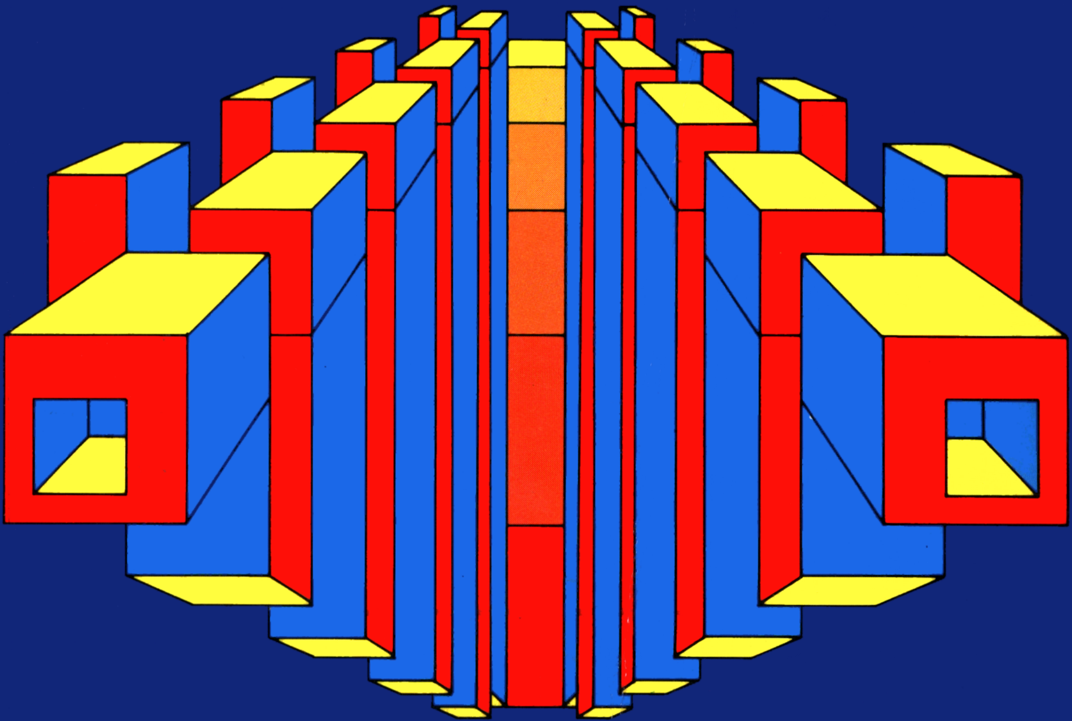
INTRODUCING AMSTRAD CPC464 MACHINE CODE



IAN SINCLAIR

Introducing Amstrad CPC464 Machine Code

Introducing Amstrad CPC464 Machine Code

Ian Sinclair

COLLINS
8 Grafton Street, London W1

Collins Professional and Technical Books
William Collins Sons & Co. Ltd
8 Grafton Street, London W1X 3LA

First published in Great Britain by
Collins Professional and Technical Books 1985

Distributed in the United States of America
by Sheridan House, Inc.

British Library Cataloguing in Publication Data
Sinclair, Ian R.
Introducing Amstrad CPC464 machine code
1. Amstrad CPC464 (Computer)—Programming
2. Machine codes (Electronic computers)
I. Title
001.64'24 QA76.8.A4

ISBN 0-00-383079-9

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

Contents

Preface

The user of an Amstrad CPC464 who has written and used BASIC programs is often reluctant to take the next obvious step, which is to learn and master the language that is used by the Z80 microprocessor at the heart of the CPC464 machine. This reluctance is understandable. Many books which deal with Z80 language seem to assume that the reader already knows all the terms that are used, and a lot about the way the microprocessor chip itself works. Other books seem, to the beginner, to be written in a foreign language, with no subtitles. Others make a promising start – then lose the inexpert reader by a sudden jump to much more difficult subjects, or to material like arithmetic routines which is of little use to most readers. Just to make things worse, the CPC464 is not such a simple design of computer as the machines of the past which used the Z80 microprocessor. This means that books which were written with other machines in mind are of very little use to the CPC464 owner.

This book is intended for the real novice to machine code – the owner who uses his or her Amstrad CPC464 for BASIC, can program in BASIC, but has absolutely no idea of what goes on inside the box. This book is not intended to make you, the reader, into an expert in programming the Z80, because only a lot of experience, a lot of reading, and a keen desire to solve problems can do that. It does not even set out to introduce you to *everything* that the Z80 can do. What I hope it will do is to introduce you to the start of a big topic, and put you in a position to understand some of the why and how of Z80 programming on the CPC464. You'll find a huge new world of computing opening up before your eyes, you'll have a much better understanding of how your computer works, and you'll be able to understand and enjoy more advanced books on machine language programming, such as these mentioned at the end of Chapter 9.

Let me make one point clear, though. This type of programming is never easy. It may become familiar, it may even become routine, but easy – not really. Learning it is also a task which requires some work, a lot of effort to understand what is going on, and a little time spent in trying things out on your computer. To help you, I've enlisted a very useful program, the HiSoft

DEVPAC, but the main effort, I repeat, must come from you, the reader. It's an effort which you will find well worth making.

As always, a book like this is the result of the efforts of a large number of people. I particularly want to thank Amsoft Ltd., who have published details of the firmware of the CPC464 so as to make life considerably easier for machine code programmers. The HiSoft DEVPAC, which is also sold by Amsoft, is a major contribution to machine code programming of this machine. At Collins Professional and Technical Books, I am most grateful to Richard Miles, Janet Murphy and Sue Moore for turning my manuscript into a real book. This is a miracle that they perform time and time again, but which never ceases to amaze me. I am also most grateful to the fastest typesetters in the business, and to the printers, for the appearance of this book in such a short time after the sheets emerged from the typewriter.

Ian Sinclair

Chapter One
ROM, RAM, Bytes and Bits

When you plug a TV receiver into a wall socket and switch it on, you're using electricity and receiving a TV signal. You don't see the machinery that generates the electricity, you don't see the TV camera that generates the TV signal, and you don't see the transmitter that sends the signal out. You can spend your life enjoying TV without ever having to worry about how the electricity and the TV signal got there, or what happens inside the TV receiver. You can also enjoy using your CPC464 computer for BASIC programming, or for running other people's programs without ever worrying about how these instructions are carried out. There is a difference, though. You can use your CPC464 computer more effectively, write programs that run faster, or which carry out operations that can't be done with BASIC if you have some understanding of what goes on inside. The most important part of that understanding is the language in which the computer is programmed when you buy it – called *machine code*.

One of the things that discourages computer users from attempts to go beyond BASIC is the number of new words that spring up. You can't do without these new words, because they are needed to describe things that are new to you. The writers of many books on computing, especially on machine code computing, seem to assume that the reader has an electronics background and will already know all of the terms. I shall assume that you have no such background. All that I shall assume is that you possess a CPC464 computer, and that you have some experience of programming your CPC464 computer in BASIC. Some experience of programming in BASIC is essential, because if you lack that, then you will have a much harder task understanding machine code. This means that we start at the beginning. I don't want in this book to have to interrupt important explanations with technical or mathematical details, and these will be found in the Appendices. This way, you can read the full explanation of some points if you feel inclined, or skip them if you are not.

To start with, we have to think about memory. A unit of memory for a computer is, as far as we are concerned, just an electrical circuit that acts like a switch. You walk into a room, switch a light on, and you never think it's remarkable in any way that the light stays on until you switch it off. You don't go around telling your friends that the light circuit contains a memory – and yet each memory unit of a computer is just a kind of miniature switch that can be turned on or off. What makes it a memory is that it will stay the way it has been turned, on or off, until it is changed. One unit of computer memory like this is called a *bit* – the name is short for binary digit meaning a unit that can be switched one of two possible ways. The memory of a computer consists of a very large number of incredibly small switches, each of which can be either on or off. There's no other possibility, no halfway position.

We'll stick with the idea of a switch, because it's very useful for explaining how we use memory. Suppose that we wanted to signal with electrical circuits and switches. We could use a circuit like the one in Figure 1.1. When the switch is on, the light is on, and we might take this as meaning YES. When the switch is turned off, the light goes out, and we might take this as meaning NO. You could attach any two meanings that you liked to these two conditions (called 'states') of the light, so long as there are only two. Things improve if you can use two switches and two lights, as in Figure 1.2.

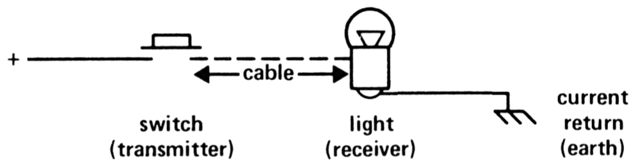
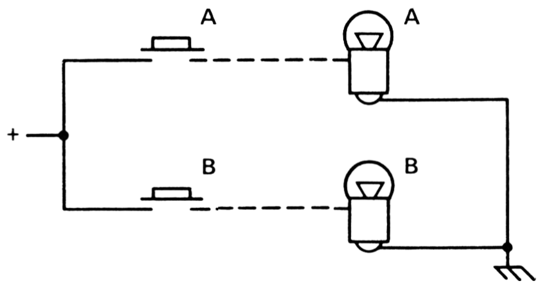


Figure 1.1. A single-line switch and bulb signalling system.



A	B
off	off
off	on
on	off
on	on

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

Now four different combinations are possible: (a) both off, (b) A off, B on, (c) A on, B off, (d) both on. This set of four possibilities means that we could signal four different meanings. Using one line allows two possible codes; using two lines allows four codes. If you feel inclined to work them all out, you'll find that using three lines will allow eight different codes. A moment's thought suggests that since 4 is 2×2, and eight is 2×2×2 , then four lines might allow 2×2×2×2, which is 16, codes. It's true, and since we usually write 2×2×2×2 as 2^4 (two to the power 4), we can find out how many codes could be transmitted by any number of lines. We would expect eight lines, for example, to be able to carry 2^8 codes, which is 256. A set of eight switches, then, could be arranged so as to convey 256 different meanings. It's up to us to decide how we might want to use these signals. The set of eight is a particularly important one, because the memory of your CPC464 computer is arranged in groups of eight bits.

One particularly useful way of using these on/off signals is called *binary code*. Binary code is a way of writing numbers using only two digits, Ø and 1. We can think of Ø as meaning 'switch off' and 1 as meaning 'switch on', so that 256 different numbers could be signalled by using eight switches by thinking of Ø as meaning off and 1 as meaning on. This group of eight is called a *byte*, and it's the quantity that we use to specify the memory size of our computers. This is why the numbers 8 and 256 occur so much in machine code computing.

The way that the individual bits in a byte are arranged so as to indicate a number follows the same way as we use to indicate a number 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 five tens, and the 2 is written one more place to the left and means two hundreds. These positions indicate the importance or significance of a digit, as Figure 1.3 shows. The 6 in 253 is

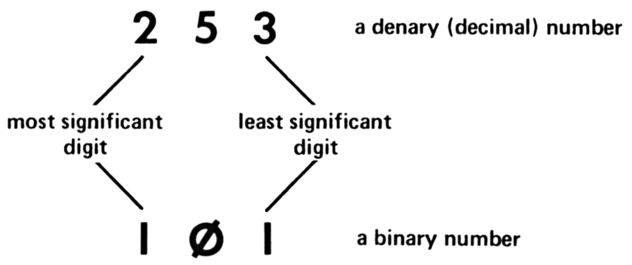


Figure 1.3. The significance of digits. Our numbering system uses the position of a digit in a number to indicate its significance or importance.

called the 'least significant digit', and the 2 is the 'most significant digit'. Change the 3 to 4 or 2, and the change is just one part in 253. Change the 2 to 1 or 3 and the change is one hundred parts in 253 which is much more important.

Having looked at bits and bytes, it's time to go back to the idea of memory as a set of switches. As it happens, we need two types of memory in a computer. One type must be permanent, like mechanical switches or fixed connections, because it has to be used for retaining the number-coded instructions that operate the computer. This is the type of memory that is called *ROM*, meaning read-only memory. This implies that you can find out and copy what is in the memory, but you cannot delete it or change it. The ROM is the most important part of your computer, because it contains all the instructions that make the computer carry out the actions of BASIC. These instructions are referred to as the 'firmware' of the computer. When you write a program in BASIC for yourself, the computer stores it in the form of another set of number-coded instructions in a part of memory that can be used over and over again. This is a different type of memory that can be 'written' as well as 'read', and if we were logical about it we would refer to it as RWM, meaning read-write memory. Unfortunately, we're not very logical, and we call it *RAM* (meaning random-access memory). This was a name that was 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 of RAM now and probably forever! The big difference between RAM and ROM is that each bit of RAM behaves like a switch only while there is an electrical supply to it. When you switch off the supply, the switch action stops. If you turn on the supply again, the switch action of the RAM will start again – but not the way it was before. Each bit of RAM may be 'on' or 'off' when power is restored, but this happens at random. When you switch off your computer, then, you lose everything that was stored in the RAM, and when you switch on again all you get is a set of random signals. It's like throwing a jigsaw puzzle into the air – you can't really expect it to land still assembled.

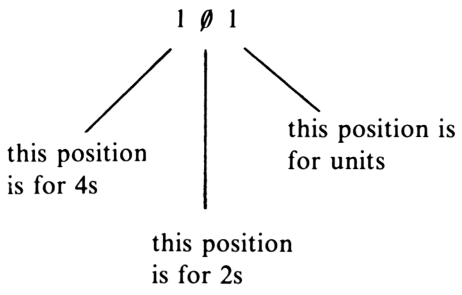
The number-code caper

Now we can get back to the bytes. We saw earlier that a byte is a group of eight bits which can be arranged in any of 256 different ways, depending on which bits are 1's and which are Ø's. The most useful way of arranging bits, however, is one that we call *binary code*.

Binary code uses the position of a digit to indicate its value. The right-hand digit can be Ø or 1, and it means just these numbers. The next digit to the left, however, can also be 1 or Ø. The Ø means Ø, but a 1 in this position means 2. In the next place to the left, a 1 means 4, and so on. The whole system is illustrated in Figure 1.4.

Each different arrangement of bits is used to represent a number which we would write in ordinary form as Ø to 255 (not 1 to 256, because we need a code for zero). Each byte of the 65536 bytes of RAM in the CPC464 computer can store a number which is in this range of Ø to 255.

Numbers by themselves are not of much use, and we wouldn't find a



The number 1Ø1 is 4+1 = 5 in denary

The position values are:

128	64	32	16	8	4	2	1

– in a byte

Example:

Ø1ØØ11Ø1

means 64 + 8 + 4 + 1 = 77

Figure 1.4. How digit positions are used in binary numbers.

computer particularly useful if it could deal only with numbers beween Ø and 255, so we make use of these numbers as codes. Each number code can, in fact, be used to mean several different things. If you have worked with ASCII codes in BASIC, you will know that each letter of the alphabet and each of the digits Ø to 9, and each punctuation mark, is coded in ASCII as a number between 32 (the space) and 127 (the left-arrow). That selection leaves you with a large number of ASCII code numbers which can be used for other purposes such as graphics characters. The ASCII code is not the only one, however. CPC464 computers use their own coded meanings for other numbers in this range of Ø to 255. For example, when you type the word PRINT in a program line, what is placed in the memory of the CPC464 computer (when you press ENTER) is not the sequence of ASCII codes for PRINT. This would be 8Ø,82,73,78,84 , one byte for each letter. What is put into memory, in fact, is one byte only, the binary form of the number 191. Memory is a precious commodity in small computers, and using a single byte in this way is an obvious saving of memory. This single byte is called a 'token' and it can be used by the computer in two ways. One way is to locate the ASCII codes for the characters that make up the word

PRINT. These are stored in the ROM, so that when you LIST a program, you will see the word PRINT appear, not a character whose code is 191. The other, even more important, use of the 'token' is to locate a set of instructions which are also held in the ROM in the form of number codes. These instructions will cause characters to be printed on the screen, and the numbers that make up these codes are what we call 'machine code'. They control *directly* what the 'machine' does. That direct control is our reason for wanting to use machine code. When we use BASIC, the only commands we can use are the ones for which 'tokens' are provided. If there's no token for an action, we can't carry out that action. What we normally do is to make up a set of BASIC instructions that will carry out the action that we want. That's what a BASIC program is. By using machine code, though, we can make up our own commands and do what we please. Incidentally, the fact that PRINT generates one 'token' is the reason why it is possible to use ? in place of PRINT. The CPC464 computer has been designed so that a ? which is not placed between quotes will also cause 191 to be put into memory.

Do-it-yourself spot

As an aid to digesting all that information, try a short program. This one, in Figure 1.5, is designed to reveal some of the stored signals that are stored in the RAM and it makes use of the BASIC instruction word PEEK. PEEK has to be followed by a number or number variable within brackets, and it means 'find what byte is stored at this address number'. All of the bytes of RAM memory within your CPC464 computer are numbered from zero upwards, one number for each byte. Because this is so much like the numbering of houses in a road, we refer to these numbers as addresses. The action of PEEK is to find what number, which must be between Ø and 255, is stored at each address. The CPC464 computer automatically converts these numbers from the binary form in which they are stored into the ordinary decimal (more correctly, denary) numbers that we normally use. By using CHR$ in our program, we can print the character whose ASCII code is the number we have PEEKed at. The program uses the variable N as an address number, and prints the characters whose codes are stored. As you can see

```
10 CLS
20 FOR N=46151 TO 46179
30 J%=PEEK(N)
40 IF J%<32 THEN 60
50 PRINT CHR$(J%);
60 NEXT
70 PRINT
```

Figure 1.5. A program that reveals the programmable key assignments.

when you run it, this prints a set of numbers and the word RUN". These numbers are stored between addresses 46151 and 46179, and they are the numbers which are allocated to the programmed function keys. They have to be stored in the RAM, because otherwise you could not alter them. Other codes are also stored along with the numbers, and the 'filter' line (line 4Ø), removes the numbers which are not ASCII codes. If you remove this line, the results will look slightly odd! What the program has done is to find each stored byte by using PEEK, and to print its ASCII character shape by using CHR$. Any part of the RAM memory can be 'peeked' in this way, provided that we know what address numbers to use.

You know that you can reprogram the function keys by using commands like KEY1,"PRINTTAB(". This reprogramming would not be possible unless the key words were held in RAM memory, because only RAM memory can be altered. The numbers, along with RUN", which are allocated to these keys must also be held in ROM, however. They have to be stored in RAM while the computer is switched on, so that you can alter them as you wish. They also have to be stored in ROM because if this were not done, the words would not be available when you switched the machine on. When you switch on, one of the first actions of your computer will be to copy the bytes that are in the ROM addresses into the addresses in RAM where they are also stored.

Now try this. Reprogram one of these keys, for example by typing:

```
KEY Ø,"RUBBISH"
```

and then RUN the program of Figure 1.5 again. This time, you will see the result of your reprogramming appearing in place of one of the numbers. If you used the example, you will find the word 'RUBBISH' appearing in place of the number zero when the program runs. You will also find that RUN" no longer appears. This is not because RUN" has been removed, it's just that you are looking at a limited range of memory. When you replace the digit Ø with the word RUBBISH, you need more memory, and as a result, all the stored characters are shifted upwards. Since the program of Figure 1.5 peeks only the addresses from 46151 to 46179, though, anything that is stored above address 46179 does not appear.

ROM and RAM

On most computers, PEEK can read a byte from ROM or RAM memory. The arrangement of memory in the CPC464 is not so simple as in other computers, however. So that you can make use of a lot of RAM, much more than most small computers, many of the address numbers are used by *both ROM and RAM*. This is done by switching the different types of memory on or off. For example, the addresses from 49152 to 65535 are used by both ROM and RAM. When you use the PEEK command, it *automatically* switches off the ROM in this range of addresses, and switches on the RAM. The result is that you cannot PEEK the ROM in this range. If you program only in BASIC, this complication is completely hidden from you, because you wouldn't normally use PEEK (to a ROM address) in a BASIC program. When the machine runs a BASIC program, it will switch the different types of memory on or off as it needs. It's only when you come to use machine code that you will find problems connected with the switching of memory. For the moment, then, you will have to accept that you cannot look at what is stored in either of the two sets of ROM memory. One set, the 'lower ROM', uses address numbers from Ø to 16384, the other (upper ROM) uses 49152 to 65535. When you PEEK memory, however, you will always peek the RAM which uses these same address numbers, not the ROM. You can find out what the ROM holds by using the HiSoft DEVPAC program, and also by the use of a short machine code program in Chapter 9.

CPC464 computer dissection

Now take a look at a diagram of the CPC464 computer in Figure 1.6. It's quite a simple diagram because I've omitted all of the detail, but it's enough to give you a clue about what's going on inside. 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 the 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 electrical signals in the computer.

The names of two of the blocks should be familiar already, ROM and RAM, but the other two are not. The block that is marked MPU is a

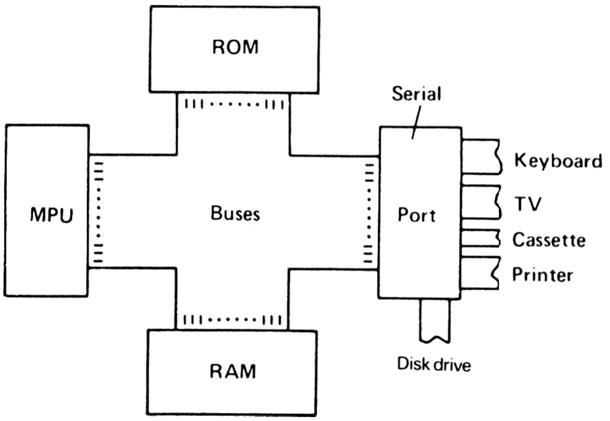


Figure 1.6. A block diagram of the CPC464 computer. The connections marked 'Buses' consist of a large number of connecting links which join all of the units of the system.

particularly important one. MPU means Microprocessor Unit – some block diagrams use the letters CPU (Central Processing Unit). The MPU is the main 'doing' unit in the system, and it is, in fact, one single unit. The MPU is a single plug-in chunk, one of these silicon chips that you read about, encased in a slab of black plastic and provided with 4Ø connecting pins that are arranged in two rows of 2Ø (Figure 1.7). There are several different types of MPU made by different manufacturers, and the one in your CPC464 is called Z80 (or Z80A). It's quite different from the MPU chips that are used in many other computers, so that books about other MPU chips such as the 6502 or the 68008 won't be of much help in understanding the Z80.

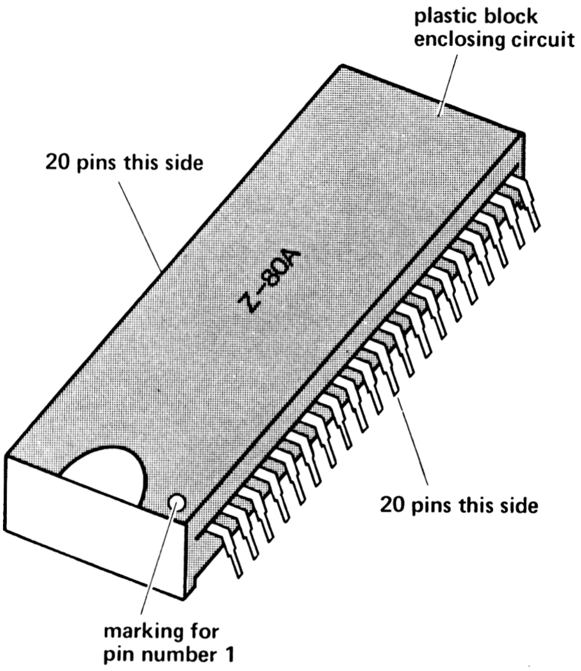


Figure 1.7. The Z80 MPU. The actual working part is smaller than a fingernail, and the larger plastic case (52 mm by 14 mm wide) makes it easier to work with.

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 which is stored in the memory can be copied into another store within the MPU. The MPU can also store a byte, meaning that a copy of a byte that is stored within the MPU can be placed in any address in the memory. These two actions (see Figure 1.8) are the ones that the MPU spends most of its working life in carrying out. By combining them, we can copy a byte from any address in memory to any other. You don't think that's very useful? That copying action is just what

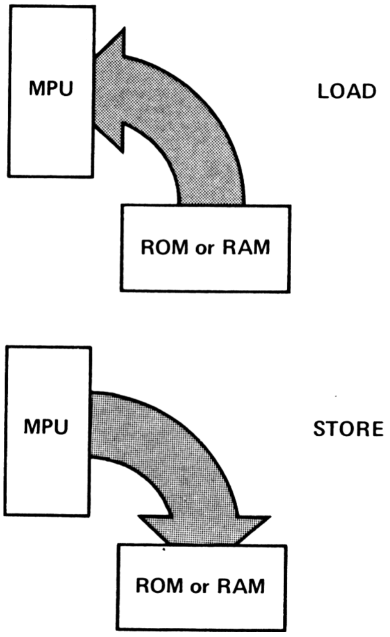


Figure 1.8. Loading and storing. Loading means signalling to the MPU from the memory, so that the digits of a byte are copied into the MPU. Storing is the opposite process.

goes on when you press the letter H on the keyboard and see the H appear on the screen. The MPU treats the keyboard as one piece of memory and the screen as another, and copies bytes from one to the other as you type. That's a considerable simplification, but it will do for now just to show how important the action is. When you think of it, everything that you do when you are typing a BASIC program is a copying action. You type a letter on the keyboard, and it appears on the screen because of this copying action. It is also stored in the memory of the computer because of a copying action. After you have typed the program, you can record it on tape by making use of another type of copying action. All of these actions make use of loading and storing, and all of them are carried out by the MPU. Even when you RUN a BASIC program, a large number of the actions are, once again, just copying actions.

Loading and storing are two very important actions of the MPU, but there are several others. One set of actions is the 'arithmetic set'. For most types of MPU, these consist of addition and subtraction only. Most of the arithmetic operations can use only single-byte numbers. Since a single-byte number means a number between Ø and 255, how does the computer manage to carry out actions like multiplication of large numbers, division, raising to powers, logarithms, sines, and all the rest? The answer is by

AND

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

$$1 \text{ AND } 1 = 1 \left\{\begin{matrix} 1 \text{ AND } 1 = \emptyset \\ \emptyset \text{ AND } 1 = \emptyset \end{matrix}\right\} \emptyset \text{ AND } \emptyset = \emptyset$$

For two bytes, corresponding bits are ANDed

$$\begin{array}{rr} & 1\emptyset11\emptyset111 \\ \text{AND} & \emptyset\emptyset\emptyset\emptyset1111 \\ \hline & \emptyset\emptyset\emptyset\emptyset\emptyset\underbrace{111} \end{array}$$

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:

$$1 \text{ OR } 1 = 1 \left\{\begin{matrix} 1 \text{ OR } \emptyset = 1 \\ \emptyset \text{ OR } 1 = 1 \end{matrix}\right\} \emptyset \text{ OR } \emptyset = \emptyset$$

For two bytes, corresponding bits are ORed

$$\begin{array}{rr} & 1\emptyset11\emptyset111 \\ \text{OR} & \emptyset\emptyset\emptyset\emptyset1111 \\ \hline & 1\underset{\uparrow}{\emptyset}111111 \end{array}$$

only
bit which
is Ø in
both.

XOR (Exclusive-OR)

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

$$1 \text{ XOR } 1 = \emptyset \left\{\begin{matrix} 1 \text{ XOR } \emptyset = 1 \\ \emptyset \text{ XOR } 1 = 1 \end{matrix}\right\} \emptyset \text{ XOR } \emptyset = \emptyset$$

$$\begin{array}{rr} & 1\emptyset11\emptyset111 \\ \text{XOR} & \emptyset\emptyset\emptyset\emptyset1111 \\ \hline & \underbrace{1\emptyset111\emptyset\emptyset\emptyset} \end{array}$$

if two bits
are identical
the result
is zero.

Figure 1.9. The rules for the three important logic actions, AND, OR and XOR.

machine code programs that are stored in the ROM. If these programs were not there you would have to write your own. There aren't many computer users who would like to set about that task, and yet for some reason, lots of books about machine code programming concentrate on these actions.

There's also the logic set. Logic means making decisions from given information. In computer terms, this means comparing two numbers so as to produce a third number. MPU logic is, like all MPU actions, simple and subject to rigorous rules. Logic actions compare the individual bits of two bytes and produce an 'answer' which depends on the values of the bits that are compared and on the logic rule that is being followed. The three logic rules are called AND, OR and XOR, and Figure 1.9 shows how they are applied. When the AND logic is in use, the 'result' of ANDing two bits is 1 only if *both* bits are 1. If either bit is a Ø, then the result is Ø, and if both bits are zero, the result is Ø. How is this used? Well, in BASIC, you may well have programmed something like:

```
IF A$="END" AND B%=1ØØ THEN 5ØØØ
```

to decide when something is to happen in a program. BASIC uses 1 and Ø codes to carry out these tests, using 11111111 to mean true and ØØØØØØØØ to mean false. Now if A$="END", then the binary number 11111111 will be stored, and if B%=1ØØ, another 11111111 will be stored. Doing an AND on these will give another 11111111, meaning true, and the result will be the jump to line 5ØØØ which has been programmed. The important point here is that the BASIC does not carry out logical actions like AND, OR, XOR, on items like A$="END", only on the 'true' or 'false' numbers 11111111 and ØØØØØØØØ.

The OR action gives a Ø only when both bits that are being compared are Ø. Only Ø OR Ø gives Ø, because 1 OR Ø is 1, and Ø OR 1 is 1. 1 OR 1 is also 1. This is used with the TRUE or FALSE number codes in the same way as AND, so that if you have a line in BASIC that reads:

```
IF A$="X" OR A$="x"
```

then if one of these is true (11111111) and the other is false (ØØØØØØØØ), the comparison gives 11111111, which means true. The XOR action is very similar to the OR action, but when both bits are 1, the result is Ø, not 1. The two points that are important about XOR are that if you XOR a binary number with itself, you get zero. If you XOR a binary number twice with another binary number, you get back to the first number (Figure 1.10). This can be used for coding and decoding purposes. If a number is coded by XORing it with a 'key' number, the result can be used as a code. When this code is XORed again with the 'key', the first number is recovered. Remember when we talk of numbers, that this is the raw material that the computer uses. By using ASCII codes, we can code any message as a number or set of numbers, so that these methods apply as much to letters of the alphabet as to simple numbers.

Binary number:	Ø11Ø1Ø11	
XOR with –	11Ø11ØØØ	Key
	1Ø11ØØ11	Result
	↓	
	1Ø11ØØ11	
XOR with –	11Ø11ØØØ	Key
	Ø11Ø1Ø11	Original number again

Figure 1.10. If you XOR one number with another (the 'key'), then the result is a number. If you XOR this number with the key again, you get the original number back again!

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

```
IF A=36 THEN GOTO 1Ø5Ø
```

so the MPU can be made to carry out an instruction which is at an entirely different address from the normal next address. The MPU is a programmed device, meaning that it carries out each of its actions 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. This is very similar to the way that BASIC carries out the instructions in a line, and then moves to the next line in order. A jump instruction would prevent this from happening, and would instead cause the MPU to read from another address, the one that was specified in the jump instruction. This jump action can be made to depend on the result of a test. The test will usually be carried out on the result of the previous action, whether it gave a zero, positive or negative result, for example.

That isn't a very long or exciting list, but the actions that I've omitted are either unimportant at this stage, or not particularly 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 programmed and that it can carry out its actions very quickly. Equally important is the fact that the microprocessor can be programmed by sending it electrical signals.

These signals are sent to eight pins, called the data pins, of the MPU. It doesn't take much of a guess to realise that these eight pins correspond to the eight bits of a byte. Each byte of the memory can therefore affect the MPU by sharing its electrical signals with the MPU. Since this is a long-winded description of the process, we call it 'reading'. Reading means that a byte of memory is connected along eight lines to the MPU, so that each 1 bit will cause a 1 signal on a data pin, and each Ø bit will cause a Ø signal on a data pin. Just as reading a paper or listening to a recording does not destroy what is written or recorded, reading a memory does not change the memory in any way, and nothing is taken out. The opposite process of writing does, however, change the memory. Like recording a tape, writing wipes out whatever existed there before. When the MPU writes a byte to an address in the RAM memory, whatever was formerly stored at that address is no longer there, it 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.

Table d'hôte?

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 if you write only in BASIC, you don't write these. All that 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. Your choice is limited to the keywords that are designed into the ROM. We can't alter the ROM, and if we want to carry out an action that is not provided for by a keyword, we must either combine a number of keywords (a BASIC program) or operate directly on the MPU with number codes (machine code). When you have to carry out actions by combining a number of BASIC commands, the result is 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 am 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 which will 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 bother about the details.

Now think of another possibility. 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, measured along the fence southwards, to a point 87 feet from the lounge end of the house measured along that fence southwards. Dig a trench eighteen inches deep and one foot wide along the path of your line. Mix three bags of sand and two of cement with four barrow-loads of pebbles for three minutes. Mix water into this until a pail filled with the mixture will take ten seconds to empty when held upside down. Fill the trench with the mixture ...'. The instructions are very detailed – they have to be for a brainless robot – but they will be carried out flawlessly and quickly. If you've forgotten anything, no matter how obvious, it won't be done. Forget to specify where the cement, sand and water are kept, 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, and the robot will keep piling one layer on top of another, like the Sorcerer's Apprentice, until someone sneezes and the whole 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, drawing on a lot of instructions that are not yours – but it may not be done as fast as you like. It might even be done in a way that you don't want. If you can be bothered with specifying the detail, machine code is a lot faster because you are giving your instructions direct to an incredibly fast but mindless machine, the microprocessor. We can stretch the similarity further. If you said to your builder 'mend the car', he might be unwilling or unable to do so. The correct set of detailed instructions to the robot would, however, get this job done. Machine code can be used to make your computer carry out actions that are simply 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 quite so important as it used to be.

One last look at the block diagram is needed before we start on the inner workings of the CPC464 computer. The block which is marked 'Port' includes more than one chip. A *port* in computing language means something that is used to pass information, one byte at a time, into or out from the rest of the system – the MPU, ROM and RAM. 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. In addition, we can isolate inputs and outputs from the normal action of the MPU. This is why nothing appears on the screen in a BASIC program except where we have a PRINT command in the program. It's also why pressing the PLAY key of the cassette recorder has no effect until you type LOAD"" and press ENTER. The port keeps the action of the computer hidden from you until you actually need to have an input or an output. The CPC464 uses several ports for such purposes as connections to the keyboard, joysticks, the cassette recorder, the printer and the disk drive.

The POKE action

The programs that we have looked at so far have read the memory of the machine, using the PEEK command. There is another BASIC command, POKE, which performs the opposite action. POKE will write a byte into the memory, and you have to specify what byte and what address. In BASIC, POKE uses ordinary (denary) numbers unless you specify otherwise, so that the address has to be between Ø and 65535, and the byte number between Ø and 255. The POKE command will *always* switch RAM memory on, because you cannot alter the ROM in any case. A typical POKE command would be POKE 3277Ø,5 with the address number following POKE, then a comma, then the byte number. This would have the effect of storing the number 5 in address 3277Ø. You can POKE to addresses Ø to 88, but you should not, because these are important addresses that are used a lot by the machine.

You see, POKE is a command that can get you into a lot of trouble unless you know what you are doing. You can PEEK the memory of the computer as much as you like, because PEEK only copies, it doesn't alter what is stored. POKE, however, can replace one byte by another. If the address that you POKE happens to contain something that is vital to the way that the computer works, then the result of your POKE will be to send the machine bananas! You can expect to see weird patterns on the screen, and to have the keyboard 'seize-up', so that pressing keys has no effect. When this happens, pressing ESC *may* restore normal operation, but very often only using CTRL SHIFT ESC will have a real effect. You may even have to switch off, and then on again. When you do either of these things, you'll lose any program that you had in the memory. Therefore, when you work with machine code – which *always* places bytes directly into memory – or when you use POKE commands from BASIC, you *must* be certain that you record any program before you try it. Failing to do this may mean losing the program and having to type it all over again. Be warned!

There's one type of POKE that you can carry out with a lot less risk, however. When you POKE addresses in the range 49152 to 65535, the effects are seen on the screen. As far as a POKE is concerned, the addresses in this range are all of RAM, and this part of the RAM is used to store the signals that control what you see on the screen. When you write to these addresses, the RAM is switched in, and the ROM is switched out. When a BASIC program runs, however, reading addresses in this range causes the RAM to be switched out and the ROM switched in. This is because the ROM which contains the instructions for using BASIC also uses the addresses 49152 to 65535.

Now the way that this part of the memory is used to store signals is very complicated indeed, and we'll do no more than glance at it in this chapter. The way that the memory is used depends on the screen MODE that is in use, and anything that you POKE will normally create some sort of pattern. Try, for example, the program in Figure 1.11. This pokes 17Ø (which in binary is 1Ø1Ø1Ø1Ø) into the addresses 55296 to 55375. Now these are eighty addresses that are used to form one line of patterns on the screen, and the effect of the program is to give a dotted line. Why is the line dotted? Because of the pattern 1Ø1Ø1Ø1Ø, which is alternately 1 and Ø, or on and off.

```
10 CLS
20 FOR N=55296 TO 55375
30 POKE N,170
40 NEXT
```

Figure 1.11. The effect of using the POKE command in this range of addresses is to place characters on to the screen.

Now try some modifications to the program. Try other numbers which give a 1∅ or ∅1 pattern in binary – Figure 1.12 shows a range of such numbers. You will find that all of them will give you a dotted line, but the colour will not always be yellow! We're not going to look at the complication of colours at this stage. The effect of POKE on this range of screen addresses is to set pixels. If you imagine that the screen of the monitor is divided into lines, then a pixel is a little slice of each line. When you switch on the CPC464, screen Mode 1 is selected, and there are 32∅ pixels in each line in this mode. Each byte of memory controls four pixels, which means that two bits control a pixel. This is why you will find that some numbers appear to give the same effect as others. We'll look at all this in much more detail later.

We have now looked at all of the important sections of your CPC464 computer. I've used some terms loosely – purists will object to the way I've used the word 'port', for example – but no-one can quarrel 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 so that it can be programmed in BASIC and can run a BASIC program. It looks like a good place to start another chapter!

∅1∅1∅1∅1	= (denary)	85
∅∅∅∅1111	= (denary)	15
1111∅∅∅∅	= (denary)	240
11∅∅11∅∅	= (denary)	204
∅∅11∅∅11	= (denary)	51

Figure 1.12. Some other numbers which contain sets of alternate 1s and 0s.

Chapter Two
Digging Inside the CPC464

I don't mean 'digging' literally – you don't have to open up the case of your computer. What I do mean is that we are going to look at how the CPC464 computer is designed to load and run BASIC programs. Once you know how this is done, you should be able to see how machine code is used, and this will be very helpful later on when we start to look at how we can use ROM routines. We'll start with a simplified version of the action of the whole system, omitting details for the moment.

The ROM of your CPC464 computer consists of a large number of short programs – *subroutines* – which are written in machine code, along with sets of values (tables) arranged like the table of words for the programmable keys that you have seen already. The ROM is, in fact, in two parts, one between addresses Ø and 16383, the other between 49152 and 65535. The upper ROM contains all the routines for BASIC, the lower one contains the 'service' routines which would be needed by any language. Any language, for example, will need to use the keyboard, the screen, and the cassette system. There will be at least one machine code subroutine for each keyword in BASIC, and some of the keywords may require the use of many subroutines in sequence. When you switch on your CPC464 computer, the piece of machine code that is carried out first of all is called the 'initialisation routine'. This is a long piece of program, but because machine code is fast, carrying out instructions at the rate of many thousands per second, you see very little evidence of all this activity. All that you notice is a very slight delay between switching on and seeing the copyright notice placed on the screen and then the 'Ready' prompt. In this brief time, though, the action of the RAM part of the memory has been checked, some of the RAM has been 'written' with bytes that will be used later (such as programmable-key words), and most of the RAM has been cleared for use. Cleared for use as far as the CPC464 is concerned means that nothing but zeros will be stored in most of the RAM. When you switch off the computer, the RAM loses all trace of stored signals, but when you switch on again the memory cells don't remain storing zeros. In each byte, some of the bits will switch to 1 and some will switch to Ø when power is applied. This happens quite at random, so that if you could examine what was stored in each byte just after switching on, you would find a set of meaningless numbers. These would consist of numbers in the range Ø to 255, the normal range of numbers for a byte of memory. These numbers are 'garbage' – they weren't put into memory deliberately, nor do they form useful instructions or data. The first job of the computer, then, is to clean up. In place of the random numbers, the computer substitutes a set of zeros, a completely clear memory. Try this – switch on, and type (with no line number):

```
FOR N=1ØØØ TO 15ØØ:?PEEK(N);:NEXT
```

– and then press ENTER. The range of memory addresses we have used is part of the 'BASIC' range, where the later bytes of a BASIC program are normally stored. I have deliberately avoided the first part of this region, because it would contain some bytes which are the result of the command. Initialisation always leaves some bytes in the first few memory addresses from Ø up to 89, and any command that you type will also be stored in memory at around address 368. The rest of the usable RAM up to 44Ø31 is cleared. The RAM above 49152 is, as we have seen, used for the screen display.

The initialising program has a lot more to do. Some of the higher section of RAM, from address 44Ø32 to 49151, is for 'system use'. This is because the machine code subroutines which carry out the actions of BASIC need to store quantities in memory as they are working. We'll look at how some of these address numbers are used in a moment. In addition, some RAM has also to be used to hold quantities that are created when a program runs. That's what we are going to look at now.

Variables on the table

BASIC programs make a lot of use of variables, meaning the use of letters to represent numbers and words. Each time you 'declare a variable' by using a line like:

```
N=2Ø
```

or

```
A$="SMITH"
```

the computer has to take up memory space with the name (N or A$ or whatever you have used) and the value (like 2Ø or SMITH) that you have assigned to it. The piece of memory that is used to keep track of variables is called the variable list table (VLT). It doesn't occupy any fixed place in the memory, but is stored in the free space just above your program. If you add one more line to your program, the VLT address has to be moved to a set of higher address numbers. If you delete a line from your program, the VLT will be moved down in the same way so that it is always kept just following the last line of BASIC.

Now because the variable list table address can and does move around as the program is altered, the computer must at all times keep a note of where the table starts. This is done by using two of the pieces of memory that are reserved for system use, the addresses 44677 and 44678. You may wonder why two addresses are used. The reason is that one byte can hold a number only up to 255 in value. If we use two bytes, however, we can hold the number of 256s in one of these bytes and the remainder in the other. A number like 257, for example, is one 256 and one remaining. We could code this as 1,1. This means that a 1 stored in the byte that is reserved for 256s, and 1 in the byte reserved for units. The order of storing the numbers is *always low-byte then high-byte*. To find the number that is stored, we multiply the high-byte number by 256 and then add the low-byte number. For example, if you found 23,163 stored in two consecutive addresses that were used in this way, this would mean the number:

```
163*256+23=41751
```

The biggest number that we can store using two bytes like this is 255,255, which means 255*256+255=65535. This is the reason that you can't use very large numbers like 7ØØØØ as line numbers in the CPC464 computer – the operating system uses only two bytes to store its line numbers. You can use any number up to and including 65535, but nothing higher.

All of this means that we can find the address that is stored in addresses 44677 and 44678 by using the formula:

```
?PEEK(44677)+256*PEEK(44678)
```

If you use this just after you have switched on your CPC464 computer, then the result is the address number 37Ø. This is just above the address at which the first byte of a BASIC program would be stored. To see this in action, type the line:

```
1Ø N=2Ø
```

and try ?PEEK(44677)+256*PEEK(44678) again. If you typed this line as I did, with a space between the line number and the 'N', then the address that you get is 382. The variable list table has moved upwards in the memory, by 12 bytes. That's more than the number of bytes that you typed, you'll notice – reasons later. The important point, however, is that this address number has changed to allow for a line of program being put into the memory.

Quite a lot of important addresses that the computer uses are 'dynamically allocated' like this. Dynamically allocated means that the computer will automatically change the address where groups of bytes are to be kept. It will then keep track of where they have been stored by altering an address that is held in a pair of bytes such as in this example. This has important implications for how you use your computer. For example, if you shift the VLT by poking new numbers into addresses 44677 and 44678, the computer can't find its variable values. Try this – after finding the VLT address, but without running the one-line program, 1Ø N=2Ø, type ?N . The answer will be zero. Why? Because the program has not been run. The address 382 is where the VLT will start, but there's no VLT created *until the program runs*. This makes it easy for you to add or delete lines at this stage. All that will have to be altered is the pair of numbers in addresses 44677 and 44678. The VLT values are put in place only when the program runs, and the table itself is never actually moved, only re-created. All that changes when you edit is the starting address in 44677 and 44678. That's why you can't resume a program after editing – you have to RUN again to create a new VLT at a new address. If you RUN the one-line program now, and then type ?N you will get the expected answer of 2Ø. Now type (no line number) POKE 44678,2 and press ENTER. Try ?N and see what you get. My CPC464 computer gave the answer Ø for this. The reason is that putting a '2' in as the higher byte of the VLT address gives a new answer when the computer looks for this address. When you command 'PRINT N', then the computer finds the value of 'N' from its VLT, and if the VLT address is changed, the values that the computer finds *must* be incorrect. In this example, the VLT address has been changed to a part of memory which will contain only zeros (assuming that the computer has just been switched on). If other bytes were stored in this region, all sorts of odd effects could arise, and you might find that the computer 'seized up' and had to be reset by using CTRL SHIFT and ESC.

A look at the table

It's time now to do something more constructive, and take a look at what is stored in the VLT. When we do these investigations, it's important to ensure that the computer is clear of the results of previous work, so it's advisable to press the CTRL SHIFT and ESC keys each time. The CPC464 is one of the few machines which allows a set of keys like this to give the effect of switching on – the effect that we call a 'cold start'. On most machines, this can be done only by switching off and then on again.

To work, then. After using the CTRL SHIFT ESC keys, type the line:

```
1Ø N=2Ø
```

again, and find the VLT address by using ?PEEK(44677)+256*PEEK (44678). This gave me the address 382 again. Now type RUN, so that values are put into the VLT, and take a look at what has been stored there. This is done by using the command:

```
FORX=382 TO 393:?X;" ";PEEK(X):NEXT
```

and pressing ENTER. This gives the listing that is illustrated in Figure 2.1. Now can we recognise anything here? We might not immediately recognise the third byte of 2Ø6, but we will if we subtract 128 from it. Subtracting 128

```
382   Ø          388   Ø
383   Ø          389   32
384   2Ø6        39Ø   133
385   4          391   Ø
386   Ø          392   Ø
387   Ø          393   216
```

Figure 2.1. The variable list table entry for an assignment to a number variable.

gives 78, which we certainly should recognise, because that's the ASCII code for N! The next byte is 4, which is a code for the type of variable. This follows the last letter of the name which in this example is also the first letter. If we used a variable name of more than one letter (up to 4Ø letters can be used), then the complete name would appear in ASCII code, with *only the last letter having 128 added to its code.* For example, if the program line had been:

```
1Ø NAMEOFIT=2Ø
```

then the sequence of codes for the name would have been 78 65 77 69 79 70 73 212, which is just the ordinary ASCII codes except for the last letter.

The next set of five bytes, then, must be the way that the number 2Ø has been coded. At this point, don't worry about how these numbers are used to represent 2Ø – just accept that they do! How do I know that it's the next five bytes that represent the number 2Ø? Easy, because the byte at address 393 is 216. Subtract 128 from this and you get the ASCII code for X, which is 88. Why should the ASCII code for 'X' appear? Simply because we used X as the variable in the command to print the VLT values! This, then, is the start of the table for the next variable that we used, and the bytes up to this point must represent the value of 2Ø. The coding that is used for these VLT entries is not exactly easy to follow unless you are really familiar with binary arithmetic. For an ordinary number which is represented by a variable name with no % sign, the CPC464 computer always uses five bytes for any value, no matter whether it's a small number like 2Ø, or a very much larger one like 1444677Ø68315, or a fraction, or negative. This makes the storage of number variables simple, and it also makes it easy for the computer to find variables. Any letter that is used for a number variable, remember, will be used for numbers of this type unless you mark it with a % sign. If, for example, the computer is looking for the value of a variable called Y, then when it finds 'N' (coded as ASCII 78 + 128 = 2Ø6) it need not waste time with the next eight bytes (one for type code, five for the value, and two zeros), but move to the next place where a variable name will be stored. If you are curious, and have a head for mathematics, Appendix A shows what method of coding is used to convert numbers into five bytes for ordinary 'floating-point' number variables. For the purposes of this book you don't, however, need to understand how the coding is done as long as you know how the code is stored and how many bytes are needed.

The storage of an integer is much more simple. Type a two-line program:

```
1Ø A%=5Ø
2Ø B%=3ØØ
```

and find the position of the VLT as before; it should be at address 395. Now use:

```
FOR N=395 TO 415:?N;" ";PEEK(N):NEXT
```

and press ENTER. This time, you'll see a different arrangement. The letter A is stored as 193 (128+65), and is followed by a 1. The '1' is the code for integer. The next number is 5Ø, which was the first integer value. The next entry is 194, the code for B, with the '1' following it to mean an integer value. This time, though, the number needs two bytes, with 44 in the first byte and 1 in the next. As always when two bytes are used, this means 1*256+44, which is 3ØØ. Integers, then are stored with just two bytes following the type number code of 1. A zero is used to separate the integer entries in the table.

Tying up a string

Now we need to take a look at how a string variable is stored. Use CTRL SHIFT and ESC again, and then type the line:

```
1Ø AB$="THIS IS A STRING"
```

RUN this one-liner, and then find the VLT address by using addresses 44677 and 44678 as before. I obtained 399 for this. Now use:

```
FOR X=399 TO 41Ø:?X;" ";PEEK(X):NEXT
```

to find what is in the VLT. This time, it's as Figure 2.2 shows. The name AB$ has been coded as 65 194, which is the ASCII code for 'A' followed by the code for 'B' with 128 added. The first value following this is 2, which tells us that this is a string variable, not a number, because number variables use 'recognition' bytes like 1 or 4. What we have to account for now is the three bytes that follow the name, because these should be the three bytes that are used to encode the string! Is there, perhaps, some clue here to the use of the 'type' numbers? We found 4 for a floating-point number which used five

```
399    0          405    123
400    0          406    1
401    65         407    0
402    194        408    0
403    2          409    216
404    16         410    4
```

Figure 2.2. The VLT entry for a string variable assignment.

bytes, 2 for a string which uses three bytes, and 1 for an integer which uses two bytes. It looks as if the type number is one less than the number of bytes that are used in the coding. This would make sense, because computer counting so often starts at zero, and Ø to 4 is five bytes, zero to 2 is three bytes, and Ø to 1 is two bytes.

Now three bytes can hardly be the ASCII codes for the letters, can they? The clue to what is being done emerges when we take a look at the numbers. The first number of the set (at address 4Ø4) is 16. Now 16 is the number of characters in the string. If you count the number of letters and spaces you'll see that this is what it comes to. The next byte is 123, and it is followed by 1. Now two bytes together are always likely to be an address, and if we combine them in the usual way, using 123+256*1, then we get 379. Next step in the trail is ?PEEK(379). Sure enough, it's 84, which is the ASCII code for 'T'. 379 is the address of the first byte of the string.

Let's gather all this up. The CPC464 computer stores an entry of four bytes in its VLT for each string. Of these four bytes, the first one is the '2' that specifies how many bytes are needed for the coding of string length and address. The next three bytes then contain the length of the string and the address in memory of its first byte. Only one byte is needed for the length because no string will be allowed to exceed 255 characters. Two bytes are needed for the address, which is stored in the usual low-then-high order. The VLT entry also includes the name, which can be of up to 4Ø characters, with 128 added to the last ASCII code.

In this example, the string is stored at an address which is lower than the address of the VLT, in the 'BASIC text' part of the memory. This is the part of the memory which contains the program, and since the ASCII codes for the string are placed here when you type the program, it's as good a resting place as any. Numbers must be transferred to the VLT because they are not stored as ASCII codes. This is why you cannot carry out string operations on a number variable, nor number operations on a string variable. The question now, is, what happens when a string is created which does not exist in the program? If you type, for example:

```
1Ø A$="AB":B$="CD":C$=A$+B$
```

and RUN this, you will find that your VLT is longer, as you might expect. You will have to look at memory addresses from 4Ø9 to 432 this time. You will find the entries for A$ and B$, just as you would expect, giving addresses inside the program memory region, as shown in Figure 2.3. The variable C$, however, gives the bytes 44 and 117 for its address. This corresponds to an address of 29996 (it's 44+256*117, remember) for this string. We can take a look at these addresses. If you type:

```
FOR X=29996 TO 29999:?X;" ";PEEK(X);" "CHR$(PEEK(X))
  :NEXT
```

then all will be revealed. The ASCII codes for letters ABCD are now stored

```
409     0               421     132
410     0               422     1
411     193             423     0
412     2               424     0
413     2               425     195
414     122             426     2
415     1               427     4
416     0               428     44
417     0               429     117
418     194             430     0
419     2               431     0
420     2               432     216
```

Figure 2.3. The VLT entry for a string which is not stored in the program part of memory.

here, and the use of CHR$ in the program reveals them. This part of memory is allocated for string storage as required. The top of this piece of memory is normally at 3ØØØ2, and the ASCII codes for strings are stored up to address 29999. The strings are stored so that the last character code will always be at address 29999. Only strings which have not been assigned in the program will be stored here. Unless you create strings by using concatenation, or slicing commands like LEFT$, MID$ and RIGHT$, or conversions like S$=STR$(V), this storage space will not be needed. For the moment, though, try this. Press CLR SHIFT and ESC to clear the memory, then type the short program:

```
1Ø A$="THIS IS A"
2Ø B$="THIS IS B"
```

Run this, and then use ?A$,B$ to show that the values have been allocated. Now find the address of the VLT, which in my example was 412. Print the VLT, using X=412 to 432, and you will see the two string entries. Now these entries are almost identical, because the strings have the same length. They are also located fairly close to each other, and the only difference between the entries is that the lower byte of the address is different. When I tried this, I found that address 417 held 122, and address 424 held 143. Now if this is the only difference between the two string entries, we could swap the strings just by swapping these bytes! You can do this by using the following steps:

```
J=PEEK(417)             (then ENTER)
POKE 417,PEEK(424)      (then ENTER)
POKE 424,J              (then ENTER)
```

Try ?A$,B$ after this runs, and you will find that the values have been swapped. A$ is now 'THIS IS B' and B$ is now 'THIS IS A'. The reason is that we have swapped the values in the VLT. Normally, we would have to swap three sets of values, one for length and two for address, but in this case only one byte was different. It casts a new light on swapping variable values when you realise that you don't have to swap every byte in the string, only the three bytes of the VLT entry. Some varieties of BASIC even have a BASIC SWAP command which does this for you. One useful command in CPC464 BASIC that is not much emphasised in the manual is the '@' command. The @ means 'find VLT entry', so if you type, for example, ?@B%, what is printed on your screen is the address of the first value byte in the VLT. For a string, @ gives the address of the length byte, the first of the three that are used. It would, for example, allow you to swap the values of two strings using only BASIC commands, and without having to find any other VLT addresses. We'll look at the use of @ again later.

PEEK-a-program time

It's time now to look at how a program is stored in the memory of your CPC464 computer. As before, we shall rely on PEEKs at parts of the memory to find out what is happening. The first thing we need to know, however, is where a program normally starts in the CPC464 machine. This is at address 367. The byte at this address is *always* Ø, and the program bytes that form the program start at the next address, 368. This part of the memory is the first set of RAM addresses that can be safely used, because addresses below this are used for other purposes.

```
10 A=10
20 PRINT A
30 C$="AMSTRAD"
```

Figure 2.4. A simple BASIC program.

We can therefore start looking at a program as it exists in the memory. Type the program as shown in Figure 2.4, but don't run it. Now the address at which the first byte of this program starts will be the normal starting address of 368. In this example, we'll look at the bytes from 368 to 411. Now when you use the usual loop to print values of the PEEK numbers from this address onwards, you get the list as shown in Figure 2.5. At first sight it looks like a stream of meaningless numbers, but when you look more carefully, you can see some pattern in it. As usual, the ASCII codes act as useful signposts. At 375, for example, you can see the number 195. Subtract 128 from this, and you get 65, which is the ASCII code for 'A'. Since we know that the line is 'A=1Ø', we can look for the rest of this line. The 1Ø is recognisable as 1Ø in address 378, so that the number 239, which immediately follows the 'A', must represent the '=' sign. Now this is *not* the ASCII code for '=', but one of these 'tokens' that I mentioned in Chapter

368	12	390	0
369	0	391	19
370	10	392	0
371	0	393	30
372	13	394	0
373	0	395	3
374	0	396	0
375	193	397	0
376	239	398	195
377	25	399	239
378	10	400	34
379	0	401	65
380	11	402	77
381	0	403	83
382	20	404	84
383	0	405	82
384	191	406	65
385	32	407	68
386	13	408	34
387	0	409	0
388	0	410	0
389	193	411	0

Figure 2.5. The bytes that represent the program in memory.

1. It's a token because the computer is required to carry out an action, not just store an ASCII code here. The 25 which lies between the 239 and the 1∅ is at present a mystery – it's a code number for the computer which notifies the machine about how big the following number is. The ∅ at address 379 marks the end of this line.

Now we have to grapple with the first four bytes of the line, starting at address number 368. Looking at the third and fourth bytes of this line, at addresses 37∅, 371, you can see the numbers 1∅,∅. If you then look over the other bytes, you will see a sequence of 2∅,∅ and 3∅,∅ following – these are the line numbers. There are two bytes reserved for the line numbers because we want to have line numbers higher than 255. For line numbers smaller than 256, the second of these bytes, the more significant byte, is set at zero. A line which was numbered 3∅∅, however, just to take an example, would be coded as 44,1, meaning 1*256+44.

Now we have to look at the first two bytes in each line. In the first line, these are 12,∅. This can't be an address pair, because it gives an address that is not in the part of RAM used for programs. We get a clue if we look at the length of the first line. It consists of twelve bytes, counting from address 368 to the zero which marks the end of the line at 379. This suggests that the first two bytes in each line are used to give the length of the line. Add this number to the address of the first byte of the line, and you get to the first byte of the next line. This is how the operating system of the CPC464 computer can pick out lines, and put them into the correct sequence, no matter what order you use to enter them. Using two bytes means that you could jump from any address to any other, and altering these bytes would make a program work (if at all) in a pretty unexpected way!

Now take a look at the other lines, as they appear stored in the memory. We have met the PRINT token of 191 before, though as it appears in line 2Ø, there are several other bytes between it and the 'A'. Note that the code for 'A' has 128 added to it once again. Note also how the sequence 13 Ø Ø occurs in lines 1Ø and 2Ø. Line 3Ø allocates a string, and the token of 239 is used here again, with the sequence 3 Ø Ø preceding it. The main novelty here is the end of the program. The last line ends with a Ø as usual, but following it, in the place where the length bytes for the next line would be, in addresses 41Ø and 411, is another pair of zeros. This is the marker that the computer uses for END. This is how the computer can recognise the end of a BASIC program, even though you do not use the word END.

We can carry out some interesting changes on a program like this. Suppose, for example, that we poke the addresses that are used to carry the line numbers. If you type:

```
POKE 382,1Ø:POKE 393,1Ø
```

and press RETURN, you will have placed the number 1Ø into each line number address for the lines 2Ø and 3Ø. Now LIST and look at the result! It's a program of line 1Øs. Contrary to what you might expect, this will RUN perfectly normally. All that a program requires to run in the correct way is that the address for the next line should be correct. The line numbers have nothing to do with this, they are simply a convenient way of labelling the lines for our purposes. Many computing languages do not use line numbers at all. A program that has been altered in this way, however, is not completely normal. The command LIST 1Ø, for example, lists all three lines! In this simple example, the alteration of the line numbers makes little difference, and even if the program had contained a GOTO, things would have not have been very different. Try, for example, typing the program again as:

```
1Ø A=2Ø
2Ø PRINT A
3Ø GOTO 1Ø
```

Run this to make sure that it is working, and then alter the second pair of line numbers to 1Ø again, using the same POKE commands. You'll find that the program still works, because the GOTO 1Ø line does not need to use line numbers again *after* it has run for the first time. If you renumber *before* running, it's a very different story. You can, of course, edit this program normally – you can even edit the line numbers to their correct values again. You can also record a program which has been altered in this way, and replay it normally, and RENUM will operate normally to restore the original line numbers. As we'll see in Chapter 9, though, making the first line number equal to zero can have very interesting consequences.

Running a program

Now that we have looked at the way in which a program is coded and stored in the memory of the CPC464 computer, we can give a bit of thought as to how it runs. This action is carried out by the most complicated part of the operating system, and it has to be given a starting address. This address is the usual 368 for all CPC464 computers. Suppose we go through the actions, omitting detail, of the three line program of Figure 2.4. At the first address of the BASIC program, the RUN subroutine will read the first two bytes, and store them temporarily. These bytes will be added to the 'start of BASIC' address when the next line is carried out. The line number bytes are then read and stored. Why? From what we have just seen, there doesn't seem to be much point in having line numbers. The reason is that if there is a syntax error in the line, the computer will be able to print out the message 'Syntax error in 1Ø' rather than 'Syntax error somewhere'! The next bytes are the 13 Ø Ø sequence, which seem to be used to indicate the use of a number variable, and the computer will take the next byte as being a variable name, with 128 added to the ASCII code. In the old days, the word LET had to be used to 'declare a variable'. This required another token, and most modern designs have dispensed with LET on the grounds that it's just as easy to arrange the subroutines differently. The special token for the '=' sign then causes a subroutine to swing into action. This one creates an entry in the variable list table, at the first available address, and puts the letter code(s) and the type number there. The number is then read and converted to the special form that is used, as noted in Appendix A. This set of bytes is also placed in the VLT as the entry for A. The next byte of the program is then read – it's Ø, which marks the end of the line. This is a signal for the address for the next line, which was calculated as the first action, to be placed into the microprocessor. The type of action that we have considered in detail for line 1Ø then repeats with line 2Ø. This time, more has to be done when the action token is read. Since this is the token for PRINT, the subroutine for PRINT must be called up. It will locate the address of the next vacant place on the screen. This is done by keeping a note of the address in a couple of bytes of RAM – read these bytes, and you have the address. The value of A is then found in the VLT, and the bytes converted back to ASCII code form. The codes are then placed, one by one, in the screen memory. After each ASCII code has been stored, the screen memory number will be incremented. Passing ASCII codes to a suitable subroutine causes patterns to be stored in the screen memory. This in turn causes the characters to become visible on the screen, because of another subroutine. Once again, the zero at the end of the line causes the next line number to be used. At the end of the third line, however, the 'next line' number is zero, and the program ends. The computer goes back to its waiting state, ready for another command.

It's not *quite* so simple as that description makes it sound, but the essentials are there. The important thing to realise is that there is a lot of action to be done, and it has to be done one step at a time. What makes this type of BASIC slow is that each token calls up a subroutine, which has to be found. For example, if you have a program that consists of a loop like:

```
1Ø FOR N=1 TO 5Ø
2Ø PRINT N
3Ø NEXT
```

then the action of reading the PRINT token of 191, and finding where the correct subroutine is stored, will be carried out 5Ø times. There is no simple way of ensuring that the subroutine is located once and then just used 5Ø times. The kind of BASIC that you have on your CPC464 computer is *interpreted* BASIC which means that each instruction is worked out as the computer comes to it. If that means finding the address of the PRINT subroutine 5Ø times, so be it. The alternative is a scheme called *compiling*, in which the whole program is converted to efficient machine code before it is run. Compiling is done using another program, called a compiler. At present, I don't know of any BASIC compiler for the CPC464 computer, but there may be one available by the time you read this! Even at the time of writing, however, there are compilers for other languages, notably the very useful and efficient language which is called 'C', and the very popular language Pascal.

Chapter Three
The Miracle Microprocessor

In this chapter, we'll start to get to grips with the Z80 microprocessor of the CPC464 computer. 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 (ports), so that what the microprocessor does will control what the rest of the computer does. Its design also decides how much memory can be used at any given time.

The MPU itself consists of a set of memory stores for numbers, but with a lot of organisation added. By means of circuits that are very aptly called 'gates', the way in which the electrical signals for bytes are shared between different parts of the MPU's own memory can be controlled. It is these sharing actions that constitute the addition, subtraction, logic and other actions of the MPU. Each of the actions is programmed. Nothing will happen unless an instruction byte is present in the form of a 1 or a Ø signal at each of the 8 data terminals, and these bytes are used to control the gates inside the MPU. What makes the whole 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 speed that has been chosen as standard for the clock of the CPC464 computers is very fast indeed. You may be used to clocks that tick once a second, but the 'clock' of the CPC464 computer ticks at four *million* times each second. This doesn't mean that it can carry out four million instructions in each second, because some instructions need many clock ticks to carry out, but it does mean that things happen fast! They have to, because the MPU operates in sequence. It can do only one thing at a time, and so each operation has to be carried out quickly. In one operation, for example, one tick of the clock may be required to feed an instruction byte to the MPU, and another to feed a data byte. The action which is needed may then need another two or more ticks, so that a complete instruction may require four or more clock ticks.

Machine code

A program for the MPU, as we have seen, consists of number codes, each being a number between Ø and 255 (a single byte number). Some of these numbers may be instruction bytes which cause the MPU to do something. Others may be data bytes, which are numbers to add, or store or shift, or which may be ASCII codes for letters. The MPU can't tell which is which – it simply does as it is instructed. It's up to the programmer to sort out the numbers and put them into the correct order. They then have to be stored in this correct order in the memory.

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 the computer or after completing another instruction, is taken as being a new instruction byte. This means a byte which will make the MPU do something. Now many of the Z80 instructions consist of just one byte, and need no data. Others may be followed by one or two bytes of data,and some instructions need two bytes (or more) rather than one, and are also followed by data. When the MPU reads an instruction byte, then it analyses the instruction number to find if the instruction is one that has to be followed by one or more other bytes. If, for example, the instruction byte is one that has to be followed by two data bytes, then when the MPU analyses the first byte, it will treat the next two bytes that are fed to it as being the data bytes for that instruction. This action of the MPU is completely automatic, and is built into the MPU. The snag is that the machine code programmer must work to *exactly* the same rules, and get the program right. 100% correct is just about good enough! If you feed a microprocessor with an instruction byte when it expects a data byte or with a data byte when it expects an instruction byte, then you'll have trouble. Trouble nearly always means an endless loop, which causes the screen to 'freeze' as it is, and the keys to have no effect. Even the CTRL SHIFT ESC keys may not be able to break the CPC464 computer out of such a loop, and the only remedy then will be to switch off. You will generally lose whatever program you had in store, so that it's vitally important to save any machine code program, or a BASIC program that causes machine code actions (by using POKE) on tape before you use it.

What I want to stress at this point is that machine code programming is tedious. It isn't necessarily difficult – you are drawing up a set of simple instructions for a simple machine – but it's often difficult for you to remember just how much detail is needed. When you program in BASIC, the machine's error messages will keep you right, and help to detect mistakes. When you use machine code, you're on your own, and you have to sort out your own mistakes. In this respect, a program called an *assembler* helps considerably. We'll look at that point again later in Chapter 8. In the meantime, the best way to learn about machine code is to write it, use it, and make your own mistakes. We'll start looking at how this is done shortly, but we have to begin with the 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 code is a way of representing the 1s and Øs in a byte, it will consist of numbers between Ø and 255 when we write it in our normal scale of ten (denary scale). The program is useless until it is fed into the memory of the CPC464 computer, because the MPU is a fast device, and the only way of feeding it with bytes as fast as it can use them is by storing the bytes in the memory, and letting the MPU help itself to them in order. You can't possibly type numbers fast enough to satisfy the MPU, and even methods like tape or disk are just not fast enough.

Getting bytes into the memory, then, is an essential part of making a machine code program work, and we shall look at several methods in more detail later on. At one time, simple and *very short* programs would be put into a memory by the most primitive possible method, using eight switches. Each switch could be set to give a 1 or Ø electrical output, and a button could be pressed to cause the memory to store the number that the switches represented, and then select the next memory address. Programming like this is just too tedious, though, and working with binary numbers of 1s and Øs soon makes you cross-eyed. Now that we have computers, it makes sense to use the computer itself to put numbers into memory, and an equally obvious step is to use a more convenient number scale.

Just what is a more convenient number scale is a matter that depends on how you enter the numbers and how much machine code programming you do. The CPC464 computer contains subroutines which convert the binary numbers in its memory to the form of denary numbers to print on the screen, and will also carry out the reverse action. When you use PEEK, the address that you want can be written in denary, and the result of the PEEK will be a number in denary, between Ø and 255. When you use POKE, similarly, you can type both the address number and the byte to be poked in denary. If you never want to use machine code for anything other than a few short subroutines, then this provision is perfectly adequate.

Serious machine code programmers, however, find the use of denary numbers anything but convenient. A denary number for a byte may be one figure (like 4) or two (like 17) or three (like 143). A much more convenient code is the one which is called *hex* (short for hexadecimal) code. All single-byte numbers can be represented by just two hex digits. In addition to this, serious machine code programmers write their programs in what is called *assembly language*. This uses command words which are shortened versions of the names of commands to the MPU. Programs that are called 'assemblers' then convert these command words into the correct binary codes. Practically all assemblers show these codes on the screen in hex form rather than in denary. In addition, when you type data numbers in assembly language, you will probably have to make use of hex code. Since the CPC464 computer also contains routines for working in hex, it seems sensible to learn how to use this facility. Not only does it make it easier for you to make progress in machine code, it allows you to make effective use of some excellent software for developing machine code, such as the HiSoft DEVPAC, which we'll look at later.

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, half of a byte, will represent numbers which lie in the range Ø to 15 in our ordinary number scale. This is the range of one hex digit (Figure 3.1). Since we don't have symbols for digits higher than 9, we have to use the letters A,B,C,D,E, and F to supplement the digits Ø to 9 in the hex scale. The advantage is that a byte can be represented by a two-digit number, and a complete address by a four-digit number. The number codes that are used as instructions have been designed in hex code, so that we can see much better how commands are related. For example, we may find that a set of related commands all start with the same digit when they are written in hex. In denary, this relationship would not appear. In addition, it's much easier to write down the binary number which the computer actually uses when you see the hex version. Converting between binary and hex is much simpler than converting between binary and denary. The use of the CPC464 computer assembler and monitor programs, such as DEVPAC, demand familiarity with hex, and books of information on the Z80 MPU will all be written assuming that you know hex. Sounds as if we ought to make a start on it!

Hex	Denary	Hex	Denary
Ø	Ø	B	11
1	1	C	12
2	2	D	13
3	3	E	14
4	4	F	15
5	5	then	
6	6	1Ø	16
7	7	11	17
8	8	to	
9	9	2Ø	32
A	1Ø	21	23
		etc.	

Figure 3.1. Hex and denary digits.

The hex scale

The hexadecimal scale consists of sixteen digits, starting as usual with ∅ and going up in the usual way to 9. The next figure is not 1∅, however, because this would mean one sixteen and no units, and since we aren't provided with symbols for digits beyond 9, we use the letters A to F. The number that we write as 1∅ (ten) in denary is written as ∅A in hex, eleven as ∅B, twelve as ∅C and so on up to fifteen, which is ∅F. 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 number that follows ∅F is 1∅, sixteen in denary, and the scale then repeats to 1F, thirty-one, which is followed by 2∅. The maximum size of byte, 255 in denary, is FF in hex. When we write hex numbers, it's usual to mark them in some way so that you don't confuse them with denary numbers. There's not much chance of confusing a number like 3E with a denary numbers, but a number like 26 might be hex or denary. The convention that is followed by many Z80 programmers is to use a capital H to mark a hex number, with the sign placed *after* the number. For example, the number 47H means hex 47, but plain 47 would mean denary forty-seven. Another method is to use the hash mark *before* the number, so that #47 would mean the same as 47H. When you write hex numbers for a Z80 program, it's advisable to follow one of these conventions. The use of the hash mark is, for example, the convention that is followed by the DEVPAC assembler for CPC464 computers. The CPC464 computer itself, however, uses a different form when you want to poke hex numbers. The CPC464 computer method is to place & *before* the number, so that POKE &8∅∅C,&A would poke the hex address of #8∅∅C with the hex number #∅A. If you are working with PEEK and POKE instructions, there's usually little advantage of working in hex like this, particularly since it involves using one extra character for each number. When data bytes are being read in a loop, though, this method can be useful because it allows you to keep data in hex, which is easier to check than denary. The CPC464 will also accept &H as well as & to mark a hex number.

Now the great value of hex code is how closely it corresponds to binary code. If you look at the hex-binary table of Figure 3.2, you can see that #9 is 1∅∅1 in binary and #F is 1111. The hex number #9F is therefore just 1∅∅11111 in binary – you simply write down the binary digits that correspond to the hex digits. Taking another example, the hex byte #B8 is 1∅111∅∅∅, because #B is 1∅11 and #8 is 1∅∅∅. The conversion in the opposite direction is just as easy – you group the binary digits in fours, starting at the least significant (right-hand) side of the number, and then convert each group into its corresponding hex digit. Figure 3.3 shows examples of the conversion in each direction so that you can see how easy it is.

The CPC464 computer is one of the better-designed computers in as much as it can convert between hex and denary by means of BASIC

Hex	Binary	Hex	Binary
∅	∅∅∅∅	8	1∅∅∅
1	∅∅∅1	9	1∅∅1
2	∅∅1∅	A	1∅1∅
3	∅∅11	B	1∅11
4	∅1∅∅	C	11∅∅
5	∅1∅1	D	11∅1
6	∅11∅	E	111∅
7	∅111	F	1111

Figure 3.2. Hex and binary digits.

Conversion: Hex to Binary

Example: 2CH .... 2H is ∅∅1∅ binary
CH is 11∅∅ binary
So 2CH is ∅∅1∅11∅∅ binary (data byte)

Example: 4A7FH .... 4H is ∅1∅∅ binary
AH is 1∅1∅ binary
7H is ∅111 binary
FH is 1111 binary
So 4A7FH is ∅1∅∅1∅1∅∅1111111 binary (an address)

Conversion: Binary to Hex

Example: ∅11∅1∅11 .... ∅11∅ is 6H
1∅11 is BH
So ∅11∅1∅11 is 6BH

Example: 1∅11∅1∅∅1∅∅1∅ ..... note that this is not a complete number of bytes.

Group into fours, starting with lsb:
∅∅1∅ is 2H
1∅∅1 is 9H
∅11∅ is 6H and
the remaining 1 is 1, making 1692H

Figure 3.3. Converting between hex and binary.

commands. If you want to find what a denary number looks like in hex, just use HEX$(number). For example, ?HEX$(46) will give on the screen 2E, which is the hex for 46 denary. You can find binary equivalents by using BIN$ in a similar way. To find the denary equivalent of hex, use &. For example, ?&3F3A will give the result 16186, the correct denary equivalent. This works for numbers up to &7FFF, but odd-looking things happen after that. If you type, for example, ?&CØØ4, then this will give you −1638Ø on the screen. When you find that a conversion like this gives a negative number, then the correct answer that you want is obtained by adding 65536. In this case, for example, the correct answer is 65536−1638Ø, which equals 49156. The reason for this curious behaviour is the way that negative signs are represented in binary numbers, and we'll look at that later. If you want to use the negative numbers in POKE commands, however, the computer will make correct use of them. For example, POKE 49156,1Ø has the same effect as POKEØ1638Ø,1Ø. The provision of a built-in converter routine means that you don't have to bother with learning the rather tedious methods for converting between denary and hex. Just in case you want to do this when you are not near a friendly CPC464 computer, though, the methods are shown in Appendix B. The CPC464 computer, however, has no built-in assembler. An assembler would allow you to write machine code almost as easily as you write BASIC, by using command-words. Since an excellent assembler is available on tape, however, in the form of DEVPAC, this is not a catastrophic loss.

Assuming, which is reasonable, that you don't want to commit yourself to the cost of a full-scale assembler at this point, what do you do to create machine code programs? The answer is that you design your program in assembly language, which is by far the easiest way to design machine code programs, and then you convert into hex code. Converting means looking up in a set of tables, called the *instruction set*, the hex number that represents each instruction. Instruction sets are provided by the manufacturers of all microprocessors, and Zilog, who designed the Z80, provide one for this chip. Just to assist you, a quick-reference guide has been included in this book, in Appendix C. Don't refer to it at the moment – it'll put you off! Though the MPU uses only a few truly different commands, there are many slight variations on each command, and each variation has a separate code number. Until you understand why so many variations are needed, then, you'll only confuse yourself by looking at the tables now.

Negative numbers

Negative numbers are very important in machine code programs, particularly if you are working without an assembler. The reason is that you sometimes want the MPU to do the equivalent of a GOTO, perhaps jumping to a step which is thirty steps ahead of its present address. This sort of thing is usually programmed by supplying a data number which is the number of steps that you want to skip. If you want to jump back to a previous step, however, you will need to use a *negative* number for this data byte. This is very common, because it's the way that a loop is programmed in machine code. We need, therefore, to know how to write a negative number in hex. In addition, because the denary-hex conversions of the CPC464 computer can give negative denary numbers (as we have just seen), it's important to know when these will arise and why.

What makes it awkward is that there is no negative sign in hex arithmetic. There isn't one in binary either. The conversion of a number to its negative form is done by a method called complementing, and Figure 3.4 shows how this is done. At first sight, and very often at second, third, and fourth, it looks entirely crazy. When you are dealing with a single byte number, for example, the denary form of the number −1 is 255! You are using a large positive number to represent a small negative one! It begins to make more sense when you look at the numbers written in binary. The eight-bit numbers that can be regarded as negative all start with a 1 and the positive numbers all start with a Ø. The MPU can find out which is which just by testing the left-hand bit, the most significant bit.

```
Binary number...................ØØ11Ø11Ø   Denary 36
inverted..............................11ØØ1ØØ1
add 1.................................11ØØ1Ø1Ø   Denary −36


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

Figure 3.4. The two's complement, or negative form, of a binary number.

It's a simple method, which the machine can use efficiently, but it does have disadvantages for mere humans. One of these disadvantages is that the digits of a negative number are not the same as those of a positive number. For example, in denary, −4Ø uses the same digits as +4Ø. In hex, −4Ø becomes #D8 and +4Ø becomes #28. The denary number −85 becomes #AB and +85 becomes #55. It's not at all obvious that one is the negative form of the other. The second disadvantage is that humans cannot distinguish between a single byte number which is intended to be negative and one which is just a byte greater than 127. For example, does #9F mean 159 or does it mean −97? The short answer is that the human operator doesn't have to worry. The microprocessor will use the number correctly no matter how we happen to think of it. The snag is that we have to know what this correct use is in each case. Throughout this book, and in others that deal with machine code programming, you will see the words 'signed' and 'unsigned' used. A signed number is one that may be negative or positive. For a single byte number, values of Ø to #7F are positive, and values of #8Ø to #FF are negative. This corresponds to denary numbers Ø to 127 for positive values and 128 to 255 for negative. Unsigned numbers are always taken as positive. If you find the number #9C described as signed, then, you know it's treated as a negative number (it's more than #8Ø). If it's described as unsigned, then it's positive, and its value is obtained simply by converting. The snag here is that when we make use of HEX$ and the & conversion methods, they will not deal with signs in single bytes. If, for example, you type: ?HEX$(−5), you get FFFB rather than FB, because HEX$ works with hex numbers of at least four digits. Similarly, if you use ?&FB, you get the number 251 rather than −5. How do we convert a signed single-byte hex number into denary, when the CPC464 computer won't accept single-byte negative numbers for conversion? Simple, just type ?&XX−256, where XX is the hex number. For example, ?&FF−256 will give you −1, which is the correct value for treated as a negative number. For conversions with HEX$, simply take the two digits on the right-hand side.

Light relief

Just to take a break from all this arithmetic, let's take another look at screen displays on the CPC464 computer. We saw from Chapter 1 that each part of the screen can be controlled by whatever is stored in part of the memory which is called the 'video memory'. This video memory starts at address 49152, which is #CØØØ in hex, and it uses 16K of the top end of RAM no matter which screen mode you are using. It is possible to shift the screen memory so as to use other addresses, but this is something that you aren't likely to do just at the moment. Since the screen memory is part of the ordinary RAM, we can peek and poke the video memory by using the ordinary PEEK and POKE commands. When you switch the computer on, the screen mode that is set is Mode 1, which uses text of 4Ø characters wide by 25 lines deep, and graphics of 32Ø dots across by 2ØØ dots vertically. It's time to take a closer look at all this, because you can produce a lot of interesting effects by making use of the screen addresses.

To start with, the way that the screen memory is used is by no means straightforward. Each byte of memory in Mode 1 is used to represent four pixels. The bits of each byte are numbered, starting with Ø for the least significant bit, and going up to 7 for the most significant bit. For each pixel, one bit is taken from each half of a byte. Referring to Figure 3.5, the left-hand pixel uses bits 3 and 7, the next one bits 2 and 6, then 1 and 5, and the right-hand pixel uses bits Ø and 4. If you select one of these pairs of bits, then

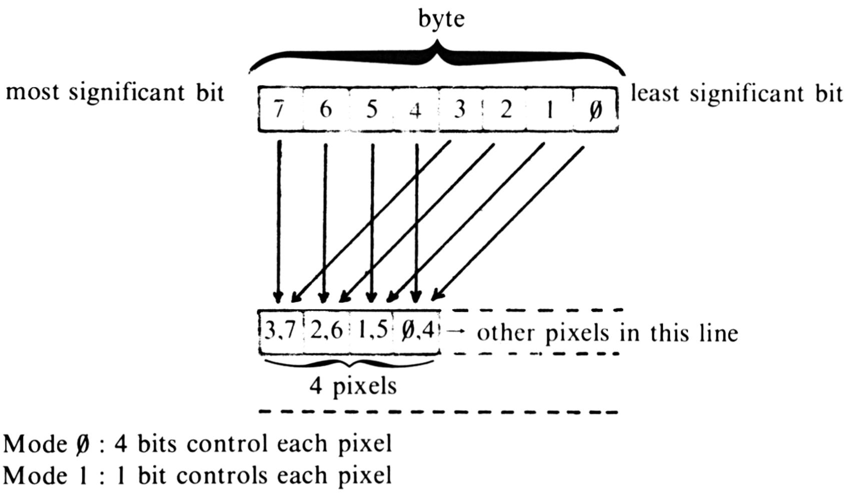

byte
most significant bit
least significant bit
7 6 5 4 3 2 1 Ø
3,7 2,6 1,5 Ø,4
other pixels in this line
4 pixels
Mode Ø : 4 bits control each pixel
Mode 1 : 1 bit controls each pixel

Figure 3.5. How each byte of screen memory controls the pixels on the screen. In Mode 1, one byte controls four pixels.

what you poke into these positions decides the position and colour of one dot on the screen. Suppose, for example, that you choose address #D8ØØ, and you take the left-hand pixel, which uses bits 3 and 7. Now two bits can be used in binary code to represent numbers from Ø to 3, and so we can poke into these bits numbers which carry the colour codes Ø to 3. These are the 'inkpot' numbers Ø to 3, and you can decide which ink colours they correspond to by using the INK command in BASIC. If this is all new to you, then it's time that you read my book *Amstrad Computing*! If, for example, you want to use inkpot 2, which is coloured cyan when no INK command has been used, then you have to remember that 2 in binary is 1Ø. To achieve this, we poke 1 into bit 3 and Ø into bit 7. Looking at the complete byte, this corresponds to the number 8. By using POKE &D8ØØ,8, then, we should create a cyan dot on the screen. Use CTRL SHIFT ESC first to make sure that everything is cleared first. You'll see the cyan dot appear on the top left-hand side of the screen. Want a different colour? Try POKE &D8ØØ,128 to get number Ø1, inkpot 1. Then try POKE &D8ØØ,136 to get inkpot 3. Why? Well 3 in binary is 11, and since we are using bits 3 and 7, this corresponds to denary 8 plus 128, giving 136. Figure 3.6 shows why in detail.

Suppose we try to brighten more pixels, so that the effects are more visible. If, for example, we want to make four pixels in one byte yellow, then we want to use inkpot 1, Ø1 in binary. Of the pairs of bytes, then, numbers 3,2,1 and Ø will be zero, and bits 7, 6, 5 and 4 will be 1. This makes the whole byte 1111ØØØØ, which in denary is 128+64+32+16=24Ø. Try POKE &D8ØØ,24Ø ,and you'll see that the result is a yellow dash, with all four pixels lit in yellow. Could we make two pixels yellow and two cyan? Yellow is

Bit 3	Bit 7	Binary 2-bit	Byte value in denary	Colour & pot number
∅	∅	∅∅ = ∅	∅	Background blue
∅	1	∅1 = 1	1∅∅∅∅∅∅∅ = 128	Yellow (pot 1)
1	∅	1∅ = 2	∅∅∅∅1∅∅∅ = 8	Cyan (pot 2)
1	1	11 = 3	1∅∅∅1∅∅∅ = 136	Red (pot 3)

Figure 3.6. In Mode 1, with two bits controlling each pixel, the arrangement of these two bits controls colour.

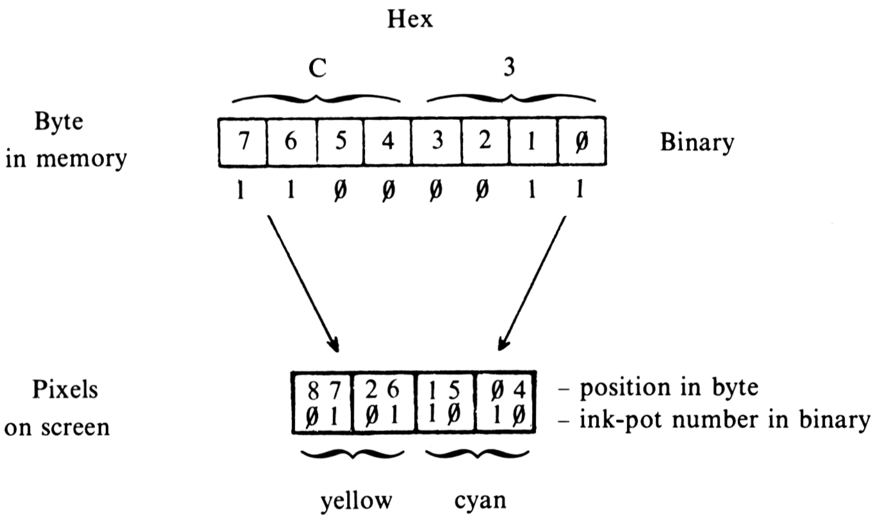


Figure 3.7. How the number &C3 can produce a two-colour bar shape.

inkpot 1 and cyan is inkpot 2. If we want the two left-hand pixels to be yellow, then they must each carry the bits ∅1. This means that bits 3 and 2 must be ∅ and bits 7 and 6 must be 1. To make the right-hand pixels use the cyan inkpot, number 2, the bits 1 and ∅ must be 1 and bits 5 and 4 must be zero. Figure 3.7 shows how this works out, giving the number &C3. To see it on screen, try POKE &D8∅∅,&C3.

Let's see how we can make use of all this. Suppose that we wanted to move a bar down the screen. One way might be to create the bar on the top line, hold it there for a time, then wipe it and create it all over again on the second screen line. Since we have 25 lines per screen, with address numbers #5∅ apart, this shouldn't be too difficult. The program of Figure 3.8 shows how it can be done. The program starts with MODE 1 to clear the screen and make

```
10 MODE 1
20 J=&C028:FOR N=0 TO 24
30 POKE J,&C3
40 FOR X=1 TO 150:NEXT
50 POKE J,0
70 J=J+&50
80 NEXT
```

Figure 3.8. Animating the bar shape – a first attempt.

---

2 pixels in mode Ø
4 pixels in mode 1
8 pixels in mode 2

Left — Right

	CØØØ	CØØ1	CØØ2	– – –	CØ4D	CØ4E	CØ4F
1st	C8ØØ	C8Ø1	C8Ø2	– – –	C84D	C84E	C84F
row of	DØØØ	DØØ1	DØØ2	– – –	DØ4D	DØ4E	DØ4F
char-	D8ØØ	D8Ø1	D8Ø2	– – –	D84D	D84E	D84F
acters	EØØØ	EØØ1	EØØ2	– – –	EØ4D	EØ4E	EØ4F
8 lines	E8ØØ	E8Ø1	E8Ø2	– – –	E84D	E8HE	E84E
deep	FØØØ	FØØ1	FØØ2	– – –	FØ4D	FØ4E	FØ4E
	F8ØØ	F8Ø1	F8Ø2	– – –	F84D	F84E	F84F

(The next line now starts with CØ5Ø)

	CØ5Ø	CØ51	CØ52	– – –	CØ9D	CØ9E	CØ9F
2nd	C85Ø	C851	C852	– – –	C89D	C89E	C89F
row of	DØ5Ø	DØ51	DØ52	– – –	DØ9D	DØ9E	DØ9F
char-	D85Ø	D851	D852	– – –	D89D	D89E	D89F
acters	EØ5Ø	EØ51	EØ52	– – –	EØ9D	EØ9E	EØ9F
	E85Ø	E851	E852	– – –	E89D	E89E	E89F
	FØ5Ø	FØ51	FØ52	– – –	FØ9D	FØ9E	FØ9F
	F85Ø	F851	F852	– – –	F89D	F89E	F89F

(The next line now starts with CØAØ)

Difference between lines in a character = #8ØØ
Difference between rows in a character = #5Ø

Last row ends with FF8Ø FF81 ... FFCF. Bytes C7DØ to C7FF, up to CFDØ to CFFF are *unused*.

Figure 3.9. The arrangement of the screen addresses. Remember that each byte can control 2, 4, or 8 pixels, depending on which Mode is chosen.

sure that the addresses correspond to the correct positions. The reason for this is that the CPC464 scrolls its screen by adding a 'displacement' number to each address. When this is in action, the address that you select will be in a different vertical position, certainly not where you expect it to be. Choosing a new MODE makes any displacement number equal to zero, and avoids these complications. We then select a starting address of &C028, which is the (hex) address of the middle of the top line. We then make a loop which counts from 0 to 24 (the number of lines), and which pokes the number &C3 each time. This gives the bar which is half-yellow and half-cyan. By adding &50 each time, we select the top line on the next character row. A time delay ensures that the movement is not too fast to follow. The result is a rather jerky movement of the bar down the centre of the screen.

Now could we make the movement rather less jerky? In the example, we have moved the bar from the top line of one set of eight to the top line of the next set. That's a movement of eight screen lines each time, which is bound to look jerky. We will get the effect of smoother movement if we make the distance less, from one line to the next. Now because of the way that the RAM addresses represent screen positions, this isn't quite so easy. Figure 3.9 shows how the addresses are arranged. The top line of the screen uses addresses &C000 to &C04F. The next line, however, uses addresses &C800 to &C84F. This is because the memory is allocated in blocks of &800, meaning that the difference between one line and the next *in one group* is always &800. The next group, however, has its first line address following where the previous first line ended. The top line ended at address &C04F, and the first address of the first line of the second group starts at &C050. Each 'group' corresponds to a 'text line', with 25 text lines per screen. If we want to make a bar appear to move smoothly down the screen, then, we have to change the POKE address by &800 for eight lines, then go back and add &50 to the original number before repeating this. The program in Figure 3.10 does what is needed. The outer loop of ROW=0 to 24 ensures that we deal with 25 groups of lines. The inner loop uses JJ, and has eight lines. The number &800 is added to JJ to make the address change from one line to the next, and then &50 is added to J to get the change from one group to the

```
10 MODE 1:J=&C028
20 FOR ROW=0 TO 24:JJ=J
30 FOR N=0 TO 7
40 POKE JJ,&CE
50 FOR X=1 TO 20:NEXT X
60 POKE JJ,0
70 JJ=JJ+&800:NEXT N
80 J=J+&50:NEXT ROW
```

Figure 3.10. An improved animation which uses the addresses in two loops.

next. Try it out, and see for yourself how it works. You should spend some time with this until you are really familiar with how the screen addressing works. You won't need to use direct screen addressing unless you are designing fast-action games in machine code. Since fast-action games are just what many users need machine code for, however, this might just be really essential information for you! Note that some address numbers are not used at all. If you use them in a program, you will see nothing appear on the screen.

Chapter Four
Z80 Details

Registers - PC and Accumulator

A microprocessor consists of sets of memories, of a rather different type to ROM or RAM, which are called *registers*. These registers can be connected to each other and also to the pins on the body of the MPU by the circuits that are called *gates*. All of the actions of the MPU are carried out by making these internal connections. In this chapter, we shall look at some of the most important registers of the Z80 and how they are used. A good starting point is the register which is called the PC – short for *program counter*.

No, it doesn't count programs – what is does is to count the steps in a program. The PC is a sixteen-bit (two-byte) register which can store a full-sized address number, up to #FFFF (65535 denary). Its purpose is to store an address number, and the number that is stored in the PC will be incremented (increased by 1) each time an instruction is completed, or when another byte is needed. For example, if the PC holds the address #1F3A, and this address contains an instruction byte, then the PC will increment to #1F3B whenever the MPU is ready for another byte. The next byte will then be read from this new address. When the computer is switched on, the PC address must be set to where the first instruction of the ROM happens to be.

What makes the PC so important is that it's the automatic way by which the memory is used both for reading and writing. When the PC contains an address number, the electrical signals that correspond to the ∅s and 1s of that address appear on a set of connections, collectively called the 'address bus', which link the MPU to all of the memory, both RAM and ROM. The number that is stored in the PC will select one byte in the memory, the byte whose address number it happens to be. At the start of a read operation, the MPU will send out a signal called the read signal on another line, and this will cause the memory to connect up the portion that has been selected to another set of lines, the data bus. The signals on the data bus then correspond to the pattern of ∅s and 1s that is stored in the byte of memory that has been selected by the address in the PC. Reading means that these signals are copied into a register within the MPU. Each time the number in the PC changes, another byte of memory is selected, so that this is the way by which the MPU can keep itself fed with bytes. When the MPU is ready for another byte, the PC increments, and another read signal is sent out. Similarly, for a write operation, the PC holds the address number, the signals on the address bus select the correct byte in the memory, and another register in the Z80 shares its electrical signals with the lines of the data bus so that the memory byte is forced to copy. Having done this, the number in the PC is incremented ready for the next action.

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 of a microprocessor is the main 'doing' register of the MPU. This means that you would normally use it to store any number that you wanted to transfer somewhere else, or add to or carry out any other operation upon. The name of accumulator comes from the way in which this register operates. If you have a number stored in the accumulator, and you add another number to it, then the result is also stored in the accumulator. The nearest equivalent in BASIC is using a variable A, and writing the line:

```
A=A+N
```

where N is a number variable. The result of this BASIC line is to add N to the old value of A, and make A equal this new value. The old value of A is then lost. The accumulator acts in the same way, with the important difference that the Z80 accumulator can't store a number greater than 255 (denary).

The Z80 has one main accumulator register, labelled A. The importance of this is that it is used much more than the other registers, because so many actions can be carried out more quickly, more conveniently, or perhaps only, in the accumulator. When we read a byte from the memory, we usually place it in the accumulator. When we carry out any arithmetic or logic action, it will normally be done in the accumulator and the result will also be stored in the accumulator. Unlike earlier designs of microprocessors, the Z80 has a large number of other registers, several of which can be used in much the same way as the accumulator, but none of them has quite such a large range of possible actions.

Addressing methods

When we program only in BASIC, we don't have to worry about memory addresses at all unless we are using PEEK or POKE. The task of finding where bytes are stored is dealt with by the operating system of the machine. When a variable is allocated a value in a BASIC program, as, for example, by a line like:

```
1Ø N%=12
```

we never have to worry about where the number 12 is stored, or in what form. Similarly, when we add the line:

```
20 K%=N%
```

we don't have to worry about where the value of N% was stored or where we will store the value of K%. Remembering our comparison with wall-building, though, 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 way in which we obtain a number, or find a place to store it, is called the 'addressing method'. What makes the choice of addressing method particularly important is that a different code number is needed for *each* different addressing method for *each* command. This means that each command exists in several different versions, with a different code for each addressing method. A list of all the Z80 addressing methods at this stage would be rather baffling, and for that reason has been consigned to Appendix D. What we shall do here is to look at some examples of selected addressing methods and the way that we write them in assembly language.

Assembly language

Trying to write down machine code directly as a set of numbers is a very difficult process which is liable to errors from beginning to end. The most useful way of starting to write a program is to write it in a set of steps in what is called *assembly language* (or *assembler language*). This is a set of abbreviated command words, called *mnemonics*, along with numbers which are the data or address numbers. The numbers can be in hex or in denary, provided they are supplied to the computer in the correct form. Each line of an assembly language program indicates one microprocessor action, and this set of instructions is later 'assembled' into machine code, hence the name. In this way, the machine carries out all of the tedious 'looking-up' actions which are so boring for a human to do. The human part then consists of planning the program and writing it in this assembly language. There is a different assembly language for each different type of microprocessor.

The aim of each line of an assembly language program is to specify the action and the data or address that is needed to carry out that action, just as when we make use of TAB in BASIC we need to complete the command with a number. Like BASIC, assembly language has to be written in the correct way (correct syntax). The part of the assembly language that specifies what is to be 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 later.

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

```
LD A,#12
```

The operator is LD, a shortened version of LOAD, meaning that a byte is to be copied from one place to another. The operand consists of two parts, A and #12. The A means that the accumulator register A is to be loaded with a byte. The other part of the operand is #12, of which the # means that this is 12 hexadecimal, rather than twelve denary. The use of a single-byte number like this shows that the addressing method that is to be employed is a method called 'immediate addressing', and that the single-byte number is to be loaded.

The whole line, then, should have the effect of placing the number #12 into the accumulator register A. It is the equivalent in machine code terms of the BASIC instruction:

```
A=&12
```

if you could imagine that the memory which held the number was inside the microprocessor rather than being part of the RAM memory, and that it was labelled A.

A command such as LD A,#12 is said to use *immediate addressing*, because the byte which is loaded into the accumulator must be stored in the memory byte whose address *immediately follows* that of the instruction byte. There is one code number for the LD A, part of the whole instruction, and this byte is #3E, so that the hex sequence in memory of 3E 12 will represent the entire command LD A,#12. It's a lot easier to remember what LD A,#12 means than to interpret the numbers #3E and #12 stored in the memory, however, which is why we use assembly language as much as possible.

Immediate addressing like this can be convenient, but it ties you down to the use of one definite number. It's rather like programming in BASIC:

```
N=4*12+3
```

rather than

```
N=A*B+C
```

In the first example, N can never be anything else but 51, and we might just as well have written: N=51. The second example is very much more flexible, and the value of N depends on what values 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 must change them, by changing the program. When the program is held in ROM, however, no change is possible – and that's just one reason for needing other addressing methods. One of these other methods is called *extended addressing* – alternative names are direct or absolute addressing.

Extended addressing uses a complete two-byte address as its operand. This creates a lot of work for the Z80, because when it has read the code for the operator, it will then have to read two more bytes to find the memory address at which the data is stored. It will then have to place this address into the PC, read in the data byte, carry out the operation, and then restore the next correct address into the PC. Figure 4.1 shows in diagram form what has to be done. An extended-addressed operation is therefore a lot slower to carry out than an immediate one, but since any byte may be stored at the address which is specified, it's easy to alter the data if we need to. We can even make the program alter the data for itself!

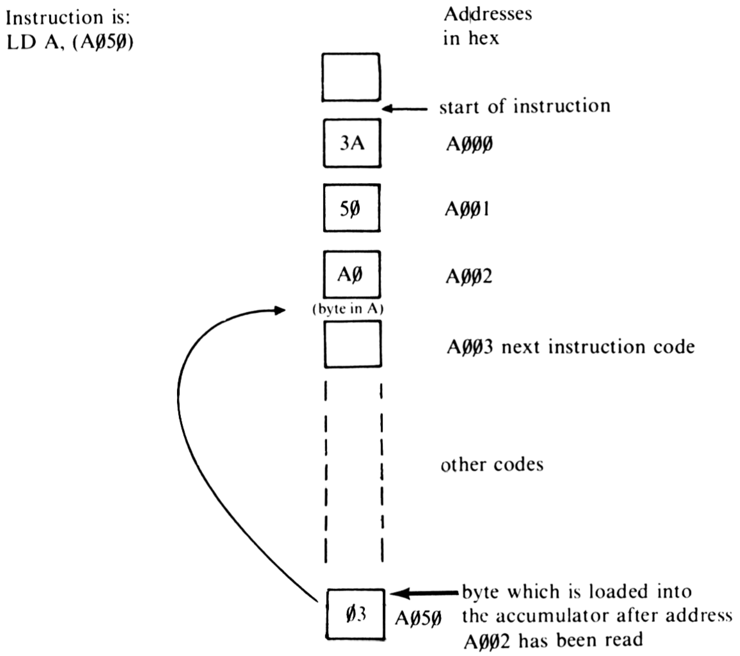


Figure 4.1. How the extended (absolute) addressing method works.

Suppose, to take an example of straightforward extended addressing, that we have the instruction:

```
LD A,(#7FFE)
```

In this slice of assembly language, the operator is LD, meaning that a load is to be carried out, and the operand consists of A, the accumulator, and the address #7FFE. The order of these two parts of the operand is important. By placing A before (#7FFE), we are commanding the computer to load what is in address #7FFE *into* the accumulator A. What you have to remember is that what is put into the accumulator A is not #7FFE, which is a two-byte address, but the *data byte* which is stored in memory at this address. The effect of the complete instruction, then, is to place a copy of the byte which is stored at #7FFE into the accumulator A of the Z80. When the instruction has been completed, the address #7FFE will still hold its own copy of the byte, because reading a memory does not change the content of the memory in any way. The content of the accumulator will have changed, however, because it must now be the byte that was held in address #7FFE. In a listing of Z80 instructions, this type of addressing would appear as LD A,(NN) (or, as shown in Appendix C as LD A,(ØØØØ)). The use of brackets around the address is not just a matter of the syntax. The brackets mean 'contents of', and wherever you see brackets used in Z80 assembly language they have this meaning. If you wanted to say what the instructions LD A,(#7FFE) meant, then, you would say 'load the accumulator with the contents of address 7FFE hex'.

We can also use the extended addressing method in a command which will store a byte into the memory. This is where the order of writing the parts of the operand becomes important. The command:

```
LD (#7FFF),A
```

means that the byte that is stored in the accumulator A is to be copied to memory at address #7FFF. This action *does* change the content of this memory address, but the accumulator A will still hold the same byte after the instruction has been carried out. Once again, brackets have been placed around the address to mean 'contents of', and the order of address, comma, A shows the direction of copying. The order, as you can see from both of the preceding examples, is *always* destination, comma, source. With the exception of immediate addressing, most of the LD commands can be written so as to mean either direction of copying. Other microprocessors use the command ST (usually with the register name attached, like STA, STX) for the action of copying a byte from the accumulator to a memory address.

Register-indirect addressing

Immediate addressing and *extended addressing* (also called direct addressing) are both useful, but the Z80 also permits another type of addressing which is called *register-indirect addressing*. Indirect addressing means going to one address to find another. It sounds very awkward and mysterious, but it's no more difficult than calling at a travel agent to find the address of a restaurant. The register-indirect addressing method is given this name because a register (more accurately, a pair of registers) will be used to hold the address. This type of addressing is a speciality of the Z80 and is found on few other microprocessors. In the Z80, it depends on being able to combine certain pairs of eight-bit registers and use them as if each pair was one sixteen-bit register. This ability to use registers singly or in pairs, as you choose, is just one of the features which has made the Z80 such a popular and widely used microprocessor. The design principles of the Z80, unlike some other early types of microprocessors, have now been carried over into newer sixteen-bit micros. By learning Z80 machine code, then, you are preparing yourself for whatever comes along next!

There are no less than three sets of these pairs of registers in the Z80, and for convenience, they are labelled as HL, BC and DE. The single registers are labelled H, L, B, C, D, and E, but we can only put them together in the three groupings that are shown. Singly, the registers can be used as if they were spare accumulators, but with fewer actions. Of the three 'double' registers or register pairs, the HL pair is the most frequently used. We can load a complete sixteen-bit address into the HL pair of registers by using a command which is written in assembly language as:

```
LD HL,#7FFF
```

– taking an example. This means that the high-byte of the address, #7F in this example, will be held in the H register (the H should remind you of *H*igh) and the low-byte of #FF is stored in the L (for *L*ow) register. You can change either the high-byte or the low-byte, incidentally, by loading the H or L register independently. Having put this address into the register pair, we can then make use of the byte which is stored at this address of #7FFF by a command such as:

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

which means that the accumulator is to be loaded from the byte whose address is stored in HL. In our example, this means the byte which is stored at the address #7FFF. If we changed the address in the HL registers, we would, of course, load a different byte from the different address. We can equally easily store a byte from the accumulator to this address by reversing the order of the parts of the operand. For example, if we use:

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

then the byte in the accumulator is copied to address #7FFF, assuming that this is still the address that is stored in HL. Remember that the brackets mean 'contents of', and that the order of the parts of the operand is destination, source.

Now you might think that this is just a rather long-winded way of writing a command such as LD A,(#7FFF), but there is a rather important difference. Once an address number has been stored in the register pair HL, we can increment or decrement that address with a single-byte instruction. If, for example, we have loaded HL with the address #7FFF, then the instruction DEC HL will make the address number which is stored in HL equal to #7FFE, one less. If we now use LD A,(HL) again, then the load will this time be from the address #7FFE, not #7FFF as it was before. If a program requires a lot of bytes to be loaded from a consecutive set of addresses, then, this allows the action to be done *in a loop*, with just one LD A,(HL) instruction and one DEC HL instruction in the loop. You could, of course, just as easily use INC HL in the loop so that the address number that is held in HL is incremented rather than being decremented. You can, incidentally, increment or decrement H or L separately, and you can use the commands INC (HL) and DEC (HL) which will increment or decrement the *byte* which is stored at the address that is held in HL! That's going too far for the moment, however, so let's get our feet back on the ground.

PC-relative addressing

PC-relative addressing is one of the first and simplest addressing methods that was ever devised, and it is not used for many commands nowadays. Relative addressing means that the operand of an instruction can consist of one-byte only, and the address that is going to be used is found by adding this number (called the *displacement* or *offset*) to the 'current address', which is the number in the program counter. The number that you end up with as a result of this addition process is then used in the load or whatever operation you have specified. It's rather like the old-style Treasure Island maps which specify 'ten steps left, three forward, five right ...' and so on. You don't know where this will get you until you know where to start, but when relative addressing is used in a microprocessor, the starting place is *always* the address in the PC. The Z80 uses relative addressing only for one small group of commands, the jump-relative (JR) commands.

Each command of this type will use a single byte *signed* number, as an offset. The use of a single byte signed number means that we can jump to a new address which is up to 127 steps forward or 128 steps back from the present one. The jump-relative commands are the machine code equivalents of GOTO, but with the difference that they can be made to depend on a condition, like the accumulator containing zero. It's as if there was one single BASIC instruction which carried out the effect of:

```
IF A=Ø THEN GOTO....
```

The complete JR instruction consists of the operator JR, and an operand which can contain a condition and a displacement byte. If the condition part of the operand is omitted, then the jump will be unconditional – it will always take place. In machine code, only two bytes are used for each type of JR command. Using the instruction without an assembler requires a certain amount of care (and experience!). This is because the displacement is treated by the Z80 as a *signed* number, meaning that if the most significant bit is 1 (number 128 or more in denary), then the byte is negative. If the byte is negative, then the jump is backwards, to a lower address number than the number in the PC. In denary numbers, then, if the displacement is 127 or less, the jump is ahead; if the number is 128 or more, the jump is backwards. After a jump instruction, the new address in the PC will be incremented in the usual way, and normal action will resume at this new address. This again is like the action of GOTO in BASIC.

When we make use of PC-relative addressing, the tricky thing is calculating the size of the displacement byte. When you use the GENA3 assembler part of the DEVPAC to write your machine code programs, the displacement byte will be calculated for you by the assembler. If you assemble 'by hand', however, you have to calculate it for yourself. The calculation is done in this way:

(1) Write down the address at which the operator byte JR, will be stored. This is called the *source address*.
(2) Write down the address to which you want to program to jump. This is called the *destination address*.
(3) Subtract source address from destination address, and then subtract two from this answer. The result is the displacement in denary. If it is positive, use it directly. If it is negative, subtract from 256, and use the result.

Why subtract 2? It's because the displacement is always calculated from the address at which the *operator* byte is placed, but the jump can't take place until the operand has been read, because the operand contains the displacement byte. To read the operand, the PC must increment, and in addition, the PC will increment *again* at the end of the instruction. By subtracting 2, we allow for these two incrementing actions and obtain the correct displacement byte. Figure 4.2 shows some examples of both positive

Address (Denary)

(a)	source	41472	JR	←instruction here
	(S)	41473	. .	←displacement byte here (226)
		.		
		.		
		.		
		.		
	destination	417ØØ		←jump to here
	(D)			

(D)−(S) is 417ØØ–41472 = 228
Then subtract 2 to get 226 which is displacement byte

(b)	Source	4147Ø	JR	←instruction here
	(S)	41471		← displacement byte here

(Destination (D) is 41362)

41362–4147Ø = −1Ø8
Subtract 2 to get −11Ø which is displacement byte

Figure 4.2. PC-relative addressing using both positive and negative displacement bytes.

and negative displacements being calculated. It's often easier when you are writing down the bytes of a 'hand-assembled' program, just to count the number of bytes that lie between the source and the destination address. Remember that PC-relative addressing cannot cause a jump to more than 127 places forward or 128 places back from the PC address, because this is the complete range of a single signed byte.

Indexed addressing

Indexed addressing is a method which is particularly useful for some applications on the Z80. The principle is that a sixteen-bit register (one of two) is used to hold an address, the address of a byte in the memory. This address can be used directly, or it can be used as a *base address*. Use as a base address means that we can add a number to the address before using it. For example, suppose that we had the number #7ØØØ stored in an index register. If we like, we can use this to specify the use of the address #7ØØØ, or alternatively we can add a number to it. If we add 4, for example, then we could load from #7ØØ4 or store to this number. The indexed loading command for the accumulator of the Z80 takes the form:

LD A,(IX+d)   or   LD A,(IY+d)

The two index registers are labelled IX and IY. Each of these is a sixteen-bit register, which *cannot* be split into two eight-bit registers. The 'd' is the 'displacement' byte which is added to the address that is held in the index register. The load operation is then from this final address number which is obtained from the addition. The use of index registers is very convenient when a set of values (like a set of BASIC tokens, for example) is held as a table. If you want the 55th token in a table that starts at #3ØØØ, for example, you use LD A,(IX+54), assuming that the IX register had been loaded with #3ØØØ earlier. This would have been done by using the command LD IX,#3ØØØ. If you're wondering why we use IX+54 to get the 55th token, remember that address IX holds the first token, IX+1 holds the second and so on. Normally, the IX and IY registers, if they are to be used in a program, will be loaded early on in the program and the contents left unchanged. Note that this is different to the use of register-indirect addressing, such as LD A,(HL), because there is no provision for leaving HL unchanged and adding a number to the HL address only when it is used. The index registers are very extensively used in the ROM routines of your CPC464 computer, as you might imagine, but they don't feature much in the short programs in this book until we get to the topic of passing variable values in Chapter 7.

The other Z80 registers

We've mentioned a number of the Z80 registers already, and a quick reminder might be useful. The PC is the addressing register, which keeps a count of the address of each instruction byte and data byte in a program. It's the 'where are we now' register of the Z80. When a machine code program is to be run, this is done by placing the address of the first instruction byte of the program into the PC. The rest is automatic, so that providing the program has been correctly written, the MPU will take over. To run a machine code program, then, we need a command which will place the correct address into the PC, and we'll look at methods for doing this later.

The accumulator is the register in which most of the work is done. It's so important that we'll devote a large chunk of the next chapter to the actions that can be carried out in the accumulator. There are six other single-byte registers, which are labelled as B, C, D, E, H and L which can be used in the way that we use the accumulator itself, though none of them offers quite such a wide range of actions. In addition, as we have seen, these registers can be grouped as HL, BC and DE to store complete sixteen-bit address numbers. A limited number of sixteen-bit arithmetic operations are also possible in the HL register pair. Remarkably enough, no computer manufacturer has ever claimed that the Z80 is a sixteen-bit chip because of this!

We have also looked briefly at the index registers, IX and IY, but this does not exhaust the register count of the Z80. Another sixteen-bit register is labelled as SP, meaning *stack pointer*. As we shall see later, the 'stack' is a piece of memory which is used for temporary storage while a machine code program is running, and the stack pointer register is used to keep track of addresses in this 'stack'. Just to illustrate how this is used, have you ever wondered how a GOSUB in BASIC could go to a new line, but afterwards RETURN to the correct place? This happens because when the GOSUB is carried out, the correct address to return to is stored in the stack memory, and the stack pointer register is used to refer to this address. When the subroutine is over, then, the stack pointer indicates what address in the stack has to be used to get the correct return address back into the PC. That's a simplified version of what happens, and we'll look at the action in a lot more detail later in Chapter 7.

There are also two registers which we very seldom use in our programs, the interrupt (I) register and the refresh (R) register. Their uses are rather specialised. Unless you are likely to write programs which allow your computer to be controlled by external devices like light pens or sensors, the interrupt register is not likely to concern you. The R register is used by the Z80 to 'refresh' the RAM memory of the computer. Modern computers use a type of RAM memory which is called dynamic RAM. This costs much less than other types, but will lose its signals after a very short time (about one thousandth of a second!) unless it is activated many times per second by a 'refresh' signal. The Z80 provides such a signal automatically. Computers which use MPU's of older design have to provide for this refresh signal by other chips. The contents of the refresh register are sometimes used in machine code programs as the source of a random number. One important register remains, however: the *flag register*.

The flag register

The *flag register*, usually referred to as the F-register but sometimes called the status register, isn't really a register like the others. You can't do anything with the bits in this register apart from testing them, and they don't even fit together as a number. What the flag register is used for is as a sort of electronic notepad. Each bit in the register (there are eight of them) is used to record what happened at the previous step of the program. If the previous step was a subtraction that left the A register storing zero, then one of the bits in the flag register will go from value Ø to value 1 to bring this to the attention of the MPU. If you add a number to the number in an accumulator, and the result consists of nine bits instead of eight (Figure 4.3)

Number in accumulator	1Ø11Ø11Ø
Number added	11ØØØ1Ø1
Result	1Ø1111Ø11

This consists of nine bits, and the accumulator can hold only eight. The most significant bit is transferred to the carry flag of the status register.

Accumulator now holds Ø1111Ø11
Carry bit is set (equal to 1)

Figure 4.3. Why the carry bit is needed.

then another of the bits in the flag register is 'set', meaning that it goes from Ø to 1. If the most significant bit in a register goes from Ø to 1 (which might mean a negative number), then another of the flag bits is set. Each bit in the flag register, then, is used to keep a track of what has just happened. In particular, the flag register keeps track of what has just happened *in the accumulator*. What makes the flag register so important is that you can make branch commands depend on whether a flag bit is set (to 1) or reset (to Ø).

Figure 4.4 shows how the bits of the flag register of the Z80 are arranged. Of these bits, numbers Ø, 6 and 7 are the ones that we are most likely to use at

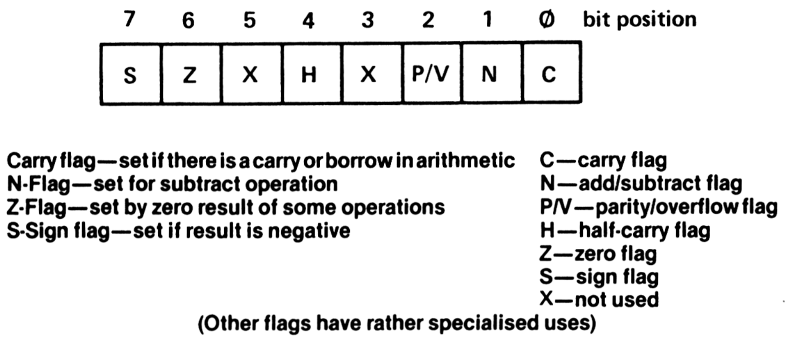

7 6 5 4 3 2 1 Ø bit position
S Z X H X P/V N C
Carry flag—set if there is a carry or borrow in arithmetic
N-Flag—set for subtract operation
Z-Flag—set by zero result of some operations
S-Sign flag—set if result is negative
C—carry flag
N—add/subtract flag
P/V—parity/overflow flag
H—half-carry flag
Z—zero flag
S—sign flag
X—not used
(Other flags have rather specialised uses)

Figure 4.4. The bits of the Flag, or Processor Status, register. Only three of these, S,Z, and C, are extensively used in most programs.

the start of a machine code career. The use of the others is rather more specialised than we need at the moment. Bit Ø is the carry flag. This is set (to 1) if a piece of addition has resulted in a carry from the most significant bit of a register. If there is no carry, the bit remains reset. When a subtraction is being carried out (or a similar operation like comparison), then this bit will be used to indicate if a 'borrow' has been needed. It can for some purposes be used as a ninth bit for the accumulator, particularly for shift and rotate operations in which the bits in a byte are all shifted by one place (Figure 4.5). The carry bit is used by all of the addition and subtraction operations of the Z80, even the operations that do not use the accumulator. This is something that you have to be careful about. Any operation that is carried out in the accumulator, apart from a load (either direction) will set flags, but many operations in other registers *do not*. This is very often a reason for using the accumulator in place of another register which appears to be equally suitable.

The zero flag is bit 6 of the flag register. It is set if the result of the previous operation was exactly zero, but will be reset (Ø) otherwise. It's a useful way of detecting equality of two bytes – subtract one from the other, and if the zero flag is set, then the two were equal. The CP (*C*om*P*are) action will set or

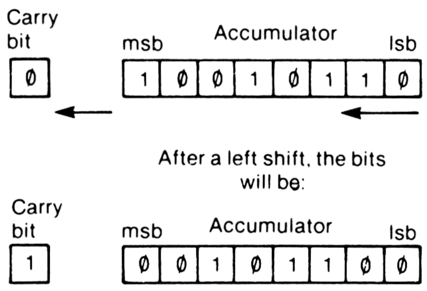


Figure 4.5. Using the carry bit in a shift operation, in which all the bits of a byte are shifted one place to the left.

reset this flag *without* actually carrying out the subtraction action. The CP instruction can be used only with the accumulator, so that it always results in flags being affected. The sign flag, number 7, is set if the number resulting in an action in a register has its most significant bit equal to 1. This is the type of number that might be a negative number if we are working with signed numbers. This bit is therefore used extensively when we are working with signed numbers.

Like most MPUs, the Z80 does not allow the programmer to work in any easy way with the contents of the flag register. You can set the carry flag by using the SCF instruction byte, and you can ensure that the carry flag is reset by using XOR A (which also zeros the accumulator). It is also possible to read the flag register contents into registers C, L or E by storing A and F on the stack and then reading into BC, HL or DE. Normally, you won't ever want to alter the contents of the flag register. You very seldom know or care about what is stored in the flag register, and its main importance to you is that it controls jumps. To make another comparison with BASIC, suppose that we had programmed:

```
100 IF A=0 THEN 300
```

which makes a 'jump' to line 300 if a variable A stores the value zero. When the program runs, we might not know at any particular instant what was stored in A, but we do know that the jump will take place if variable A stores zero. The machine code version of this action is:

```
JR Z,d
```

where JR is the operator, meaning jump-relative, and the operand consists of Z and d. Of these two, Z (it must be the *first* part of the operand) means that the Z-flag must be tested. The 'd' part indicates a displacement – you would write a single-byte number here. If the jump is taken (if the accumulator contains zero), then the displacement number will be added to the number in the PC to give the new address to which the program will jump. If Z=0, meaning that the accumulator does *not* store a zero, then the displacement will be ignored, and the PC will increment in its usual way. When the microprocessor carries out this action, then, the single byte of code that represents the JR Z part will cause the Z80 to check the Z-flag in the flag register.

One peculiarity of the Z80 is that only certain actions, mainly actions that affect the accumulator, will cause flags in the flag register to be affected. This takes a lot of getting used to if you have ever programmed some other types of microprocessors, particularly the 6502 or 6809. In particular, load and store operations *never* affect flags in any way, so that a flag which has been set before a load or store operation will still be set after that operation. This can at times be very useful. Suppose, for example, that we have a piece of assembly language which reads:

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

– then the flags are unaffected by the LD A,(DE) step. If the DEC A (decrement byte in the accumulator) step had caused the contents of the accumulator to become zero, then the zero flag would be set at this stage. This zero flag will not be affected by the LD A,(DE) step, even though this step places a new byte into the accumulator so that it no longer contains zero. The jump at the JR Z,disp step will therefore take place, even though the accumulator now contains a byte which is not zero. This can be very useful at times, because it can save having to repeat a loading step. It's something that you'll appreciate later if you become really hooked on machine code.

To sum up all of this information on registers, Figure 4.6 shows a map of all the Z80 registers. The registers are shown as main set and alternate set, and in each group, the registers are shown in pairs. The groupings HL, BC

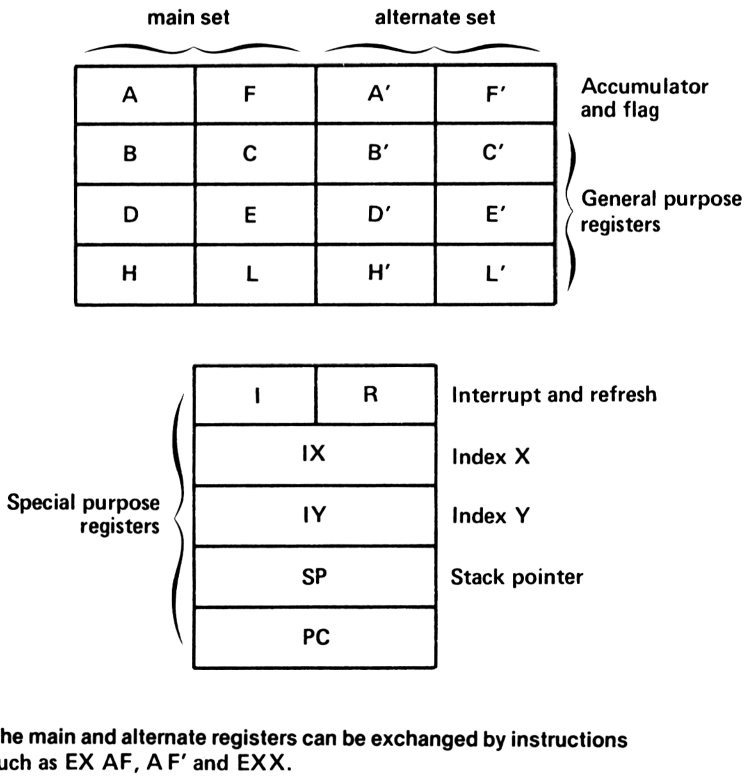

main set
alternate set
A
F
A'
F'
B
C
B'
C'
D
E
D'
E'
H
L
H'
L'
Accumulator and flag
General purpose registers
I
R
IX
IY
SP
PC
Interrupt and refresh
Index X
Index Y
Stack pointer
Special purpose registers
The main and alternate registers can be exchanged by instructions such as EX AF, A F' and EXX.

Figure 4.6. A 'map' of the Z80 registers.

and DE are already familiar to you, but you will see that the assumulator A and flag register F are also grouped as AF. This is because there are a few operations which treat these two registers as if they were one single sixteen-bit register. We'll come to that later. Meantime, a word about the alternate set. Normally, we make use of the main register set of the Z80, but each of the registers in the main set is duplicated in the alternate set. You can't use the main set registers and the alternate set registers at the same time, you have to switch to one or the other. Two commands are used for this, EX AF,AF′ and EXX. The command EX AF,AF′ swops the main set A and F registers with their alternates, but leaves other registers unchanged. If you have been using A and F, for example, then the command EX AF,AF′ allows you to use the other A and F, while the original A and F remain unaffected, storing whatever was placed in them before you made the exchange. Another EX AF,AF′ will swop again. The operating system of the CPC464 makes a lot of use of the alternate registers, so you should leave them strictly alone. If you must make use of them, you will have to follow very closely the advice which is given in Appendix XI of *The Concise Firmware Specification* (Amsoft Ref. SOFT 158). The other registers, B, C, D, E, H and L will be swopped by using the command EXX. In addition to these registers, we have the special purpose registers I,R, IX, IY, PC and SP. Apart from I and R, these are all sixteen-bit registers which are used for storing addresses.

Chapter Five
Register Actions

Accumulator actions

In this chapter, we're going to look in a lot more detail at what can be done in the registers of the Z80. By this time, you're probably getting impatient with having nothing that you can try for yourself on your own computer. That's because machine code isn't something that you can dip into in easy stages. There isn't much point in having an example to try, if everything about it is baffling, and that's just how it is until you know what you are doing. Slog on, you're nearly there!

Since the accumulator is the main single-byte register, we can list its actions and describe them in detail, knowing that whatever holds good for the accumulator of the Z80 will also be a useful guide for the other single-byte registers. Of all the accumulator actions, simple transfer of a byte is by far the most important. We don't, for example, normally carry out any form of arithmetic on ASCII code numbers, so that the main actions that we perform on these bytes are loading and storing. We load the accumulator with a byte copied from one memory address, and store it at another. Very few computer systems allow a byte to be moved directly from one address to another, so that the rather clumsy-looking method of loading from one address and storing to another is used almost exclusively. In assembly language, and using direct addressing, this would look like:

```
LD A,(SOURCE)
LD (DEST),A
```

The first line copies a byte in address 'SOURCE' into the accumulator. The same byte is then copied to address DEST in the next line. Remember the order of the items in the operand. I have used the words, SOURCE and DEST, in place of actual address numbers for two reasons. One is that it helps you to remember that these can be *any* address numbers that you want to use. The other is that this is just how we use assembly language, using words as 'labels' wherever possible rather than definite address numbers. If you use a number, you're stuck with it, but if you use a word, you can allocate a number to it only when you actually have to enter the program into the computer. This makes the design, planning, and alteration of a program a lot easier. It's like using variables in place of constants in BASIC. Suppose, for example, that you have a BASIC program that carries out a lot of VAT calculations. Do you put steps like X=A*.15 in? Not if you know what you're doing! What you should use is X=A*VT. This way, the variable VT carries the value of VAT. The two good reasons for this are that a variable takes up less memory space, and that it needs to be assigned just once. If the rate of VAT changes to .18, all you have to do is to alter a line that says VT=.15 to one that reads VT=.18. Using a 'label' (a variable name) is better than using a specific number here, and the same applies to assembly language.

The next most important group of accumulator actions is the arithmetic and logic group, which contains addition, subtraction, INC, DEC, AND, OR and XOR. We can add to it the SHIFT and ROTATE actions which we looked at briefly in the previous chapter. What we decide to place in this group is rather arbitrary, and a lot of books also place the CP command in this group. We'll start with the add and subtract operations. The Z80 has two varieties of these commands, the difference being the use of the carry bit in the F register. When the accumulator is used, as it is for most operations, the accumulator contains a byte before the action starts. Another byte is then added, from a different register or memory address, and the *result* of the action (addition or subtraction) is also stored in the accumulator. Sticking to immediate addressing for the moment, the effect of ADD A,#ØA will be to add the number ten (#ØA) to the contents of the accumulator. The carry flag is ignored when the addition is carried out, but it might be set *after the addition.* For example, if the accumulator register contained #2B (denary 43) before ADD A,#ØA, then the result will be that the accumulator contains #35 (denary 53), and the carry bit is reset (to Ø) because there is no carry. On the other hand, if the accumulator had contained the number #F9 (denary 249), then the effect of ADD A,#ØA would be to leave the accumulator storing #Ø3, and the carry bit set. Why? Because #F9+#Ø A=#1Ø3, and since the accumulator can store only one byte (the lower two digits of this hex number), then it stores the #Ø3, and the '1' causes the carry bit to be set. Once again, it doesn't matter whether the carry bit was set or not *before* the addition.

It's very different if we use the other add command, ADC. ADC means 'add with carry', and it will *always* add the carry flag number to the other number. If, for example, we have the number #2B in the accumulator and then use ADC A,#ØA, then the result will be #35 as before if the carry bit was reset (Ø) before the addition. If the carry bit was *set* however (equal to 1), then the result of the addition would be #36, one more. The reason for having two sets of addition commands is that we sometimes need to add in a carry (arithmetic on numbers that use more than one byte, for example), but just as often we don't want one (arithmetic with one byte numbers, for example). By having two sets of commands, we don't need to know in advance whether the carry bit is set or not. Microprocessors which don't use two sets of commands like this need to have commands which will set or reset the carry bit. The subtraction commands SUB (ignore carry) and SBC (use carry) perform in the same way. When SBC is used, the carry value is also subtracted. All of these arithmetic commands exist in a variety of addressing methods, though I have used immediate addressing in the illustrations for the sake of simplicity. ADD and SBC can be used with some sixteen-bit registers, notably HL. When this is done, the result is also stored in the same sixteen-bit register.

The INC and DEC commands are straightforward by comparison. INC and DEC can be used to increment or decrement any of the single-byte registers, so that we can use commands such as INC A, DEC C, INC H, and so on. These commands do not affect the carry bit but will *always* affect the Z and S bits of the flag register. INC and DEC can also be used with (HL), and with the index registers, to increment or decrement a byte which is held in the memory rather than in a register. The Z and S flags will be affected by these commands also, but the C flag is not. There are also INC and DEC commands for the double registers (HL, BC, DE, IX, IY and SP). These INC and DEC steps do *not* affect any of the flags in the status register. As we'll see in Chapter 7, this is something that can be a nuisance, but we can get around it.

The logic commands AND, OR, XOR *always* operate on the byte which is stored in the accumulator, and the result of the operation is also left in the accumulator. For example, suppose that the accumulator contains the byte #3E, which in binary is ØØ11111Ø, and that the C register contains the byte #AB, binary 1Ø1Ø1Ø11. The result of AND C will therefore be the AND of #3E and #AB which is #2A, binary ØØ1Ø1Ø1Ø, and this number will be stored in the accumulator. Look back to Chapter 1 if you have forgotten how the AND action works. The OR and XOR actions use the registers in the same way. This time, I've used an example which uses register addressing rather than memory addressing, but all of these commands can make use of the whole range of addressing methods.

Shift and rotate

The effects of the Z80's shift and rotate commands, with their assembly language mnemonics, are shown in Figure 5.1. A shift always results in a register losing one of its stored bits, the one at the end which is shifted out. Most types of shifts – the arithmetic shift right (SRA) is the exception – cause the register to gain a zero at the opposite end. The carry bit is used as a ninth bit of the accumulator in all of these shifts. The shift action can be carried out on the accumulator, on a byte in any of the other eight-bit general-purposes registers, or on a byte that is stored in the memory and addressed by (HL), or by using indexing. The effect of a shift on a binary

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

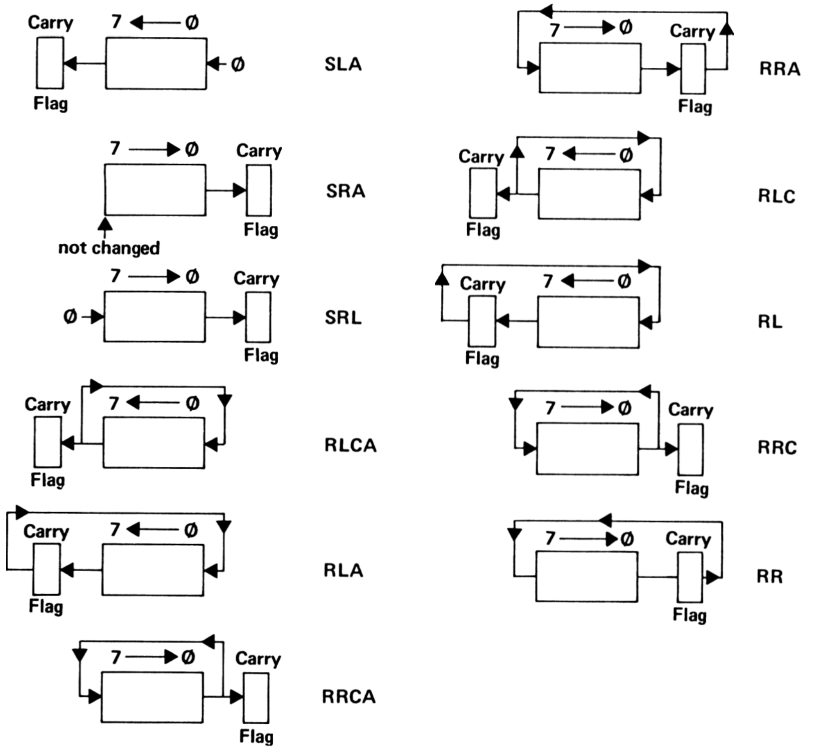


Figure 5.1. The Z80 shift and rotate instructions. You would not normally need to use more than a couple of these commands in your own programs.

number stored in the register is to multiply the number by two if the shift is left, or to divide it by two if the shift is right (Figure 5.2).

A rotation, by contrast, always keeps the same bits stored in the register, but the *positions* of the bits are changed. The Z80 has two directions of rotation, left and right, and three ways of using the carry flag – at the least significant end or at the most significant end of the byte and also as part of the rotation or not. There are eight possible Rotate commands in all, of which you are likely to use no more than two in the course of most of your

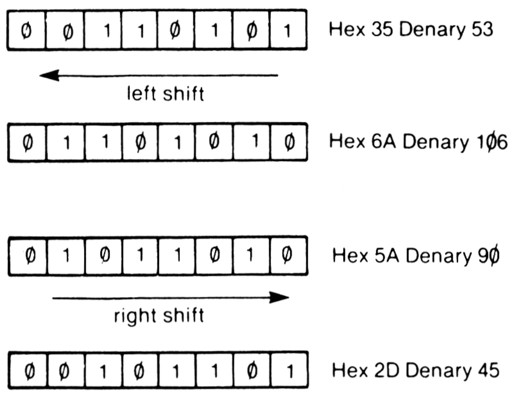


Figure 5.2. The effect of a shift on a number.

programs. Some of the rotations are carried out exclusively in the accumulator. These are marked by the use of 'A' in the mnemonic, so that rotations such as RLCA, RLA, RRCA, RRA are available only to bytes that are stored in the accumulator. The other Rotate commands are available to a wider range of addressing methods.

The CP (Compare) instruction is a particularly useful one which appears in almost every program. The command applies *only* to a byte that is stored in the accumulator. It can use any one of the standard memory addressing methods, and its effect is to compare the byte that was copied from the memory with the byte that is already present and stored in the accumulator. Compare in this respect means that the byte copied from memory is subtracted from the byte in the accumulator. The difference between this instruction and a true subtraction is that the result is *not* stored anywhere! The result of the subtraction is used to set flags, but nothing else, and the byte in the accumulator is unchanged. For example, suppose that the accumulator contained the byte #4F, and we happen to have the same size of byte stored at an address #727F which is held in the HL register pair. If we use the command:

```
CP (HL)
```

then the zero flag in the F register will be set (to 1), but the byte in the accumulator will still be #4F, and the byte in the memory will still be #4F. A subtraction would have left the content of the accumulator equal to zero.

Why should this be important? Well, suppose you want a program to do one thing if the 'Y' key is pressed, and something different if the 'N' key is pressed. If you arrange for the machine code program to store into the accumulator the ASCII code for the key that was pressed, you can compare it. By comparing it with #4E (the ASCII code for 'N'), we can find if the 'N' key was pressed. If it was, the zero flag will be set. If not, we can test again. By comparing with #59, we can find if the 'Y' key was pressed – once again, this would cause the zero flag to be set. If neither of these comparisons caused the zero flag to be set, we know that neither the 'Y' nor the 'N' key was pressed, and we can go back and try again. If it looks very much like the action that you can get with the INKEY$ loop in BASIC, you're right – it is.

Finally, we have the jump group of actions. These, as the name suggests, allow the flags in the F register to be tested, and will make the program jump to a new address if a flag was set. Which flag? That depends which jump-test instruction you use, because there's a different one for each flag, and for each state of a flag. For example, consider the two tests whose mnemonics are JR Z,d and JR NZ,d. JR Z,d means 'if zero flag set, then jump "d" places'. As this suggests, it will cause a jump if the result of a subtraction or comparison is zero. Its 'opposite number', JR NZ,d means 'if the zero flag is *not* set, then jump'. There are , therefore, two branch instructions which test the zero flag, but in opposite ways. The same sort of thing goes for most of the other flags. There's also a jump relative instruction, mnemonic JR, with no flag name following, which doesn't carry out any tests, like a GOTO with no IF preceding it. As well as these JR jump instructions, which use PC-relative addressing, there are JP instructions, which use other addressing methods such as direct addressing, HL addressing, and indexed addressing. The JP instructions can also test a wider range of flags.

The complete list of all the available jump instructions is shown in Figure 5.3. Many of these are instructions that you'll probably never use, and the really important ones are the ones that use the zero, carry and sign flags. For short programs, you will usually only ever need to use the JR instructions, because these are always adequate for jumps of up to 127 steps ahead or 128 back. The JP jumps are used in long programs, or when it is essential to jump from a program in RAM to a routine in the ROM *without returning to the RAM*. When you want to jump and return, there's a

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.

Figure 5.3. The complete list of Z80 jump instructions.

command CALL which corresponds to the use of GOSUB in BASIC. We'll deal with that one later, however.

Interacting with the CPC464 computer

The time has come at last to start some very elementary practical machine code programming of your CPC464 computer. This is not simply a matter of typing the assembly language lines as if they were lines of BASIC. Unless you happen to have the GENA3 assembler program in operation, the CPC464 computer will simply give you 'Syntax error' messages when you try to run these programs. Since we want to start on a small scale, we'll forget about assemblers at the moment, and assemble 'by hand'. This means that we find the machine code bytes that correspond to the assembly language instructions by looking them up in a table. We then poke them into the memory of the CPC464 computer, place the address of the first byte into the PC of the Z80, 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, the CPC464 computer uses quite a lot of its RAM, as we have seen, for its own purposes. If we simply POKE a number of bytes into the memory without heeding which part of memory we use, the chances are that we shall either replace bytes that the CPC464 computer needs to use, or our program bytes will be replaced by the action of the CPC464 computer. What we need is a piece of memory that is safely roped off for our use only.

This can be done by making use of the HIMEM command in BASIC. As the name suggests, HIMEM controls the highest address in the memory that can be used by BASIC. When you switch on the computer, typing ?HIMEM will produce the answer 439Ø3. This then is the highest memory address which any BASIC program can make use of. Addresses above this will be used by the machine for keeping note of essential addresses and other numbers. These are the sort of actions which are often referred to as 'housekeeping'.

Now you can alter this HIMEM number for yourself. Suppose, for example, that you alter it to 42999. You can do this by typing MEMORY42999 and then pressing ENTER. After you have done this, the new value of HIMEM will be 42999. Since BASIC will not use memory addresses above 42999, and the operating system will not use addresses below 43903, you have roped off some memory to use for machine code programs, or for storing any bytes that you want to be secure. This is a total of 9Ø3 bytes, which is more than enough for anything we shall be trying in this book. Do not attempt to use an address above the normal HIMEM number unless you are *very* sure about what you are doing, because if you corrupt the contents of some of these addresses, the operating system will go completely bananas. You will then have to switch off, and then switch on again to get back to normal operation, and reload your program. You *did* save it, didn't you?

Having protected a space in the memory so that we can store the bytes of a program, the other problem is how to place the starting address for your program into the program counter of the Z80. Fortunately, the designers of the CPC464 computer have been kind to you. There is a simple command word CALL which will do this for you. The principle is that CALL is followed by the first address of the program. For example, if you have a program which starts at address 43ØØØ (which in hex is #A7F8), you would start the program by using the BASIC command CALL 43ØØØ (or CALL &A7F8). The CALL command normally uses denary numbers, which is why the & has to be used when you work in hex. The PC is loaded with the address 43ØØØ and so the machine code program is started.

Using CALL 43ØØØ will therefore cause 43ØØØ to be used as the address of the first byte of your program. You must be quite sure that this is the correct address. Incidentally, I've taken the first byte as meaning 'starting byte'. It's possible to write programs in which the first few bytes are data, so that the program starts at, say, the tenth byte. This creates no problems, you simply use the address of the real starting byte as the number for CALL.

Lastly, for the moment at least, you have to ensure that your machine code program will stop in an orderly way. Nothing that we have done so far will indicate to the Z80 of the CPC464 computer where your program ends. As a result, the Z80 could continue to read bytes after the end of your program, until it encounters some byte which causes a 'crash'. This might, for example, be a byte which causes an endless loop. Some programmers doubt if there are any bytes which do *not* cause an endless loop in these circumstances! To return correctly to the operating system of the CPC464 computer, you need to end each machine code program with a 'return from subroutine' instruction, whose mnemonic is RET and whose code is #C9, denary 2Ø1. If you PEEK at the routines in the ROM, you'll see these '2Ø1' bytes scattered all over the code, marking the end of each section of program.

There's another headache that we don't have to worry about at the moment. When you run a machine code program along with a BASIC program in your CPC464 computer, you are using the same Z80 microprocessor for both jobs. It can't cope with both at the same time, so it runs one, then the other. If you make use of the Z80 registers in your machine code program, as you are bound to do, then you have to be quite certain that you are not destroying information that the BASIC program needs. For example, if at the instant when your machine code program started, the registers of the Z80 contained the address of a reserved-word in the ROM, then it will need this address in these registers when your machine code program ends. When a machine code program is called into action by using the CALL command, this is taken care of automatically. The CPC464 operating system uses the alternate registers of the Z80 very extensively. If you avoid using these registers there should be no conflicts between your own machine code programs and the routines that the machine uses. Later we shall see that you can use CALL to make use of machine routines in ROM, and when this is done, the alternate registers will be switched into use as needed, with no effort on your part. Other essential quantities are placed into a part of the RAM memory which is called 'the stack'. This, incidentally, is another good reason for being careful where you place your machine code in the memory. If you wipe out the stack, the CPC464 computer will quite certainly not like it! When the RET instruction is encountered at the end of your machine code, normal action resumes. If you call a machine code program into action by any other method, not using CALL, you may have to attend to this salvage operation for yourself as part of your machine code program. This is rather more advanced programming than we want to get into in this book, however.

Practical programs at last

With all of these preliminaries out of the way, we can at last start on some programs which are *very* simple, but which are intended to get you familiar with the way in which programs are placed into the memory of the CPC464 computer. You will also get some experience in the use of assembly language and machine code, and with how a machine code program can be run.

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

```
ORG 43ØØØ         ;start placing bytes here
LD A,#55          ;place hex 55 in accumulator
LD (43Ø5Ø),A      ;store it at 43Ø5Ø
RET               ; go back to BASIC
```

The first line contains a mnemonic, ORG, which you haven't seen before. It *isn't* part of the instructions of the Z80, but it is an instruction to the assembler, which in this case is you! ORG is short for origin, and it's a reminder that this is the first address that will be used for your program. We've chosen to use an address which leaves space for longer programs than we shall be writing in the course of this book, and we could have chosen a higher number. It will do as well as any other, however, and it leaves plenty of room for longer programs. When you program using an assembler, this line can be typed and the assembler will then automatically allocate the bytes of the program to the memory starting at this address. As it is, with assembly being done 'by hand' it simply acts as a reminder of what addresses to use. In particular, you have to remember to use the MEMORY command to rope off this piece of memory. Note the comments which follow the semicolons. The semicolon in assembly language is used in the same way as a REM in BASIC. Whatever follows the semicolon is just a comment which the assembler ignores, but which the programmer may find useful.

Now we need to look at what the program is doing. The first real instruction is to load the number #55 into the accumulator. This uses immediate addressing, so the number #55 will have to be placed immediately following the comma. The next line commands the byte in the accumulator (now #55) to be stored at address 43Ø5Ø. In hex, this is #A82A. It's an address well above the ones that we shall use for the program. Obviously, we wouldn't want to use an address which was also going to be used by the bytes of the program. This instruction uses direct addressing. There are more elegant methods, but not for beginners! Finally, the program ends with the RET instruction, essential for ensuring that CPC464 life continues normally after our program ends.

The next step in programming is to write down the codes. Each code has to be looked up, taking care to select the correct code for the addressing method. The code for LD A immediate is #3E, 62 denary, so that is the first byte of the program which will be stored at address (denary) 43ØØØ. We can start a table of address and data numbers with this entry:

```
43ØØØ    62
```

and then move on. The byte that we want to load is #55, 85 denary, and this has to be put into the next memory address, because this is how immediate addressing works. The table now looks like this:

```
43ØØØ    62
43ØØ1    85
```

The next byte we need is the instruction byte for LD to memory, with extended addressing. This byte is #32, 5Ø denary, and it has to be followed by the two bytes of the address at which we want the bytes stored. The address 43Ø5Ø translates into hex as #A82A, so we can use the bytes #2A and #A8 following the LD instruction. Remember the 'low-high' arrangement of the two byte of an address here. In denary, these are bytes are 42 and 168. The last code has to be the RET code of #C9, denary 2Ø1, so that the table now looks as in Figure 5.4. It uses addresses 43ØØØ to 43ØØ5, six bytes in all, and will place a byte into 43Ø5Ø, using denary numbers. Now we have to put it into memory and make it work!

This requires a BASIC program which will clear the memory, and poke

Address	Byte
43ØØØ	62
43ØØ1	85
43ØØ2	5Ø
43ØØ3	42
43ØØ4	168
43ØØ5	2Ø1

Figure 5.4. The coded program, using denary addresses and bytes of data. Each instruction byte has been chosen from the list in Appendix C.

the bytes in one by one. The program is shown in Figure 5.5. By using MEMORY 42999 we ensure that all memory addresses above 42999 are left unused by the CPC464 computer. That means addresses 43000 to 43903 in denary. We declare the variable M as 43000, so that we can make use of this in the POKE commands. Lines 30 to 50 then poke data numbers into addresses that start at 43000. We have used POKE M+N%, and with M=43000 and N%=0, the first address just has to be 43000. We have written this whole program in denary, which caused complications when we came to write address numbers in two-byte form. Since you have to make use of hex when you graduate to the DEVPAC assembler, it's as well to start getting familiar with the principles as soon as possible. In the following programs, then, we shall use addresses in hex, and hex data. Such hex data has to be read as a *string*, because it will not be accepted as a number by a command such as READ N%. This will mean using VAL("&"+D$) to find the number value of the hex codes so that they are poked in denary. In this way, a code which is read as "3E", for example, will have the & added to it, and VAL will then find the *denary* value for the POKE command, which cannot accept hex numbers.

```
10 MEMORY 42999
20 M=43000
30 FOR N%=0 TO 5
40 READ J%
50 POKE M+N%,J%:NEXT
60 CALL M
100 DATA 62,85,50,42,168,201
```

Figure 5.5. The BASIC program which puts the bytes into place.

Line 60 contains the CALL M, which loads this number into the PC of the Z80. This is the BASIC instruction which will actually cause your machine code program to run, with the start address specified by the value of the variable M. Line 100 then contains the six bytes of data that we have worked out. When you have entered the program, *but before you run it*, type ?PEEK(43050) to find out what is in address 43050. It should be zero. When you now RUN the program, there's no obvious effect. That's because you can't see what is in address 43050. If you use:

?PEEK(43050)

then you should now find the value of 85, which is the denary version of #55, the number that the program put there. Now try this: type POKE 43050,255 (ENTER) and then delete line 60 of your program. This is the CALL line. RUN the program again, and use ?PEEK(43050) to find what's there. It should still be 255. Now type CALL M and press ENTER. You will see the usual 'Ready' message appear on the screen. Using ?PEEK(43050) now should now give you 85 again. This is because poking the bytes of the program into memory won't make the program run, only CALL (with the correct address number) does this. You can therefore poke values into memory early in a BASIC program, and then make use of them later with an CALL wherever you like. Even if you now delete all the rest of the BASIC program, the machine code will stay in the memory until you POKE something else there, or until you switch off. Using CALL 43ØØØ will always make this piece of machine code run until some other set of codes is placed in this part of memory. We can make a lot of use of this to place a piece of machine code into memory, delete the BASIC program that put it there (by putting NEW in place of END at the end of the BASIC program), and use only the machine code. You can preserve the machine code program on tape if you like, and this is a technique that we'll look at later. One step at a time, if you please!

Now this first example of machine code isn't an ambitious piece of work, it does no more than POKE 43Ø5Ø,85 would do in BASIC, but it's a start. The main thing at this point is to get used to the ways in which you write machine code and how you place it into memory and run it. Now let's try something a lot more ambitious in terms of our use of machine code – though the example is simple enough. Figure 5.6 shows the assembly language version of the program. What we are going to do is to load a byte into the accumulator, shift it one place left, and then put it into memory at an address one step higher than the address from which we took it. This looks like a good opportunity to show an example of indexed addressing, so we shall start by placing an address into the X register. This is the LD IX, #A82A step, and its effect will be to place the address #A82A into the IX index register. The next line, LD A,(IX+Ø) means that the accumulator is to be loaded from the address in the X register, with Ø added to this number. This makes the load from #A82A, the same as the address in the IX register. The third step is SLA, arithmetic left shift, so that the bits of the byte are shifted left. Fourthly, we store this at address #A82B by using LD (IX+1),A. This time, 1 is added to the number in the IX register, so that the byte is stored at address #A82B. We end, as always, with the RET instruction.

```
LD      IX, #A82A
LD      A, (IX + Ø)
SLA     A
LD      (IX + 1), A
RET
```

Figure 5.6. The assembly language program for 'multipıy by two'. The listing shows the assembly language as it is written for an assembler.

Now we can put this into code form, using hex for all the bytes this time. It's not quite so easy as before, because of the use of indexing. The LD IX,#A82A instruction needs the codes of DD21, and these have to be followed by the two bytes of the address, 2A and A8 *in that order*. The LD A with indexed addressing is coded as DD7E, but that isn't the end of it. When you use an indexed instruction of this sort, the instruction byte has to be followed by another byte. This, called the *displacement*, is needed to specify the number that has to be added to the address in the index register. In this example, we are using numbers like ∅ and 1, but we can't omit the zero. This therefore follows the DD7E code for the indexed load. The SLA A bytes are CB27 (we've really picked all the multiple-byte instructions this time!), and then we get to the LD(IX+1). The instruction part is DD77 and this has to be followed by the displacement byte 1. This will make the load to an address which is the index address *plus one*. Finally, C9 is the RET command.

Now we have to code this in BASIC. If we choose a small number to place into #A82A, the effect of the left shift will be to double the number, so we can use this to obtain a bit of arithmetic wizardry. The BASIC program is shown in Figure 5.7. We start, as usual, by clearing memory space. You needn't worry if you have had another program in this part of the memory before. The new program will replace it completely, and provided that your program ends correctly with the RET instruction byte, the old program bytes cannot interfere with the new ones. The values are poked into place in the usual way in lines 3∅ and 4∅. In line 2∅, however, we place a number, 1∅ denary, into the address #A82A. Now this is the address which will be used by the program, and the byte which is #∅A (in binary form, ∅∅∅∅1∅1∅) will be placed in this address. In line 5∅, CALL M will carry out the machine code program, which should left-shift this byte, making it ∅∅∅1∅1∅∅. In denary, this is 2∅, twice 1∅. Line 1∅∅ contains the data bytes. When the

```
10 MEMORY 42999
20 M=43000:POKE &A82A,10
30 FOR N%=0 TO 12:READ D$
40 POKE M+N%,VAL("&"+D$):NEXT
50 CALL M
60 CLS
70 PRINT"Twice ";PEEK(&A82A);" is";PEEK(
&A82B)
80 END
100 DATA DD,21,2A,A8,DD,7E,00,CB,27,DD,7
7,01,C9
```

Figure 5.7. The BASIC program which pokes the bytes into place and then makes use of the machine code program.

program runs and line 7Ø proves what has happened by printing the result, you get the message:

```
Twice 1Ø is 2Ø
```

when the program runs.

It's simple enough, but if you knew nothing about machine code, you would wonder how on earth the number became multiplied by two. Once again, the program does nothing that could not be done more easily and as quickly by using BASIC. The important thing, from our point of view, is that you have now used indexed addressing, and a shift instruction, as well as getting more experience in putting a machine code program into your CPC464 computer by the hardest method of all. You can try poking different numbers into &A82A in line 2Ø. The answers will be correct for numbers up to 127 (denary), but from 128 up, you will get incorrect answers because the accumulator can hold only one byte. If you still don't see why, write the numbers in eight-bit binary, and then you'll see. If, incidentally, you have made any mistakes, particularly with DATA, then it's likely that the CPC464 computer will go into a trance and refuse to do anything. When you have typed in a BASIC program like this which pokes bytes into the memory, *always* record the BASIC program before you RUN it. This way, if the effect of an incorrect byte is to zonk out half the RAM, you can use CTRL SHIFT ESC to recover. Alternatively, you may have to switch off, then on again, and reload your program. If you didn't record it, then you'll have to type it all over again. That's hard work.

More examples

At this point, it's important to try a lot of simple programs to make sure that you are familiar with these methods. By the time you have finished this chapter, you will have a much better idea of how to approach machine code from a practical point of view, and you will be able to make up more exercises for yourself. The next chapter will then help you with the design of machine-code programs, which is the most difficult part of all. After that, it's all downhill!

Getting back to examples, it's time to do a little arithmetic with a register-indirect load. Figure 5.8 shows the assembly language version, which starts by loading an address, #A82A, into the HL register pair. The next step is to load the accumulator from this address, using LD A,(HL). Remember that the brackets mean 'contents of'. What we want to do is to load the byte that is stored at address #A82A into the accumulator. Having done that, the next step is INC HL. This will increment the address to #A82B. Note that we *don't* use INC(HL), because that would increment *the byte* at address #A82A, not the address itself. With the HL address incremented, the byte in the new address #A82B is added to the byte in the accumulator. The address

		Hex codes		
LD	HL, #A82A	21	2A	A8
LD	A, (HL)	7E		
INC	HL	23		
ADD	A, (HL)	86		
INC	HL	23		
ADD	A, (HL)	86		
INC	HL	23		
LD	(HL), A	86		
RET		C9		

Figure 5.8. An assembly language program which uses register-indirect loading to perform some additions.

is incremented again, and another byte is added to the accumulator. The accumulator, remember, will always contain the result of the addition, so it is accumulating the numbers for us. Finally, the address is incremented again and the accumulator is loaded into memory at #A82D, after which the program returns.

Now Figure 5.9 shows this in BASIC POKE form. Lines 1Ø to 4Ø prepare space, and poke the bytes of the program into memory. Line 6Ø then pokes three values into consecutive memory positions so that the program can use them. Line 6Ø runs the machine code part of the program, and line 7Ø proves that it has all worked by printing the sum of the three numbers. You can try changing the numbers for yourself, but remember that if the sum exceeds 255 denary, then what is printed will be only the remainder, the lowest eight bits, that remains in the accumulator. If the sum is 3ØØ, for example, then what remains will be 3ØØ−256=44. If the sum is 7ØØ, then what remains will be 7ØØ−(2*256)=188.

```
10 MEMORY 42999
20 M=43000
30 FOR N%=0 TO 10
40 READ D$:POKE M+N%,VAL("&"+D$):NEXT
50 POKE &A82A,15:POKE &A82B,12:POKE &A82
C,27
60 CALL M
70 PRINT"Sum is ";PEEK(&A82D)
100 DATA 21,2A,A8,7E,23,86,23,86,23,77,C
9
```

Figure 5.9. The BASIC program which pokes the machine code into place.

For a last example in this chapter, let's look at something a little bit less numerical. You can never get very far away from numbers when you are dealing with machine code, but at least this time we're not doing any arithmetic. Also, this time, we will use a loop in BASIC to call the machine code program twenty-six times. Ideally, we would perform the loop in machine code, but we're not quite ready for that yet. Figure 5.10 shows the assembly language version of what we're doing in the machine code portion of the program. The DE register pair (makes a change!) is loaded with an address, #A82A. This address will be used to store temporarily an ASCII code number for a letter. The C register is then loaded with the number #2Ø, using immediate loading. The principle here is that if the ASCII code for an upper-case (capital) letter is ORed with #2Ø, the result is the ASCII code for the same letter in lower-case. The same result can be achieved by adding #2Ø, but this seems a good chance to see the OR action working. The third line of the assembly language program loads the accumulator from the address in DE, #A82A, so that this will place an ASCII code into the accumulator. The OR C action then ORs the byte in the accumulator with the byte (#2Ø) in register C, and places the result back into the accumulator. We then load this byte back into the DE address of #A82A, and return to BASIC. All we need now is a BASIC program to poke the machine code bytes into place, and to place bytes into #A82A for the machine code program to operate on.

```
                              Hex Codes

LD      DE, #A82A       11    2A    A8
LD      C, #2Ø          ØE    2Ø
LD      A, (DE)         1A
OR      C               B1
LD      (DE), A         12
RET                     C9
```

Figure 5.10. An assembly language program for converting from upper-case to lower-case letters.

Figure 5.11 shows the BASIC program. Lines 1Ø to 4Ø follow paths which should be familiar to you by now, and no further comment is needed. Line 5Ø clears the screen and starts a loop, using J%=65 to 9Ø to cover the ASCII codes for A to Z, upper-case letters. Line 6Ø prints each letter, and line 7Ø pokes the ACSCII code into the address #A82A. This is the address which is used by the machine code program, and line 8Ø calls it into action. This will convert the code number into the ASCII code for a lower-case letter, so that A becomes a, B becomes b and so on. Line 9Ø prints the result by finding the number that is stored at #A82A and printing the CHR$ of this number. It's simple, but very effective – watch it in action!

```
10 MEMORY 42999
20 M=43000
30 FOR N%=0 TO 8
40 READ D$:POKE M+N%,VAL("&"+D$):NEXT
50 CLS:FOR J%=65 TO 90
60 PRINT CHR$(J%);
70 POKE &A82A,J%
80 CALL M
90 PRINT CHR$(PEEK(&A82A));
100 NEXT
110 DATA 11,2A,A8,0E,20,1A,B1,12,C9
```

Figure 5.11. The BASIC program which performs a loop to print the upper-case letters, and calls the machine code to convert to lower-case.

Chapter Six
Taking a Bigger Byte

The simple programs that we looked at in Chapter 5 don't do much, though they are useful as practice in the way that machine code programs are written. Practising assembly language writing and its conversion into machine code is essential at this stage, because you can more easily find if you are making a mistake when the programs are so simple. It's not so easy to pick up a mistake in a long machine code program, particularly when you are still struggling to learn the language!

Most beginners' difficulties arise, oddly enough, because machine code is so *simple*, rather than because it is difficult. Because machine code is so simple, you need a large number of instruction steps to achieve anything useful, and when a program contains a large number of instruction steps, it's more difficult to plan. The most difficult part of that planning is breaking down what you want to do into a set of steps that can be tackled by assembly language instructions. For this part of the planning, flowcharts are the traditional method of finding your way around. I never think that flowcharts are well suited for planning BASIC programs, but they come more into their own for planning machine code.

Flowcharts

Flowcharts are to programs as block diagrams are to hardware – they show what is to be done (or attempted) without going into any more detail than is needed. A flowchart consists of a set of shapes, with each shape being the symbol for some type of action. Figure 6.1 shows some of the most important flowchart shapes for our purposes (taken from the British Standards set of flowchart shapes). These are the terminator (start or stop), the input/output, the process (or action) and the decision steps. Inside the shapes, we can write brief notes of the action that we want, but once again without details.

A simple example is always the best way of showing how a flowchart is used. Suppose that you want a machine code program that takes the ASCII code for a letter from an address in memory, sets bit 7, and then replaces the

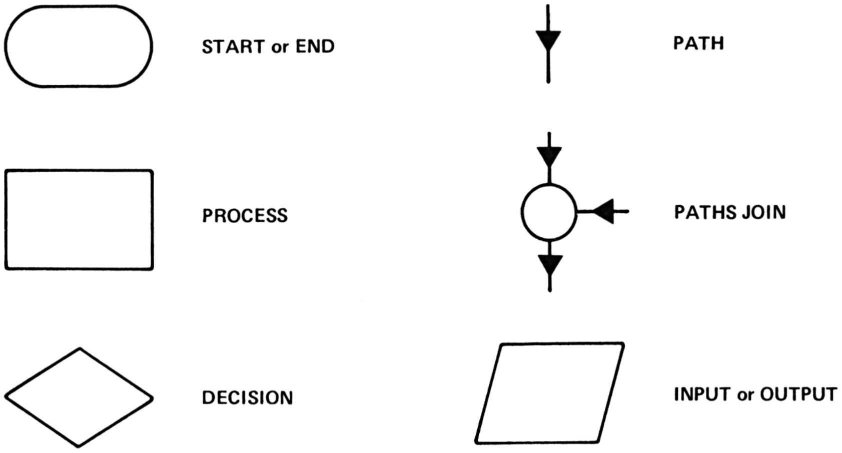


Figure 6.1. The main flowchart shapes.

number. Setting bit 7 (the most significant bit) is equivalent to adding 128 to the ASCII code, and will convert a letter code into a 'special character' code, or a graphics code. A flowchart for this action is shown in Figure 6.2. The first terminator is 'START', because every program or piece of program has to start somewhere. The arrowed line shows that this leads to the first 'input/output' block, which is labelled 'read code'. This describes what we

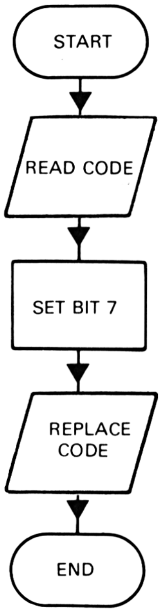


Figure 6.2. A flowchart for a program which will read a byte, set bit 7, and store the byte again.

want to do – get the code number for a character that is stored in a memory address. We don't know what the address is, nor does it matter at this point. After getting the character, the arrow points to the next action, setting Bit 7 of the ASCII code number. This is represented in the flowchart by an 'action' box, with the actual action written alongside. Next, as the arrow shows, the altered code number is replaced in the memory at the same address. The END terminator then reminds us that this is the end of this piece of program, it's not an endless loop.

This is a very simple flowchart, but it is enough to illustrate what I mean. The arrows are *very* important, because they show the direction of 'flow' (hence the name flowchart) of the program. You don't need to be reminded about the order of actions in an example as simple as this, but as your programs get more adventurous these arrows become more important. Note too that the descriptions are fairly general ones – you don't ever put assembly language instructions inside the boxes of your flowchart. A flowchart should be written so that it will show anyone who looks at it what is going on. It should never be something that only the designer of the program can understand and use, and which just confuses anyone else. A good flowchart, in fact, is one that could be used by any programmer to write a program in any variety of machine code – or in any other computer 'language', such as BASIC, FORTH, Pascal and so on. A lot of flowcharts, alas, are constructed after the program has been written (usually by lots of trial and error) in the hope that they will make the action clearer. They don't, and you wouldn't do that, would you?

Once you have a flowchart, you can check that it will do what you want by going over it very carefully. In such a simple example there isn't much to do, because the only thing that needs to be checked is that the order of actions is correct. In fact, if you write your flowcharts well, this is about all you ever have to do! That's because you will write a program by first drawing up a flowchart of the main actions, and there are surprisingly few main actions in a machine code program. Most programs, in fact, have only three boxes in their first flowcharts: an input, a process, and an output. Once you have decided on this outline, you then draw separate flowcharts for the separate sections. Each box of these flowcharts might then need a flowchart to itself, and so on until you have a set of steps that can easily be put into machine code. This shouldn't come as a surprise to you if you have written programs in BASIC, because if you read the right books (mine, I hope!) about CPC464 BASIC, you will already know how to plan a program by breaking it into smaller and smaller pieces. The main difference about machine code programs is that the pieces are very much smaller!

Warnier-Orr diagrams

A lot of programmers never get on well with flowcharts, and have turned to a different way of program design. The title, Warnier-Orr diagram, comes from the inventor Warnier, and the populariser Orr. One of the advantages of this method is that you don't, unless you are writing for a professional magazine, have to stick too closely to any set method. A Warnier-Orr diagram, despite its name, relies on words to describe the program plan rather then symbols. Its great advantage as compared to flowcharts is that it is better suited to the idea that a program is designed 'top-down', planning outlines first, and then filling in detail later.

Figure 6.3 shows a very simple example of a form of Warnier-Orr

```
START           { load HL

                ( in A82A
READ CODE       {
                ( use (HL)

SET BIT 7

                ( to A82A
REPLACE CODE    {
                ( use (HL)

END             { RET to BASIC
```

Figure 6.3. A simplified version of a Warnier-Orr diagram for the program which sets bit 7.

diagram for the same problem as we looked at in Figure 6.2. This time, the main parts of the process are shown written down the left-hand side. Curly brackets are then used to show how each action is broken down, or to give more information. I should stress that this is an *adaptation* of the Warnier-Orr method; professors and other purists should look away. It suits me, though, and it may suit you too. In a simple program plan like this, there's no need to look for any greater detail, but we could, if we needed to, use further sets of curly brackets for more detail. In this way, the diagram grows from the left-hand side of the paper to the right-hand side, with the finest detail on the right-hand side. The great advantages of this method are that you can see both the outline plan and the detail on one sheet of paper. From now on, I shall illustrate this method of planning for the example programs in this book. If you are happier with flowcharts, then all you need to do is draw the flowchart shapes alongside the notes.

Having shown the plan of this program in two different ways, we had better look at how to carry it out. If we follow the outline strictly, then we arrive at Figure 6.4(a). This puts the address #A82A into HL, then loads the accumulator from this address, using LD A,(HL). Setting the number 7 bit

```
(a) LD     HL, #A82A
    LD     A, (HL)
    SET    7,A
    LD     (HL), A
    RET
                          Hex codes
(b) LD     HL, #A82A     21   2A   A8
    SET    7, (HL)       CB   FE
    RET                  C9
```

Figure 6.4. Implementing the plan in assembly language: (a) reading the byte from the address, (b) a simpler method which sets bit 7 without reading the byte into the accumulator.

is done by the SET 7,A command, and the byte is then put back into memory by using LD (HL),A. This is what our experience so far leads to, but the Z80 allows a simpler method. This is because the SET command allows register indirect addressing. Figure 6.4(b) shows this in action, with HL being loaded, and then the SET action being carried out directly on the byte in the (HL) address. This gets rid of any need to load to and from the accumulator. Since the motto of all machine code programmers should be 'simplicate and add lightness', this second version is the one that we'll use.

Figure 6.5 shows a BASIC program which makes use of this action. The usual set-up lines are 1Ø to 4Ø, and the action starts in line 5Ø. The screen is cleared, and you are asked to enter your name. In line 6Ø, an INKEY$ loop makes the machine scan the keyboard until a key is pressed. Line 7Ø is used

```
10 MEMORY 42999
20 M=43000
30 FOR N%=0 TO 5
40 READ D$:POKE M+N%,VAL("&"+D$):NEXT
50 CLS:PRINT"Please enter your name"
60 K$=INKEY$:IF K$="" THEN 60
70 IF K$=CHR$(13) THEN 120
80 J%=ASC(K$):POKE &A82A,J%
90 CALL M
100 PRINT CHR$(PEEK(&A82A));
110 GOTO 60
120 PRINT:PRINT"That's your name in code
200 DATA 21,2A,A8,CB,FE,C9
```

Figure 6.5. The BASIC program which pokes the machine code bytes into memory and activates the program.

to detect the ENTER key, which stops the action. The action is in lines 8∅ to 12∅. The key that has been pressed will give K$, and we find the ASCII code for this character. This code is poked into address #A82A for the machine code to operate on. Line 9∅ runs the machine code, which sets bit 7 of the ASCII code and puts it back into address #A82A. When the machine code returns to BASIC, it lets line 1∅∅ run, which prints the character whose code is stored in #A82A. The semicolon keeps the printing in the same line, and line 11∅ causes the action to repeat until the ENTER key has been pressed. Line 12∅ then tells you that what you see on the screen is your code. It certainly won't look like your name! The useful thing about this is that it isn't obvious to anyone who reads the listing what is being done.

Loop back in hope

The examples which we have looked at so far are of linear programs, not of loops. Now it's very seldom that we make much use of linear machine code programs, because loops are much more common. A loop action in BASIC can be very slow, and it is only in looping programs that you can really appreciate the speed of machine code. This looks like a good opportunity to get in an introduction to looping. If you have done anything more than the most elementary BASIC programming, you will know what a loop involves. A loop exists when a piece of program can be repeated over and over again until some test succeeds. In BASIC, you can cause a loop to happen by using a line which might read, for example:

```
2∅∅ IF A=∅ THEN GOTO 1∅∅
```

This contains a test (is A=∅?), and if the test succeeds (yes, A is ∅), then the program goes back to line 1∅∅ and repeats all the steps from there to line 2∅∅ again. That sort of loop in BASIC corresponds very closely to how we create a loop in machine code. Instead of using line numbers, however, we are using address numbers. Instead of testing a variable called 'A', we shall test the contents of a register, which in most cases is the 'A' register.

Let's start the proper way with a planning diagram. Figure.6.6 shows how this might look. The first step is to load a register. The next step is to decrement the number which is in the register. The third step is to test what remains. If it is not zero, then we must return to the decrement stage. I have used a 'label letter', A, to show where the loop returns. This is not a standard Warnier-Orr marking but it's very convenient because it marks where the loop begins, something that the use of GOTO in BASIC lacks. Having put down this very simple outline, then, we use the curly brackets to enlarge on it. At the loading stage, we'll place the byte #FF into register B. This is the largest size of single byte that we can have. The decrement step will then be done using DEC B, and the test will use JR NZ, and will return to the point A, the decrement step.

```
LOAD REGISTER           } FFH into B

A: DECREMENT            } use DEC B

IF NOT ZERO, GOTO A     } use JR NZ

END                     } RET
```

Figure 6.6. A plan for a one-byte counting loop.

The action, then, will be that the B register is loaded immediate with the byte #FF, and the byte in this B register is then tested to see if it is zero. If it is, we return to BASIC. If it isn't (which means that the countdown is not complete), then we decrement and try again. Now we have to put this into assembly language form – and that's going to introduce some new items to you. We could, knowing what we know so far, program this in assembly language as:

```
      LD B,#FF
Loop: DEC B
      JR NZ,Loop
      RET
```

and this would carry out the action that we want. The Z80, however, was designed so as to allow you easier ways of programming counting loops. One of these ways is the instruction DJNZ. The letters DJNZ refer to a command which automatically decrements the B register and will jump to a new address if the byte in the B register has not reached a value of zero. The name DJNZ, in fact, stands for 'Decrement and Jump if Not Zero'. This command applies *only* to the B register.

Figure 6.7 shows an assembly language program which makes use of this useful DJNZ step. The LD B,#FF step loads the B register with the

```
                       Hex codes
      LD B, #FF        Ø6   FF
LOOP: DJNZ,LOOP        1Ø   FE
      RET              C9
```

Figure 6.7. The assembly language for a counting loop using DJNZ.

maximum possible size of byte. The new step is the 'DJNZ LOOP', and this step now has the word LOOP: stuck in front of it. This word LOOP is a 'label', and the colon is a separator. LOOP is being used here in place of an address, and it means the address at which the instruction starts. The colon is used to separate the label word LOOP from the instruction word DJNZ. With LOOP: placed in front of the DJNZ LOOP instruction, the word LOOP means the address at which the DJNZ instruction byte is stored. By using words in this way, we avoid having to think about address numbers until we actually write the machine code. If we use an assembler, we usually don't have to worry about address numbers *at all* – the assembler automatically puts in address numbers in place of label words.

In assembly language, this all looks quite neat and straightforward. If we were using an assembler it would be straightforward, but when we assemble by hand, it's not so simple. The reason is that we have to follow the DJNZ instruction by a single byte which will give the address of the DJNZ instruction byte. This is PC-relative addressing, so that we have to use a signed byte that can be added to the address in the program counter to give the address of the LDA step. We have looked at the formula for this earlier, but we'll go through the method again. Assuming that we are going to place the first byte of the program at address #A7F8 (43ØØØ in denary) then the address of the DJNZ instruction is at #A7FA, and the displacement byte will be at #A7FB. #A7FA is the source address, where we're coming from and #A7FA is also the destination address, where we're going to. Note that we *always* use the address of the instruction byte. Subtract source from destination numbers, and we get Ø. Subtract another 2 from this, and we get −2. The figure −2 in hex is FE, so that's the displacement byte that is placed following the DJNZ instruction code. If you are in doubt about these address numbers, then write them in denary. Dedicated hex-programming nuts (yes, like me) buy special calculators (the TI LCD Programmer Calculator) which carry out arithmetic in hex, but you have to be keen to use it all that much, and fairly well-heeled to buy it (at around £45).

Now it's time to try out this looping business in a program. Figure 6.8 shows a program which pokes the machine code into memory, and then carries out the machine code loop. It then carries out the same action in BASIC. Now when you run this program, you'll find that the machine code doesn't seem to have a very noticeable advantage. Both versions, in fact, deal with the count from 255 (hex #FF) to zero pretty quickly. What makes the BASIC count fairly fast is the use of integer numbers, but in fact, most of the time that is taken by both programs is due to printing on the screen. The results are therefore misleading, and if we want to see how fast machine code is compared to BASIC, we must use much longer counts, so that the time delays that are caused by actions like printing are not so significant. First of all, though, we need to find out how to carry out a longer count in machine code.

```
10 MEMORY &A7F7
20 M=&A7F8
30 FOR N%=0 TO 4
40 READ D$:POKE M+N%,VAL("&"+D$):NEXT
50 PRINT"Start count...";
60 CALL M:PRINT"End"
70 PRINT"Now the same in BASIC..."
80 INPUT"Press ENTER to start";a$:PRINT"
START..."
90 FOR Q%=255 TO 0 STEP -1:NEXT
100 PRINT"End."
110 DATA 06,FF,10,FE,C9
```

Figure 6.8. A BASIC version of the machine code counting program, along with a similar count in BASIC. The count is too fast to time in either case.

A look at longer loops

The most obvious way of carrying out a longer countdown in machine code is to load one of the double registers and count that down. Unfortunately, though, there is no command like DJNZ for the double registers. Equally unfortunately, though we can DEC the double registers, the DEC action on a double register does *not* affect the flags in the status register, so that we can't use JR NZ following DEC to loop back until the count is finished. This lapse on the part of the designers can be remedied with a little bit of programming cunning. If we load the contents of one half of a double register into A, and then OR with the other half, only one condition will result in zero. That's when both halves of the double register contain zero. Suppose, for example, that we are counting down in BC (the favourite for this action). If we carry out LD A,B, then the accumulator will be loaded from B. If we then OR C, then if there is a 1 anywhere in either A or C registers, there will be a 1 in the result (stored in A). Only if both registers are zero will the result be zero. The point here is that a zero in A after OR C *will* set flags. We can therfore follow the OR C by a JR NZ to do the looping action.

Let's look at it, in Figure 6.9. The first action is to load the BC register pair with the number #FFFF, 65535 in denary. The label word 'LOOP' marks where the loop starts, and in this line the BC register pair is decremented. The result is then tested by using LD A,B and OR C, as described, and this is followed by JR NZ,LOOP to make the program loop back if the count has not reached zero. The program returns to BASIC after the end of the countdown. Figure 6.10 is the poke program which puts the bytes into place. This time, you have to press ENTER to start each countdown, and you'll notice a very marked difference between the machine code version and the

```
                                   Hex codes
            LD BC, #FFFF      ∅1    FF    FF
LOOP:       DEC BC            ∅B
            LD A, B           78
            OR C              B1
            JR NZ, LOOP       2∅    FB
            RET               C9
```

Figure 6.9. A counting program which uses a register pair, BC. This takes considerably longer to execute.

```
10 MEMORY &A7F7
20 M=&A7F8
30 FOR N%=0 TO 8
40 READ D$:POKE M+N%,VAL("&"+D$):NEXT
50 CLS
60 INPUT"Press ENTER to start";A$:PRINT"
START...";
70 CALL M:PRINT"STOP."
80 PRINT"Now the same in BASIC..."
90 INPUT"Press ENTER to start";A$:PRINT"
START..."
100 FOR N=65535 TO 0 STEP -1:NEXT
110 PRINT"STOP"
120 DATA 01,FF,FF,0B,78,B1,20,FB,C9
```

Figure 6.10. The BASIC POKE program for the double-register count-down, and a BASIC count for comparison. The speed advantage of machine code is pretty obvious!

BASIC version. The machine code version appears to end in about half a second – it's still almost too fast to time. The BASIC version takes about 73 seconds. In this example, then, machine code is roughly 146 times faster than BASIC! The advantage would be less, about 64 times, if we could use integers for the BASIC loop. This illustrates the sort of speed advantages that you can expect when you are dealing with a program which is purely machine code.

Trying to time the machine code version with a stop-watch is very inaccurate because of the time that it takes you to start and stop the watch. We can find *exactly* how long a countdown in machine code takes, however. The clock-rate of the CPC464 computer is given as 4.00 MHz. This means 4.00 *million* clock pulses per second, so that the time between pulses is approximately 0.25 *millionths* of a second. Now for each Z80 instruction, there is a time which is measured in terms of the number of clock cycles. These times are shown in detail in Appendix E, and Figure 6.11 shows how much time is needed for each instruction in the machine code loop. This is approximate, because the JR NZ instruction takes less time on the last run, when it doesn't loop back. The correction is *very* small, though, and the figure of 26 clock cycles per loop is a good guide. 26 clock cycles, with each clock cycle taking 0.25 millionths of a second, gives us a time of 6.5 millionths of a second for each loop, so that 65536 loops take a time of 0.426 seconds, under half a second. Counter loops like this can be used to produce *very* precise time delays, because the clock rate is controlled by a quartz crystal which is as precise as the one which would be used in a modern watch or clock. These time delays are used to a considerable extent in the ROM routines. The sound routines, BEEP, cassette input and output and printer routines, just to name a few, make use of precise timing to achieve their actions, and loops such as the ones we have used here are the basis of that accuracy.

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

Figure 6.11. The times for each instruction can be summed to find how much time is needed for each loop. Times are expressed in clock cycles.

Using loops

Now that we have seen the speed of a machine code loop when it's used as a time delay we need to take a look at other applications of loops. One of the most obvious applications of a loop is to screen operations, because the time that BASIC needs to carry out screen actions can be quite limiting. Try, for example, a BASIC program which fills the screen with the letter 'A', as Figure 6.12 shows. This program in BASIC takes about six seconds to fill the screen with the letter 'A'. We could expect that a machine code version of

```
10 CLS
20 FOR N%=1 TO 1000
30 PRINT"A";:NEXT
```

Figure 6.12. A BASIC program that fills the screen with the letter 'A'.

this program would take rather less time. The problem is, how do we go about it.

There are two ways, the *very* difficult way and the easy way. The *very* difficult way is to make use of the screen addresses directly. The easy way is to find the routine in the ROM which carries out the 'write to screen' action. Both methods require information, and the difficulty about the first method is that we need to place numbers into the screen memory to draw out the shape of the letter 'A'. At first sight, the 'easy' method doesn't look much better, because you need to know an address and some information about registers. As it happens, there is a lot of published information here. This is obtainable from Amsoft in the form of the book which is titled *The Concise Firmware Specification*. This book lists all of the routines in the ROM that are likely to be useful to the machine code programmer, and a lot of other very useful information. Don't feel that you have to rush out and buy this book right away, but if you are going to become a serious machine code programmer of the CPC464, then it's a must. The routine which starts at address #BB6C will clear the screen, and the routine which starts at #BB5A will write a character on the next available screen space. The character ASCII code has to be in the accumulator before this second routine is used.

That's one step onwards, but we then have to know how to make use of these routines. Assembly language has two commands, CALL and RET, which are almost exact parallels of GOSUB and ENTER in BASIC. CALL has to be followed by a starting address, which can be in ROM or RAM. Its effect will be to jump to that address, but to store the current address in the 'stack' memory. The subroutine that is called by CALL should end in RET (or some variation of RET, like RET Z, meaning 'return if the zero flag is set). When the RET command is executed, the Z80 will take the address from the stack, increment it, and then carry on as if nothing had happned. When I say that CALL and RET is the parallel of GOSUB and ENTER, incidentally, I mean a very close parallel, because GOSUB/ENTER is carried out by using CALL and RET in machine code. The BASIC interpreter, remember, is just a very long program in machine code! Don't confuse the machine code instruction CALL with the BASIC CALL which is used to start a machine code program running.

With all of that to digest, let's plan a program which will fill the screen with the letter 'A'. Figure 6.13 shows the planning, with the outline, as usual, on the left-hand side. We want to clear the screen, write 1ØØØ ASCII 65s to the screen, and then return to BASIC. The figure of 1ØØØ comes from the rows and columns that we use. There are 25 rows, each of 4Ø characters in the normal text screen and 25×4Ø is 1ØØØ. The next stage in planning shows more detail. The screen will be cleared by using CALL ØBB6C. The action of placing letter 'A' on the screen will be done by loading the accumulator with the number #41 (ASCII 'A'), setting a counter to 999, printing to the screen with CALL #BB5A, and then decrementing and testing the counter. If the counter has not reached zero, the program returns to the point that is

```
START

CLEAR            { use CALL #BB6C

                 ( set counter to 999      { use BC→#Ø3E7
WRITE ASC 65     ( put 65 in A→#41
1000 TIMES       ( * CALL #BB5D            { DEC & use
                 ( DJNZ to *               { LD A,B OR C JR NZ

END              { RET
```

Figure 6.13. A plan for a program that will use machine code to fill the screen with the letter 'A'.

labelled (X), the screen print CALL. Note that we need to load the accumulator on each pass through the loop, because the number #41 will be replaced when the LD A step (for testing the count) is carried out. Normally, we would try to avoid having to reload a register in a loop, because it wastes time. The further detail is then shown in the next set of brackets, which shows the numbers in hex, and reminds us to use LD A,B and OR C followed by JR NZ for the loop control. Now we can write the assembly language program.

This is shown in Figure 6.14, and it follows the lines of the plan so exactly that there isn't much point in making any further comments. The first few lines allocate numbers to label names. This is rather pointless when we assemble by hand, but it's very useful when an assembler is used. Once again, you can see the use of semicolons in the assembly language program.

```
ORG      #A7F8
COUNT:   EQU #Ø3E7                              Hex  codes
START:   CALL #BB6C       ; clear screen        CD   6C   BB
         LD BC, COUNT     ; counter             Ø1   E7   Ø3
LOOP:    LD A, #41        ; ASC 'A'             3E   41
         CALL #BB5A       ; print it            CD   5A   BB
         DEC BC           ; decrement count     ØB
         LD A,B           ; test - destroy A    78
         OR C             ; compare             B1
         JR NZ, LOOP      ; back if not zero    2Ø   F6
         RET              ; to BASIC            C9
```

Figure 6.14. The assembly language program for the screen-fill plan. This has been written for an assembler. ORG is a reminder of the starting address, and the label COUNT has been used for the counting number.

In assembly language, remember, the semicolon acts like a REM in BASIC. Anything that follows the semicolon will be printed in the listing, but the assembler ignores it. Since we're assembling by hand at the moment, the semicolons are just a convenient reminder of what the program is about. Always put these comments in, because when you look at an uncommented assembly program in a few weeks time, you won't have a clue about what it was supposed to do! Trying to follow a BASIC program that you wrote last year is nothing compared to trying to understand a machine code program that you wrote half an hour ago.

Figure 6.15 shows the program converted into BASIC form, using POKE, and with an INKEY$ step that allows you to see how fast the machine code is. When you use this, the speed with which the screen fills with the letters is limited only by the speed of the CPC464 screen print routine. This is not as fast as we could arrange if we could access the screen addresses directly, but it's roughly twice as fast as BASIC. For a lot of purposes, this rate of screen filling would be fast enough for characters like this. A lot of action on the text screen can be organised with the aid of this type of routine, and we can now take a look at an extension to this program. Suppose that instead of filling the screen with the letter 'A', we print out the entire character set? This again is a rather slow action in BASIC, as you'll see if you write a BASIC program to do it. In assembly language, it's not so very different from what we have just done. This is important, because if you can develop a new program out of one which you have tested and which you know to be reliable, then your new program is more likely to be successful.

```
10 MEMORY &A7F7
20 M=&A7F8
30 FOR N%=0 TO 16:READ D$
40 POKE M+N%,VAL("&"+D$):NEXT
50 PRINT"Press any key to start"
60 K$=INKEY$:IF K$="" THEN 60
70 CALL M
80 END
100 DATA CD,6C,BB,01,E7,03,3E,41,CD,5A,B
B,0B,78,B1,20,F6,C9
```

Figure 6.15. The screen-fill program in BASIC POKE form.

Figure 6.16 shows the planning stage of this program. In outline, we shall set a counter, print a character, select another character and repeat this to the end of the count. In more detail, we shall print #EØ characters (224 denary), using a count from #DF to zero. The first character code number is 2ØH, the spacebar character. The printing can be done as usual by CALL #BB5A, and on each loop we shall have to increment the character number and decrement the counter. The test can be done by using DJNZ, because

```
   START

                  ( # characters
   SET COUNT      {                       { use B register
                  ( start at # 2Ø

*  PRINT CHAR     { use CALL # BB5A       { use C register

                  ( INC char
   NEXT CHAR      {
                  ( DEC count

   TEST COUNT     { DJNZ to  *

   END            { RET
```

Figure 6.16. A plan for printing out the entire character set of the CPC464 computer.

the count is less than #FF so it will fit in the B register. This point is noted in the third layer of detail, and we can also decide on a register to hold the character code number. We might be able to hold it in A, but if we wanted to extend this program to cover the screen we couldn't use A, so we'll use C right from the start. All of this planning leads to the program in Figure 6.17.

The BASIC POKE program is illustrated in Figure 6.18, and it will carry out the action of printing the entire character set. It's fast, and you can compare the speed to that of a BASIC program which carries out the same

```
ORG       #A7F8
COUNT:    EQU #DF
STRT:     EQU #2Ø                                Hex  codes
          CALL #BB6C        ; clear              CD   6C   BB
          LD B, COUNT       ; counter            Ø6   DF
          LD C, STRT        ; 1st char.          ØE   2Ø
LOOP:     LD A,C            ; char in A          79
          CALL #BB5A        ; print it           CD   5A   BB
          INC C             ; next char          ØC
          DJNZ LOOP         ; back till ended    1Ø   F9
          RET               ; to BASIC           C9
```

Figure 6.17. The assembly language for the plan of Figure 6.16.

```
10 MEMORY &A7F7
20 M=&A7F8
30 FOR N%=0 TO 14:READ D$
40 POKE M+N%,VAL("&"+D$):NEXT
50 PRINT"Press any key to start"
60 K$=INKEY$:IF K$="" THEN 60
70 CALL M
100 DATA CD,6C,BB,06,DF,0E,20,79,CD,5A,B
B,0C,10,F9,C9
```

Figure 6.18. The BASIC POKE program for the character set plan.

action. One great benefit of having something like this in machine code is that it is independent of BASIC. Suppose, for example, that you replaced line 7ø in the program of Figure 6.18 with the line:

7ø KEY 128,"CALL &A7F8"+CHR$(13)

and you added a line 75 NEW. The effect would be to put the machine code into memory, and set up the zero key on the number-pad to make the machine code execute. The program would then wipe itself out, leaving the machine code in place, and the function key loaded. You could then load in another BASIC program, but any time that you wanted to display the full character set, you would only need to press the zero key on the number-pad. Magic! One point to note, incidentally, is the use of CALL #BB5A. If you have the *The Concise Firmware Specification* book, you will see that there is another character printing routine at address #BB5D. If you try this one, you'll find that your character printing routine jams. This is because the routine at #BB5D allows register values to be corrupted, so that counters which use BC will not work. The routine at #BB5A restores register values, and that's why I have used it. This is just one example of the sort of thing that makes machine code programming more difficult, even when you have a lot of information available.

Reading the keyboard

The routine at #BB5A is very useful for printing a character, and we could make a lot more of this if we could also make use of the keyboard. As it happens, there is a routine in the ROM for doing just that. It's at #BB18, and it carries out a complete INKEY$ routine. In other words, when you call this, it will read the keyboard and wait for a key to be pressed. If a key is pressed, then the ASCII code for the character of that key will be placed in the accumulator. If no key is pressed, then the routine keeps looping round, waiting for you. Unlike INKEY$, then, we don't have to place this CALL in a loop to make use of it. A loop of this type is called a 'holding loop', because it keeps the machine held up until you do something – in this case, pressing a key.

Let's start the proper way with a bit of planning. Figure 6.19 shows how this might look. The first step is to get the character, the second is to print it. Now we can look in more detail at each of these. Getting the character consists of testing the keyboard, and then looping back if the number that is returned is zero. In more detail, we shall use the routine at #BB18 to do all of this. The 'Print Character' part is done by the routine at #BB5A as before.

```
START
                    ( * test keyboard      } use # BB18
GET CHARACTER      {
                    ( if zero, then  *     { this is automatic

PRINT IT           } use #BB5A

END                } RET
```

Figure 6.19. A plan for reading the keyboard, and printing the character for whichever key is pressed.

Figure 6.20 shows the assembly language version of all this. The routine at #BB6C is used to clear the screen, and then the keyboard CALL starts. In the CALL, the keyboard is tested, and a byte is put into the accumulator when a key is pressed. The accumulator will then contain the ASCII code for that key. When a byte is put into the accumulator, the routine at #BB18 returns, and the byte is printed by the CALL #BB5A routine.

```
ORG:     #A7F8
KEYB :   EQU #BB18
CLEAR :  EQU #BB6C
PRNT :   EQU #BB5A                     Hex  codes
         CALL CLEAR     ; cls          CD   6C   BB
         CALL KEYB      ; get key      CD   18   BB
         CALL PRNT      ; print it     CD   5A   BB
         RET                           C9
```

Figure 6.20. The assembly language, which makes use of calls to the ROM routines.

```
10 MEMORY &A7F7
20 M=&A7F8
30 FOR N%=0 TO 9:READ D$
40 POKE M+N%,VAL("&"+D$):NEXT
50 CALL M
100 DATA CD,6C,BB,CD,18,BB,CD,5A,BB,C9
```

Figure 6.21. The BASIC POKE version.

Figure 6.21 shows the program put into the form of a set of BASIC poke instructions. When this runs, pressing a key will cause a letter or other character to appear on the screen, and the program then ends. Not all keys will produce a result (find out which ones don't), and some keys produce unexpected effects. Try the ESC, DEL, CLR and COPY keys, for example, and also try the effect of pressing the SHIFT key along with other keys. You'll notice that there is no 'Syntax error' message following the letter on the screen. This is because the machine code acts directly, cutting out the action of BASIC. The letter has not been produced by the normal BASIC action, so the BASIC error message does not appear! By making use of these three routines in the ROM, however, we can carry out a surprising amount of machine code actions that require input or output. While we're on the subject of these CALL addresses, incidentally, it's useful to note what they consist of. The actual routines that carry out screen clear, get character and print character are *not* at the addresses #BB6C, #BB18, and #BB5A. What is placed at each of these addresses is an RST command. RST means 'reset', and it allows a special form of jump to be used to addresses in either ROM. This is *not* something that you will find in every computer that uses the Z80, it is a particular feature of the CPC464. The 'address' number which follows the RST is not, in fact, simply an address number but a combination of an address and a code number to show which ROM is to be used. Details of this coding are given in Chapter 9. If you investigate the memory around each of these addresses, then, you will find another pair of bytes stored following the RST code of #CF (2Ø7 denary). The routine starts at another address, so that the addresses #BB6C, #BB18 and #BB5A (along with others) are just staging posts. The reason for this is that it allows for changes to be made in the ROM. If a ROM is changed, then planning is needed to make sure that the machines which use it can still use the new routines correctly. For example, the RST at #BB6C is RST 8 #954Ø, which is a jump to a routine at address #154Ø. Suppose the ROM were changed so that this routine started at #1542. Any jump to the old address of #154Ø would then be wrong, but if the bytes in #BB6C to #BB6E were also changed so as to hold the new address, then all calls to #BB6C would still locate the correct address for the routine. This sort of thing is often referred to as a 'terminal block' or 'jump-block' by comparison with electrical circuits, where conections can be made to a terminal block and altered at that point if necessary. Another name is 'hook'. A jump block in ROM is not, however, quite so useful to us as one in

```
   START

   CLEAR              { use #BB6C

*  GET CHARACTER      { use KEYB subroutine

   TEST FOR C/R       { use CP #ØD

   BACK IF NOT        { to  *{JR NZ used
      ⊕
   END                { RET
```

Figure 6.22. A plan for testing the keyboard and reading each key until the ENTER key is pressed.

RAM. A RAM terminal block allows us to put in a JP instruction of our own, so that the normal action of the machine is diverted to a routine of our own choosing. That's something that we'll look at later, in Chapter 9.

Meantime, how about developing this fragment of program action into something more useful. Suppose that we wanted to print on to the screen everything that was entered until the ENTER key was pressed? This means using a loop, to test the key character that has been obtained to find if it is a ENTER code (ØDH, 13 denary). Figure 6.22 shows the plan for this action, which contains one novelty. On the last section of the outline plan, you'll see a symbol which looks like a plus sign with a circle round it. This means Exclusive OR, and when used in a plan like this it means that one statement or the other must be true. You can't have *both* statements true, and you can't have *neither* statement true. The effect here is that the program must loop back (no return character) or end.

Figure 6.23 then shows the assembly language which corresponds to this action, which is straightforward. When the program of Figure 6.24 runs, the screen clears, and you see the cursor at the top left-hand corner. You can then type anything that you like, and it will appear on the screen. The crunch comes when you press ENTER. Once again, there is no 'Syntax error' message, such as you would expect to see when you type anything that isn't a BASIC command, and the 'Ready' prompt appears at the left-hand side of the line, replacing whatever you typed there. This is because you have typed the letters using machine code. Once again, when you use machine code, you have completely bypassed BASIC. Since what you have typed has not been

```
ORG      #A7F8
KEYB     : EQU #BB18
CLEAR    : EQU #BB6C
PRNT     : EQU #BB5A
C/R      : EQU #ØD                           Hex  Codes
START    : CALL CLEAR       ; cls            CD   6C    BB
GET IT   ; CALL KEYB        ; get key        CD   18    BB
           CALL PRNT        ; print it       CD   5A    BB
           CP C/R           ; is it ENTER    FE   ØD
           JR NZ, GET IT    ; back if not    2Ø   F6
           RET              ; to BASIC       C9
```

Figure 6.23. The assembly language version, which is a development of Figure 6.20.

```
10 MEMORY &A7F7
20 M=&A7F8
30 FOR N%=0 TO 13:READ D$
40 POKE M+N%,VAL("&"+D$):NEXT
50 CALL M
100 DATA CD,6C,BB,CD,18,BB,CD,5A,BB,FE,0
D,20,F6,C9
```

Figure 6.24. The BASIC POKE program for Figure 6.23.

checked by the BASIC interpreter of the ROM, there can be no 'Syntax error' message. In the same way, the BASIC routines which keep a check on where the prompt should appear have also been bypassed, and the prompt therefore appears at the left-hand side. Normally, when you use a routine like this you would *not* be returning to BASIC at this stage, so these points present no problems. I have emphasised these points, however, to show that you are very much on your own when you write machine code. The normal BASIC error messages do not apply, and you will have to create routines for yourself if you want to detect errors in anything that is entered.

Chapter Seven
Cassettes and Parameters

Save it!

At this point, when our programs are getting slightly longer, and doing more interesting things, it's time to look at the topic of saving machine code programs on tape. Now you aren't in any way obliged to do this. Our machine code programs have all been, so far, in the form of a BASIC program which assembles machine code numbers into the memory. You can, obviously, just save this BASIC program, and it will create the machine code for you whenever you want. When you are using a mixture of BASIC and machine code, this is the ideal way of saving and loading the machine code bytes. There are times, however, when you need as much of the memory as possible, and another piece of BASIC program is as welcome as an elephant in a space capsule. You may also want to put on cassette a program that is a mixture of BASIC and machine code, and to make it difficult for anyone to copy it. Either way, you'll want to make a direct recording of the bytes that are stored in the memory. The ordinary BASIC commands of the CPC464 computer can cope with this, but, as we'll see, in a rather different way from the conventional LOAD and SAVE.

Like most other machines, the CPC464 computer has special BASIC commands for saving or reloading a machine code program. There are considerable advantages in doing this. One is that bytes of machine code take up a lot less memory than the assembler instructions or the BASIC lines that generate the code. If you generate the code separately, using a piece of assembly language, then record the code (as distinct from recording the assembly language version), you can then place that code into any BASIC program. This will give you the advantage of having a piece of machine code but without the amount of space that the assembly language would take up. Another advantage is that machine code takes up very little space on a cassette, and is replayed very quickly. A third advantage is that your machine code is much more secure from prying eyes – a quick LIST on club meeting night will not reveal much to the would-be copier! Nothing is ever totally secure, but using machine code direct from the cassette makes a program much less easy to copy in a hurry than using a BASIC POKE program. In the following chapter, we'll look at the topic of generating machine code from assembly language by using the GENA3 assembler. If you use a BASIC POKE program, then it can delete itself by making the last line NEW instead of END. In this way, the code will be poked into memory, but the program that pokes it disappears, along with all trace of the addresses and data.

Down to business, then. Machine code on cassette (or on disk, when that becomes possible) uses the modified command LOAD in exactly the same way as BASIC, but the SAVE command is very different. We'll start, then, with the SAVE command, and I'll use as an example the program which we met in the previous chapter, the one which prints characters on to the screen as you type them (Figure 6.24). As you might expect for a machine code operation, SAVE requires a lot more information to be supplied than the normal SAVE command in BASIC. Following SAVE, you need a 'filename', a starting address, and a length number. These are the *minimum* requirements; we'll look at other possibilities later! The filename obeys exactly the same rules as a BASIC filename. A comma must follow the closing quotemark of the filename, and must be followed by a letter B (B for Binary). This is followed by another comma, then you need the starting address. The starting address is the address, in denary or hex, of the first byte of your machine code program. If you use hex, you must remember to place '&' in front of the hex number. This is followed by another comma, and the length number, which is the number of bytes in the program, again in denary or hex. The program in Figure 6.24 uses a start address of &A7F8 and consists of 14 bytes (denary), so we could record it using the command:

```
SAVE "TEST",B,&A7F8,14
```

There's nothing like trying it for yourself. Load in the BASIC program which produces the characters, and remove the CALL M line. RUN the program, so that the machine code assembles, then type the SAVE command as above. Make sure that the address corresponds to the address on your *own* listing if you used different address numbers. It's very important to get the starting address correct. If you specify a length number which results in saving several bytes beyond the end of the program, it doesn't matter. If, however, the length number gives a finishing address which is short of the true value, then you may find yourself with no RET, or with other vital pieces missing, and this will cause trouble later.

The SAVE command for machine code allows one small but useful variation. This is that you can use another address number following the length number. This is the 'execution address', and it can be included if the machine code program does not start with the first byte. At the moment, this is not a matter of concern, but you will at some stage meet a program which has data bytes as its first few bytes, and has the true start later. The execution address is the address of the true start of the program. If the program starts with the first byte, then there is no need to include this address.

Try the SAVE command, then, on your program of Figure 6.24, and then perform the ultimate test. Switch the machine off and then on again, or press CTRL SHIFT and ESC together. You have to do this, because it's the easiest way of wiping out the machine code that is stored in the memory. Now type:

```
MEMORY &A7F7      (Press ENTER)
LOAD "TEST"       (Press ENTER)
```

– that's all, no 'B' and no address. When you press ENTER after the LOAD instruction, you get the usual cassette system messages, and you can then run the machine code by typing CALL&A7F8. You can't LIST anything, however, because there's no BASIC program in the memory.

Just in case you thought you knew it all by now, there are a couple of extra twists to the LOAD command. Normally, when you use LOAD with only the filename following it, the machine code loads into the starting address that has been specified in the SAVE command. You can, however, move a program. That means that you can load it into a different set of memory addresses. Moving the bytes of the program does *not* guarantee that it will run correctly at the new address, but many programs will; they are called 'relocatable programs'. Suppose, for example, that you wanted to move the "TEST" program so that it started at the address &9ØØØ. You will first have to make certain that you can use this part of memory. If you restart (CTRL, SHIFT, ESC) and then use MEMORY &8FFF, then &9ØØØ onwards will be free for use. You can then use:

```
LOAD"TEST", &9ØØØ
```

This will only load the program, and you have to start it running by using CALL&9ØØØ. You can, however, use a modification of SAVE which will create a tape which can be loaded and then run automatically. This is done by following the length number with a comma and then by the execution address. For example, SAVE "TEST",B,&A7F8,14,&A7F8 would create a tape which would contain &A7F8 as a starting address. You could then load this by using RUN" (or by pressing CTRL and the ENTER key on the number-pad). The prgram will then load *and run*. The snag here is that when the program ends, the machine will initialise, clearing the memory. This, however, can be an advantage if you want to make your program secure against copying.

Take a message...

After that interlude on the subject of saving and reloading machine code, let's get back to the programs. We left Chapter 6, you remember, writing characters on the screen. It's time now that we looked at ways of putting something more interesting on the screen, and words look like a reasonably simple start to this type of programming. What do we have to do? Well, to start with, we need to store some ASCII codes for letters somewhere in the memory; we can't just use a string variable as we would in BASIC. We will have to know the address at which the first of the letters is stored, and we also need some way to stop the process. Having done that, we should be able to design a loop which takes a byte from the 'text space' (where the letter codes are stored) and passes it to the subroutine that prints the character.

We start, as always, with a program plan. It's not so easy this time, because we need a different way of ending the loop. We could count the number of letters that we want to place on the screen, but I want to look at a different technique this time – using a terminator. You are probably familiar with this idea used in BASIC programs. A 'terminator' is a byte which the program can recognise as a special character, one which is not, for example, part of a message. A convenient terminator for numbers is Ø, but for words, the 'carriage return' code of &ØD is quite useful, so we'll try that. We can do this by testing the accumulator after it has been used for the printing routine. After, you note, not before, because the carriage return is a valid character, whose effect we need in the string of characters, placing it into the writing routine. We also need the 'line-feed' character, ØAH, to ensure that the cursor will move down by one line after printing the phrase. This will avoid having the 'Ready' prompt blot out the message!

The plan that we need is shown in Figure 7.1. What we have to do is to store an address for the message we want to print on the screen. This will be the address of a string of bytes of ASCII codes. I've chosen LETR as the label name for this piece of memory. If we were using an assembler, then this piece of assembly language could be used almost as it is. In GENA3 assembler language, DEFM means 'define message', and is used to allocate

```
START

GET LETR          { put into HL
ADDRESS

                  ( load A from (HL)
READ &            ) check for terminator
PRINT             ) increment HL
                  ( print letter

REPEAT IF         { end if terminator
NOT TERMINATOR
    ⊕
END               { RET
```

Figure 7.1. A plan for printing a message on to the screen.

```
          ORG #A7F8
LETR;     EQU #A83∅
PRNT:     EQU #BB5A
C/R:      EQU #∅D
L/F :     EQU #∅A
START :   LD HL, LETR   ; get address
LOOP :    LD A, (HL)
          INC HL
          CALL PRNT
          CP C/R
          JR NZ, LOOP
          RET
LETR :    "THIS IS THE TEXT" ∅A ∅D
```

Figure 7.2. The assembly language program for printing the message. The ASCII codes of the letters, followed by #∅A and #∅D, are stored starting at address LETR.

bytes into memory. This is *not* part of Z80 standard assembly language, simply a convenient action of this assembler. You need quotes around the string, as you would use when assigning a string variable. By having the label name of LETR ahead of the command, we make sure that it is placed at the correct starting address of #A83∅.

Since we're using a BASIC POKE, however, the next problem is then how we store the bytes starting at this address, using only BASIC. This is fairly easy, using POKE, and we'll deal with it later. Getting back to the assembly language, Figure 7.2 shows what we need. The HL register pair is loaded with the address LETR. In the loop, the accumulator is loaded from (HL), which means that it copies the first ASCII code in the text. The HL address is incremented, then the printing subroutine is called to print the character. The character code is then compared with ∅DH, the terminator, and if the test fails (it's not ∅DH) then the program loops back. When the ∅DH character is read, it is printed, and the program then ends, returning to BASIC.

Figure 7.3 shows the BASIC POKE version of all this. Lines 5∅ to 7∅ define a string, with characters ∅AH and ∅DH at the end, and poke this into memory. This is done by a loop which starts at address &A83∅+∅, and pokes the first ASCII code here. By incrementing the address on each pass, and using the loop counter (plus 1) as the selecting number in MID$, we successively poke each ASCII code of the string into memory starting at &A83∅. This places the string of bytes ready for the machine code. If we had been using the assembler, then the bytes would have been put into memory by the DEFM action. We have placed the machine code bytes with lines 3∅ and 4∅ as usual, and we can now call the machine code. The CLS action is carried out in BASIC this time. It all looks rather clumsy because of the size

```
10 MEMORY &A7F7
20 M=&A7F8:P=&A830
30 FOR N%=0 TO 12:READ D$
40 POKE M+N%,VAL("&"+D$):NEXT
50 T$="THIS IS THE TEXT"+CHR$(11)+CHR$(1
3)
60 FOR N%=0 TO LEN(T$)-1
70 POKE P+N%,ASC(MID$(T$,N%+1,1)):NEXT
80 CLS:CALL M
100 DATA 21,30,A8,7E,23,CD,5A,BB,FE,0D,2
0,F7,C9
```

Figure 7.3. The BASIC poke program. Remember that once this has run, the machine code can be saved separately.

of the BASIC program, but remember how this can be recorded. If you use SAVE "name",B to record the machine code and the text codes, the whole thing can be run in from tape without the need for any BASIC. Once you have created a machine code program, it can be saved and loaded without any need to have BASIC present. Another boon is that, provided you have used MEMORY to reserve a memory space, you can LOAD in a machine code program *without affecting a BASIC program which is in the memory.* This means that your BASIC program can have LOAD instructions in it so as to load in machine code as it is needed. If you make a tape of a program which has a line such as:

LOAD"CHARS"

in it, then at that point in the BASIC program, the cassette motor will start (if the PLAY key is still down), and a machine code program called "CHARS" will be loaded. It will not be RUN, however, until your BASIC program uses a line which has the CALL command in it.

Passing values

The instruction CALL has been used so far purely to start a machine code program. It can, however do rather more than that, it can *pass a parameter* to the machine code routine. To do this, the 'parameter' must follow the CALL address, with a comma separating the two. By 'passed to', I mean that if, for example, you use CALL &A7F8,5 , then the figure 5 is placed at an address in memory which can be used by a machine code program. In addition, a 'number of parameters' figure is also placed into the accumulator. The accumulator then holds the number of items that are to be passed, and the address of the last item is held in the X-register. The main restriction about all this is that each parameter must consist of no more than two bytes. A parameter in this sense means a number. This might, for

```
START
GET VALUE            } FROM (IX + Ø) address
COMPLEMENT           } use NEG
PRINT ON SCREEN      } use #BB5A
STOP
```

Figure 7.4. A plan which involves passing a number quantity from BASIC to a machine code routine.

example, be an integer, or it might be the one byte of the address of a string. If you simply use an integer number, then the X register will contain an address at which the low-byte of this number is stored. Your machine code program can then make use of this number by loading from (IX+Ø) and (IX+1) as required.

In this very simple form, it's quite straightforward to pass a number to a machine code routine, and make the routine do something with the number. An example will make this clearer. Take a look at the program plan of Figure 7.4. This outline proposes to get the value, make the complement (the negative form), and then print the result on the screen. The address of the value is obtained from (IX+Ø), the complement by using NEG, and the value is then printed on the screen by using the familiar CALL #BB5A.

Figure 7.5 illustrates the assembly language for the action that has been planned in Figure 7.4. Once again, this is a simple action which could just as easily have been carried out using BASIC, but an example like this is short and easy to type, whereas a more complicated example would take longer to type (more mistakes!) and would be full of items which are not essential to understanding the principles. The assembly language program starts with LD A,(IX+Ø), so that the accumulator is loaded from the address which is held in the IX register. We shall make things simpler still here by restricting the number to one byte only. If we use two-byte numbers, then we will find the lower byte at (IX+Ø) and the higher byte at (IX+1). The allocation of a single-byte number has been done in the BASIC part of the program. NEG will then complement the number. The result is then printed on to the screen by CALL #BB5A, and the RET returns to BASIC.

```
          ORG #A7F8
PRNT:     EQU #BB5A
          LD A, (IX + Ø)
          NEG
          CALL PRNT
          RET
```

Figure 7.5. The assembly language routine for implementing the plan of Figure 7.4.

```
10 MEMORY &A7F7
20 M=&A7F8
30 FOR N%=0 TO 8:READ D$
40 POKE M+N%,VAL("&"+D$):NEXT
50 CALL &A7F8,21
100 DATA DD,7E,00,ED,44,CD,5A,BB,C9
```

Figure 7.6. The BASIC POKE version of this program.

In the BASIC program of Figure 7.6, the assembled machine code is poked into memory in the usual way, and then the CALL #A7F8,21 passes the number 21 (denary) to the machine code. The machine code program will then pick this up, complement the number, and print the result on the screen. What you see is the biological female symbol. This has the ASCII code of 235, which is 256−21, as you might expect from the NEG action. It's no more than you would get if you had used PRINT CHR$(256−21), but it does at least show what is involved.

Being able to pass a value directly like this can sometimes be useful, but it's a lot more useful to be able to pass a variable value. It's even more useful if the variable value can be altered and then used by the BASIC program. The Amstrad manuals do not describe this action very well, so hang on to your hats for an account of what happens. To start with, you can pass on any variable value, and what is passed is the address of the VLT entry. You don't have to find this by any special trickery, because the BASIC of the CPC464 has a command to find VLT addresses. This is @, and if you type, for example, ?@X%, then what appears on the screen, remember, is the VLT address of the value of X%, assuming that you have assigned such a variable. The address that you get in this way is the address of the *first byte of the VLT entry*. For an integer entry, this would be the low-byte, with the high-byte in the next address up. For a floating-point number, the VLT address would be the lowest of the four bytes of the mantissa, with the other three bytes of mantissa, and the exponent byte, stored at consecutive higher addresses. For a string variable, the address is the address of the first of the three bytes of the string block. You may remember that this string VLT entry consists of the length number followed by two bytes of the string address.

The important point about all this is that @ obtains VLT entry *addresses*, not the numbers stored in them, and that these can be passed to a machine code routine. The machine code can then read values from these addresses, *and can alter values in the VLT*. In this way, the values of variables in a BASIC program can be altered by the action of a machine code program. For example, if you use A%=5, followed by CALL&A7F8,@A%, then the machine code routine which starts at &A7F8 might read the '5' from memory and put '25' back into the same address. If the next BASIC line is PRINT A% then this will give the number 25! Here, then, is the way that you can make machine code routines interact with your BASIC programs.

ASSIGN VAR	X% = 21
PASS TO M/C	CALL & A7F8, @ X% Read address from (IX + Ø), (IX + 1) into HL Read byte from (HL)
MAKE NEGATIVE	NEG
PASS BACK	use LD (HL),A

Figure 7.7. How a single-byte number value can be passed to and from a machine code program.

Figure 7.7 gives an example of this type of technique. The plan is to pass a number 21 (denary) to the machine code routine. This number will be stored in the VLT at an address which can be obtained from @X%. By using a CALL&A7F8,@X%, the VLT address can be passed to the address in the IX register. Since an address consists of two bytes, this means that (IX+Ø) will hold the low byte of the VLT address and (IX+1) will hold the high byte. We can assemble these two bytes into HL, so getting the VLT address into HL. Reading from (HL) will then give the value of the low byte of X%. If X% is less than 256, then only this byte needs to be used.

Figure 7.8 shows the assembly language for the machine code part of the program. The L register is loaded from (IX+Ø) and the H register from (IX+1), so that HL contains the full VLT address. These actions should be the first of the program. We can then use LD A,(HL) to get the byte which is stored at this address. NEG will then form the complement of this number, and by using LD(HL),A, we then *put the complemented number back into the VLT.* This last step means that the value of the variable will now have altered. If the value of the variable is printed by the BASIC program after the machine code has run, the value will be the altered one, the complement. Figure 7.9 is the BASIC POKE version, which demonstrates the action. The codes are poked into place as usual, and a value is assigned to X%. To keep things simple, this value is less then 256. The VLT value is then passed to the machine code by using @X%, and then the complemented value is printed.

```
LD L, (IX + Ø)          ; VLT address low
LD H, (IX + Ø)          ; VLT address high
LD A, (HL)              ; get byte in VLT
NEG                     ; negative
LD (HL), A              ; put back
RET
```

Figure 7.8. An example of passing a number which is manipulated by the machine code and then returned.

```
10 MEMORY &A7F7
20 M=&A7F8
30 FOR N%=0 TO 10:READ D$
40 POKE M+N%,VAL("&"+D$):NEXT
50 X%=21
60 CALL &A7F8,@X%
70 PRINT X%
100 DATA DD,6E,00,DD,66,01,7E,ED,44,77,C
9
```

Figure 7.9. The BASIC POKE version.

Passing strings

Passing a string value to a machine code routine involves a little bit more work. Once again, we find the VLT address stored in the (IX+Ø) and (IX+1) addresses. We then have to pick up this address and find out what is stored at *this* new address to get the string length, then get the next two bytes for the address of the string. These three bytes are known as the 'string descriptor' in the Amsoft books. The Amsoft books caution you *not to alter the string descriptor in any way*, but say that the string bytes themselves can be altered.

We can illustrate the action of passing string values by a program which will 'code' a string message. The idea is that each byte will have 3 added to its ASCII code – not exactly a baffling code, but it's principles we want to illustrate. Figure 7.10 shows the plan for this action. This goes further than a typical plan might, because I want to emphasise what's going on – it's not all that easy to follow at first. Throughout, I'll refer to the addresses as VLT, meaning the Variable List Table address of the string length and situation and STR, meaning the address of the string itself.

GET VLT ADDR	(IX + Ø) → L	
	(IX + 1) → H	
	HL is address of start	
READ STRING CHARACTERS	1st byte = length	put into B
	2nd byte = low byte addr	put into DE
	3rd byte = high byte addr	
ADD 3	get (DE) → A	
TO EACH	A → A + 3	
PUT BACK	A → (DE)	
RETURN	to BASIC	

Figure 7.10. A plan which requires a string value to be passed.

```
        LD    L, (IX+Ø)      ; low byte
        LD    H, (IX + 1)    ; high byte
        LD    B, (HL)        ; length
        INC   HL             ; next
        LD    E, (HL)        ; address
        INC   HL             ; into
        LD    D, (HL)        ; DE
LOOP:   LD    A, (DE)        ; get character
        ADD   A, #Ø3         ; add 3
        LD    (DE), A        ; put back
        INC   DE             ; next
        DJNZ  LOOP           ; until all done
        RET                  ; back to BASIC
```

Figure 7.11. The assembly language for the 'message coding' plan.

The first action is GET VLT ADDRESS. This, you remember, is held at the address whose bytes are held in (IX+Ø) and (IX+1) and we can get to this most easily by loading L and H from these addresses. This VLT address which is in HL can now be used. The first byte is a length byte, and we want it in B. Why? Because B is used for the counter action of DJNZ, and if we load from (HL) into B, we shall have the character count all ready to go! The next two addresses in the VLT we read and place in DE. This pair of bytes makes up the STR address. We can now form a loop which will use DE for addressing, read from (DE) into the accumulator, add 3, put it back, and repeat until the B register is at zero. That's all, folks! Figure 7.11 shows what it all boils down to in assembly language. Note that though there are a lot of steps in the assembly language, they create very little machine code, only 19 bytes of machine code, less than the average line of BASIC! Figure 7.12 shows the BASIC POKE program which will put the machine code into place and test it with a string. Once again, it doesn't do anything that you couldn't do in BASIC, but it shows how you go about locating and using a

```
10 MEMORY &A7F7
20 M=&A7F8
30 FOR N%=0 TO 18:READ D$
40 POKE M+N%,VAL("&"+D$):NEXT
50 INPUT"Your name, please ";NM$
60 CALL M,@NM$
70 PRINT"Code is ";NM$
100 DATA DD,6E,00,DD,66,01,46,23,5E,23,5
6,1A,C6,03,12,13,10,F9,C9
```

Figure 7.12. The BASIC POKE version of Figure 7.11.

string, which is the point of the exercise. It's also the longest assembly language program that we've dealt with so far, and in the following chapter, we'll start looking at how you can use an assembler to create much longer programs.

I have avoided the topic of passing floating-point variables to machine code programs. There are several good reasons for this. One is that writing programs to make use of floating point numbers is very hard work, and practically every one that you might need is already in the ROM. The other is the very simple reason that we hardly ever need to pass floating point numbers. One of the main uses of machine code is to provide fast screen displays. As you have seen, even the machine code routines which print characters on to the screen are fairly slow. To control items like screen positions we may have to pass numbers to a machine code program, but these numbers will *always* be integers. The complication of learning how to make use of floating point numbers, then, is practically never necessary. If you want to write a new routine to calculate the value of PI to a million places, using machine code, then I apologise. If you're writing that sort of program, though, you don't need this kind of book! Meanwhile, we're going to divert to some topics that need to be dealt with before we can make much more progress in Z80 code.

The stack

The stack is part of the normal RAM memory of the computer, and the only thing that makes it special is the fact that it is used in a special way. Early on in this book, we looked at the variable list table which sits just above a BASIC program. This was one example of a piece of memory which was reserved for use by a BASIC program. The stack is another part of the RAM, but unlike the VLT, its starting address is fixed unless you alter it by a command. That doesn't mean that it will *always* be in the same place. If we look at a 'map' of the top of the memory (Figure 7.13), we can see that the memory between #B1ØØ and #CØØØ is reserved as a 'work area' for the BASIC ROM – we've looked at a few of these addresses previously. The position of the start of the stack when the machine is switched on is somewhere around #BFFF. From this address, lower address numbers are used as required, with something over 256 bytes available for stack use. This is not a restriction for most machine code programs, but if you wrote routines in BASIC which depended on recursion (where a routine calls itself, leaving an address on the stack), it could be. The starting position of the stack can be moved by machine code commands. The position of the stack, however, *must remain fixed* for the duration of a program. If you do anything to alter it during a program, then this is guaranteed to cause a crash. When you work in BASIC, you can be prevented from doing anything like this, because the program will stop with an error message if

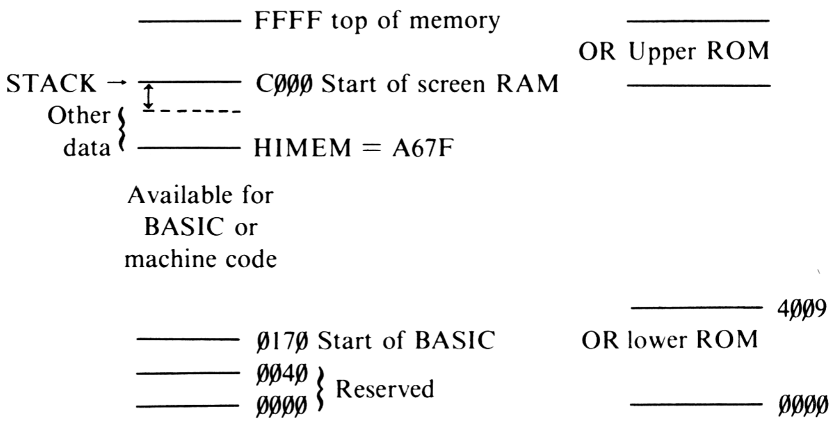


Figure 7.13. A 'map' of the top of memory, showing how the stack is normally positioned.

you have any instruction which might cause the stack to be overwritten with other data. In machine code, you have no such protection.

Our programs so far have used addresses of #A7F8 upward. The stack 'grows downwards', meaning that as you store bytes on the stack, the next address that will be used is a *lower* address. For example, if the stack address is #BFFF and a pair of bytes is stored, then the address changes to #BFFD, two bytes lower. This address for the next available place in the stack is held in a register of the Z80 which is called the 'stack pointer'. By loading this register with other values, we can change the position of the stack. For your own programs, unless there is some very pressing reason for shifting the stack pointer, *leave it alone*. The stack pointer setting for BASIC can normally be used with no problems in your machine code programs. If you shift it, then you must store its value in RAM, and restore this value before your machine code program jumps back to BASIC.

Using the stack

You can make use of the stack in programs without having to worry about where the stack is located, unless there is a danger of the stack growing down into the program area. If you use a MEMORY command, such as MEMORY &A7F7, and place your program only in addresses above the area that is cleared (#A7F8 upwards in this example), then you are not likely to corrupt the stack, and you can use it as you please. You can, of course, corrupt a lot of other addresses if you make your program use addresses above the normal HIMEM address of #AB7F. If you want to write long machine code programs, clear a lower starting address, as for example by MEMORY &7FFF.

The Z80 uses the stack in two ways, automatic and manual. The automatic use is by commands such as CALL #BB5A. Just before this is executed, the PC will have an address in it which is of the next instruction to be executed. Executing the CALL causes the Stack Pointer (SP) to be decremented and the high-byte of the PC stored at the address held in the SP. The SP is then decremented again, and the low-byte of the PC stored. The SP is then left with the address in the stack where the low-byte of the PC was stored. The new CALL address is then loaded into the PC, and the subroutine is executed. When the RET instruction has been executed, the byte from the address that is held in the SP is loaded into the PC low-byte, the SP is incremented, and then the byte at this new address is loaded to the PC high. This restores the correct PC address that existed just before the CALL was executed, and the SP is then incremented. This 'top of stack' position is never used, it is kept empty so as to mark the position of the top of the stack memory. Figure 7.14 summarises the process.

As well as this automatic use of the stack by CALL and some other instructions (such as RST), we can carry out stack commands in the form of PUSH and POP. PUSH means putting the contents of a double register onto the stack, using two bytes of the stack memory. In a PUSH, the stack pointer is decremented, the high-byte placed into the stack address, then the SP decremented again to store the low-byte. This is the same order of storing as is used in the CALL instruction. The POP action follows the RET action, with the low-byte being returned, then the SP incremented, the high-byte returned, and the SP incremented again. The PUSH and POP operators can be used with the operands AF, BC, DE, HL, IX and IY, so that all of the registers that are important to a program can be saved in this way. It's time, then, to see *why* we need these commands and how we use them.

One very common requirement is to save BC because of a DJNZ count. Suppose that we have a loop which has been programmed by loading the B register with #FF, and using DJNZ. Now this would normally prevent us from making any use of the BC register pair just before and after the loop. You couldn't, for example, do what is shown in Figure 7.15(a), where an address is loaded into BC just before a loop, and then used afterwards. Using DJNZ will have changed the number in the B register, so that BC will quite certainly not hold the address #A8BC after the loop; it will hold #ØØØØ because B has been decremented to zero. The program sample of Figure 7.15(b) gets around this by pushing BC on to the stack just before the loop, and pulling it off the stack immediately afterwards. Another use for PUSH and POP is in preserving flags. Suppose that you have just carried out a CP action that set flags, but that you want to carry out another two actions before testing for a jump. If these actions alter the flags, the jump will not be correct, but by using PUSH AF just after the CP, and POP AF just before the JR step, you can restore the values in A and F that were present immediately after the CP, *no matter how much has happened between the PUSH and the POP.*

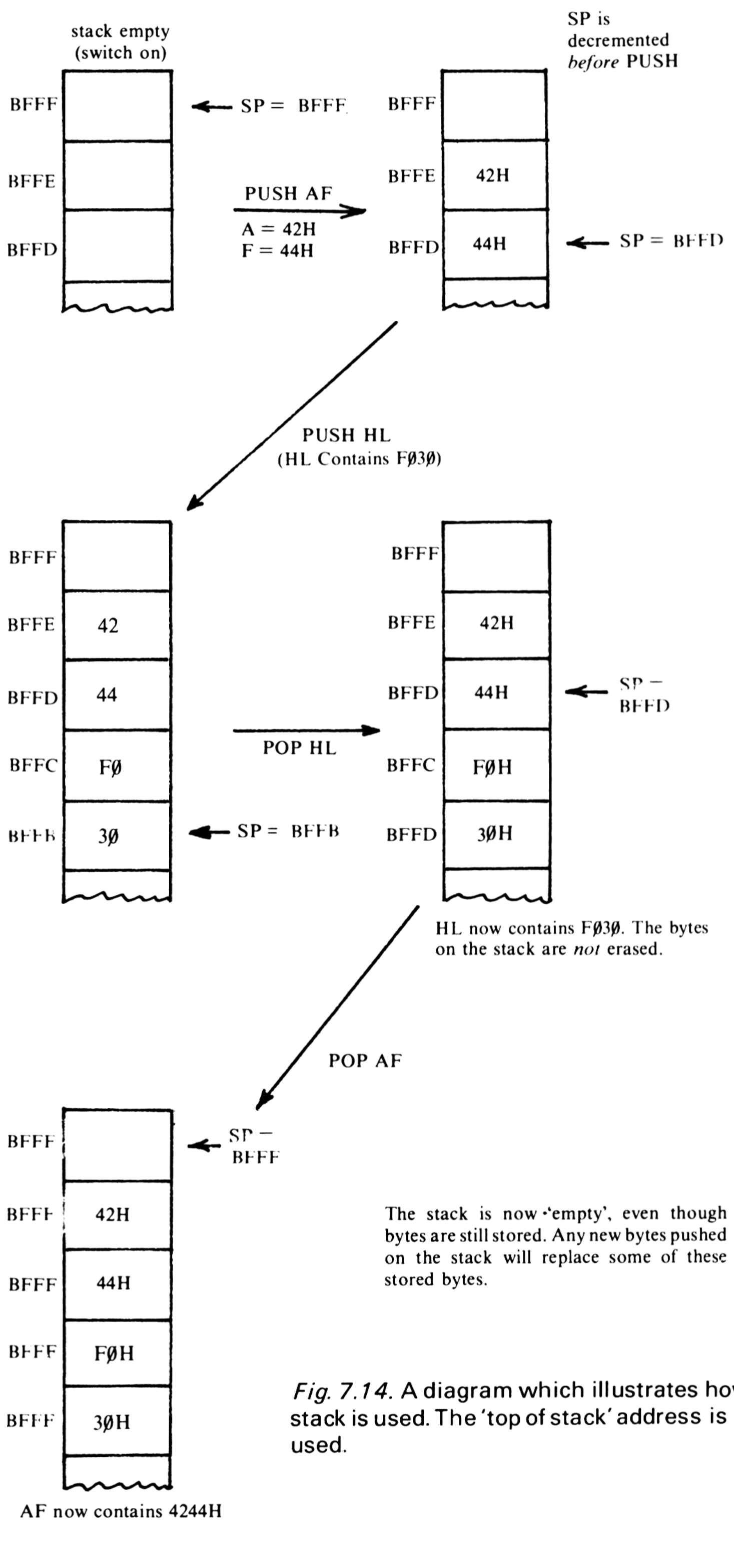


Fig. 7.14. A diagram which illustrates how the stack is used. The 'top of stack' address is never used.

```
(a)         LD BC, ØF1ØØH
            LD B, ØFFH              ← this replaces byte F1H
                                    in 'B'
   LOOP:        .
                .
                . (instructions in loop)
                .
            DJNZ LOOP               ← end of loop
            LD A, (BC)              ← next instruction

(b)         LD BC, ØF1ØØH
            PUSH BC                 ← save on stack
            LD B, ØFFH
   LOOP:        .
                . (instructions in loop)
                .
                .
                .
                .
            DJNZ, LOOP              ← end of loop
            POP BC                  ← get BC contents back
                                    from stack
            LD A,(BC)               ← next instruction
```

Figure 7.15. (a) Incorrect programming, because the byte in B is re-loaded. (b) How the stack can be used to save the correct values and restore them later.

```
(a)                                  (b)
   PUSH BC                              PUSH BC
   PUSH DE                              PUSH DE
     .                                    .
     .                                    .
     .                                    .
     .                                    .
     .                                    .
   POP DE                               POP BC
   POP BC                               POP DE
   (bytes back in correct registers)    (bytes exchanges)
```

Figure 7.16. (a) The normal first-in-last-out use of the stack. (b) Using a different order to shift two bytes into a different pair of registers.

As you might expect, PUSH and POP have to be used with some care, particularly when more than one set of registers is pushed. A command like POP AF, for example will read two bytes from the stack and place them in the AF registers *whether or not they belong there*! For example, suppose that you have a piece of program which looks like Figure 7.16(a). This pushed BC on to the stack, and then DE. When the time comes to pop these, the bytes which will be returned first are the ones that came from DE. The rule of the stack is strictly last-in-first-out, and DE was last in. By using POP DE and then POP BC, the bytes will be restored to the register they came from. If you used the sequence which is shown in Figure 7.16(b), however, the bytes would be interchanged in the registers. The BC register would hold what was originally in DE, and the DE register would hold what was originally in BC. This *can* be used as a way of exchanging register contents.

One very common use of the stack is connected with CALL instructions. When you use a CALL to the ROM or to your own subroutines in RAM, these subroutines may corrupt the registers that you are using. You may, for example, have an important address in HL when you call a subroutine. If the subroutine loads and uses HL, then you are in trouble unless you use PUSH HL just before the CALL and POP HL just after. Because the HL register pair is used so much in many routines, this requirement is a very common one. Another common one is PUSH AF and POP AF, because a lot of subroutines will alter both the A and F registers. *The Concise Firmware Specification* book details the important ROM calls, and shows which registers will have to be protected.

The block commands

While we're looking at a selection of items that have not been dealt with earlier, the block commands of the Z80 are worth mentioning, because we shall be using one of them in Chapter 9. We have already had a hint of one block command in the shape of DJNZ, but there are several more. The DJNZ command is simply a convenient one-byte counter, saving you from having to carry out DEC and CP commands. You have to program for yourself what you put into these loops. By contrast, the other block commands contain register loading and transfer instructions which are carried out automatically. One of the most important sets of these commands concerns the HL and DE registers.

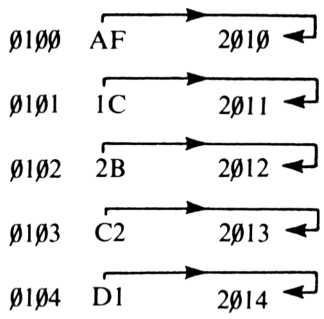
Figure 7.17 shows a summary of the block shift commands of the Z80. Block shift is a good summary of what these instructions are about, which is the ability to move code from one part of the memory to another simply by setting up registers with memory addresses and using a single assembly language instruction. The block shift commands exist in two main forms, automatic or programmed repeat, and in increment and decrement forms of each. We'll examine each of them, one at a time. LDI means Load,

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.

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

The action starts by copying the byte in address 235∅∅ to address 236∅∅, then the addresses are incremented to 235∅1 and 236∅1, and the number in BC is decremented to 4. The transfer is repeated, so that the byte stored at 235∅1 is copied to 236∅1, with BC decremented to 3. The action stops when the content of BC has been reduced to ∅, with 5 bytes copied.

Another example



HL -- ∅1∅∅
DE -- 2∅1∅
BC -- ∅∅∅5

LDIR carries out the transfer of bytes from addresses starting at ∅1∅∅ to addresses starting at 2∅1∅, as indicated by the arrows.

Note: All numbers in hex.

Figure 7.17. The block-shift commands of the Z80 summarised.

Decrement, Increment. When this instruction is used, the HL and the DE registers should each have been loaded with an address, and BC with a count. At the LDI command, the byte at the (HL) address is loaded to the (DE) address, and the counter, BC, is decremented. The HL and DE registers are then incremented. It's up to the program to determine when the count should end, or where the program loops back to. LDD is another version of this in which the transfer from (HL) to (DE) is made, but then both registers are decremented as well as BC. The addition of 'R' to either of these commands makes the repeat action automatic, and also means that you can't choose a loop back address. LDIR, for example, will cause the (HL) to (DE) transfer, and will increment both of these registers, and decrement BC. This action, however, will then continue until BC reaches zero. Similarly, LDDR will transfer and then decrement HL and DE, along with BC, until BC reaches zero.

The block shift commands are particularly useful for shifting a complete group of bytes from one set of memory addresses to another. You can, for example, shift a BASIC program out of the way, so that you can LOAD another BASIC program, RUN this second program, and then recover your first program! You can also store the VLT of a BASIC program, then LOAD in another program which will use the variable values from the first one. Both of these actions can be carried out by reserving memory at the top of the address range (for example, by using MEMORY &A7F7) and using a short machine code program to shift bytes into this protected area. A second piece of machine code can then shift the bytes back. We'll look at one example of this type of action in Chapter 9.

Meantime, there's another type of block action to consider. It's the 'block comparison' action, typical of which is CPI. To set up for this command, you must have an address loaded into HL, a count number in BC, and a byte in the accumulator. When CPI is executed, the byte in (HL) is compared with the byte in the accumulator. If the two match, then the Z-flag is set, otherwise the Z-flag remains reset. The HL address is then incremented, and the count in BC is decremented. It's up to you to use a JR Z or JR NZ step to make a loop out of it. If you use CPIR, then the action of compare, increment HL, decrement BC will be repeated until *either* BC is decremented to zero, *or* a match is found. This is a splendid way of looking through the memory for some specified byte. The other two commands in this group are CPD and CPDR. In the CPD action, the bytes are compared as before, but the HL register pair is then decremented, and BC is also decremented. The CPDR action is the same, except that the action continues until BC reaches zero or a match is found. You must be able to tell which event happened, and in all of these instructions both the Z-flag and the P/O flag (number 2 flag) are used. The Z-flag signals a match found, and the P/O flag signals the state of the count. The P/O flag is set to 1 for as long as the BC register is counting, but the flag is reset (to Ø) when the count is complete. If you find Z−1, P/O=1, then a match was found. If you find Z=Ø, P/O=Ø, then no match was found. One thing that you have to watch is that a JR instruction can test the Z-flag, but only a JP instruction can test the P/O flag directly.

Chapter Eight
Debugging, Checking, and DEVPAC

Debugging delights

Now that you have experienced some of the delights of machine code programming, it seems fair to mention some of the drawbacks. One of these is debugging. A 'bug' is a fault in a program, and debugging is the process of finding it and eliminating it. The guy who puts the bugs into the program is, of course, called a programmer. It all sounds rather insecticidal, but it's nothing like as easy as that!

It's easy to say, I know, but the first part of effective debugging is prevention. Check your program plan or flowchart carefully to make sure that it really describes what you want to do. When you are satisfied with the plan, turn to the assembly language to make sure that it will carry out the instructions of the plan. When you are happy with this, then check that the bytes you intend to poke into memory are the bytes which correspond to the assembly language instructions. One thing to watch very carefully is that you have the correct code for the addressing method that you are using. If you want to use direct addressing, but you have used the instruction byte for indexed addressing, you can't expect the program to work. Remember that when a machine code program doesn't work, the usual result is that the machine locks up. You don't get any of the polite messages that you have in BASIC! If you check each stage in the development of a program in this way, you will eliminate a lot of bugs before they are up and flying. Don't feel that you are a failure if the program still doesn't run – unless a machine code program is very simple, there's a very good chance that there will be a bug in it somewhere. It happens to all of us – and it's only by experience that you can get to the stage where the bugs will be few in number and easy to find.

If you use an assembler, one source of bugs completely disappears. Human frailty means that the process of converting assembly language instructions into machine code bytes is error-prone. That's because it means looking up tables, and anything which involves looking from one piece of paper to another is highly likely to introduce mistakes. I shall briefly describe the action of the GENA3 assembler later in this chapter. At the time of writing, GENA3 was one of two assembler programs that was available for CPC464 machines. It's the assembler which is recommended and sold by Amsoft as part of the DEVPAC set, but there is another excellent available assembler program, called ZEN. I have used ZEN for years on other machines, and I can testify (hand on heart) that it is also a really good assembler program. If you are using ZEN rather than DEVPAC, then this chapter, plus the brief ZEN manual, will help you. If machine code has really caught your imagination, and you feel that you want to branch out into more advanced work than we have space for in this book, then a good assembler/editor/monitor program is an essential, not just a luxury. If, however, you intend to be just a dabbler, spawning the odd drop of machine code now and again, then the poke-to-memory methods that we have used so far will be perfectly adequate. Using these methods, however, means that there will be bugs lurking in each corner of the code. The main cause of these bugs is weariness. Converting an assembly language program into hex bytes, and writing them in the form of DATA lines for a BASIC poke program is a tedious job, and all tedious jobs result in mistakes (ever driven a 'Friday' car?). Faulty address methods are one common result of tedium, and simply writing down the wrong code is another. One very potent source of trouble is with JR displacements. You may get the number wrong somewhere between subtracting addresses and converting a number (particularly a negative number) to hex. Another problem arises when you modify a program, and add code between a jump instruction and its destination. Having done that, you then forget to alter the size of the displacement byte! This is a problem which simply doesn't arise when an assembler is used. An incorrect jump will nearly always cause the computer to lock up. You will probably then have to switch off and on again, because the CLR-SHIFT-ESC keys do not always allow recovery from endless loops. You'll be thankful then that you remembered to record your program before you ran it! Another form of incorrect jump is doing the opposite of what you intended, like using JR Z in place of JR NZ or the other way round. Careful thought about what the jump will do for different sizes of bytes should eliminate this one.

A lot of problems, as I have already said, can be eliminated by meticulous checking, and it pays to be extra careful about branch displacements, and about the initial contents of registers. A very common fault is to make use of registers as if they contained zero at the start of the program. You can never be really certain of this. It's safer, in fact, to assume that each register will contain a value that will drive the computer bananas if it is used. With all that said, and with all the effort and goodwill in the world, though, what do you do if the program still won't run?

There's no simple answer. It may be that your program planning doesn't do what you expect it to do, and if you didn't draw up a plan, then you have got what you deserve. It may be that you are trying to make use of a CPC464 ROM routine and it doesn't operate in the way that you expect. In particular, you have to be careful of corruption of registers when you make use of these routines. A lot of them will change the contents of every register that you are using, so that a number of PUSH and POP actions will need to be included in your program. *The Concise Firmware Specification* notes the conditions for all of the published ROM routines, but if you are using a ROM routine which is not noted in this manual, you are likely to find that this is a matter of trial and error. All I can do here is to give you general guidance on removing the bugs from a program that seems to be well-constructed but which simply doesn't work according to plan.

The first golden rule is never to try out anything new in the middle of a large program. Ideally your machine code program will be made up from subroutines on tape, each of which you have thoroughly tested before you assembled them into a long program. In real life, this is not so easy, particularly when the subroutines exist only as DATA lines for BASIC poke programs. As usual, users of an assembler have the best of it, because they can keep assembly language instructions stored like BASIC programs, and merge and edit them as they choose. The next best thing to keeping a subroutine library on tape is to have extensive notes about subroutines. In addition to routines of your own, you can keep notes on routines which you have seen in magazines. *Personal Computer World* runs a series called *SUBSET*. This consists of several general-purpose machine code routines each month. Most of these are for the Z80, though nowadays a lot are for the new sixteen-bit chips. Another excellent source of useful subroutines is the book *Z80 Machine Code for Humans*, which is by Alan Tootill and David Barrow, the joint editors of *SUBSET*. Even if you don't use many of the routines, the way in which they're documented should give you some ideas about how you should keep a record of your own routines – I personally would buy *PCW* for *SUBSET* alone! If you are going to use a new routine in a program, it makes sense to try it out first on its own so that you can be sure of what has to be in each register before the routine is called, and what will be in the registers after. Look at the examples in *SUBSET*, and see how well this information is presented.

Planning of this type should eliminate a lot of bugs, but if you are still faced with a program that doesn't work, and which you don't want to have to pull apart, then you will have to use breakpoints. A breakpoint, as far as the CPC464 operating system is concerned, is the byte #C9 (denary 2Ø1) replacing another instruction byte. This is the RET byte, and its effect is to return to BASIC. When you are back in BASIC, you can examine the contents of memory by using PEEK instructions. The principle is to pick a point in the program at which something is put into memory. If you place a #C9 byte following this, then when the program runs, it will return to BASIC immediately after the #C9 instruction is executed. By using a PEEK, you can then check that what has been loaded into the memory is what you expect. If it isn't, you should know where to look for the fault. If all is well at this point, then substitute the original byte that belongs in place of the #C9, and place the #C9 at the next place following a memory store command. The thing that you must be careful about when you do this is that #C9 must replace an *instruction* byte, not a data byte.

The most awkward fault to find by this or any other method is a faulty loop. A faulty loop always causes the computer to lock up, which means the usual CTRL SHIFT ESC sequence, or even switching off and reloading. If you are using a disk system, this is just about tolerable, but if you use cassettes, it can waste a lot of time. The main cause of this sort of thing is a loop back to the wrong position. For example, if we had a program, part of which read:

```
       LD B,#FF
LOOP:  DJNZ
       LD A,#A65Ø
```

we could encounter problems. Suppose that this was assembled by hand, and we made the branch back to the LD B,#FF instruction rather than to the DJNZ instruction. This would result in the B register being kept 'topped up', and never decremented to zero, so that the loop would be endless. A mistake like this is easily spotted in assembly language, because the position of the label name is easy to check. It is very much more difficult to find when you have only the machine code bytes to look at. As always, taking care over loops is the only answer, and the method that has been shown in this book of calculating and checking displacements is a good precaution.

Monitors

I mentioned monitors briefly earlier on in this chapter. This has nothing to do with a TV Monitor, which is a sort of superior quality TV display for signals. A monitor in the software sense is a program, one which checks (or monitors) each action of a machine code program. A monitor is (or should be!) a machine code program which can be put into the memory at a set of addresses that you aren't likely to use for anything else. Once there, a monitor allows you to display the contents of any section of memory (in hex), alter the contents of any part of RAM,and inspect or alter the register contents of the Z80. These are the most elementary monitor actions, and it's useful if a section of program can be run, breakpoints inserted, and registers inspected on a working program. The ideal monitor is one which can carry out the steps of a machine code program one at a time, displaying the register and memory contents at each step. The GENA3 assembler comes with a monitor package called MONA3 that does almost all that we would like, and so we'll look at the monitor facilities as we come to them. The combination of the assembler GENA3 and the monitor MONA3 is sold by Amsoft as the DEVPAC (Ref. SOFT 116).

Starting with DEVPAC

The GENA3 assembler is a large chunk of machine code, which sits in addresses starting at a place that you can specify, between 1ØØØ and 3ØØØØ denary. When you buy GENA3, you get the program on cassette, with instructions on how to load it, and the manual. The manual is quite a good one, and it's simple to follow if you have used an assembler before. You wouldn't be reading this book if you had, so we'll look at the processes in detail here!

Now it's one thing to write down instructions, quite another to enter code at the keyboard and make it work, so let's go through the steps of loading and using the program. You'll find that you need to do this rather a lot, because if one of your programs contains an endless loop, then the act of using CTRL SHIFT ESC to break out will also wipe out GENA3. You may feel, when you have rather more experience, that you will want to make another copy of GENA3, using the fast load speed of the tape, or on disk. In any case, it's always wise to have a backup copy of any tape which is expensive and which might take some time to replace.

Using GENA3

Now if you have a copy of GENA3, it's time to take a look at it in action. You don't have to do anything special to load GENA3, except to specify a starting address in memory after the first block of the tape has loaded. This BASIC piece of program simply reserves memory, and creates a space for loading and saving bytes for the cassette system. It then loads the main machine code section at the address that you have specified. Throughout the rest of this book, we'll assume that you have specified address 1ØØØ, which in hex is Ø3E8. We'll now go through the routine of using the GENA3 assembler to generate code for one simple program. Once you have been through this routine of using GENA3, you can repeat it for other programs until you feel confident in using this excellent assembler. It's like buying your first Black & Decker, you wonder ever after how you could possibly have managed without it! Load GENA3, using the address 1ØØØ when you are prompted. When the tape has loaded, assuming that all is well, you will see the screen clear, and the HISOFT copyright notice, along with the list of commands. Below this is the 'sign-on' indication (an arrowhead >) with the cursor immediately following the arrowhead. This arrowhead is always used to mean 'instruction, please', it's a signal that GENA3 is waiting for you to issue a command. Suppose, then that we take the program of Figure 6.9, the delay loop. It's a simple start, and we know already that it works, so we are following the main rule of machine code programming – don't try too many new things at once!

To enter this lot into the GENA3 assembler, we first type I (for INSERT), followed by 1Ø,1Ø and then press the ENTER key. All of the GENA3 commands need the use of the ENTER key, so that you can correct an entry by using the backspace if you have second thoughts. The result of pressing ENTER this time is that the >I1Ø,1Ø command moves up, and a 1Ø appears, several spaces in from the left-hand side, waiting for you to type the first line of the assembly language program, which should be ORG #A7F8. This indicates where the assembled code has to be placed in the memory. While you have nothing but GENA3 in the memory, you don't have to protect any addresses by using MEMORY, but you will have to remember to do so if you use your programs along with BASIC. Addresses from #Ø3E8 to about #274B are used by GENA3, so you should not attempt to put ORG addresses in this range. You will be reminded if you do, however. If you use ENT $ in place of ORG, the assembler will place your code at the end of the memory that it uses. You *must remember* that in an assembly language program GENA3 expects a hex number to start with the hash-mark, #. It won't accept the & in front of the number, as the CPC464 machine uses in PEEK and POKE instructions, nor will it accept numbers like AØØØ or F2. All commands must be in upper-case letters. The commands are the normal Z80 mnemonics. If you find that the assembler will not accept a mnemonic, then you have a faulty tape. It's important to check this, because a faulty copy may work perfectly in all other respects. When you've typed the ORG #A7F8 command, press ENTER to put it into memory. It goes into a section of memory (called the "file"), which is controlled by GENA3. Nothing is loaded into the section of memory which starts at #A7F8 denary at this stage, because GENA3 does *not* assemble (create machine code) until you command it to do so.

Once you have entered this line, the screen shows a 2Ø underneath the 1Ø. It's rather like typing a BASIC program with AUTO line numbering, and the 'I' command works just like the AUTO command in BASIC. GENA3 is now waiting for you to type the next line, LD BC,#FFFF and (ENTER). Be careful about spacing – GENA3 expects one space between LD and BC, so that it can distinguish between the operator and the operand. Don't forget the hash mark to indicate hex numbers. Keep typing now, but watch the spaces and commas, which are so important. The label word is typed with a colon following it, just as we write it in assembly language. The last line is the RET, and when you have entered this, you can press CTRL C to return to the waiting state with the > prompt showing.

What exactly do we have now? We have a set of characters in a piece of memory – the buffer – that's all. We can re-examine these characters from ORG to RET, modify them, edit them, add to them as we want. We can select one line or several lines to view, and make any changes we want, just as we could on a BASIC program; the main difference is that the instructions are different.

GENA3 allows you to list any line or set of lines, or the whole program as you choose by using the 'L' command. Try typing L (ENTER) at this stage, and you will see your program neatly set out. You can change a line in several ways. One is simply to type the line number again, then a space, then the corrected command. You *must* remember the space between the line number and the command. Another possibility, when the > prompt is visible, is to use the cursor keys. You can use the up arrow to place the cursor at the start of a line, and then copy the line using the COPY key. Note that *you don't see the copy of the line*, only the cursor position. When you get to the incorrect part of the line, you can make a correction by typing. This typing will over-write what is already there, so that your new characters replace the old ones. The other editing method is to type E followed by the line number. This places the offending line on the screen, and lets you use the range of editing commands that is detailed in Figure 8.1.

Let's assume, then, that it's all as it should be. You can find how much memory has been used by this set of instructions, the 'text-file', by typing X (ENTER). For this example, I obtained the address numbers 8836 to 8911. Note that these *are denary numbers*, not the hex numbers that GENA3 displays for other purposes. This is rather confusing unless you are on your guard. Now how do we get GENA3 to generate the machine code and run it? The first step is to make GENA3 assemble the program – convert the typed instructions into machine code. The GENA3 command for that is A (for ASSEMBLE), (ENTER). It doesn't happen right away,though. The phrase

After using E5Ø, the line number 5Ø appears at the bottom of the screen, followed by a cursor. The editing keys have the following effects:

SPACEBAR	; moves cursor along line until text appears; causes no changes
DEL	; moves cursor back
ENTER	; ends editing, confirms all changes
Q	; ends editing, restores line as it was
R	; start again with unchanged line
L	; list rest of line so that you can see it
K	; delete the character under the cursor (invisible)
Z	; delete all characters from cursor position to end of line
I	; start inserting characters. The cursor changes to an asterisk. Ended by pressing ENTER once. To end editing you have to press ENTER again
X	; move to end of line *and then* insert
C	; replace characters. The character under the cursor is replaced by the character that you type until you press ENTER. The cursor changes to a + shape while replacing

F and S are less useful 'block editing' commands

Figure 8.1. The commands which can be used to edit a line after typing, for example, E50.

Table size? appears, asking you how much memory you would like to be allocated for the symbols that are used in the program. Normally, you can ignore this, and press ENTER. The next request is for Options:, meaning whether you want the assembly to appear on the video screen or to a printer or in what form you want it. You can also ignore this option, and simply press ENTER. This is the simplest option, which shows any errors, and which will display a full assembly if no errors exist. At this stage in your experience, always go for this 'default' option of video-screen assembly, so that you can see what happens. The instruction lines, along with the code that they generate, will then appear on the screen, and should look as shown in Figure 8.2. We'll deal with the other options later.

```
Hisoft GENA3 Assembler. Page     1.

Pass 1 errors: 00

A7F8               10          ORG   #A7F8
A7F8   01FFFF      20          LD    BC,#FFFF
A7FB   0B          30 LOOP     DEC   BC
A7FC   78          40          LD    A,B
A7FD   B1          50          OR    C
A7FE   20FB        60          JR    NZ,LOOP
A800   C9          70          RET

Pass 2 errors: 00

Table used:     24   from    109
```

Figure 8.2. A typical assembly as it appears on the screen.

Because you included the instruction ORG #A7F8, the code which was generated by the assembler will be displayed at addresses that start with #A7F8. These codes, however, *have not been placed in the memory*, only printed on the screen. Before you do anything else, look again at the assembled program. Notice that no code has been generated for the ORG command. That's because this, along with several others, is a command to the GENA3 program, not part of our machine code that has to be assembled. The addresses are shown on the assembler listing and the code is actually put there only if the ENT $ command follows the ORG line. The addresses in the listing are in hex, and when you have used ENT $, the codes are entered into memory, and an 'execution address' is given. The 'execution address' which is shown below the listing (when ENT $ has been used) is in denary – it's the equivalent of the first address of the program in this case. If no execution address appears, it's because *no code has been put into memory*.

Now there are a lot of tricks and additional options that we could employ even on such a straightforward assembly. At the moment, though, all we're interested in is the few bytes that start at address #A7FB. These should be the bytes of code that were generated by the program, and you can use these in a BASIC poke version if you want. Alternatively, you can add a new line:

```
15 ENT $
```

to make the next assembly put the bytes into memory. When you do this, you will see the 'Execution address' notice appear following the listing, and you can run the program by typing 'R' (ENTER).

We can look now at some of the other messages that accompany the assembly. The first line carries the Hisoft copyright marking, and a page number. This allows long programs to be neatly divided into (long) pages if you are using a printer. If you are not using a printer, then the assembly is continuous. If you want to see the code in screenfuls, however, you can stop assembly at any point by pressing a key – the spacebar is a good choice. The assembly can be continued by pressing the spacebar again.

The next message in the assembly is 'Pass 1 errors: ØØ'. The assembler works by going over the instructions (or 'source text') twice. In the first pass, no code is generated, because the assembler is looking for errors, and also creating a table of all the symbols (such as labels) that are used. If no errors are found on this pass, the assembler will then continue, and will then generate code. What follows is then the assembly code. The left-hand column contains the address, starting with the address that was selected by ORG. This address will be shown against the ORG line and the ENT line, and then the code starts in line 2Ø. The second column now shows what code the assembler has generated, the third column shows the line number of the assembler listing, the fourth column shows label names, and the remaining columns show the assembly language. You will find that when this appears on the ordinary forty-column screen, some of the assembly language and any comments will overlap on to another line. If this becomes annoying, you can switch to 8Ø-column display by pressing the 'W' key (then ENTER) when the assembler is waiting for a command. Unfortunately, the 8Ø-column display of the CPC464 is not too easy to read. You can switch back to 4Ø-character display by pressing W (ENTER) again. The assembly will end with another error report, 'Pass 2 errors: ØØ'. It's rather unusual to get any error report in Pass 2 – one certain way is to pick an ORG address which will make the assembled code use the addresses that are occupied by the assembler program. Finally, you get a report on 'Table used', the number of bytes that were devoted to making a table of labels, how many bytes were allocated, and finally, the Execute address. This will appear only if ENT was used, and its value *is in denary* – don't ask me why! Now that we know for certain that the program exists in the memory (and believe it or not, some assembler programs require you to record it on a cassette and load it back in just to do this!), we can run it. At the moment, if we run it, the RET command will make it return to GENA3.

The GENA3 command for 'RUN' is simply R, so we type R, and (ENTER). No screen messages appear – you will simply notice a short time delay before the arrowhead and the cursor appear again. Now whether you use a short and simple program like this one, or one which has hundreds of instructions, the steps of loading GENA3, typing in the assembly language program, entering it into the buffer, assembling and testing, are pretty much the same. All that differs significantly is the actual assembly language, so we're going to concentrate on that from now on, with no BASIC POKE programs. If you do not have GENA3, then you can still read the hex listings, which I have printed with GENA3, and you now know enough about poking hex bytes into memory to carry this out for yourself. The steps in the table of Figure 8.3 should help to remind you of what has to be done, but if you want to keep your computer switched on with GENA3 in place, you can run through several lots of assemblies without any need to change anything. When you have finished with one assembly for good, just type D and follow it with the first line number, a comma, then the last line number. When you press ENTER, the 'text buffer' will be cleared, and GENA3 will be ready to start a new entry of assembly language. If, of course, you want to preserve your first efforts at assembly language, then GENA3 provides for this also. Before you Kill the program, insert a cassette in the recorder, and wind it to a blank piece of tape. Press the RECORD and PLAY keys of the recorder, then type P followed by the first line number, last line number and filename. These should be separated by commas, so if you are recording the time delay program, you would use P1Ø,7Ø,DELAY. You don't need any inverted commas around the filename. When you press the ENTER key you will see the usual message about pressing REC and PLAY, and then any key. When you do this, the recorder will start, record the whole of the assembly language program, then stop. At any time, when you have GENA3 loaded into your machine, you can play this back by typing G,, (two commas) then typing the filename and pressing ENTER, or just pressing ENTER alone. Note that this recording can be read *only* by GENA3, and it

1. Decide start address (ORG), A.
2. Reserve memory by using MEMORY = A − 1 where A is starting address.
3. Count number of bytes, starting with zero.
4. Program loop Ø TO COUNT and use READ D$ followed by POKE A + LP, VAL ("&" + D$) where A = start address, LP = loop counter.
5. Put hex codes in DATA line.
6. Start machine code by CALL A (A is start address).

Figure 8.3. A reminder of how the bytes of a program can be poked into memory from BASIC.

is *not* a recording of the machine code bytes – that's made in a different way. The recording that is made using P and read by G is of the assembly language text, and it's in ASCII code. You cannot read it with the ordinary LOAD command of the CPC464 computer, because it is not a BASIC program. Saving this text allows you to reload your assembly language at any time so that it can be modified into a different program. You should *always* save any assembly language that you make in this way. With your assembly language in place, make a recording, using P, and then rewind the tape. Now type V,, (two commas following V) for Verify. You should then enter the filename, and when you have done this, press ENTER. You will be prompted to press the PLAY key on the tape recorder as usual. You should see the program name appear, and when the message 'Verified' appears, you can be sure that the recording is good. If it isn't, then you will have to try another recording. The cassette recording routines of the CPC464 are very good, and it's most unlikely that you will have any trouble.

You can also use GENA3 to create a machine code tape. This is a recording of the bytes of the machine code program only, the type of program that can be loaded by using LOAD in BASIC. To make this type of tape, you must first of all have created machine code in the memory by assembling your text using the ENT command following ORG. When this has been done, the machine keeps a record of the start, finish and execution addresses of the program. The execution address is the address of the first *command* of the program, and for a lot of programs it will be the same as the starting address. Many programmers, however, like to start their programs with a table of bytes, and the execution address may be several bytes into the program. The code is then saved by typing the letter O,, (two commas again) then the filename, and pressing ENTER. You then go through the normal recording routine as prompted.

To test this form of writing, use the same delay loop program as before, and assemble it. When the program has been assembled, prepare a cassette with some blank tape. Type O,,DELAY (ENTER) and then press RECORD and PLAY on the recorder when you are prompted. The machine code program will then be recorded. To check this, you can use the same verification command as you used for the text-file. You can also check the program by using it from BASIC. Type B (ENTER) to return to BASIC from GENA3. Now clear the memory that was used for the code by typing:

```
FOR N=&A7F7 TO &A8Ø1:POKE N,Ø:NEXT
```

and pressing ENTER. Then rewind the cassette, type LOAD"DELAY" (ENTER) and load in the usual way. The code should enter, and you can prove that it has done so by using CALL &A7F8. This should return to the 'Ready' prompt after a short delay. If you want to return to GENA3, you can do so with a CALL. This should be to an address 4 steps above the address at which GENA3 was loaded. If, for example, you loaded GENA3 to address 1ØØØ, then you can use CALL 1ØØ4 to get back to it from BASIC.

Other commands

GENA3 is such a comprehensive assembler/editor that it needs a lot of space just to describe the features that are extra to the essentials that we have dealt with so far. We can start with the editor commands, of which we've used only E (Edit), I (Insert), D (Delete) and A (Assemble) so far. You can renumber the lines of your assembly language by using Na,b followed by ENTER. In this case, a represents the starting line number (usually 1∅ or 1∅∅) and b represents the step, which is usually 1∅. If you have some text already in the memory, and you use the 'G' command to load more text from tape, then this new text is *loaded so that it follows the existing text*. In other words, new text from tape does not replace old text, unlike the action of LOAD in BASIC. In addition, when this is done, the whole of the amalgamated text will be renumbered, starting at line 1, and in steps of 1. Because of this, the renumbering command is often very useful. These are the commands of the editor section of GENA3 which you are most likely to use each time you assemble a program. There are many other commands which you may use from time to time. Since these are explained in the manual, and since you won't really need them until you are a fairly experienced programmer, we'll leave them out here.

There are still a number of the assembler commands which we have not looked at so far, and which are useful. To start with, there are the assembly options. These are coded by number, so that by adding the number codes you can specify more than one option. The options are listed in the Manual, and also in Figure 8.4. At this stage, Option 1 is useful. This causes the assembly to be followed by a listing of symbols. This is *in alphabetical order* and shows each label name followed by the address or byte to which it refers. This can be very useful when you are assembling a long program. Option 2 prevents any code from being generated. This over-rules ENT, so that you will see addresses and bytes appear, but no code is placed in memory. Option 4 suppresses the assembly listing. Because screen actions on the CPC464 are slow, this greatly speeds up assembly. It's a good one to use when you are trying to assemble a long listing for the first time. Error reports

1 –	Produce symbol table printout following assembly.
2 –	Do not generate any code.
4 –	No listing of assembly language or code.
8 –	Listing on printer, not on screen.
16 –	Put code in RAM following GENA3, but show addresses specified by ORG.
32 –	Turn off checking routine to speed assembly of very large programs.

These numbers can be added to produce more than one effect.

Figure 8.4. The assembly options of GENA3.

will be printed as usual, but no assembly. You can then re-assemble with a listing when the program is sorted out.

Option 8 allows the listing to go to a printer. The CPC464 which I used sent two line feeds to the printer so that it always placed two spaces between lines. In addition, the GENA3 program is arranged to that the printer rolls out a lot of paper after a short listing. Figure 8.5 shows how these problems can be overcome when you are using the popular Epson printers (MX-80, RX-80 and FX-80). Option 16 places the machine code immediately following the text in memory, starting beyond the end of the symbol table. This option is not quite so likely to be used as the others. Finally, option 32 speeds up assembly by skipping the action of checking where code is loaded. Combining this with Option 4 (no listing) gives the number 36, and when this is used, the assembly is very fast, but with no listing on the screen.

In addition to the options that you can specify for assembly, you can place instructions within the assembly code itself. These commands consist of letters (and possibly other data) which are placed in the text with an asterisk preceding them. Once again, a lot of these are more useful to you when you are working with long programs. For example, putting in a line with *E in it forces the assembly to skip three lines on the screen. If you are listing to a printer, a new page will be taken at this point. If you used a line with *H early on, you can cause each page to carry a heading. For example, the line:

```
5 *H MY DELAY
```

will cause each page of the listing to carry the heading of MY DELAY. A line with *S will cause the assembly listing to halt at this line. It's useful, as the manual notes, if you particularly want to find what address has been used at this point. Of the other commands, the *T is interesting. This allows a machine code (object) tape to be made as the program assembles. When you want to write very long programs, the *F command allows an assembly language text tape to be *read and assembled* in the middle of another assembly. You might, for example, want to place a short routine called PRINTY in several places in a longer program. By placing lines such as

```
1ØØ *F PRINTY
```

in your listing, this can be done. When the assembler comes to line 1ØØ, you

```
1Ø PRINT#8, CHR$ (27); CHR$(69)
2Ø PRINT#8, CHR$(27); "Q", CHR$(80)
3Ø PRINT#8, CHR$ (27); "A"; CHR$(7)
4Ø PRINT#8, CHR$ (27); "C"; CHR$ (5)
```

Figure 8.5. How to configure an Epson printer so as to make use of GENA3 output. This BASIC program switches to bold type, 80 characters per line, narrow line spacing, and short gaps.

will get the 'Press PLAY then any key' message. You can then put into the recorder a tape with the PRINTY routine on it, press PLAY, then any key, and the tape will load. More important, the listing on the tape *will be assembled as the tape is read* and the code put into memory. This means that the text which is on the tape does not have to be put into the memory, taking up valuable memory room. You can use this to generate very long programs whose text could not be fitted into the memory.

The MONA3 monitor

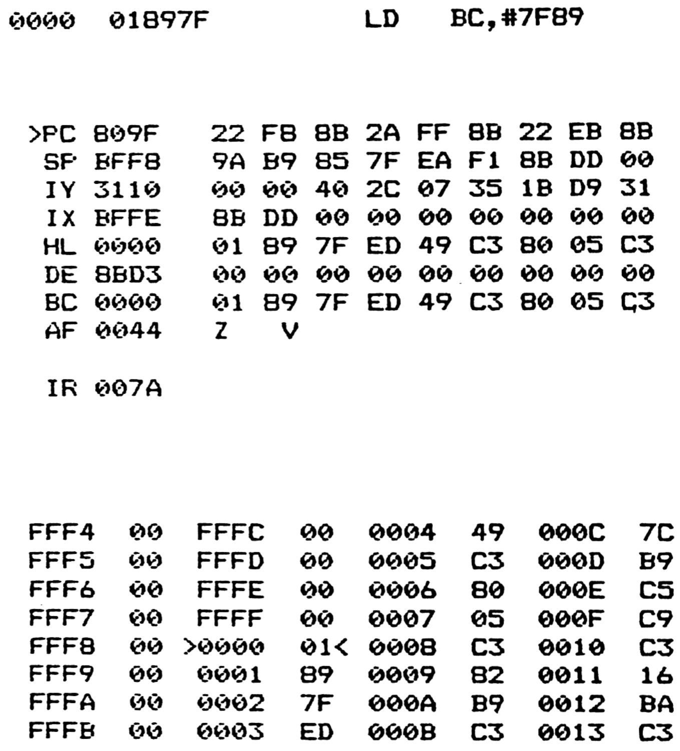
The monitor commands of the companion program MONA3 allow you to work with a program that is in machine code, as distinct from one which you have in assembly language form. This monitor program is extremely valuable for finding faults in a program that has been created by GENA3, and it is a pity that the functions could not have been combined. As it is, though, you can have both programs in the memory at the same time. This restricts the size of program that you can work with, but allows you to carry out tasks which would otherwise be difficult or impossible. This includes, for example, the disassembly of the CPC464 ROM routines, and adding such disassembled text to the text in a GENA3 file. In addition, the MONA3 routines allow you to test and follow the action of a machine code program, step by step if need be. You can look at the contents of memory and also at the contents of the Z80 registers while you execute a program step by step, so that faults are very clearly visible.

You have to start by loading the MONA3 program. The Manual advises you that if you want to use GENA3 and MONA3 together, you should load MONA3 first in high memory (around 3ØØØØ denary) and then GENA3 next in low memory (around 1ØØØ). Both MONA3 and GENA3 have commands which allow you to switch from one to the other, but these do not work unless they have been correctly set up. Suppose, for example, that you load MONA3 first. Like GENA3, this uses a BASIC loader, and you are asked for a load address. The manual suggests 3ØØØØ, and this should be used unless you have any pressing reason to use a different address. The program will then take up addresses to about #8FCC, leaving plenty of room for all but the longest type of machine code programs in the space above #8FCC. When this has loaded, MONA3 runs, displaying a set of addresses and their contents. If you want to use MONA3 by itself, you can go ahead, and this will be dealt with shortly. If you want to use GENA3 with MONA3, however, you will now have to load GENA3. Return to BASIC by pressing CTRL X, and then load GENA3 in the usual way, with a load address of 1ØØØ (denary). When this has loaded, you will be in GENA3, and you now have to set up the 'J' key actions that transfer you from one program to the other. The Manual suggests that if you have loaded MONA3 and used it, then when you have loaded GENA3, you should be able to get to MONA3 from GENA3 by pressing the 'J' key. It's not quite so straightforward. The best way to set up these actions, assuming that MONA3 is at 3ØØØØ and GENA3 is at 1ØØØ, is to use 'B' to return to BASIC from GENA3. Now enter MONA3 by typing CALL 3ØØØ2 (ENTER). This takes you to MONA3, and you should return to BASIC again by pressing CTRL X. Now use CALL 1ØØ4 (ENTER) to enter GENA3. From now on, everything works as the Manual says it should. In GENA3, pressing J (ENTER) will take you to MONA3. From MONA3, CTRL J will take you to GENA3. When you do this, only the cursor can be seen, and you will have to press 'H' (ENTER) to get the GENA3 menu. With both programs set up like this, you can tackle virtually any form of trouble-shooting on machine code programs. Remember, though, that if anything goes wrong and you have to reset the computer using CTRL SHIFT ESC, you will have lost both programs.

Using MONA3

When the MONA3 program starts running, the screen display is of the type that used to be common on large computers, a 'front panel'. This is a display of register contents and of a selected area of memory, as illustrated in Figure 8.6. When you first switch to MONA3, the part of memory that is shown on the lower part of the screen is centred around address ØØØØ. This is ØØØØ in the RAM, *not in the ROM*, because the operating system automatically selects RAM. The upper part of the screen shows a list of double-byte registers and register pairs and the two-byte number that is stored in each. For the registers and pairs of PC to BC, the display also shows the byte which is stored at the address in the register, and a further eight bytes on. For example, you will see the HL address of ØØØØ with the byte Ø1 and the list of the next eight bytes. This list should agree with the list that is shown in the memory display. The first instruction is shown in disassembled form at the top of the screen. For the AF pair, the display consists of the bytes in AF, and the flags that are set. This is because the number in AF can *never* be an address. This 'front-panel' is the display that MONA3 will return to after completing any other command. Its importance is that it allows you to see what is contained in the registers and in any selected part of memory. Let's look, then, at a few applications of MONA3 to machine code programs.

Very often, you may have a long machine code program in which you want to alter only a few bytes. It would be very time-wasting to try to make assembly code of it all, and re-assemble. Several commands, therefore, are designed so that you can amend machine code rather than a text file. For all the uses of MONA3, the 'M' command is the key one. When you type 'M', you will see the letter appear on the screen, with a colon following it. The colon is a prompt which is there to remind you that you have to enter an address *which must be in hex*. You can alter the *display addresses* to denary by using CTRL D (use CTRL D also to return to hex), but when an address



```
0000  01897F          LD    BC,#7F89


   >PC 809F   22 FB 8B 2A FF 8B 22 EB 8B
    SP BFF8   9A B9 85 7F EA F1 8B DD 00
    IY 3110   00 00 40 2C 07 35 1B D9 31
    IX BFFE   8B DD 00 00 00 00 00 00 00
    HL 0000   01 89 7F ED 49 C3 80 05 C3
    DE 8BD3   00 00 00 00 00 00 00 00 00
    BC 0000   01 89 7F ED 49 C3 80 05 C3
    AF 0044   Z   V

    IR 007A


   FFF4  00  FFFC  00   0004  49  000C  7C
   FFF5  00  FFFD  00   0005  C3  000D  B9
   FFF6  00  FFFE  00   0006  80  000E  C5
   FFF7  00  FFFF  00   0007  05  000F  C9
   FFF8  00 >0000  01<  0008  C3  0010  C3
   FFF9  00  0001  89   0009  82  0011  16
   FFFA  00  0002  7F   000A  B9  0012  BA
   FFFB  00  0003  ED   000B  C3  0013  C3
```

Figure 8.6. A typical MONA3 'front panel' display.

has to be entered, hex must be used. When you enter such an address, you do not have to bother with leading zeros. In other words, you can enter 1F rather than ØØ1F if you like. When you press ENTER, you will see a display of the hex bytes that are stored in the memory starting at this address. You will also see the contents of address before and beyond your stated starting point. This action is called 'setting the memory pointer'.

Once the memory pointer is set, you can take advantage of all the actions of the monitor. Suppose, for example, that you want to alter the byte at the memory pointer address. All you have to do is to type the new byte *in hex* and then press ENTER. The display will then show the effect of the change. You can then move the memory pointer forward or backward by one address number using the cursor left and right keys, and change other bytes. If you want to move the memory pointer by eight bytes, you can do so with the cursor up and down keys.If you don't want to change a byte, you can simply press ENTER, which will leave the byte unchanged, and move to the byte at the next consecutive address.

The type of display that you get using 'M' alone may not be informative enough, and you can get a more extensive hex listing by pressing 'L'. You don't have to use ENTER for this command, and the result is a display of

16Ø bytes, starting at the memory pointer address. The bytes are shown in hex, with the starting address for each row of eight at the left-hand side of the screen. On the right-hand side, the characters that correspond to the ASCII codes of the bytes are displayed. For a machine code program, this display is meaningless, but if any messages are present in the program, they will be revealed. This is, for example, an excellent way of checking that any message bytes in a program are correct, and it's a lot quicker than wading through lots of codes. You can also use this to spot the positions of BASIC keywords in the ROM. Pressing any key will continue the listing, and you escape from it by using the ESC key. If you have a printer connected, you can use CTRL L to get a printed listing.

If hex listing is not enough, you can display a disassembly by using CTRL A. The disassembly shows what assembly language will produce the hex listing, with sixteen lines produced at a time. The disassembly starts, as usual, at the memory pointer address. To continue the disassembly, press the spacebar (or any letter key); to return to the 'front panel' display, press CTRL A again. This disassembly is a 'dumb' disassembly, in the sense that each byte will be treated as if it were machine code. You can obtain a more elaborate disassembly by using the 'T' command. This asks for start and finish addresses, and allows a printer option. More important, it allows you to specify if any of the code that you will be looking at is ASCII text – something you would have discovered by using the 'L' option. This is done by filling in values for 'First' and 'Last' addresses of the text areas. Even more important than a disassembly is the fact that this command can be used to generate a textfile that *can be put directly into the GENA3 memory space*. This is one to remember when you have shed your L-plates.

Analysing a program

The main use that many owners will find for MONA3 is sorting out a program that refuses to do what it is supposed to. This is something that MONA3 does better than most monitors; as well, in fact, as some of the best monitors I have ever used. To start with, MONA3 allows you to run any program that you happen to have in memory, loaded from tape or placed in memory by GENA3. Before you do so, though, you may want to select a breakpoint, meaning, do you want to break the program at any place to examine what it has done so far? If the program is faulty, you will not wish to let it run completely in case of an endless loop. You have to select the position of a breakpoint by moving the memory pointer to an instruction, then pressing the ! key. For example, suppose that we are investigating the delay program, and that we want to stop it at the OR C stage. Now the OR C instruction is at address #A7FD, so we use M:A7FD to set the memory pointer, and then press the ! key. This replaces the OR C instruction by a jump back to MONA3. Now you can run the program by typing, which shows as J: on the screen. Now type the starting address of A7F8, and press ENTER. The program will run, stopping at address A7FD.

Now how do we know that it has run and stopped at the correct place? The screen will be blank, with a line cursor showing at the top. Now hit any key to bring up the 'front panel' display again. This allows you to examine the registers of the Z80. You will find them neatly set out, and beside the letters PC you should find A7FD. This is the address at which the program switched back to MONA3, because you put the breakpoint there. The only other registers with anything of interest are BC, with FFFE, and AF, which contains FFØ4. This means that the BC register pair has been decremented, and that B has been copied to A. The F register contains Ø4 (hex numbers, remember), which means that flag 2 is set. This is the P/O flag – not important in this program.

Now try again, but this time set a breakpoint at A7FE. When you examine the Z80 registers this time you will find a slightly different picture. Once again, BC contains FFFE and the A register contains FF, as it should after LD A,B. The F register, however, now contains AC, because the OR C action has set another flag, the sign flag. The display indicates this by displaying an 'S' – note that two unused bits of the F register are also set. This should not cause any confusion, because the display shows that only two flags are represented.

This action of running a program to a breakpoint is useful, but it may not do all that you might want. The ultimate method of debugging is to run a program step by step, and MONA3 allows this by using the CTRL S command. This requires careful setup. You can use 'M' to get the memory pointer to the starting address of the program, but this is not enough. You also have to get the PC containing this address. If you look at the front panel display, you will see an arrowhead against the PC display. This means that you can alter this number, and you do so by typing a new address *followed by a full-stop*. In this example, you can type A7F8. With the PC showing the same address as the memory pointer, the program start address, you can then single-step by pressing CTRL S. At each step, the top line on the screen will show the *next* instruction that will be executed, and you can, of course, read the contents of all the registers. Want to see what happens when the loop ends? You don't have to single-step around the loop 65536 times. Just press the full-stop, which will shift the arrowhead on the register display. Keep pressing until the arrowhead is pointing at BC. Now alter the BC address to ØØØ1 by typing ØØØ1. – you *must not* forget the full-stop. If you don't use the full-stop, you can find that you have changed the memory byte, not the register. You can now continue single-stepping with CTRL S and watch what happens when the loop terminates. With power like this over your programs, there's no excuse for having faulty programs! One thing that you *must* watch is that you do not attempt to single step a CALL to the ROM. When the top line of the screen shows such a CALL, press the '>' key, then ENTER, to step over this complete CALL and on to the next instruction in your own program. If you don't do this, the result is inevitably a lock-up.

Two more commands are very useful for dealing with blocks of code. If you want to move a block of code, perhaps from the ROM of the computer into RAM, then you can use I (Intelligent copy).You will be asked for the 'First' address of the block that you want to copy, then the 'Last' address of this block. You are then asked for the 'To' address, which is the starting address at which you want to have the copy. The shift is done when you press ENTER after typing the 'To' address. What makes the shift action 'intelligent' is that you can use a 'To' address which will cause code to overlap the original addresses. In other words, you could start at A7F8, end at A8ØØ and shift to A7FØ, with the last bytes of the shifted program replacing the first bytes of the original. A similar command, P, will fill a specified range of memory with any byte that you type.

Last, but certainly not least, MONA3 allows you to read or write machine code tapes. The read command is R, and you don't have to specify a filename if you want to read whatever is on the tape. You *must*, however, specify the loading address in hex. The write command is W, and you will be asked for the first and last addresses of the code, and a filename. You can, incidentally, save the screen addresses in this way so as to save a screen display (with the 'Start' address of F1EA) but this takes a lot of time. If you ever doubted how complicated the screen layout is, however, it's a good exercise to watch a screen recording replaying! Finally, Figure 8.7 shows what you have to do to switch on a ROM so that you can use MONA3 to examine what is stored there. The example shows the lower ROM, which is the more interesting one, but the same methods can be used for the upper ROM.

Lower ROM
Use M: BA53 to position memory pointer
Press '!' key to put in breakpoint
Use J: BA4F to switch on ROM

Upper ROM
Use M: BA67 to position memory pointer
Press '!' key to put in breakpoint
Use J: BA5E to switch on ROM

Figure 8.7. How to use MONA3 to switch on a ROM so that ROM contents can be examined.

Chapter Nine
Last Round-up

This chapter is devoted to programs, mostly in assembly language, but with the odd one in BASIC. They all have one factor in common, which is that they use knowledge of the internal workings of the CPC464 computer to achieve what they have to do. All of them, then, are useful even if you aren't particularly interested in what they do. Each one of them shows how some useful action can be achieved, or some useful piece of data extracted. Because of this, you will probably find that the programs which *you* want to write can be adapted from some of the ones that follow. I have tried to vary the programs as much as possible, but when a computer system is fairly new, as the CPC464 is at the time of writing, it's not so easy to find out *everything* about it. When the system has been established for several years, it is always much easier to find from magazines, user-group newsletters, and books, what routines are stored in the ROM, where their starting addresses are, and how they make use of the Z80 registers. This is why the older computers are always a machine code programmer's dream! *The Concise Firmware Specification* manual is *very* useful, however, and a must for the keen programmer.

Shift the program

In Chapter 7, I mentioned the possibility of shifting a program written in BASIC to a different part of memory. If this is done, you can carry out tasks like loading and running another BASIC program, and then replacing the original one. This is much faster than recording the first program on tape, and then replaying it afterwards, because tape is slow and not always as reliable as we would like. As well as saving the lines of BASIC, the program which is shown here saves the VLT, so that the variables of the program are saved as well. This allows you to resume the program if it has been interrupted, provided that you use GOTO (with a line number) rather than RUN to restart. If you use RUN, all variable values are reset, and the program starts from scratch. The main problem about such a program is the use of memory, and there's more on that later.

The assembly language for the BASIC saver is shown in Figure 9.1. This, like all of the assembly language programs in this chapter, has been written using GENA3, and printed out by using the number 8 option for assembly. The program is commented to make it slightly easier to follow, but you need not include the comments if you are using GENA3 for yourself. The code is assembled at #753Ø, which is denary 3ØØØØ and consists of 57 bytes, small enough to tuck into any spare piece of memory if you want to put it elsewhere. In its present location, it leaves a reasonable amount of space above it for storing short programs, but you might have to relocate it if you want to use MONA3 at 3ØØØØ. Later on, we shall look at methods of starting such programs by pressing the TAB key, but remember that you can always assign the function keys to CALL statements. You can have KEY139, "CALL 3ØØØØ"+CHR$(13) and KEY14Ø,"CALL 31ØØØ"+CHR$(13) and so on, so that the function keys will start your machine code routines.

Getting back to the program, lines 6Ø to 9Ø are concerned with allocating values, and you can ignore these if you are using a BASIC POKE program with the bytes that are shown in the assembly. The label PTRS indicates the address which is the start of the pointers, in this case #AE81. The DEST address is the address to which a BASIC program is going to be shifted, and NUM is the number of variable pointer addresses to save, ten in all. Finally, BAS is the address of the start of BASIC, #Ø17Ø. The one that you are most likely to want to alter is DEST. This controls where the BASIC is stored. For many purposes, however, the address that is shown in the listing is adequate.

Lines 1ØØ to 13Ø are concerned with saving the pointers to the VLT entries. These pointers are located from #AE81 onwards, and if these are not restored after you restore your program lines, you will get a 'Syntax error' message if you try to run the program again. The BC register is used to hold the count of ten, HL is loaded with the starting address of #AE81, and DE with the first destination address of #757Ø. This address has been chosen arbitrarily – you could use any address that has been protected by a MEMORY statement, and what you choose depends on the size of the BASIC program that you are saving. If you want to save very long programs, you might have to come down to lower numbers, possibly with the machine code located higher in memory. Obviously, if a BASIC program takes up more than half of the available RAM, you can't use this technique. When you are writing and using programs of such a size, though, it's likely that you will be using disks, and saving to disk will be just as easy as saving to memory. The LDIR command in line 13Ø carries out the transfer of the pointer bytes.

Following LDIR, the DE register is left pointing to the next free address in the range we are using to save the BASIC program. By using PUSH DE, we save this address on the stack, ready to use later. The next thing is to find how long the BASIC program, plus VLT, is. To do this, we read the address of the end of the VLT from the last pair of pointer bytes. Because LDIR has

```
                    10 ;Program to shift a program
                    20 ;in BASIC to high memory and
                    30 ;then restore it.
7530                40        ORG  #7530
7530                50        ENT  $
AEB1                60 PTRS   EQU  #AEB1
7570                70 DEST   EQU  #7570
000A                80 NUM    EQU  #0A
0170                90 BAS    EQU  #0170
7530  010A00       100 SAVIT  LD   BC,NUM;get number of
7533  2181AE       110        LD   HL,PTRS;pointers to save
7536  117075       120        LD   DE, DEST;to memory
7539  EDB0         130        LDIR
753B  D5           140        PUSH DE:save address
753C  2B           150        DEC  HL;get last high byte
753D  56           160        LD   D,(HL);into D
753E  2B           170        DEC  HL;now low byte
753F  5E           180        LD   E,(HL); into E
7540  217001       190        LD   HL,BAS;start of BASIC
7543  E5           200        PUSH HL;save it
7544  EB           210        EX   DE,HL;swap registers
7545  ED52         220        SBC  HL,DE; subtract for count
7547  4D           230        LD   C,L;count into
7548  44           240        LD   B,H; BC pair
7549  E1           250        POP  HL;restore address
754A  D1           260        POP  DE ;and the other
754B  79           270        LD   A,C;to save the
754C  12           280        LD   (DE),A;count in
754D  13           290        INC  DE;the memory
754E  78           300        LD   A,B;before
754F  12           310        LD   (DE),A;shifting bytes
7550  13           320        INC  DE; ready now
7551  EDB0         330        LDIR
7553  C9           340        RET
7554  010A00       350 GETBAK LD   BC,NUM
7557  217075       360        LD   HL,DEST;return all
755A  1181AE       370        LD   DE,PTRS;pointers first
755D  EDB0         380        LDIR
755F  4E           390        LD   C,(HL); get the count
7560  23           400        INC  HL; from memory
7561  46           410        LD   B,(HL); into BC
7562  23           420        INC  HL
7563  117001       430        LD   DE,BAS; destination BASIC
7566  EDB0         440        LDIR
7568  C9           450        RET
```

Figure 9.1. The BASIC saver program in assembly language, printed by GENA3. Since the bytes are also shown, you can easily create a BASIC POKE program if you do not use GENA3.

left HL pointing to one address further on from the last one in the pointer table, we need DEC HL to restore the correct address, and we can read the two bytes of the address into D and E, making sure that the low byte is in E and the high byte in D. Line 19∅ then loads the start of BASIC address of #∅17∅ into HL, and this address is saved on the stack by using PUSH HL in line 2∅∅. Line 21∅ then swaps the addresses in HL and DE. This is necessary because we want to subtract, and the only sixteen-bit subtract action that uses these registers takes the DE address away from the HL address, and puts the answer into HL. The subtraction is done in line 22∅, and then lines 23∅ and 24∅ place the result into BC. This number can now be used as a count for the transfer of all of the bytes of the BASIC program and the VLT.

Now we must set up for moving the BASIC bytes. By using POP HL, we get back the address of start of BASIC (#∅17∅) into HL, and by using POP DE, we get the address of the next available piece of free memory. We are now going to save the count number, because we will need it to return the BASIC bytes in the second part of the program. We don't have a command which will load into the DE address from B or C, only from A. We therefore transfer from C to A, and so to (DE), increment DE, then transfer B to A to (DE) and increment again. It looks clumsy, but very few bytes of code are needed. By line 32∅ this has been done, and everything is ready for the transfer. The LDIR in line 33∅ carries it out, and the program returns to BASIC.

After this piece of program has been run with a CALL command, you can then DELETE your BASIC program, or CLOAD in another one as you please. *Don't* use NEW, because this can clear memory rather too effectively! When you want to return the original program, you will have to type CALL &7554. This will start the second part of the program running. This starts (line 35∅) by transferring the saved bytes of the pointer table to addresses #AE81 onwards. This is very much a mirror image of the first piece of program, with the addresses in HL and DE reversed. The LDIR in line 38∅ carries out the transfer. The BC register then is loaded with the bytes of the count which have also been saved in the memory. When this is done, with these addresses held in HL this time, DE is loaded with the destination address of #∅17∅, and another LDIR carries out the transfer. Now we return to BASIC again, and a LIST will reveal the original program.

Variable Lister

This is a rare example of a BASIC program in this chapter. The reason is that it is so much easier to use BASIC for this job, and little point in using machine code. The task is to print the names of variables that have been used in a BASIC program which is also in memory. To use this program, Figure 9.2, you have to make sure that the program whose variables are to be listed is in the memory, and that it has no line numbers above 9999 – you can

```
10010 CLS:PRINT TAB(15)"VARIABLES":PRINT
:PRINT
10020 VS=PEEK(&AE83)+256*PEEK(&AE84)
10030 VF=PEEK(&AE87)+256*PEEK(&AE88)-25
10040 VS=VS+2:TB%=5
10050 CH%=PEEK(VS):IF CH%>128 THEN 10080
10060 PRINT TAB(TB%);CHR$(CH%);
10070 TB%=TB%+1:VS=VS+1:GOTO 10050
10080 PRINT TAB(TB%);CHR$(CH%-128);:TB%=
TB%+1
10090 VS=VS+1:VP%=PEEK(VS):VT$=""
10100 IF VP%=2 THEN VT$="$" ELSE IF VP%=
1 THEN VT$="%"
10110 PRINT TAB(TB%);VT$
10120 VS=VS+VP%+2:IF VS<VF THEN 10040
10130 PRINT:PRINT"End of simple variable
s"
```

Figure 9.2. A Variable Lister program which uses BASIC.

renumber if necessary. You should have saved the Variable Lister by using the 'A' option following the filename, rather than SAVE by itself. This ensures that the program is saved as a set of ASCII codes, rather than a mixture of ASCII and tokens, so that you can load it by using MERGE. This will attach the Variable Lister to the end of the main program. You should then put in a line 9999 STOP. You will then have to RUN the main program so as to establish the variables in the VLT, and then you can use CONT (ENTER) to call up the Lister. Only simple variables are listed, not arrays. Since most programs use a lot of simple variables and few arrays, this is no great loss. The variables which are used in the Lister itself are *not* shown, because the addresses have been chosen so as to avoid doing so.

The principle is simple. Addresses &AE83, &AE84 give the address of the start of the VLT, which is assigned to the variable name of VS. Similarly, the addresses &AE87 and &AE88 give the address of the end of the simple variables list table, and this is assigned to VF. In line 1ØØ3Ø, 25 is subtracted from this address so as to ensure that the variables of the Lister program do not appear. The reason for using 25 is that by the time the allocations of VF has been made, any changes that are made by allocating other variables will not cause the value of VF to change. This means that the bytes for VS and VF have been allocated, a total of 25 from the table. The later allocations, such as VP%, VT$ and TB%, do not appear in any case even if you omit the 25.

Line 1ØØ4Ø adds 2 to the starting address, so that VS now stores the address of the first byte of the variable name. If the name consists of one byte only, this will consist of the ASCII code for the letter with 128 added. In line

1ØØ4Ø also, a TAB number, TB% is assigned. The next thing is to print the name. We have to start by checking for PEEK(VS) being more than 128. If it is, we have reached the last letter of the name, or perhaps the only letter. If it isn't, the loop which uses lines 1ØØ6Ø and 1ØØ7Ø will print each character and increment the TAB and address numbers. When the last character is reached, the THEN 1ØØ8Ø will cause the correct character to be printed, by subtracting 128 from the code number. Line 1ØØ9Ø then increments the address number again, so as to read the type number. This will be 1 for an integer, 2 for a string and 4 for a floating-point number. Line 1Ø1ØØ assigns values to the 'marker' string, VT$, for integer or string variables, and line 1Ø11Ø prints this string following the variable name. Line 1Ø12Ø then tests to find if VS has exceeded VF, and stops the program if it has. If there are more variables to read, then the program loops back to line 1ØØ4Ø.

This is a good example of a program which makes use of PEEK in BASIC, and which would be rather pointless in machine code. To write this in machine code would require you to plan much more carefully and, in particular, you would have to write more elaborate routines to assign the type symbols. Nevertheless, it's not particularly difficult to do, and you might like to try it as an exercise. You will find, though, that the machine code version is not noticeably quicker, so there is little point in using it. Don't feel that because you *can* program in machine code, that you *must*!

Find the line

Just to turn the argument on its head, here's a routine which can just as easily be written entirely in BASIC (try it!) but which is shown here in a mixture of BASIC and assembly language. The aim is to find where a line of a BASIC program is stored. You enter the line number as requested, and the machinecode then finds the address of the start of the line. There *must* be a STOP between the BASIC program whose lines you are checking, and the piece of BASIC program which is shown here in Figure 9.3. The line number

```
10000 STOP
10010 PRINT"Which line number, please?"
10020 INPUT D%
10030 CALL &A7F8,@D%
10040 IF D%=0 THEN 10070
10050 PRINT"Line starts at ";d%;" (Hex "
;HEX$(D%);")"
10060 END
10070 PRINT"No such line"
10080 END
```

Figure 9.3. The line-finder. This consists of a mixture of BASIC and machine code, and the BASIC portion is shown here.

that you choose is allocated to variable name D%, which limits the maximum line number to 32767, since the CPC464 computer will not accept a number higher than this as an integer. Using an integer, however, makes it much easier to pass the value to the machine code. The line number is returned as an integer with the same variable name of D%. If the line does not exist, the number ∅ is returned, and line 1∅∅4∅ detects this so that the message in line 1∅∅7∅ is printed in place of line 1∅∅5∅.

The assembly language is shown in Figure 9.4. When you use a program which passes values, this *must be attended to first of all.* The program starts then by loading H and L with the address bytes that are obtained from IX. You will remember that this will leave HL holding the address of the integer variable in the VLT. This value, which can be up to two bytes, is then read into BC by lines 9∅ to 11∅, and the starting HL address is then restored. This address will not be needed again until the end of the program, so it is saved on the stack (line 13∅).

The HL register pair is loaded with the address of the start of BASIC, and the DE register pair with the bytes from the VLT pointers. Note carefully the difference between a command like LD HL,BAS and LD DE,(ENDPTR). The first places the two-byte address #∅17∅ into HL, so that H contains #∅1 and L contains #7∅. The LD DE,(ENDPTR) command places into E the byte that is *stored* at #AE87, and into D the byte that is stored at #AE88. This means that *two* addresses are read. In line 16∅, then, a loop starts by loading the byte from the first address of the BASIC text into A. This is then compared with the byte that is held in the C register, which is the low-byte of your selected line number. If the two match, then another test has to be made, because the line number is correct only if both low-byte *and* high-byte match. The JR Z,TSTHI step will cause a jump to the second test if a match is found for the low-byte. If no match is found, then line 19∅ increments HL to get the next address in the BASIC text. The HL value is then saved on the stack so that the DE address can be subtracted from the HL address. This result will always be negative or zero until HL carries a higher address than DE. A negative or zero result will set the carry flag, and after restoring the original HL value (line 22∅), the JR C,LOOP step will cause the action to repeat from line 16∅. Note that the carry flag has been tested rather than the negative flag. The carry flag can be tested by using a JR test, the negative flag needs the longer JP test.

Meanwhile, if a match has been found for the low-byte, the program moves from line 18∅ to line 35∅. This increments HL, loads A from the next address where the high-byte is stored, and compares with the byte in B. If a match is found, then HL contains the address at which the high-byte of the line number is stored. We have to subtract three from this address to get the first byte of the line, and this is done by the jump to line 26∅. The resulting value in HL is then transferred to BC. We then have to return this value back to the BASIC program. This is done by recovering the VLT address from the stack in line 3∅∅. The address is now in HL, and the bytes in B and in C

```
Hisoft GENA3 Assembler. Page     1.

Pass 1 errors: 00

                       10 ;Program to find line
                       20 ;address from line number.
A7F8                   30         ORG   #A7F8
A7F8                   40         ENT   $
0170                   50 BAS     EQU   #0170
AE83                   60 ENDPTR  EQU   #AE83
A7F8  DD6E00           70 START   LD    L,(IX+0);low addr.
A7FB  DD6601           80         LD    H,(IX+1);high byte
A7FE  4E               90         LD    C,(HL);for VLT
A7FF  23              100         INC   HL
A800  46              110         LD    B,(HL);value in BC
A801  2B              120         DEC   HL; as it was
A802  E5              130         PUSH  HL;save till later
A803  217001          140         LD    HL,BAS;start of BASIC
A806  ED5B83AE        150         LD    DE,(ENDPTR);end of BASIC
A80A  7E              160 LOOP    LD    A,(HL); get byte of text
A80B  B9              170         CP    C;equal to low byte?
A80C  2818            180         JR    Z,TSTHI;if so, test high
A80E  23              190         INC   HL;next byte
A80F  E5              200         PUSH  HL;save temporarily
A810  ED52            210         SBC   HL,DE; test for end
A812  E1              220         POP   HL;recover address
A813  38F5            230         JR    C,LOOP; back if not finished
A815  010000          240         LD    BC,#0000;not found
A818  1807            250         JR    EXIT;return with zero
A81A  110300          260 GOTIT   LD    DE,#03;step back
A81D  ED52            270         SBC   HL,DE;start of line addr.
A81F  44              280         LD    B,H; get into
A820  4D              290         LD    C,L; BC register
A821  E1              300 EXIT    POP   HL;VLT address
A822  71              310         LD    (HL),C;put BC number
A823  23              320         INC   HL
A824  70              330         LD    (HL),B;back in VLT
A825  C9              340         RET
A826  23              350 TSTHI   INC   HL;get high byte
A827  7E              360         LD    A,(HL); from (HL)
A828  B8              370         CP    B; compare
A829  28EF            380         JR    Z,GOTIT; matches
A82B  18DD            390         JR    LOOP  ;doesn't match

Pass 2 errors: 00

BAS     0170  ENDPTR AE83  EXIT    A821  GOTIT  A81A
LOOP    A80A  START  A7F8  TSTHI   A826

Table used:    94  from   212
Executes: 43000
```

Figure 9.4. The assembly language for the line-finder routine. The symbol table has been printed to show how this appears when this option is selected.

are loaded into the two addresses of the VLT in lines 31∅ to 33∅. The last step is then the return to BASIC. What if no match is found? If the high-byte does not match, then line 39∅ returns to the loop so that the low-byte can be tested at the new address. The loop which goes down to line 23∅ will be repeated until the HL address just exceeds the DE address. If no match has been found, then the line number does not exist. Line 24∅ then loads zero into HL, and the jump to EXIT then loads zero into the two addresses of the VLT for D%. This will cause the NO SUCH LINE message to appear in the BASIC portion.

You can probably think of ways to improve this program. For one thing, it's not infallible. If you are looking for line 4∅∅, for example, which in hex is the two-bytes #9∅ #19 (low-high order), then it will 'find the line' if this combination of numbers just happens to be in the program, not as a line number. You could get around this by using the 'length' byte at the start of each line and adding this to HL instead of incrementing HL through every byte of the program. If you look only at the line numbers, then you can't confuse any other number with the line numbers. In addition, you can use the marker to find where the program ends instead of all the SBC HL,DE stuff.

Another challenge is either to make the program work with fewer bytes, or in a faster time, or both. You might want to make this a completely BASIC program, using POKEs to get the code into place from DATA lines. Another possibility that you might like to ivestigate at a later stage is of writing the program entirely in machine code. Placing the messages on the screen is easy enough, the problems start when you try to enter the line numbers. These will be passed as ASCII codes, and they have to be converted into a two-byte binary number. Similarly, when the line address is passed back, it has to be converted into the form of ASCII code. You will find subroutines for doing this type of action in books such as Tootill and Barrow's *Z80 Machine Code for Humans*. Unless your binary arithmetic is good, though, don't try to understand how the routines work, just use them and be duly thankful that you didn't have to invent them!

One point that I must deal with here is my preference for JR jumps rather than JP. The reason is that it helps to make code 'relocatable'. A relocatable program is one whose code bytes can be poked into any set of RAM addresses and which will then work. Now if you have JP or CALL instructions in a program which jumps to or calls addresses *which are in the program itself*, then the program cannot be easily relocated. Imagine, for example, that you have a JP #A∅5∅ in a program which is located between #A∅∅∅ and #A#F#. If you placed this program starting at #9∅∅∅, then the JP #A∅5∅ would still cause a jump to #A∅5∅, but this is no longer the correct address (it should now be #9∅5∅). JR instructions only specify a number of steps forward or backward, so if your program uses only JR jumps, and any CALLs or JPs are to addresses in the ROM, the program is relocatable, and you can poke it anywhere you like. By contrast, if the program is not relocatable, you will have to re-assemble it with the new ORG address before you can get code that will work. If only a few JP instructions are used, however, it is possible to correct them when the program is relocated.

The zero-line caper

We can make use of our knowledge of line numbering to make all the line numbers of a BASIC program equal to zero. In BASIC, this takes a lot of time, but in machine code, it's as good as instant, and a very easy task to program. In addition, as you will see, it has some interesting side-effects. Figure 9.5 shows the method, using assembly language which has been printed out from the GENA3 program. We start by loading the start-of-BASIC address of #Ø17Ø into the HL register pair, and zero into the C register. In the main loop, the byte at this address is loaded into E and then

```
Hisoft GENA3 Assembler. Page     1.  pyright Hisoft 1984*


Pass 1 errors: 00

                          10 ;Program to zero line numbers
                          20 ;of a BASIC program.
A7F8                      30         ORG  #A7F8
A7F8                      40         ENT  $
0170                      50 BAS     EQU  #0170
A7F8   217001             60 START   LD   HL,BAS
A7FB   0E00               70         LD   C,0
A7FD   5E                 80 BACK    LD   E,(HL);displacement bytes
A7FE   E5                 85         PUSH HL
A7FF   23                 90         INC  HL
A800   56                100         LD   D,(HL);for next line
A801   23                110         INC  HL
A802   71                120         LD    (HL),C; aero it
A803   23                130         INC  HL;get h.b.
A804   71                140         LD    (HL),C;zero it
A805   E1                150         POP  HL
A806   19                160         ADD  HL,DE; get next line
A807   7A                170         LD   A,D;test
A808   B3                180         OR   E;for end
A809   20F2              190         JR   NZ,BACK;back if not
A80B   C9                200         RET

Pass 2 errors: 00

Table used:     46  from    145
Executes: 43000
```

Figure 9.5. How to make every line of a BASIC program carry a line number of zero!

the value of the HL address is saved on the stack. The HL address is then incremented, and the byte at the next higher address is then loaded into D. This makes DE hold the displacement byte(s) which when added to the address on the stack will give the address of the next program line. The value of zero in the C register is then loaded into the next two addresses, which are the line number addresses. The HL address of the start of the line is then recovered from the stack, and the displacement number is added from DE, which leaves HL holding the address of the start of the next line. The displacement bytes in DE then have to be tested to find if there is a 'next line'. If DE holds zero, we have reached the end of the program; if not, we loop back. It's simple and effective – try it! The technique of getting from one line to the next is one which you might like to adapt for the line-finder program.

The quite unexpected thing is that the program which has been treated in this way disappears! If you LIST, nothing happens, though if the program is one which prints something on the screen, you will see that it can still be RUN! You cannot list a program whose line numbers have been zeroed in this way unless the *first line* has a line number. If, for example, you carry out:

```
POKE &172,1Ø     (ENTER)
```

then you will see when you LIST a first line which is numbered 1Ø, and the remaining lines numbered Ø. It could be very useful for program protection, because even though the program cannot be listed, it can be recorded and replayed and run. If you want to see the listing, you can poke the first line number in place, or you can use the RENUM command to put all the line numbers back into place again.

The character matrix

Earlier, we looked briefly at how the screen memory of the CPC646 is organised. Even a brief glimpse is enough to persuade us that we don't meddle with this lot without a lot of care. It's time now to investigate just how the CPC464 puts characters on to the screen, and the first thing to look at is the character matrices. Each character that you see on the screen is made up from a set of dots, and the dots are arranged in lines of eight. In addition, there are eight dots to a line, so that each character can consist of up to 64 dots. Since each bit of a byte can be used to signal whether a dot is in ink colour (1) or paper colour (Ø), one byte can be used for each line of a character, giving eight bytes to hold all the information that is needed for the shape of the character. The thing that we want to know now is where these bytes are stored. We might expect that all the bytes for the standard alphabetical characters could be held in ROM, but that some RAM might be reserved for shapes that we wanted to create for ourselves.

Now as you know, you can't use PEEK on the CPC464 ROMs, because the PEEK command is operated by machine code, and it always selects

RAM. It's not difficult to find out where the bytes that form a character on the screen are stored, however. If we work in machine code, we can make use of one of the CALLs which has been thoughtfully listed by Amsoft in their manual. This CALL is to #BBA5, and has to be made with an ASCII code in the accumulator. After the CALL, the HL register pair will hold the address of the start of the set of eight bytes which forms the shape of the character. The carry flag is also used to show whether this is in ROM or RAM, but we don't really need that refinement at the moment. What we have to do, then, is to write a short routine which will allow us to use this call, and then take it from there.

Figure 9.6 shows the short assembly language routine which will accept

```
                    ;program to find character
                    ;matrix in memory.
A7F8                          ORG   #A7F8
A7F8                          ENT   $
A7F8   DD6E00       START     LD    L,(IX+0);addr low
A7FB   DD6601                 LD    H,(IX+1);and high
A7FE   E5                     PUSH  HL;save vlt address
A7FF   7E                     LD    A,(HL);get ASCII code
A800   CDA5BB                 CALL  #BBA5;subroutine
A803   EB                     EX    DE,HL;address into DE
A804   E1                     POP   HL;get VLT again
A805   73                     LD    (HL),E;address
A806   23                     INC   HL;into
A807   72                     LD    (HL),D;VLT for return
A808   C9                     RET

Pass 2 errors: 00

Table used:     25  from    139
Executes: 43000
```

Figure 9.6. A program which will find where the 'character matrix' is stored.

an ASCII code passed from a BASIC program, and will return an address. The address will be the address of the bytes (the 'matrix') which hold the character shape. The routine starts as usual by getting the VLT address into HL from (IX+Ø) and (IX+1). This address is then saved on the stack, because the subroutine which we shall call needs to use HL. The accumulator is then loaded from (HL), getting an ASCII code if the BASIC has done its work, and CALL #BBA5 will carry out the work of finding the address. This will be in HL now, and if the matrix is in RAM, the carry bit will be set. We ignore the carry bit, and pass the address into DE by using EX DE,HL. The POP HL step then gets the VLT address back, so that we can load the address bytes from DE into the VLT. At the RET step, these will be returned to the BASIC program as a value of an integer.

Figure 9.7 shows the BASIC part of this exercise. Line 3Ø tests the character in case of errors in entry, and line 4Ø calls up the machine code to

```
10 CLS:PRINT TAB(15)"CHARACTERS":PRINT
20 INPUT"Type ASCII code for character";
A%
30 IF A%>255 OR A%<1 THEN PRINT"MISTAKE-
 TRY AGAIN":GOTO 20
40 CALL &A7F8,@A%
50 PRINT"Address of matrix is ";A%;"  ,He
x #";HEX$(A%)
60 END
```

Figure 9.7. The BASIC part of the 'character matrix' program.

do the work of finding the address. Line 5∅ then prints the address that is obtained. This is also A%, because A% contains the new value that has been returned to the VLT by the machine code part of the program. You'll find when you work with this that standard alphabetical codes like 65 ('A') give addresses in the lower ROM, around #3A∅∅. The codes for the user-programmable keys, codes 24∅ to 255, give addresses in high memory, which read as negative in denary. For example, code 244 gives the address −216∅∅, hex #ABA∅. You can see the shape of this character on the screen by using ?CHR$(244), and you can also examine the matrix. The best way to do this is by making use of the BIN$ command of Amstrad BASIC. If you take the matrix starting address of #ABA∅, then the following line will display the whole matrix on the screen –

```
FOR N=&ABA∅ TO &ABA7:?BIN$(PEEK(N),8):NEXT
```

This display of 1s and ∅s gives the shape of the character, as you know from using the SYMBOL command in BASIC.

The next thing to tackle in this investigation is some peeking into the ROM. You could think of these routines as tools for programmers who want to find out how the machine works. If you approach it as a game, you will find it more interesting and more rewarding than any other game you could play with a computer. Since the BASIC PEEK command does not work on the ROM, we'll have to make one that does, using machine code. This, after all, is precisely what machine code is about. We can have a BASIC part which defines an integer variable as the ROM address that we want to PEEK, and which will return with the byte that is stored in ROM at that address. Since the lower ROM seems to hold some interesting data, like the matrix for the standard character set, we can look at this one first.

Figure 9.8 shows the sort of BASIC routine that we might like to use. The call to the machine code has been put into a subroutine, so that a set of addresses can be fed to the subroutine and a set of bytes will be returned. This subroutine will be used as we might use a PEEK, except that a variable N% is being used to hold the PEEKed character. You have to be careful here to separate X%, the loop counter, from N%, the variable which is used to

```
10 INPUT"Start, finish ";A%,B%
20 FOR X%=A% TO B%:N%=X%:GOSUB 1000
30 PRINT N%:NEXT
40 END
1000 CALL &A7F8,@N%
1010 RETURN
```

Figure 9.8. A BASIC program which uses a CALL as the equivalent of a PEEK for the lower ROM.

pass numbers, because N% will be changed by the machine code routine. The result will be a subroutine which we can use in place of a PEEK for ROM, and which will run just as fast as a PEEK in a loop like this!

The important part, of course, is the machine code, which is shown in Figure 9.9. This gets into HL the address in the VLT where the value of N% has been stored. The bytes of the address are then read from HL and HL+1 into DE, so that this register pair now holds the address in ROM that we want to PEEK. While HL holds the upper byte of the address, we can make this byte zero. This is because we want to return only one byte of data to the

```
Hisoft GENA3 Assembler. Page     1.  pyright Hisoft 1984*


Pass 1 errors: 00

                     10 ;Program to PEEK ROM
A7F8                 20         ORG  #A7F8
A7F8                 30         ENT  $
A7F8  DD6E00         40         LD   L,(IX+0);get
A7FB  DD6601         50         LD   H,(IX+1);VLT
A7FE  5E             60         LD   E,(HL);address in HL
A7FF  23             70         INC  HL
A800  56             80         LD   D,(HL);DE holds PEEK addr.
A801  3600           90         LD   (HL),0;zero upper byte
A803  2B            100         DEC  HL;restore VLT addr.
A804  CD06B9        110         CALL #B906;switch on ROM
A807  F5            120         PUSH AF;preserve byte
A808  1A            130         LD   A,(DE);get PEEK byte
A809  77            140         LD   (HL),A;put in VLT
A80A  F1            150         POP  AF;restore
A80B  CD0CB9        160         CALL #B90C;switch off ROM
A80E  C9            170         RET

Pass 2 errors: 00

Table used:     13  from   144
Executes: 43000
```

Figure 9.9. The assembly language program which carries out the PEEK action on ROM.

BASIC program. If the upper byte of the address is left in (HL), it will also be returned, making the data look decidedly peculiar. HL is then decremented so as to point at the lower VLT byte again. The next step is a CALL to #B9Ø6. This switches on the lower ROM, so that all addresses from Ø to 3FFF now use this ROM. When this call returns, it puts a byte into the accumulator which is a code number for this ROM. We shall use this byte to restore things to normal afterwards, so it is saved on the stack. The accumulator is then loaded from DE, and since this DE address is now an address in the ROM, this loads A with the PEEK number that we want. The number in A is then returned to (HL) to get it back into the VLT, and the code byte is popped from the stack. Another CALL, this time to #B9ØC, will now restore the ROM switching to normal, so that the routine can return.

Now that you can peek your way around the lower ROM, you can use this to look at the addresses which hold the character matrix bytes. Try the range &#AØ8 to &#AØF, for example, to see the character numbers for the letter A. It looks better if you print in binary as before, so that line 3Ø now starts: PRINT BIN$(N%,8). This way, you'll see the pattern on the screen in 1s and Øs rather than a set of denary numbers. Needless to say, you can use this routine to read a load of bytes from the ROM into memory. From there, you can record them, or use MONA3 to disassemble them as you wish. You can, of course, make use of the CALL #B9Ø6 from MONA3 if you want to investigate the ROM, and this routine is really intended for the owners who want to do some peeking without using MONA3.

Cassette capers

If the layout of the screen memory of the CPC464 were rather simpler, it might be worthwhile looking at ways of animating screen displays. As it is, the complications of the screen layout make this a job for the much more experienced machine code user, and we'll switch to a new topic, the machine code control of the cassette system. The BASIC of the CPC464 allows a number of formats of tape to be made, of which the three main varieties are a BASIC program, a data file, and a machine code program. Many of the subroutines for the cassette system are published in the great Amstrad book of wisdom, however, and so we can make use of them. If you write your own cassette routines, you can create cassettes which other CPC464's can't read, and it's even possible to write routines which can read cassettes which have been made by other types of machines. At this stage, though, we're not into such high-flying activities, and we'll confine ourselves to looking at some variations on reading a cassette.

The first program I want to lead you through is one which will load bytes from a cassette and put them into a 'buffer'. A buffer in this sense means a spare piece of memory. Why should we want to do this? One good reason is curiosity, because it allows us to see just what is on a tape. When you read a

BASIC tape, for example, you end up with lines of BASIC stored starting at address #∅17∅, but that isn't all that's on the tape. Just to give one example, the filename must be read from the tape; there must also be a byte count so that the machine can read the correct number of bytes, and presumably there must be signals about what type of tape it is and whether it is protected or not. Normally when you load a tape, this information is read and used by the machine. If we load into a buffer, all of this will be stored for us to inspect.

The program is illustrated in Figure 9.10, and consists largely of calls to the ROM. The main call is to #BCA1, which is a reading routine. This will read blocks of data, and it is the routine that is used by most of the other tape-reading routines. It requires some setting up in the form of numbers in the HL, DE, and A registers. The HL register pair needs to be loaded with the starting address of a buffer, which can be anywhere in the RAM. The DE register is supposed to be loaded with the number of bytes of data. Since you don't normally know this, it can be loaded with ∅, which will *permit* 65536 bytes to be loaded. The word 'permit' is important here. The loading will, in fact, stop either when the system detects the last byte in the recording, or when DE has been counted down. If you load DE with ∅, then you ensure that only the end of the recording stops the cassette. When it does so, an error code is loaded into the accumulator, but unless you write a routine to

```
Hisoft GENA3 Assembler. Page     1.

Pass 1 errors: 00

                         10 ;read cassette to #200
A7F8                     20          ORG  #A7F8
A7F8                     30          ENT  $
0200                     40 BUF      EQU  #0200;memory space
A7F8   210002            50          LD   HL,BUF;set up
A7FB   110000            60          LD   DE,0;call to
A7FE   3E2C              70          LD   A,#2C;cassette
A800   CDA1BC            80          CALL #BCA1;read header
A803   214002            90          LD   HL,BUF+64;room for text
A806   110000           100          LD   DE,0;set up
A809   3E16             110          LD   A,#16;for text read
A80B   CDA1BC           120          CALL #BCA1;and read it
A80E   C9               130          RET

Pass 2 errors: 00

Table used:    23  from   132
Executes: 43000
```

Figure 9.10. A program which will load a piece of 'buffer' memory from cassette. Only one block can be read with this program. To read more, you will have to increment the buffer address and repeat the reading steps in lines 50 to 120 until the 'last byte' indicator appears in the header.

read this and act on it, it has no effect. This is machine code, remember, not BASIC!

The byte in the accumulator is more important. If this is #2C, the routine will stop after reading the 'header'. This is the block of 64 bytes which is recorded on to the tape before the main text, and it holds all the information about the tape. We can, for example, read from this header the number of bytes in the tape, and use this to load DE for reading the main text. In this case, we'll keep to the simpler method of just loading zero, however. Lines 9∅ to 11∅ set up again, this time placing the buffer address 64 bytes higher up to avoid overwriting the header. This time, the accumulator has to be loaded with #16, the signal for the end of the main text.

Assemble this, and get back to BASIC if you are using GENA3. Now put a short BASIC tape into the cassette player, and press PLAY. By short, I mean a tape that consists of one block only. Now use CALL 43∅∅∅ to start the routine, and watch what happens. The header will be read, and then the main text, and the tape will stop with the usual 'Ready' message. Apart from the lack of messages, it's rather like an ordinary tape read. The fun starts when you start to look at what you have in the buffer. The first sixteen bytes, #∅2∅∅ to #∅2∅F consist of the filename of the tape followed by zeros. The zeros are padding only, and are used to reserve the space for longer filenames of up to sixteen characters. The next interesting byte is in #∅21∅, and shows which block has been read. For a one-block tape, this will be ∅1. Address #∅211 will be FF if the end of the tape has been read.

So far, so good, but the really interesting part is #∅212. This carries a code which distinguishes the type of tape. The most interesting thing here is that if the number here is odd, then the tape is a protected one. Normally, the machine reads this byte and acts on it. Because we are by-passing the normal action, though, and reading into a buffer, this byte can have no effect. Figure 9.11 illustrates how the bits of this byte are used. The next two bytes, #∅213, #∅214 make up the length-of-file number, equal to the number of bytes recorded *following* the header. The next two bytes, in #∅215, #∅216 contain the address from which the data was read. For a BASIC tape, this will normally be #∅17∅. Address #∅217 contains FF once again, showing that this is the first block, and #∅218, #∅219 contains the byte count again. The next two addresses will be used for execution address of a machine code tape, and the rest of the header is not used. This means that you could set up routines to put your own data into the unused part of the header.

The text of the program itself will start at #∅24∅. This, remember, is just a set of bytes in the memory – you can't list it or run it. You can, however, shift it to #∅17∅, the normal starting place for BASIC. If you are using MONA3, you can do this very simply by using the 'I' command. You fill in the address ∅24∅ for the 'First' prompt, and add the length byte to this number for the 'Last' prompt. For the 'To:' prompt, you enter ∅17∅, and then press ENTER. The bytes will then be shifted down. Since the header has not been read by the normal BASIC routine, any protection bit setting will have no effect.

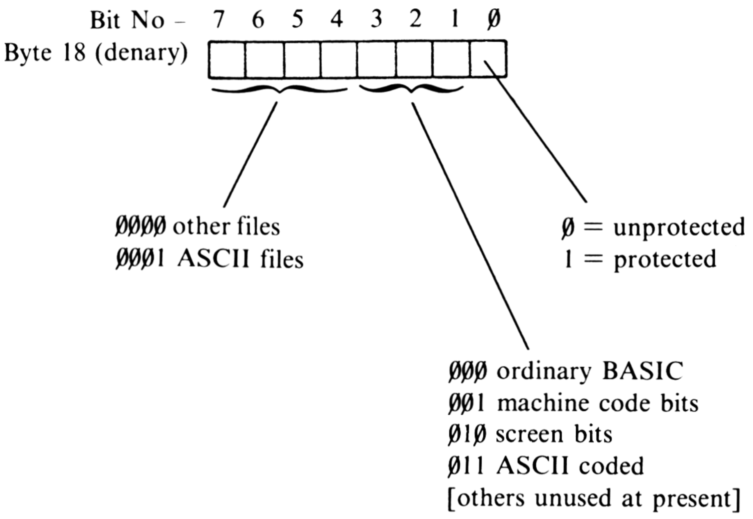


Figure 9.11. How to analyse byte 18 in the header.

The program can be listed and run as normal, because it *is* as normal as it would be if you had typed it from the keyboard.

Load and shift

All of the information about shifting bytes is useful if you have MONA3, not quite so useful if you haven't. You can, of course, shift the bytes using PEEK and POKE in a direct BASIC command (not a program!), but it's even easier to do it in a machine code program. Figure 9.12 shows what you need to add. The routine for reading from the cassette is as before, and all we need to add is a byte-shift routine. This means that DE is to be loaded with a destination address, HL with a source address, and BC with a byte count. Since we know the two addresses, and we can read the byte count from the buffer, this is easy to arrange, and with the registers set up, LDIR does the transfer, so that the program can be listed whenever the cassette has finished. Needless to say, you can use methods like this with machine code tapes as well, and you can also read the execution address and load address from the header. This can be extremely useful if you want to make copies of a machine code program with the fast cassette speed selected. It's particularly handy for MONA3 and GENA3, because the time that they take to load can be a real pain.

```
Hisoft GENA3 Assembler. Page     1.

Pass 1 errors: 00

                         10 ;read cassette to #200
A7F8                     20          ORG  #A7F8
A7F8                     30          ENT  $
0200                     40 BUF      EQU  #0200;memory space
A7F8  210002             50          LD   HL,BUF;set up
A7FB  110000             60          LD   DE,0;call to
A7FE  3E2C               70          LD   A,#2C;cassette
A800  CDA1BC             80          CALL #BCA1;read header
A803  214002             90          LD   HL,BUF+64;room for text
A806  110000            100          LD   DE,0;set up
A809  3E16              110          LD   A,#16;for text read
A80B  CDA1BC            120          CALL #BCA1;and read it
A80E  117001            130          LD   DE,#0170;BASIC start
A811  214002            140          LD   HL,#0240;BUF start
A814  ED4B1302          150          LD   BC,(#0213);get length
A818  EDB0              160          LDIR
A81A  C9                170          RET

Pass 2 errors: 00

Table used:    23  from   143
Executes: 43000
```

Figure 9.12. A program which loads from tape to buffer, then shifts the bits of the file to the start of BASIC. This, once again, operates on a single block only. The program can be listed, but it cannot be RUN unless the VLT is poked with the correct ending addresses.

Screenload

Sometimes, a sequence of load to buffer, then peek, is just too longwinded, and you'd like to look at the bytes of a program directly from the tape. Here's a program (Figure 9.13) which will let you do just that. It makes use of the main cassette routines of the machine, which means that the header will *not* appear on screen. The header will be read and used by the machine, so that you will see the filename printed, but no other information. Following the filename, though, you'll see the program bytes in hex on your screen. It's quite a long program compared to what we've used before, but the principles are straightforward.

It all starts with calling #BB6C. This clears the screen, and places the cursor at the top left-hand side. We then load a buffer address into DE. Since you don't have to use BASIC with this program, you can place the buffer in fairly low memory, at #Ø2ØØ. Even if you use BASIC to load the machine code, you can still use this space. The set-up routine for reading the

```
Hisoft GENA3 Assembler. Page     1.

Pass 1 errors: 00

                      10 ;Program to load from tape
                      20 ;and display on screen.
A7F8                  30          ORG  #A7F8
A7F8                  40          ENT  $
A7F8   CD6CBB         50          CALL #BB6C;clear and home
A7FB   110002         60          LD   DE,#0200;buffer
A7FE   010000         70          LD   BC,#00;load first file
A801   210000         80          LD   HL,00;no flename to get
A804   CD77BC         90          CALL #BC77:open stream
A807   CD80BC        100 READ     CALL #BC80;fill BUF and read
A80A   F5            110          PUSH AF; save flags
A80B   4F            120          LD   C,A;save byte
A80C   E6F0          130          AND  #F0:select high nibble
A80E   CB2F          140          SRA  A;shift
A810   CB2F          150          SRA  A;to
A812   CB2F          160          SRA  A;low
A814   CB2F          170          SRA  A;nibble
A816   CD31A8        180          CALL ASCI;convert
A819   CD5ABB        190          CALL #BB5A;and print
A81C   79            200          LD   A,C;get byte back
A81D   E60F          210          AND  #0F;low nibble
A81F   CD31A8        220          CALL ASCI;convert
A822   CD5ABB        230          CALL #BB5A;and print
A825   3E20          240          LD   A,32;space
A827   CD5ABB        250          CALL #BB5A;print
A82A   F1            260          POP  AF;restore flags
A82B   38DA          270          JR   C,READ;back for next
A82D   CD7ABC        280          CALL #BC7A;close
A830   C9            290          RET
A831   FE0A          300 ASCI     CP   #0A;decimal?
A833   3802          310          JR   C,DEC;jump on
A835   C607          320          ADD  A,07;for letters
A837   C630          330 DEC      ADD  A,48;to ASCII
A839   C9            340          RET

Pass 2 errors: 00

Table used:    45  from   187
Executes: 43000
```

Figure 9.13. A program which displays the contents of a tape on the screen, in hex. Note the different subroutines which have been used.

tape requires BC to be loaded with the number of characters of the filename, and HL with the filename address. To avoid difficulties of passing two sets of parameters here, we can load BC with zero, and set HL also to zero. This has the effect of loading the first file that is found on the tape, just as if you had typed LOAD"" in BASIC. The CALL to #BC77 then opens the stream for reading the tape.

The main loop then starts in line 1ØØ. The CALL to #BC8Ø will first of all fill the buffer from the tape, then read the first character. Subsequent calls will read from the buffer until the end marker is found. If there is no end marker in the buffer, another block will be read from the tape, and so on. The result of using #BC8Ø will therefore put a byte into the accumulator. It will also affect flags, with the C flag set until the end marker is read. We need to save this flag condition on the stack before we start to operate on the byte in the accumulator. The problem now is that we have a number in the accumulator. We can't just print this on the screen, because the screen-print routines print the *character whose ASCII code* is in the accumulator. What we have to do is to select the top four bits, and convert from this number into the ASCII code for a hex digit, print this, and then do the same with the lower four bits.

This is quite easy, because the ASCII codes for numbers Ø to 9 are obtained by adding 48 to the numbers. The codes for letters A to F, corresponding to denary 1Ø to 15, require another 7 to be added. For example, the number 1Ø needs 48 and 7 to be added, giving 65, the ASCII code for 'A'. First of all, we select the top four bits (the upper nibble, as it's called) of the byte. This is done by saving a copy in C, and then using AND #FØ to blank out the lower nibble. The byte is then shifted right four times to get the upper nibble into the lower position. If the byte had been #A7, for example, then AND #FØ would make it #AØ, and the shift would convert this to #ØA. By calling a conversion routine, ASCI, this number is replaced by the correct ASCII code, and the routine at #BB5A prints it on the screen. The whole byte is then loaded back from C to A, and this time, the higher nibble is zeroed. The conversion to ASCII code is done again, and the result is printed. Once the complete byte has been printed like this, we then have to see if there is another byte in the buffer. This is done by getting the flags back from the stack. If the carry is set, then there is another byte to read, and the program loops round again. When the carry flag is cleared by the call to #BC8Ø, the last byte has been read.

It's a simple routine once you know what is going on, and it can be very useful for inspecting what is in a BASIC program, without the usual loading and peeking. Note that the program as it stands in not easily relocatable, because of the CALL ASCI routine. Suppose that you wanted to print the program listing on the screen rather than the bytes? The trouble with this is that the recorded program is normally in coded form, with token bytes used for all the keywords. If you have recorded the tape using the ASCII 'A' option, then all keywords are in ASCII rather than as tokens, and the routine works well. If the program is coded, you can still make out some features, like strings and REMs. You will have to replace any number less than 32 or greater than 128 by a space or a dot code. This is because the numbers less than 32 will cause odd effects if you print them on the screen, and numbers greater than 128 will give graphics characters. Perhaps you might like to experiment for yourself – you have all the clues before you!

Screen-printing?

I'm going to wind down with a routine which will, I admit, be useless to many readers. By way of compensation, it will be very valuable to many others, and it will still illustrate a lot about machine code even if you can't use it. The program is designed to allow you to send whatever normal characters are on the screen to a printer. It's particularly useful with MONA3, because MONA3 makes no provision for printing the display of memory and register information. With this routine, it should be possible to print anything which uses ASCII codes 32 to 128 (denary). The main problem is that printers generally use 8Ø or more characters per line, and the screen uses a maximum of 4Ø characters per line when MONA3 is running. This implies that the machine code must send the line feed (#ØA) and carriage return (#ØD) characters to the printer after each forty characters. How do we do it? The simple answer is to use a count in one of the double registers. As it happens, this is convenient, because the ROM contains a routine for positioning the cursor which requires row and column numbers in HL. This routine is at #BB75, and it requires the column number in H and the row number in L. These numbers start at 1, not at zero, so to position the cursor at the top left-hand corner of the screen, we need to load HL with #Ø1Ø1.

There are two other routines now that we need. One is the routine to find what character is placed on the screen at the cursor position, and this is at #BB6Ø. The other is to send a character to the printer, and this is at #BD2B. If you have no printer attached, or have not switched the printer on, you can't use the routine. You don't get the usual error messages about 'No printer' unless you build them into your machine code routines. For the moment, we have not bothered with this complication. The routine will position the cursor, get the character under the cursor, and then print it. It must then test the numbers in HL and increment the column number before getting the next character. When the column number increments to 41, the H register has to be reloaded with Ø1, and the L register (the row register) incremented, and this number tested. Because the cursor positioning call corrupts the numbers in H and L, we have to save these registers on the stack each time through the loop. At the end of each row, the line feed and carriage return characters are sent to the printer. This makes the printer carry out one line feed – it should be set so that it does not cause another line feed when it gets the carriage return. The CPC464 BASIC print routines seem to send two line feeds to my printer, which I get around by using closer line spacing. If you also are using this dodge, you will have to use normal spacing for this routine, because the normal single line feed is sent.

The complete assembly language routine is shown in Figure 9.14. Line 40 loads the 'cursor home' numbers into HL so that reading starts on the top left-hand corner. The loop then starts in line 50 with a PUSH, so that the HL contents are saved for the counter increment steps later on. The next three steps use the calls to the ROM to position the cursor, read the character under the cursor, and then send the character to the printer. After this lot, we can be sure that all the registers will be thoroughly corrupted, but the only one we need is HL, and it has been saved on the stack.

Now we have to check the cursor position. Line 100 recovers the HL bytes from the stack, and lines 110 to 130 increment and test the column number which is stored in H. If this is in the range 1 to 40, the program simply loops back to get another character. When the column number has been incremented to 41, it's time to go to the next row of screen *and* printer. For the printer, this means sending out the line-feed and carriage return

```
Hisoft GENA3 Assembler. Page     1.

Pass 1 errors: 00

                        10 ;program to print screen data
A7F8                    20          ORG  #A7F8
A7F8                    30          ENT  $
A7F8   210101           40          LD   HL,#0101;row, column
A7FB   E5               50 LOOP     PUSH HL;save it
A7FC   CD75BB           70          CALL #BB75;position cursor
A7FF   CD60BB           80          CALL #BB60; get character
A802   CD2BBD           90          CALL #BD2B;print it
A805   E1              100          POP  HL;get row,col
A806   24              110          INC  H;next column
A807   7C              120          LD   A,H;test
A808   FE29            130          CP   41;too much?
A80A   20EF            140          JR   NZ,LOOP;back if not
A80C   3E0A            150          LD   A,#0A;linefeed
A80E   CD2BBD          160          CALL #BD2B;print
A811   3E0D            170          LD   A,#0D;c/r
A813   CD2BBD          180          CALL #BD2B;print
A816   2601            190          LD   H,01;reload
A818   2C              200          INC  L ;next row
A819   7D              210          LD   A,L;test
A81A   FE1A            220          CP   26;too much?
A81C   20DD            230          JR   NZ,LOOP;back if not
A81E   C9              250          RET

Pass 2 errors: 00

Table used:     24  from    156
Executes: 43000
```

Figure 9.14. A screen-print routine which can be used with any 80-column printer.

characters in lines 15∅ to 18∅. For the screen, we have to reload the first column number into H, increment the row number in L, and then test for the bottom of the screen. This means testing for the row number reaching 26. If it hasn't done so, we can loop back for another set of columns, but if we have reached the end, we can return to BASIC. The whole routine has been written in the assumption that it will be used from BASIC. You can set up a programmable key to carry out the CALL &A7F8+CHR$(13) action so that the routine is called up by pressing this key.

Getting hooked

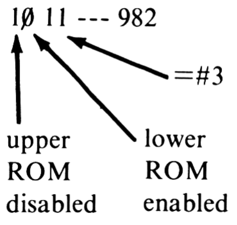
This is definitely the last item, but it's a very important one. So far, you have always started a machine code program going by a CALL, which could be assigned to one of the programmable keys. For most purposes, this is quite satisfactory, but printer-owners will have noticed a snag with the previous program. The snag is that when you assign a CALL command like CALL &A7F8 to a programmable key, then when the key is used, the command CALL &A7F8 is printed on the screen, and if you are sending everything on the screen to the printer, this gets sent as well. In addition, if the screen is full at the time, it will scroll, and you may miss whatever was on top.

We'll end, then, with an automatic method of starting a program. This gets you into the inner depths of the operating system, and when you start making use of programming methods like this you have definitely taken your 'L' plates off! The principle is that the CPC464 computer is designed to be adaptable, so many of the routines which exist in the ROM make calls to the RAM. When I say 'make calls', I mean that literally. At several places in the RAM you will find commands such as:

```
    JP XXXX      or      RST 8 XXXX
```

– where XXXX represents an address. There is a whole set of these jump and reset commands, arranged in groups. One small group of jumps starts at #B9∅∅ and ends at #B92∅. The main group of RST commands starts at #BB∅∅ and ends at #BD3C. There is yet another set of JP comands which start at #BDCD and end at #BDF3. The RST commands in the main set are not what you might expect from reading textbooks about the Z80. The CPC464 is designed so that RST 8 can be followed by two bytes which carry a code and an address. The code is for which of the two ROMs is to be enabled, and the address is the address to jump to. Remember that a simple JP will normally go to an address in RAM, not in ROM. The method that is used in the RST calls is illustrated in Figure 9.15. You write the two bytes that follow the RST code of #CF in the usual way for an address, by reversing the order of the bytes. For example, CF96A4 means RST8 followed by #A496. Now this address is not quite so straightforward as it looks, because the first two bytes are *not* part of the address. If we write the

CF 82 B9
'Address' is #B982
Separate first digit, and write in binary



∴ Address is #3982, lower ROM enabled

Top two bits :
ØØ both ROMs enabled
Ø1 upper ROM enabled, lower disabled
1Ø lower ROM enabled, upper disabled
11 both ROMs disabled

Figure 9.15. How the RST 8 calls are decoded.

first *nibble*, hex 'A', in binary, we get 1Ø1Ø. Of this, the first '1' means that the upper ROM (#CØØØ upwards) is disabled, and the following Ø means that the lower ROM (up to ##FFF) is enabled. The rest is part of the address. The rest of this nibble is the number #2, so that address is #2496, which is indeed in the lower ROM.

Now each of these calls in RAM must be used by the routines in the ROM. A ROM routine will therefore jump to the RAM, and then return at once. What does the RAM do with the routine? Answer – nothing! A better question however, is 'what *might* it do?'. Since these addresses which store the JP and RST8 bytes are in RAM, they can be altered. We can, for example, change the two address bytes that follow a JP, or we can replace an RST8 call by a JP to another address in RAM. The effect will be to divert any call to a routine of our own instead of just returning. These addresses, then, allow us to 'hook in' any routines of our own, so that they run automatically each time the ROM makes a call to RAM. The next thing is to illustrate just how these 'hook' calls are used. The Manual on the firmware lists each of these calls in detail, but we'll stick with just one, the jump at #BDD3.

This is a call to the screen routines which is made each time a key is pressed. You can, however, be caught out here. The screen routines of the CPC464 are in groups. One group will not accept control codes with ASCII numbers below 32, another group will not accept codes above 128. The routine at #BDD3 is used by BASIC and by MONA3, but not by GENA3. Since GENA3 has sufficient print driving routines of its own, we don't need to worry about this. The principle we are following is to change things so that pressing the TAB key will produce a printed copy of what is on the screen. If you are using MONA3, you may have to re-initialise your printer first by switching off and on again.

We have to start by altering the code at the address #BDD3, and substituting the starting address of the screen-print routine. If we alter the routine so that it tests for the TAB byte, ASCII ∅9, then it's possible to reject other keys, and make a routine run only when the TAB key was pressed. I have chosen the TAB key because it's not used for anything else in normal use. Having loaded these new address bytes in at #BDD4, #BDD5, we can then return to BASIC. At the starting address for our routine, we place a filter. This will test for one specific byte and return if this byte is not present in the accumulator. The rest of the routine will consist of the instructions which are to be run if the selected byte *is* present. In this example, the selected byte is #∅9, the byte which is present when the TAB key has been pressed. The routine will end with a JP #134A so that the program goes back to the ROM at the address which would have been used by #BBD3.

Figure 9.16 shows the screen-print routine adapted for this method. Line 3∅ is very important. DI means 'disable interrupts', and it means that nothing will interrupt the program while the addresses are being changed. This is to avoid the catastrophe if something were to stop the program and leave the address half-changed. This would immediately cause a crash. I'm not convinced that this step is essential for CPC464 machines, but it certainly is for others, and it's easier to be on the safe side. The HL pair is then loaded with START, the starting address of the routine. This is then placed into #BDD4 and #BDD5. Note that using HL for this lets us do the load of two bytes in one operation. Once this has been done, interrupts can be re-enabled, and the routine returns to BASIC. When this is called, once only, by a CALL 43∅∅∅ statement in BASIC, it will change the address so that the routine which exists at address START will run each time a key is pressed. We now have to amend the screenprint routine so as to make it run only when the TAB key is pressed, and we must also protect the registers. We know that we have to preserve the HL and AF registers if the eventual jump to #134A is to work, so these registers are pushed on to the stack whenever the TAB key is detected, in line 1∅∅.

The screenprint routine now starts (line 12∅) and follows the pattern of the screenprint routine that we looked at earlier. This particular version uses two line-feeds, but this is not necessary, and in fact, only one was obeyed by the printer. You can therefore omit one in your own version. Lines 12∅ to 32∅ mark out where you can place your own routine which you want to run when the TAB key is pressed. After the routine has run, the registers are restored, and the JP #134A sends the Z80 back to the ROM again. Once you have assembled this, the screenprint routine is now ready to hook in. If you have used GENA3 to assemble the machine code, then return to BASIC, and use CALL 43∅∅∅ to make the first part of the program run – assuming that you assembled at #A7F8. This will hook in the second part of the

```
Hisoft GENA3 Assembler. Page     1.

Pass 1 errors: 00

A7F8               10        ORG  #A7F8
A7F8               20        ENT  $
A7F8   F3          30        DI
A7F9   2101A8      40        LD   HL,START;routine start
A7FC   22D4BD      50        LD   (#BDD4),HL;into jump hook
A7FF   FB          60        EI
A800   C9          70        RET
A801   FE09        80 START  CP   9;is it TAB key?
A803   202F        90        JR   NZ,EXIT;out if not
A805   F5         100        PUSH AF;preserve
A806   E5         110        PUSH HL;registers
A807   210101     120        LD   HL,#0101;row, column
A80A   E5         130 LOOP   PUSH HL;save it
A80B   CD75BB     140        CALL #BB75;position cursor
A80E   CD60BB     150        CALL #BB60; get character
A811   CD2BBD     160        CALL #BD2B;print it
A814   E1         170        POP  HL;get row,col
A815   24         180        INC  H;next column
A816   7C         190        LD   A,H;test
A817   FE29       200        CP   41;too much?
A819   20EF       210        JR   NZ,LOOP;back if not
A81B   3E0A       220        LD   A,#0A;linefeed
A81D   CD2BBD     230        CALL #BD2B;print
A820   3E0A       240        LD   A,#0A;again
A822   CD2BBD     250        CALL #BD2B;for uniformity
A825   3E0D       260        LD   A,#0D;c/r
A827   CD2BBD     270        CALL #BD2B;print
A82A   2601       280        LD   H,01;reload
A82C   2C         290        INC  L ;next row
A82D   7D         300        LD   A,L;test
A82E   FE1A       310        CP   26;too much?
A830   20D8       320        JR   NZ,LOOP;back if not
A832   E1         330        POP  HL;restore
A833   F1         340        POP  AF;registers
A834   C34A13     350 EXIT   JP   #134A; rejoin routine

Pass 2 errors: 00

Table used:    47  from   186
Executes: 43000
```

Figure 9.16. Calling a routine with the TAB key. In this example, the routine is the screen print routine.

program by placing its address into #BDD4 and #BDD5. The second part of the program will now run each time a key is pressed. This will happen no matter whether you are using BASIC or MONA3, as long as addresses #BDD4 and #BDD5 are not altered by anything else. The result is that you can use the computer perfectly normally, but when you press TAB, the printer will start up, and the text on the screen will be copied to the printer. It's very useful, because you don't need to program with a whole set of PRINT and PRINT#8 commands if you want both screen and printed output. Because the screenprint is called by pressing TAB, which doesn't place anything on the screen until *after* the routine has run, you get a printout of the screen just as it is, unaffected by any CALL command or by scrolling.

You don't have a printer? If you're serious about computing, you soon will, but in the meantime, the method which I have demonstrated here is still useful. Perhaps you want to print a character with the TAB key – so you substitute a character-print routine in place of the screen-print routine. You might want to sound a noise as each key is pressed, so you omit the test for the TAB key, and put a sound routine into the program. You can make the TAB key clear the screen, fill it with one character, print out a character set, load a tape, anything, in fact, that can be programmed using machine code. That's what machine code programming is all about.

We've now come to the end of this particular road. It's the end of the beginning, because by now you should have a good idea of how to write a machine code program, and what can be gained by doing so. Even more important, you know a lot more about how your CPC464 computer works, and how you can dig some of its secrets out. Your next steps depend a lot on what your interests are. A lot of books on Z80 programming are advanced, but within your reach now. I particularly recommend Tootill and Barrow's book: *Z80 Machine Code for Humans* (Granada-Collins). The book I cut my teeth on many years ago was Lance Leventhal's *Z80 Assembly Language Programming*, which is excellent for its summary of the effect of Z80 commands, particularly if you want to know about effects on flags. I wouldn't be without it. Another book that I have used for a long time is *Z80 CPU – Instruction Set* which was published by the semiconductor manufacturer SGS-ATES in this country. I don't know if it is generally available, or even in print, now. If, as I suspect, it is out of print, then perhaps some day it will be reprinted – it is certainly very useful. For more advanced knowledge of your CPC464, particularly all of these hook addresses, you will need *The Concise Firmware Specification*, from Amsoft. This is expensive, but essential if you are to make the most of your machine. Among magazines, *Personal Computer World* is of particular interest to you now at the start of your career as a machine code programmer. This is mainly because of its *SUBSET* series, but you'll probably enjoy the rest of the magazine as well! Happy assembly, and may you never run out of RAM.

Appendix A
Coding of Numbers in the VLT

The CPC464 uses five bytes of memory to store any number. This Appendix describes how numbers are stored, but if you have no head for mathematics, you may not be any the wiser!

To start with, floating point (not integer) numbers are stored in mantissa-exponent form. This is a form that is also used for denary numbers. For example, we can write the number 216ØØØ as 2.16 ×1Ø^{5}, or the number .ØØØ12 as 1.2×1Ø$^{-4}$. When this form of writing numbers is used, the power (of ten in this case) is called the *exponent*, and the multiplier (a number greater than 1 and less than 1) is called the *mantissa*. Binary numbers can also be written in this way, but with some differences. To start with, the mantissa of a binary number that is written in this form is always fractional, but no point is written. Secondly, the exponent is a power of two rather than a power of ten. We could therefore write the binary number 1Ø11ØØØØ as 1Ø11E1ØØØ. This means a mantissa of 1Ø11 (imagine it as .1Ø11) and exponent of 1ØØØ (2 to the power 8 in denary). There's no advantage in writing small numbers in this way, but for large numbers, it's a considerable advantage. The number:

11Ø1Ø1ØØØØØØØØØØØØØØØØØØ

for example, can be written as 11Ø1Ø1E11ØØØ (think of it as .11Ø1Ø1×2^{24}).

This scheme is adapted for the CPC464, and other machines which use Microsoft BASIC. Since the most significant digit of the mantissa (the fractional part of the number) is always 1 when a number is converted to this form, it is used as a sign bit, and converted to a Ø if the number is positive. The exponent then has (denary) 128 added to it before being stored. This allows numbers with negative exponents up to −128 to be stored without complications, since a negative exponent is then stored as a number whose value is less than 128 denary. The CPC464 uses four bytes to store the mantissa of a number, and one byte to store the exponent.

To take a simple example, consider how the number 2Ø (denary) would be coded. This converts to binary as 1Ø1ØØ, which is .1Ø1ØØØØØ×2^{5}, writing it with the binary point shown, using eight bits, and with the exponent in denary form. The msb of the fraction is then changed to Ø, so that the number stored is ØØ1ØØØØØ. Peeking this memory will therefore produce the number (denary) 32 in the mantissa lowest byte. Meantime, the exponent of 5 is in binary 1Ø1. Denary 128 is added to this, to make 1ØØØØ1Ø1. Peeking this memory will give you 133 (which is 128+5). If you use MONA3 for looking at memory, remember that the codes appear in hex. This can be misleading, because 32 denary then appears as #2Ø, and you might think that you were seeing the denary number stored.

Integers need only two bytes for storage. The integer must have a value between −32768 and +32767. To convert an integer to the form in which the CPC464 stores it,proceed as follows:

(1) If the number is negative, subtract it from 65536 and use the result,
(2) Divide the number by 256 and take the whole part. This is the most significant byte.

(3) Subtract from the number 256*(most significant byte). This gives the least significant byte.

Example
Convert 9438 into integer storage form.
The number is positive, so it can be used directly.

9438/256=36.867187.

The msb is therefore 36.

36*256=9216, and 9438−9216=222.

The low byte is 222.

Appendix B

Converting Between Denary and Hex

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. The number is therefore 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. The number is therefore 3AH.

(b) Double bytes – numbers between 256 and 65535
Example: convert 23815 to hex
23815/16 = 1488.4375. Ø.4375*16 = 7, lowest digit.
1488/16 = 93. Ø is 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.

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*4096 = 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

Appendix C
Z80 Operating Codes, with Mnemonics, Hex Codes, Denary Codes and Binary Codes

The following list of all Z80 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 ØØ for a data or displacement byte, or ØØØØ 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 as 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
	00	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	DEC(HL)	35	53	00110101
	79	121	01111001	DEC(IX+0)	DD	221	11011101
BIT 7,D	CB	203	11001011		35	53	00110101
	7A	122	01111010		00	0	00000000
BIT 7,E	CB	203	11001011	DEC(IY+0)	FD	253	11111101
	7B	123	01111011		35	53	00110101
BIT 7,H	CB	203	11001011		00	0	00000000
	7C	124	01111100	DEC A	3D	61	00111101
BIT 7,L	CB	203	11001011	DEC B	05	5	00000101
	7D	125	01111101	DEC BC	0B	11	00001011
CALL 0000	CD	205	11001101	DEC C	0D	13	00001101
	00	0	00000000	DEC D	15	21	00010101
	00	0	00000000	DEC DE	1B	27	00011011
CALL C,0000	DC	220	11011100	DEC E	1D	29	00011101
	00	0	00000000	DEC H	25	37	00100101
	00	0	00000000	DEC HL	2B	43	00101011
CALL M,0000	FC	252	11111100	DEC IX	DD	221	11011101
	00	0	00000000		2B	43	00101011
	00	0	00000000	DEC IY	FD	253	11111101
CALL NC,0000	D4	212	11010100		2B	43	00101011
	00	0	00000000	DEC L	2D	45	00101101
	00	0	00000000	DEC SP	3B	59	00111011
CALL NZ,0000	C4	196	11000100	DI	F3	243	11110011
	00	0	00000000	DJNZ,00	10	16	00010000
	00	0	00000000		00	0	00000000
CALL P,0000	F4	244	11110100	EI	FB	251	11111011
	00	0	00000000	EX(SP),HL	E3	227	11100011
	00	0	00000000	EX(SP)IX	DD	221	11011101
CALL PE,0000	EC	236	11101100		E3	227	11100011
	00	0	00000000	EX(SP)IY	FD	253	11111101
	00	0	00000000		E3	227	11100011
CALL PO,0000	E4	228	11100100	EX AF,AF'	08	8	00001000
	00	0	00000000	EX DE,HL	EB	235	11101011
	00	0	00000000	EXX	D9	217	11011001
CALL Z,0000	CC	204	11001100	HLT	76	118	01110110
	00	0	00000000	IM 0	ED	237	11101101
	00	0	00000000		46	70	01000110
CCF	3F	63	00111111	IM 1	ED	237	11101101
CP(HL)	BE	190	10111110		56	86	01010110
CP(IX+0)	DD	221	11011101	IM 2	ED	237	11101101
	BE	190	10111110		5E	94	01011110
	00	0	00000000	IN A,(C)	ED	237	11101101
CP(IY+0)	FD	253	11111101		78	120	01111000
	BE	190	10111110	IN A(PORT)	DB	219	11011011
	00	0	00000000		00	0	00000000
CP A	BF	191	10111111	IN B,(C)	ED	237	11101101
CP B	B8	184	10111000		40	64	01000000
CP C	B9	185	10111001	IN C,(C)	ED	237	11101101
CP D	BA	186	10111010		48	72	01001000
CP 00	FE	254	11111110	IN D,(C)	ED	237	11101101
	00	0	00000000		50	80	01010000
CP E	BB	187	10111011	IN E,(C)	ED	237	11101101
CP H	BC	188	10111100		58	88	01011000
CP L	BD	189	10111101	IN F,(C)	ED	237	11101101
CPD	ED	237	11101101		70	112	01110000
	A9	169	10101001	IN H,(C)	ED	237	11101101
CPDR	ED	237	11101101		60	96	01100000
	B9	185	10111001	IN L,(C)	ED	237	11101101
CPI	ED	237	11101101		68	104	01101000
	A1	161	10100001	INC(HL)	34	52	00110100
CPIR	ED	237	11101101	INC(IX+00)	DD	221	11011101
	B1	177	10110001		34	52	00110100
CPL	2F	47	00101111		00	0	00000000
DAA	27	39	00100111	INC(IY+00)	FD	253	11111101
					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
JP 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
	00	0	00000000
	00	0	00000000
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),D	DD	221	11011101
	72	114	01110010
	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	ED	237	11101101
	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,(IX+00)	DD	221	11011101
	7E	126	01111110
	00	0	00000000
LD A,(IY+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,L	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(00),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
	86	134	10000110
RES 0,(IY+00)	FD	253	11111101
	CB	203	11001011
	00	0	00000000
	86	134	10000110
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
	8E	142	10001110
RES 1,(IY+00)	FD	253	11111101
	CB	203	11001011
	00	0	00000000
	8E	142	10001110
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
	AF	175	10101111
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
RETI	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
	C6	198	11000110
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

Appendix D
Z80 Addressing Methods

There are some differences in the names used for the Z80 addressing methods. *Z80 Assembly Language Programming* 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. It is used extensively by the operating system of the Amstrad CPC464 but only one call can be adapted for other uses.

Appendix E
Times for Operations

This table shows the time in terms of clock pulses for a few instructions. For the CPC464 clock operating at 4.00 MHz, the clock pulse time is 0.25 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 at the end of Chapter 9.

Index

Step by step this book lifts the wraps off subjects like the microprocessor, the ROM, RAM and ports, and explains what the all-important registers are and how they are used. Readers are taken through the design of simple machine code programs and introduced to the type of software tools that the professionals use, such as assembler-editors and monitors. The use of an assembler and monitor is clearly described and the book ends with a set of examples which take the beginner well past the L-plate stage.

By the time you have finished this book, only practice will be needed to turn you into a proficient machine code programmer.

The Author
Ian Sinclair has regularly contributed to journals such as *Personal Computer World, Computing Today, Electronics and Computing Monthly, Hobby Electronics* and *Electronics Today International*. He has written over fifty books on aspects of electronics and computing, mainly aimed at the beginner.

COLLINS
Printed in Great Britain
0 00 383079 9

£8.95 net

SINCLAIR INTRODUCING AMSTRAD CPC464 MACHINE CODE COLLINS

AMSTRAD CPC

MÉMOIRE ÉCRITE
MEMORY ENGRAVED
MEMORIA ESCRITA

https://acpc.me/