX+00)	DD	221	11011101
	66	102	01100110
	00	0	00000000
LD H,(IY+00)	FD	253	11111101
	66	102	01100110
	00	0	00000000
LD H,A	67	103	01100111
LD H,B	60	96	01100000
LD H,C	61	97	01100001
LD H,D	62	98	01100010
LD H,00	26	38	00100110
	00	0	00000000
LD H,E	63	99	01100011
LD H,H	64	100	01100100
LD H,L	65	101	01100101

LD HL,0000	ED	237	11101101
	6B	107	01101011
	00	0	00000000
	00	0	00000000
LD HL,(0000)	2A	42	00101010
	00	0	00000000
	00	0	00000000
LD HL,0000	21	33	00100001
	00	0	00000000
	00	0	00000000
LD I,A	ED	237	11101101
	47	71	01000111
LD IX,(0000)	DD	221	11011101
	2A	42	00101010
	00	0	00000000
	00	0	00000000
LD IX,0000	DD	221	11011101
	21	33	00100001
	00	0	00000000
	00	0	00000000
LD IY,(0000)	FD	253	11111101
	2A	42	00101010
	00	0	00000000
	00	0	00000000
LD IY,0000	FD	253	11111101
	21	33	00100001
	00	0	00000000
	00	0	00000000
LD L,(HL)	6E	110	01101110
LD L,(IX+00)	DD	221	11011101
	6E	110	01101110
	00	0	00000000
LD L,(IY+00)	FD	253	11111101
	6E	110	01101110
	00	0	00000000
LD L,A	6F	111	01101111
LD L,B	68	104	01101000
LD L,C	69	105	01101001
LD L,D	6A	106	01101010
LD L,00	2E	46	00101110
	00	0	00000000
LD L,E	6B	107	01101011
LD L,H	6C	108	01101100
LD L,I	6D	109	01101101
LD R,A	ED	237	11101101
	4F	79	01001111
LD SP,(0000)	ED	237	11101101
	7B	123	01111011
	00	0	00000000
	00	0	00000000
LD SP,0000	31	49	00110001
	00	0	00000000
	00	0	00000000
LD SP,HL	F9	249	11111001
LD SP,IX	DD	221	11011101
	F9	249	11111001
LD SP,IY	FD	253	11111101
	F9	249	11111001
LDD	ED	237	11101101
	A8	168	10101000
LDDR	ED	237	11101101
	B8	184	10111000
LDI	ED	237	11101101
	A0	160	10100000

LDIR	ED	237	11101101
	B0	176	10110000
NEG	ED	237	11101101
	44	68	01000100
NOP	00	0	00000000
OR(HL)	B6	182	10110110
OR(IX+00)	DD	221	11011101
	B6	182	10110110
	00	0	00000000
OR(IY+00)	FD	253	11111101
	B6	182	10110110
	00	0	00000000
OR A	B7	183	10110111
OR B	B0	176	10110000
OR C	B1	177	10110001
OR D	B2	178	10110010
OR 00	F6	246	11110110
	00	0	00000000
OR E	B3	179	10110011
OR H	B4	180	10110100
OR L	B5	181	10110101
OTDR	ED	237	11101101
	BB	187	10111011
OTIR	ED	237	11101101
	B3	179	10110011
OUT(C),A	ED	237	11101101
	79	121	01111001
OUT(C),B	ED	237	11101101
	41	65	01000001
OUT(C),C	ED	237	11101101
	49	73	01001001
OUT(C),D	ED	237	11101101
	51	81	01010001
OUT(C),E	ED	237	11101101
	59	89	01011001
OUT(C),H	ED	237	11101101
	61	97	01100001
OUT(C),L	ED	237	11101101
	69	105	01101001
OUT(Port),A	D3	211	11010011
	00	0	00000000
OUTD	ED	237	11101101
	AB	171	10101011
OUTI	ED	237	11101101
	A3	163	10100011
POP AF	F1	241	11110001
POP BC	C1	193	11000001
POP DE	D1	209	11010001
POP HL	E1	225	11100001
POP IX	DD	221	11011101
	E1	225	11100001
POP IY	FD	253	11111101
	E1	225	11100001
PUSH AF	F5	245	11110101
PUSH BC	C5	197	11000101
PUSH DE	D5	213	11010101
PUSH HL	E5	229	11100101
PUSH IX	DD	221	11011101
	E5	229	11100101
PUSH IY	FD	253	11111101
	E5	229	11100101
RES 0,(HL)	CB	203	11001011
	86	134	10000110
RES 0,(IX+00)	DD	221	11011101
	CB	203	11001011
	00	0	00000000
	00	0	00000000

RES 0,(1Y+00)	FD	253	11111101
	CB	203	11001011
	00	0	00000000
	00	0	00000000
RES 0,A	CB	203	11001011
	87	135	10000111
RES 0,B	CB	203	11001011
	80	128	10000000
RES 0,C	CB	203	11001011
	81	129	10000001
RES 0,D	CB	203	11001011
	82	130	10000010
RES 0,E	CB	203	11001011
	83	131	10000011
RES 0,H	CB	203	11001011
	84	132	10000100
RES 0,L	CB	203	11001011
	85	133	10000101
RES 1,(HL)	CB	203	11001011
	8E	142	10001110
RES 1,(IX+00)	DD	221	11011101
	CB	203	11001011
	00	0	00000000
	00	0	00000000
RES 1,(IY+00)	FD	253	11111101
	CB	203	11001011
	00	0	00000000
	00	0	00000000
RES 1,A	CB	203	11001011
	8F	143	10001111
RES 1,B	CB	203	11001011
	88	136	10001000
RES 1,C	CB	203	11001011
	89	137	10001001
RES 1,D	CB	203	11001011
	8A	138	10001010
RES 1,E	CB	203	11001011
	8B	139	10001011
RES 1,H	CB	203	11001011
	8C	140	10001100
RES 1,L	CB	203	11001011
	8D	141	10001101
RES 2,(HL)	CB	203	11001011
	96	150	10010110
RES 2,(IX+00)	DD	221	11011101
	CB	203	11001011
	00	0	00000000
	96	150	10010110
RES 2,(IY+00)	FD	253	11111101
	CB	203	11001011
	00	0	00000000
	96	150	10010110
RES 2,A	CB	203	11001011
	97	151	10010111
RES 2,B	CB	203	11001011
	90	144	10010000
RES 2,C	CB	203	11001011
	91	145	10010001
RES 2,D	CB	203	11001011
	92	146	10010010
RES 2,E	CB	203	11001011
	93	147	10010011
RES 2,H	CB	203	11001011
	94	148	10010100

RES 2,L	CB	203	11001011
	95	149	10010101
RES 3,(HL)	CB	203	11001011
	9E	158	10011110
RES 3,(IX+00)	DD	221	11011101
	CB	203	11001011
	00	0	00000000
	9E	158	10011110
RES 3,(IY+00)	FD	253	11111101
	CB	203	11001011
	00	0	00000000
	9E	158	10011110
RES 3,A	CB	203	11001011
	9F	159	10011111
RES 3,B	CB	203	11001011
	98	152	10011000
RES 3,C	CB	203	11001011
	99	153	10011001
RES 3,D	CB	203	11001011
	9A	154	10011010
RES 3,E	CB	203	11001011
	9B	155	10011011
RES 3,H	CB	203	11001011
	9C	156	10011100
RES 3,L	CB	203	11001011
	9D	157	10011101
RES 4,(HL)	CB	203	11001011
	A6	166	10100110
RES 4,(IX+00)	DD	221	11011101
	CB	203	11001011
	00	0	00000000
	A6	166	10100110
RES 4,(IY+00)	FD	253	11111101
	CB	203	11001011
	00	0	00000000
	A6	166	10100110
RES 4,A	CB	203	11001011
	A7	167	10100111
RES 4,B	CB	203	11001011
	A0	160	10100000
RES 4,C	CB	203	11001011
	A1	161	10100001
RES 4,D	CB	203	11001011
	A2	162	10100010
RES 4,E	CB	203	11001011
	A3	163	10100011
RES 4.H	CB	203	11001011
	A4	164	10100100
RES 4,L	CB	203	11001011
	A5	165	10100101
RES 5,(HL)	CB	203	11001011
	AE	174	10101110
RES 5,(IX+00)	DD	221	11011101
	CB	203	11001011
	00	0	00000000
	AE	174	10101110
RES 5,(IY+00)	FD	253	11111101
	CB	203	11001011
	00	0	00000000
	AE	174	10101110
RES 5,A	CB	203	11001011
	5F	95	01011111
RES 5,B	CB	203	11001011
	A8	168	10101000

RES 5,C	CB	203	11001011
	A9	169	10101001
RES 5,D	CB	203	11001011
	AA	170	10101010
RES 5,E	CB	203	11001011
	AB	171	10101011
RES 5,H	CB	203	11001011
	AC	172	10101100
RES 5,L	CB	203	11001011
	AD	173	10101101
RES 6,(HL)	CB	203	11001011
	B6	182	10110110
RES 6,(IX+00)	DD	221	11011101
	CB	203	11001011
	00	0	00000000
	B6	182	10110110
RES 6,(IY+00)	FD	253	11111101
	CB	203	11001011
	00	0	00000000
	B6	182	10110110
RES 6,A	CB	203	11001011
	B7	183	10110111
RES 6,B	CB	203	11001011
	B0	176	10110000
RES 6,C	CB	203	11001011
	B1	177	10110001
RES 6,D	CB	203	11001011
	B2	178	10110010
RES 6,E	CB	203	11001011
	B3	179	10110011
RES 6,H	CB	203	11001011
	B4	180	10110100
RES 6,L	CB	203	11001011
	B5	181	10110101
RES 7,(HL)	CB	203	11001011
	BE	190	10111110
RES 7,(IX+00)	DD	221	11011101
	CB	203	11001011
	00	0	00000000
	BE	190	10111110
RES 7,(IY+00)	FD	253	11111101
	CB	203	11001011
	00	0	00000000
	BE	190	10111110
RES 7,A	CB	203	11001011
	BF	191	10111111
RES 7,B	CB	203	11001011
	B8	184	10111000
RES 7,C	CB	203	11001011
	B9	185	10111001
RES 7,D	CB	203	11001011
	BA	186	10111010
RES 7,E	CB	203	11001011
	BB	187	10111011
RES 7,H	CB	203	11001011
	BC	188	10111100
RES 7,L	CB	203	11001011
	BD	189	10111101
RET	C9	201	11001001
RET C	D8	216	11011000
RET M	F8	248	11111000
RET NC	D0	208	11010000
RET NZ	C0	192	11000000
RET P	F0	240	11110000
RET PE	E8	232	11101000
RET PO	E0	224	11100000
RET Z	C8	200	11001000
RET1	ED	237	11101101
	4D	77	01001101
RETN	ED	237	11101101
	45	69	01000101
RL(HL)	CB	203	11001011
	16	22	00010110
RL(IX+00)	DD	221	11011101
	CB	203	11001011
	00	0	00000000
	16	22	00010110
RL(IY+00)	FD	253	11111101
	CB	203	11001011
	00	0	00000000
	16	22	00010110
RL A	CB	203	11001011
	17	23	00010111
RL B	CB	203	11001011
	10	16	00010000
RL C	CB	203	11001011
	11	17	00010001
RL D	CB	203	11001011
	12	18	00010010
RL E	CB	203	11001011
	13	19	00010011
RL H	CB	203	11001011
	14	20	00010100
RL L	CB	203	11001011
	15	21	00010101
RLA	17	23	00010111
RLC(HL)	CB	203	11001011
	06	6	00000110
RLC(IX+00)	DD	221	11011101
	CB	203	11001011
	00	0	00000000
	06	6	00000110
RLC(IY+00)	FD	253	11111101
	CB	203	11001011
	00	0	00000000
	06	6	00000110
RLC A	CB	203	11001011
	07	7	00000111
RLC B	CB	203	11001011
	00	0	00000000
RLC C	CB	203	11001011
	01	1	00000001
RLC D	CB	203	11001011
	02	2	00000010
RLC E	CB	203	11001011
	03	3	00000011
RLC H	CB	203	11001011
	04	4	00000100
RLC L	CB	203	11001011
	05	5	00000101
RLCA	07	7	00000111
RLD	ED	237	11101101
	6F	111	01101111
RR(HL)	CB	203	11001011
	1E	30	00011110
RR(IX+00)	DD	221	11011101
	CB	203	11001011
	00	0	00000000
	1E	30	00011110
RR(IY+00)	FD	253	11111101
	CB	203	11001011
	00	0	00000000
	1E	30	00011110
RR A	CB	203	11001011
	1F	31	00011111
RR B	CB	203	11001011
	18	24	00011000
RR C	CB	203	11001011
	19	25	00011001
RR D	CB	203	11001011
	1A	26	00011010
RR E	CB	203	11001011
	1B	27	00011011
RR H	CB	203	11001011
	1C	28	00011100
RR L	CB	203	11001011
	1D	29	00011101
RRA	1F	31	00011111
RRC(HL)	CB	203	11001011
	0E	14	00001110
RRC(IX+00)	DD	221	11011101
	CB	203	11001011
	00	0	00000000
	0E	14	00001110
RRC(IY+00)	FD	253	11111101
	CB	203	11001011
	00	0	00000000
	0E	14	00001110
RRC A	CB	203	11001011
	0F	15	00001111
RRC B	CB	203	11001011
	08	8	00001000
RRC C	CB	203	11001011
	09	9	00001001
RRC D	CB	203	11001011
	0A	10	00001010
RRC E	CB	203	11001011
	0B	11	00001011
RRC H	CB	203	11001011
	0C	12	00001100
RRC L	CB	203	11001011
	0D	13	00001101
RRCA	0F	15	00001111
RRD	ED	237	11101101
	67	103	01100111
RST 00	C7	199	11000111
RST 08	CF	207	11001111
RST 10	D7	215	11010111
RST 18	DF	223	11011111
RST 20	E7	231	11100111
RST 28	EF	239	11101111
RST 30	F7	247	11110111
RST 38	FF	255	11111111
SBC A,(HL)	9E	158	10011110
SBC A,(IX+00)	DD	221	11011101
	9E	158	10011110
	00	0	00000000
SBC A,(IY+00)	FD	253	11111101
	9E	158	10011110
	00	0	00000000
SBC A,A	9F	159	10011111
SBC A,B	98	152	10011000
SBC A,C	99	153	10011001
SBC A,D	9A	154	10011010
SBC A,00	DE	222	11011110
	00	0	00000000
SBC A,E	9B	155	10011011
SBC A,H	9C	156	10011100
SBC A,L	9D	157	10011101
SBC HL,BC	ED	237	11101101
	42	66	01000010
SBC HL,DE	ED	237	11101101
	52	82	01010010
SBC HL,HL	ED	237	11101101
	62	98	01100010
SBC HL,SP	ED	237	11101101
	72	114	01110010
SCF	37	55	00110111
SET 0,(HL)	CB	203	11001011
	C6	198	11000110
SET0,(IX+00)	DD	221	11011101
	CB	203	11001011
	00	0	00000000
	C6	198	11000110
SET0,(IY+00)	FD	253	11111101
	CB	203	11001011
	00	0	00000000
	23	35	00100011
SET0,A	CB	203	11001011
	C7	199	11000111
SET0,B	CB	203	11001011
	C0	192	11000000
SET0,C	CB	203	11001011
	C1	193	11000001
SET0,D	CB	203	11001011
	C2	194	11000010
SET0,E	CB	203	11001011
	C3	195	11000011
SET0,H	CB	203	11001011
	C4	196	11000100
SET0,L	CB	203	11001011
	C5	197	11000101
SET1,(HL)	CB	203	11001011
	CE	206	11001110
SET1,(IX+00)	DD	221	11011101
	CB	203	11001011
	00	0	00000000
	CE	206	11001110
SET1,(IY+00)	FD	253	11111101
	CB	203	11001011
	00	0	00000000
	CE	206	11001110
SET1,A	CB	203	11001011
	CF	207	11001111
SET1,B	CB	203	11001011
	C8	200	11001000
SET1,C	CB	203	11001011
	C9	201	11001001
SET1,D	CB	203	11001011
	CA	202	11001010
SET1,E	CB	203	11001011
	CB	203	11001011
SET1,H	CB	203	11001011
	CC	204	11001100
SET1,L	CB	203	11001011
	CD	205	11001101
SET2,(HL)	CB	203	11001011
	D6	214	11010110

SET2,(IX+00)	DD	221	11011101
	CB	203	11001011
	00	0	00000000
	D6	214	11010110
SET2,(IY+00)	FD	253	11111101
	CB	203	11001011
	00	0	00000000
	D6	214	11010110
SET2,A	CB	203	11001011
	D7	215	11010111
SET2,B	CB	203	11001011
	D0	208	11010000
SET2,C	CB	203	11001011
	D1	209	11010001
SET2,D	CB	203	11001011
	D2	210	11010010
SET2,E	CB	203	11001011
	D3	211	11010011
SET2,H	CB	203	11001011
	D4	212	11010100
SET2,L	CB	203	11001011
	D5	213	11010101
SET3,(HL)	CB	203	11001011
	DE	222	11011110
SET3,(IX+00)	DD	221	11011101
	CB	203	11001011
	00	0	00000000
	DE	222	11011110
SET3,(IY+00)	FD	253	11111101
	CB	203	11001011
	00	0	00000000
	DE	222	11011110
SET3,A	CB	203	11001011
	DF	223	11011111
SET3,B	CB	203	11001011
	D8	216	11011000
SET3,C	CB	203	11001011
	D9	217	11011001
SET3,D	CB	203	11001011
	DA	218	11011010
SET3,E	CB	203	11001011
	DB	219	11011011
SET3,H	CB	203	11001011
	DC	220	11011100
SET3,L	CB	203	11001011
	DD	221	11011101
SET4,(HL)	CB	203	11001011
	E6	230	11100110
SET4,(IX+00)	DD	221	11011101
	CB	203	11001011
	00	0	00000000
	E6	230	11100110
SET4,(IY+00)	FD	253	11111101
	CB	203	11001011
	00	0	00000000
	E6	230	11100110
SET4,A	CB	203	11001011
	E7	231	11100111
SET4,B	CB	203	11001011
	E0	224	11100000
SET4,C	CB	203	11001011
	E1	225	11100001
SET4,D	CB	203	11001011
	E2	226	11100010
SET4,E	CB	203	11001011
	E3	227	11100011
SET4,H	CB	203	11001011
	E4	228	11100100
SET4,L	CB	203	11001011
	E5	229	11100101
SET5,(HL)	CB	203	11001011
	EE	238	11101110
SET5,(IX+00)	DD	221	11011101
	CB	203	11001011
	00	0	00000000
	EE	238	11101110
SET5,(IY+00)	FD	253	11111101
	CB	203	11001011
	00	0	00000000
	EE	238	11101110
SET5,A	CB	203	11001011
	EF	239	11101111
SET5,B	CB	203	11001011
	E8	232	11101000
SET5,C	CB	203	11001011
	E9	233	11101001
SET5,D	CB	203	11001011
	EA	234	11101010
SET5,E	CB	203	11001011
	EB	235	11101011
SET5,H	CB	203	11001011
	EC	236	11101100
SET5,L	CB	203	11001011
	ED	237	11101101
SET6,(HL)	CB	203	11001011
	F6	246	11110110
SET6,(IX+00)	DD	221	11011101
	CB	203	11001011
	00	0	00000000
	F6	246	11110110
SET6,(IY+00)	FD	253	11111101
	CB	203	11001011
	00	0	00000000
	F6	246	11110110
SET6,A	CB	203	11001011
	F7	247	11110111
SET6,B	CB	203	11001011
	F0	240	11110000
SET6,C	CB	203	11001011
	F1	241	11110001
SET6,D	CB	203	11001011
	F2	242	11110010
SET6,E	CB	203	11001011
	F3	243	11110011
SET6,H	CB	203	11001011
	F4	244	11110100
SET6,L	CB	203	11001011
	F5	245	11110101
SET7,(HL)	CB	203	11001011
	FE	254	11111110
SET7,(IX+00)	DD	221	11011101
	CB	203	11001011
	00	0	00000000
	FE	254	11111110
SET7,(IY+00)	FD	253	11111101
	CB	203	11001011
	00	0	00000000
	FE	254	11111110
SET7,A	CB	203	11001011
	FF	255	11111111
SET7,B	CB	203	11001011
	F8	248	11111000
SET7,C	CB	203	11001011
	F9	249	11111001
SET7,D	CB	203	11001011
	FA	250	11111010
SET7,E	CB	203	11001011
	FB	251	11111011
SET7,H	CB	203	11001011
	FC	252	11111100
SET7,L	CB	203	11001011
	FD	253	11111101
SLA(HL)	CB	203	11001011
	26	38	00100110
SLA(IX+00)	DD	221	11011101
	CB	203	11001011
	00	0	00000000
	26	38	00100110
SLA(IY+00)	FD	253	11111101
	CB	203	11001011
	00	0	00000000
	26	38	00100110
SLA A	CB	203	11001011
	27	39	00100111
SLA B	CB	203	11001011
	20	32	00100000
SLA C	CB	203	11001011
	21	33	00100001
SLA D	CB	203	11001011
	22	34	00100010
SLA E	CB	203	11001011
	23	35	00100011
SLA H	CB	203	11001011
	24	36	00100100
SLA L	CB	203	11001011
	25	37	00100101
SRA(HL)	CB	203	11001011
	2E	46	00101110
SRA(IX+00)	DD	221	11011101
	CB	203	11001011
	00	0	00000000
	2E	46	00101110
SRA(IY+00)	FD	253	11111101
	CB	203	11001011
	00	0	00000000
	2E	46	00101110
SRA A	CB	203	11001011
	2F	47	00101111
SRA B	CB	203	11001011
	28	40	00101000
SRA C	CB	203	11001011
	29	41	00101001
SRA D	CB	203	11001011
	2A	42	00101010
SRA E	CB	203	11001011
	2B	43	00101011
SRA H	CB	203	11001011
	2C	44	00101100
SRA L	CB	203	11001011
	2D	45	00101101
SRL(HL)	CB	203	11001011
	3E	62	00111110
SRL(IX+00)	DD	221	11011101
	CB	203	11001011
	00	0	00000000
	3E	62	00111110
SRL(IY+00)	FD	253	11111101
	CB	203	11001011
	00	0	00000000
	3E	62	00111110
SRL A	CB	203	11001011
	3F	63	00111111
SRL B	CB	203	11001011
	38	56	00111000
SRL C	CB	203	11001011
	39	57	00111001
SRL D	CB	203	11001011
	3A	58	00111010
SRL E	CB	203	11001011
	3B	59	00111011
SRL H	CB	203	11001011
	3C	60	00111100
SRL L	CB	203	11001011
	3D	61	00111101
SUB(HL)	96	150	10010110
SUB(IX+00)	DD	221	11011101
	96	150	10010110
	00	0	00000000
SUB(IY+00)	FD	253	11111101
	96	150	10010110
	00	0	00000000
SUB A	97	151	10010111
SUB B	90	144	10010000
SUB C	91	145	10010001
SUB D	92	146	10010010
SUB 00	D6	214	11010110
	00	0	00000000
SUB E	93	147	10010011
SUB H	94	148	10010100
SUB L	95	149	10010101
XOR(HL)	AE	174	10101110
XOR(IX+00)	DD	221	11011101
	AE	174	10101110
	00	0	00000000
XOR(IY+00)	FD	253	11111101
	AE	174	10101110
	00	0	00000000
XOR A	AF	175	10101111
XOR B	A8	168	10101000
XOR C	A9	169	10101001
XOR D	AA	170	10101010
XOR 00	EE	238	11101110
	00	0	00000000
XOR E	AB	171	10101011
XOR H	AC	172	10101100
XOR L	AD	173	10101101

Appendix G
ASCII Codes

SPACE	20	32	00100000	P	50	80	01010000
!	21	33	00100001	Q	51	81	01010001
"	22	34	00100010	R	52	82	01010010
£	23	35	00100011	S	53	83	01010011
$	24	36	00100100	T	54	84	01010100
%	25	37	00100101	U	55	85	01010101
&	26	38	00100110	V	56	86	01010110
'	27	39	00100111	W	57	87	01010111
(	28	40	00101000	X	58	88	01011000
)	29	41	00101001	Y	59	89	01011001
*	2A	42	00101010	Z	5A	90	01011010
+	2B	43	00101011	[	5B	91	01011011
,	2C	44	00101100	/	5C	92	01011100
-	2D	45	00101101	]	5D	93	01011101
.	2E	46	00101110	^	5E	94	01011110
/	2F	47	00101111	_	5F	95	01011111
0	30	48	00110000	`	60	96	01100000
1	31	49	00110001	a	61	97	01100001
2	32	50	00110010	b	62	98	01100010
3	33	51	00110011	c	63	99	01100011
4	34	52	00110100	d	64	100	01100100
5	35	53	00110101	e	65	101	01100101
6	36	54	00110110	f	66	102	01100110
7	37	55	00110111	g	67	103	01100111
8	38	56	00111000	h	68	104	01101000
9	39	57	00111001	i	69	105	01101001
:	3A	58	00111010	j	6A	106	01101010
;	3B	59	00111011	k	6B	107	01101011
<	3C	60	00111100	l	6C	108	01101100
=	3D	61	00111101	m	6D	109	01101101
>	3E	62	00111110	n	6E	110	01101110
?	3F	63	00111111	o	6F	111	01101111
@	40	64	01000000	p	70	112	01110000
A	41	65	01000001	q	71	113	01110001
B	42	66	01000010	r	72	114	01110010
C	43	67	01000011	s	73	115	01110011
D	44	68	01000100	t	74	116	01110100
E	45	69	01000101	u	75	117	01110101
F	46	70	01000110	v	76	118	01110110
G	47	71	01000111	w	77	119	01110111
H	48	72	01001000	x	78	120	01111000
I	49	73	01001001	y	79	121	01111001
J	4A	74	01001010	z	7A	122	01111010
K	4B	75	01001011	{	7B	123	01111011
L	4C	76	01001100	¦	7C	124	01111100
M	4D	77	01001101	}	7D	125	01111101
N	4E	78	01001110	~	7E	126	01111110
O	4F	79	01001111	©	7F	127	01111111

Index