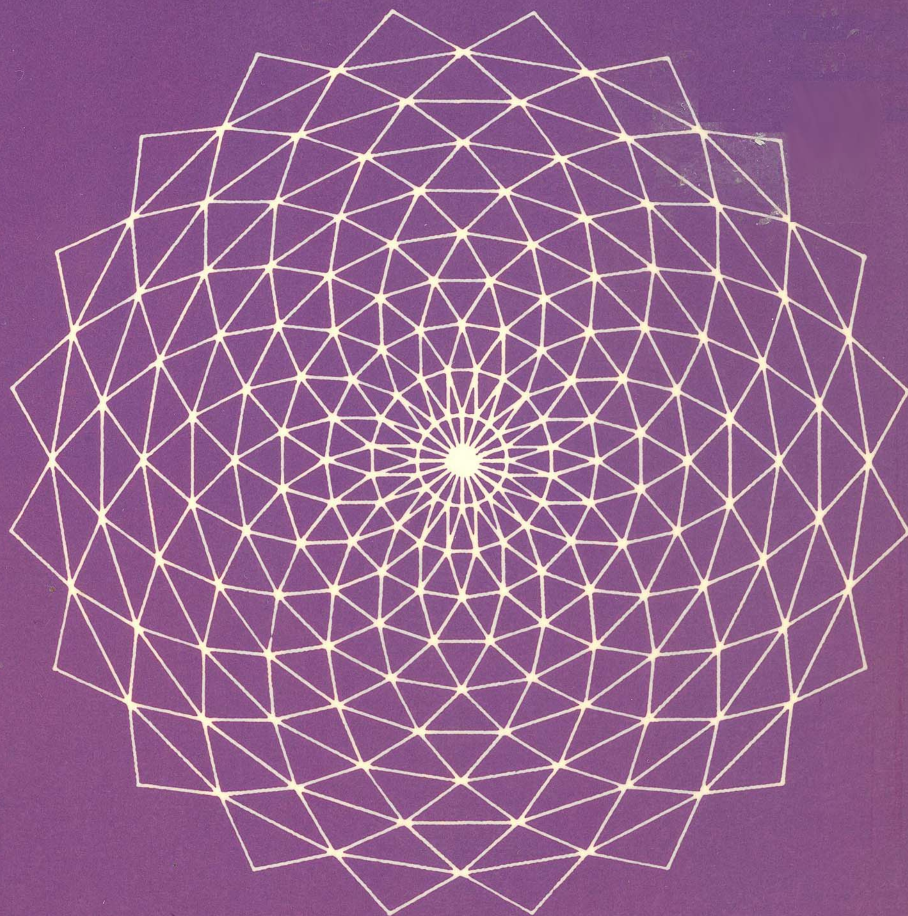
INTRODUCING AMSTRAD CP/M ASSEMBLY LANGUAGE



IAN SINCLAIR

Introducing Amstrad CP/M Assembly Language

Other books for Amstrad users

Introducing Amstrad Machine Code
Ian Sinclair
0 00 383079 9

Advanced Amstrad CPC6128 Computing
Ian Sinclair
0 00 383300 3

Amstrad Word Processing – on the PCW 8256
Ian Sinclair
0 00 383328 3

Introducing C
Boris Allan
0 00 383105 1

Using Amstrad CP/M Business Software
Ian Sinclair
0 00 383359 3

Introducing Amstrad CP/M Assembly Language

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 1986
Reprinted 1986

Copyright © Ian Sinclair 1986

British Library Cataloguing in Publication Data
Sinclair, Ian R.
Introducing Amstrad CP/M assembly language.
1. CP/M (Computer operating system)
2. Microcomputers
I. Title
005.4'46 QA76.6

ISBN 0-00-383309-7

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

All rights reserved. No part of this publication may
be reproduced, stored in a retrieval system or transmitted,
in any form, or by any means, electronic, mechanical, photocopying,
recording or otherwise, without the prior permission of the
publishers.

The cover illustration was taken from *Analysis, Design and Construction of Braced Domes*, courtesy of the Editor, Professor Z.S. Makowski.

Other books for Amstrad users

Introducing Amstrad Machine Code
Ian Sinclair
0 00 383079 9

Advanced Amstrad CPC6128 Computing
Ian Sinclair
0 00 383300 3

Amstrad Word Processing – on the PCW 8256
Ian Sinclair
0 00 383328 3

Using Amstrad CP/M Business Software
Ian Sinclair
0 00 383359 3

Contents

Preface vii

1 ROM, RAM, Bytes and Bits 1

2 Digging Inside the CPC6128 and PCW 8256 16

3 The Miracle Microprocessor 26

4 Register Actions 39

5 Taking a Bigger Byte 54

6 CP/M Interactions 66

7 More Routines 85

8 SID, ED and Family 101

9 Disc Use and Utility Programs 114

Appendix A: Languages Under CP/M 132

Appendix B: Addressing Methods of the 8080 133

Appendix C: 8080 Instructions 134

Appendix D: The ED Commands 137

Appendix E: The ASCII Codes in Hex 139

Appendix F: Effect on Flags 141

Appendix G: Calls to 0005H 142

Index 143

Preface

The user of an Amstrad CPC6128 or PCW 8256 who has used programs running under CP/M is often, very reasonably, inclined to treat CP/M as one big mystery. After all, the original CP/M system, in 1972, was designed for professional programmers in order to allow them to write programs which would run on a wide range of the microcomputers which were then becoming available. At one time, it appeared that CP/M for 8-bit microprocessors might die out, but the appearance of CP/M, particularly the new CP/M 3.0, on the Amstrad, Einstein and Commodore machines has changed all this. There is now a new generation of CP/M users, and among them must be many who would like to know more about CP/M, how it works, and how CP/M programs are arranged.

The first obvious step in this process is to learn and master the language that is used by the 8080 microprocessor which is the language of CP/M. This is not so easy as it sounds. The 8080 microprocessor, has not been used in computers for many years, and its programming language is not one that is dealt with by many modern books. The reason is that the 8080 was superseded by another chip, the Z80, which is the heart of the CPC6128 machine. The Z80 is, however, completely compatible with the 8080, using the same set of number codes, with several enhancements. Any program that is written for the 8080 will run on the Z80, but a program written for the Z80 may not run on the 8080. If you write programs in 8080 language, then, they will run on a machine which uses CP/M on the Z80, or on another chip, the 8085.

This still leaves the snag of learning the 8080 language. Many books seem to be written with the assumption 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 material without explanations, or to material like arithmetic routines which is of little use to most readers. In addition, most books that deal with CP/M were written around the very early versions, rather than around the new issue that Amstrad owners enjoy. Even the CP/M 2.2 of the CPC464 disc system is an older version, though much in this book will be applicable to it.

This book is intended for the real novice to CP/M machine code - the owner who uses his or her Amstrad CPC6128 or PCW 8256 for programs running under CP/M, 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 in CP/M, 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 can be done with CP/M, because very few people can ever hope to do that. What I hope it will do is introduce you to the start of a big topic, and put you in a position to understand some of the whys and hows of CP/M programming on the CPC6128. You'll find a huge new world of computing opening up before your eyes, you'll have a much better understanding of how the CP/M system works, and you'll be able to understand and enjoy the hints and programs that you'll find in the magazines, particularly in the CP/M User Group magazine. In particular, you'll have a clearer understanding of how to modify CP/M programs to your own needs, something that continually turns up when the system is adapted to yet another computer.

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 own computer. To help you, the system itself has several very useful programs, named ED, ASM, SID, and HEXCOM, 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 Digital Research, who have published sufficient details of the structure of CP/M to make life considerably easier for machine code programmers. I must also acknowledge the enormous help that any CP/M programmer gets from the CP/M User Group, among whom Andrew Clarke is a constant fount of wisdom. 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 daisywheel printer.

Ian Sinclair

Note: PCW 8256

The version of CP/M supplied with the Amstrad PCW 8256 is the same as that on the CPC6128. The keys of the PCW 8256 are differently marked, however. The CTRL key is marked ALT and the ESC key is marked EXIT. For details about the use of other keys, see the item in the PCW 8256 manual concerning the SETKEYS file, page 108.

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 out the signal. 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 CPC6128 and PCW 8256 for running programs under CP/M without ever worrying about how these instructions are carried out. There is a difference, though. You can use CP/M on your PCW 8256 or CPC6128 computer much more effectively, modify programs to suit you better, or write routines to carry out operations that are not provided for in standard CP/M, 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 just making use of CP/M 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 the terms. I shall assume that you have no such background. All I shall assume is that you possess a CPC6128 or PCW 8256, and that you have some experience of programming it 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 correct place, which is the beginning. I don't want to have to interrupt important explanations with technical or mathematical details, so these will be found in the Appendices. That 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 on a light, and you never think it's remarkable 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 *b*inary dig*it*, 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 half-way position.

We'll stick with the idea of a switch, because it's very useful for explaining how we use memory. Suppose we want 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 you like 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. 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

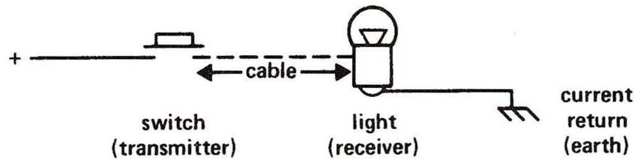
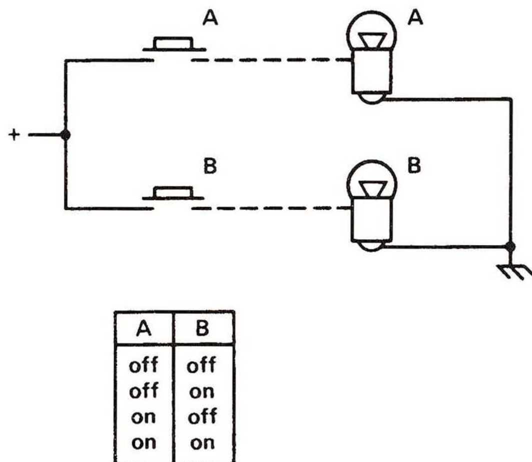


Figure 1.1 A single-line switch and bulb signalling system.



A	B
off	off
off	on
on	off
on	on

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

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 four is 2×2, and eight is 2×2×2, then four lines might allow 2×2×2×2, which is sixteen, codes. Since we usually write 2×2×2×2 as 2^4 (two to the power four), 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 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 CPC6128 and PCW 8256 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, 0 and 1. We can think of 0 as meaning 'switch off' and 1 as meaning 'switch on', so that 256 different numbers could be signalled using eight switches by thinking of 0 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 to indicate a number is the same 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 256 is called the 'least significant digit', and the 2 is the 'most significant digit'. Change the 6 to 7 or 5, and the change is just one part in 256. Change the 2 to 1 or 3 and the change is one hundred parts in 256, 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

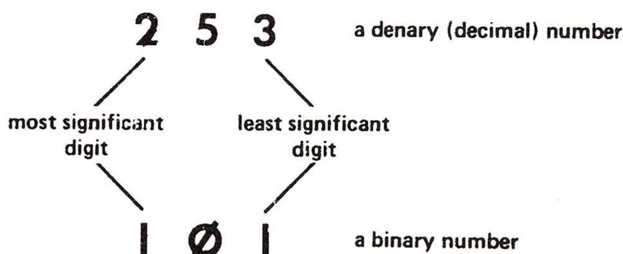


Figure 1.3 The significance of digits. Our numbering system uses the position of a digit in a number to indicate its significance or importance.

connections, because this type has to retain the number-coded instructions that operate the computer. This type of memory 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, and a few parts of CP/M. These instructions are referred to as the 'firmware' of the computer.

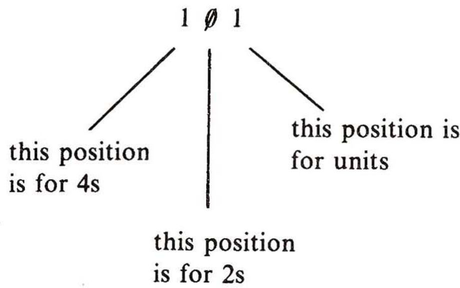
When you write a program in BASIC for yourself, the computer stores it in the form of another set of number coded instructions in a part of memory that can be used over and over again. This is a different type of memory that can be 'written' as well as 'read', and if we were logical about it we would refer to it as RWM, meaning read-write memory. It's this read-write memory that is used mainly by CP/M programs, and the CP/M operating system itself. 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 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 the program will not appear as 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. This is also why the programs in CP/M which occupy this part of memory are called 'transient' – they will always vanish when you switch off, and they will be replaced when you load in a new program.

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 1s and which are 0s. 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 0 or 1, and it means just these numbers. The next digit to the left, however, can also be 1 or 0. The 0 means 0, 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 eight bits is used to represent a number which we would write in ordinary form as 0 to 255 (not 1 to 256, because we need a code for zero). Each byte of the 65536



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.

bytes of RAM in the CPC6128 computer can store a number which is in this range 0 to 255.

Numbers by themselves are not much use, and we wouldn't find a computer particularly useful if it could deal only with numbers beween 0 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 0 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 the one we use for programming words into CP/M, and anything that consists purely of characters in ASCII code is called a *textfile*. All of the program sections that we'll write for CP/M will start out as a textfile, and will be recorded on the disc in that form. This is not the same as the form which is used for storing BASIC programs under AMSDOS, however. When you switch out of CP/M into BASIC, the CPC6128 computer uses its own coded meanings for BASIC words in this range of 0 to 255. For example,

when you type the word PRINT in a BASIC program line, what is placed in the memory of the CPC6128 computer (when you press RETURN) is not the sequence of ASCII codes for PRINT. This would be 80,82,73,78,84, one byte for each letter. What is put into memory, in fact, is one byte only – the binary form of the number 191, meaning 'print'. Memory is a precious commodity in small computers, and using a single byte in this way is an obvious saving of memory. Since this applies only to BASIC, that's the end of it as far as we're concerned, because you can't write programs in Amstrad BASIC when you are using CP/M. That's not quite true, because you can buy a version of BASIC which will run under CP/M, but there isn't much point in doing this unless it is the type of BASIC that is called 'compiled'. There will be more about this in Appendix A.

Back to CP/M, then. CP/M programs are instructions which are written in the form of number codes. These instructions will cause actions to be carried out, 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 important, and it's one reason for using CP/M. A program written in CP/M takes direct control over the microprocessor which is the heart of the computer, and will be able to run very fast, much faster than a program in Amstrad BASIC.

Do-it-yourself spot

As an aid to digesting all that information, try a short bit of self-help. Place the CP/M System disc copy, side 1 uppermost, in the drive, and type:

TYPE LANGUAGE.COM

You can use either lower-case or upper-case for this, as you probably know, but commands will be printed in upper-case in this book to make it clear that they *are* commands. The effect of the **TYPE** command, as you know, is to place on the screen the characters in any file. When you press RETURN on this effort, you'll see some intelligible characters appear, such as a title, a copyright notice and a few error messages, but the rest looks like gibberish. In addition, the loudspeaker hoots at you briefly. This is because only a small part of the file consists of ASCII codes, and these have been printed as the messages that you see. The rest consists of machine code characters, the codes that are the instructions to the microprocessor when it carries out this program.

Now try another tactic. Turn the disc over to side 2, and type:

DUMP B:LANGUAGE.COM (RETURN)

The disk spins, and you will get a message scrolling sideways at the foot of the screen inviting you to insert side B. This you do by turning the disc over, and then pressing any key. The screen display then shows something quite

different. 'Dump' means 'find what bytes are stored in the file', and this is what the display shows. The four-digit numbers down the left-hand side of the screen are reference numbers, a way of checking the position of the codes of the file. The codes themselves are the two-digit numbers, arranged in lines of 16. The numbers are written in a rather special way, called 'hexadecimal' (or hex), and that's something we'll need to come back to later. For the moment, though, you'll see that this is a way of discovering what is contained in a CP/M file without causing a strange characters to appear, or odd sounds. Numbers which correspond to ASCII codes are shown in character form on the right-hand side of the screen.

Using SID

At this point, however, it's more important to start getting used to a utility called SID, the symbolic debugger. SID is a program of the type called *monitor*, one which can tell what is going on in a program. The older version in CP/M 2.2 is called DDT. Since SID is specifically written for CP/M 3.0, it's a vital asset to the understanding of our CP/M programs. SID is a large and complicated program, and we'll look at it in easy stages – some of its facilities will not be needed in this book, and others may never be needed in your applications. The important point about SID is that it can load in a program and then investigate it. To show this in action, turn to side 2 of the System discs. Now type:

SID B:LANGUAGE.COM and (RETURN)

This first loads SID, and then prompts you to put in side 1 of the disc in order to load in **LANGUAGE.COM**.

When all the disc spinning is finished, you will see the message:

NEXT	**MSZE**	**PC**	**END**
0500	0500	0100	DAFF

appear on the screen. These numbers are called *address numbers*. All of the bytes of RAM memory within your CPC6128 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. One action of SID is to find what number, which must be between 0 and 255, is stored at each address. SID automatically converts these numbers from the binary form in which they are stored into the hexadecimal numbers that are normally used for working with CP/M codes. What the message tells us is that the next free address in the memory (NEXT) is 0500 hex, the location following the largest file (MSZE) is also 0500, the microprocessor will start its action (PC) at address 0100, and the end of the usable memory is at DAFF. These numbers show the situation when the file (LANGUAGE.COM in this example) is in memory *by itself*. As it happens, when you are using

SID, the addresses near the top of the memory are occupied by the SID program, and they are quite definitely *not* free! Because of the way that SID loads in and then shifts, the memory addresses just above 0500 appear to be used, but the numbers that you see there are just left-overs. I know these don't look like numbers if you aren't used to hex, but be patient and all will be revealed.

The next thing to do is to try a SID action. SID indicates that it is ready for a command by printing the hash sign (#) on the screen. Try the command:

D0100 01FF then (RETURN)

This will produce a display that at first sight looks very much like the one you obtained using **DUMP**. There is a difference, though. The numbers on the side do not start from zero this time, but from 0100. This is because SID reports the actual location of each byte in the memory, and these numbers show where each byte of the file **LANGUAGE.COM** is located in the memory of your CPC6128. Each of these bytes is stored in RAM memory, and can be altered. To prove this, type **S0102** following the # prompt. This will bring up the address 0102, and the byte which is stored there, **4C**. Now type **53**, and you will see this number printed alongside the other one, so that the line looks like:

0102 4C 53

Now press RETURN, and you will see the next address number, **0103**, appear, with its byte **41**. Type **49** this time, and press RETURN. When the next address comes up, don't type anything, just leave it. In the subsequent addresses, type **43, 4C, leave, 49** and **52**. The word 'leave' is a reminder not to type anything in address 0107, just press RETURN. What you have done is to replace some of the ASCII codes in a program by others. To see the effect, escape from the memory-change routine by pressing ESC, then RETURN. Now use **D0100 01FF** (RETURN) to display your work. This has made a noticeable change to the title has it not? We could even record this altered file back on disc - but that's something for later. The important point is that each program is stored in memory that can be altered, and SID offers one way of carrying out that alteration.

Computer dissection

Now take a look at a diagram of the CPC6128 or PCW 8256 in Figure 1.5. It's quite a simple diagram because I've omitted all 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 is inside. Block diagrams are like large-scale maps which show the main routes between towns but don't show sideroads or town

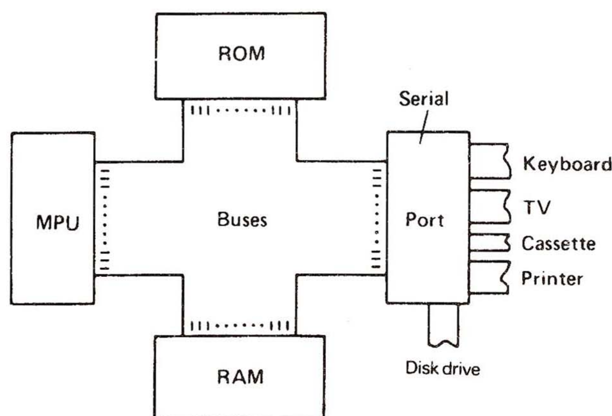


Figure 1.5 A block diagram of any computer. The connections marked 'Buses' consist of a large number of connecting links which join all the units of the system.

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 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 plug-in chunk, a silicon chip encased in a slab of black plastic and provided with 40 connecting pins arranged in two rows of 20. There are several different types of MPU made by different manufacturers; the one in your CPC6128 computer is called Z80 (or Z80A), and it will work with the codes of the old 8080 microprocessor.

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 1.6) are the ones that the MPU spends most of its working life 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 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 CP/M program name is a copying action. You type a letter

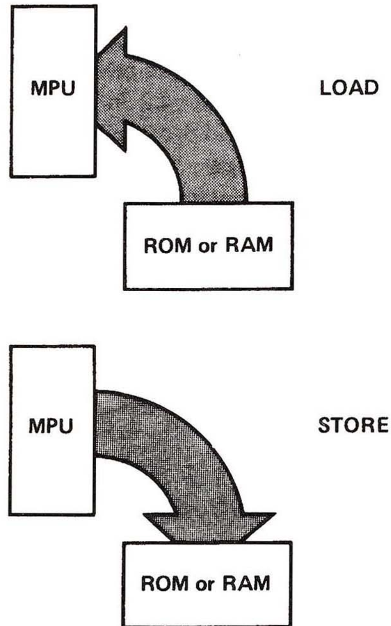


Figure 1.6 Loading and storing. Loading means signalling to the MPU from the memory, so that the digits of a byte are copied into the MPU. Storing is the opposite process.

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 another copying action. After you have typed the program name, it is fetched from the disc by making use of another type of copying action. All of these actions make use of loading and storing, and all are carried out by the MPU. Even when you run a 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 0 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 machine code routines that are part of the program. These routines already exist in the ROM of the CPC6128, but the ROM routines are not used in CP/M, so many CP/M programs need to carry their own arithmetic routines with them.

There's also the *logic set*. Logic means making decisions from given information. In computer terms, this means comparing two numbers so as

AND

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

$$1 \text{ AND } 1 = 1 \quad \left\{ \begin{array}{l} 1 \text{ AND } \emptyset = \emptyset \\ \emptyset \text{ AND } 1 = \emptyset \end{array} \right\} \quad \emptyset \text{ AND } \emptyset = \emptyset$$

For two bytes, corresponding bits are ANDed

```
        1Ø11Ø111
AND     ØØØØ1111
        --------
        ØØØØØ111
```

only these bits exist in both bytes.

OR

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

$$1 \text{ OR } 1 = 1 \quad \left\{ \begin{array}{l} 1 \text{ OR } \emptyset = 1 \\ \emptyset \text{ OR } 1 = 1 \end{array} \right\} \quad \emptyset \text{ OR } \emptyset = \emptyset$$

For two bytes, corresponding bits are ORed

```
        1Ø11Ø111
OR      ØØØØ1111
        --------
        1Ø111111
         ↑
```

only bit which is Ø in both.

XOR (Exclusive-OR)

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

$$1 \text{ XOR } 1 = \emptyset \quad \left\{ \begin{array}{l} 1 \text{ XOR } \emptyset = 1 \\ \emptyset \text{ XOR } 1 = 1 \end{array} \right\} \quad \emptyset \text{ XOR } \emptyset = \emptyset$$

```
        1Ø11Ø111
XOR     ØØØØ1111
        --------
        1Ø111ØØØ
```

if two bits are identical the result is zero.

Figure 1.7 The rules for the three important logic actions, AND, OR and XOR.

Binary number:	∅11∅1∅11	
XOR with –	11∅11∅∅∅	Key
	1∅11∅∅11	Result
	↓	
	1∅11∅∅11	
XOR with –	11∅11∅∅∅	Key
	∅11∅1∅11	Original number again

Figure 1.8 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!

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; Figure 1.7 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 0, then the result is 0, and if both bits are zero, the result is 0. The OR action gives a 0 only when both bits being compared are 0. Only 0 OR 0 gives 0, because 1 OR 0 is 1, and 0 OR 1 is 1. 1 OR 1 is also 1. The XOR action is very similar to the OR action, but when both bits are 1, the result is 0, 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 1.8). 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 XORd again with the 'key', the first number is recovered. Remember when we talk of numbers that this is the raw material the computer uses. By using ASCII codes we can code any message as a number or set of numbers, so these methods apply as much to letters of the alphabet as to simple numbers.

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 the MPU carries out its decision steps. Just as you can program in BASIC:

```
IF A = 36 THEN GOTO 1050
```

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 (in the RAM, when we use CP/M), it carries out the instruction and then reads the instruction byte stored in the next address up. This is very similar to the way 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 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 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 0 bit will cause a 0 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?

Does any CPC6128 owner 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 you do is 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 much 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 18 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 would 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. A CP/M program has been written in machine code (or converted into machine code) so that it will run fast and efficiently, taking up only as much of the memory as is needed. 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 CP/M in the CPC6128. 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 of 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 keys has no effect while a program is being loaded. The port keeps the action of the computer hidden from you until you actually need to have an input or an output. The CPC6128 and PCW 8256 use several ports for such purposes as connections to the keyboard, joysticks, the printer and the disk drive.

Altering memory

We have seen how some of the memory that is used by a CP/M program can be altered, using the **S**(et) command of SID. Now this **S** command is one that can get you into a lot of trouble unless you know what you are doing. You can look at the memory of the computer as much as you like, because the **D** command only copies; it doesn't alter what is stored. Using **S**, however, can replace one byte by another. If the address that you alter happens to contain something that is vital to the way the computer works, then the result will be to send the machine bananas when you try to run the program! 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. When you work with writing or altering CP/M, then, (which *always* involves placing bytes directly into memory) 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. Even if all appears to be well, it's possible to corrupt the memory so that some parts of CP/M do not work. Be warned!

We have now looked at all of the important sections of your Amstrad machine. 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 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!

Chapter Two

Digging Inside the CPC6128 and PCW 8256

I don't mean that 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 CPC6128 and PCW 8256 are designed to load and run CP/M 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 when we start to look at how we can use CP/M routines. We'll start with a simplified version of the action of the whole system, omitting details for the moment.

The ROM of your CPC6128 computer is switched out of action for much of the time when you are using CP/M, but it can be called on at intervals. This ROM consists of a large number of short programs – subroutines – which are written in machine code. The ROM is, in fact, in two parts, one between addresses 0 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, and which are essential at the moment when you switch on the computer. Any language, for example, will need to use the keyboard, the screen, and the disc 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 CPC6128 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 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, and most of the RAM has been cleared for use.

Cleared for use as far as the CPC6128 is concerned means that nothing but zeros will be stored in most of the main bank of 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 continue to store zeros. In each byte, some of the bits will switch to 1 and some will switch to 0 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 0 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. When you switch to CP/M, however, the pattern becomes quite different. For one thing, the computer switches to its second bank of RAM, the spare 64K that BASIC can use only for screen displays or as a RAM-disc. CP/M can use practically all of this memory, though sections at the bottom and at the top are reserved for special purposes. Try this – switch on, select CP/M, and load SID. Now type:

DC000 FFFF

and then press RETURN. The range of memory addresses we have used starts with cleared memory, and ends with a part that is reserved for CP/M use. If this were the main bank of RAM, the locations that we're looking at would correspond to characters on the screen. Since it's in the second bank of RAM, though, it doesn't correspond to anything visible. You'll see at the start of this memory range that the stored characters are either **00** or **FF**, though in the middle of each FF block there are some other characters, always in the same relative positions among the FFs. These are part of the system that CP/M uses to identify parts of the memory. Each block of 00 or FF consists of 256 bytes in all.

The initialising program for CP/M has a lot more to do. Some of the higher section of RAM, from address **DAFF** to **FFFF**, is for 'system use'. This is because the machine code subroutines which carry out the actions of CP/M are stored here, rather than in the ROM, and they also 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. Before we do, though, it's essential to get to grips with the way that these numbers are written in CP/M terms, using the hexadecimal scale.

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 0s in a byte of eight bits, it will consist of numbers between 0 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 CPC6128 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 disc 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 0 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 0s 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. As far as working with CP/M is concerned, though, you don't really have any choice. CP/M was written for the convenience of professional programmers, who use hex numbers, so hex numbers it has to be. 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. A program that is called an *assembler*, then, converts these command words into the correct binary codes. The CP/M assembler, like the other CP/M utilities, shows these codes on the screen in hex form rather than in denary. In addition, when you type data numbers in assembly language, you must make use of hex code. 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 be obvious. Moreover, it's much easier to write down the binary number which the computer actually uses when you see the hex version.

The hex scale

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 0 to 15 in our ordinary number scale. This is the range of one hex digit (Figure 2.1). 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 0 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. Converting between binary and hex is much simpler than converting between binary and denary. The number that we write as 10 (ten) in denary is written as 0A in hex, eleven as

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

Figure 2.1 Hex and denary digits. Note the range of one hex digit.

0B, twelve as 0C and so on up to fifteen, which is 0F. 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 0F is 10, sixteen in denary, and the scale then repeats to 1F, thirty-one, which is followed by 20, thirty-two.

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 number, but a number like 26 might be hex or denary. The convention that is followed by many programmers is to use a capital H to mark a hex number, with the H-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 hashmark *before* the number, so that #47 would mean the same as 47H. When you write hex numbers for a CP/M program on paper, it's a good idea to follow one of these conventions. When you are actually typing hex numbers into the computer under CP/M, though, you don't need to add the 'H' or the '#'. Most of the CP/M utilities assume that you will type in hex numbers, and they will not work with anything else. You might think that this could be awkward, but it's not, because SID provides you with some useful assistance, like a hex arithmetic calculator, of which more later. The only awkwardness is to remember when you *must* use the letter H following a hex number, but we'll come to that later.

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 2.2, you can see that #9 is 1001 in binary and #F is 1111. The hex number #9F is therefore just

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

Figure 2.2 Hex and binary digits. Any single hex digit can be written using up to four binary digits.

Conversion: Hex to Binary

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

So 2CH is ØØ1Ø11ØØ binary (data byte)

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

So 4A7FH is Ø1ØØ1Ø1ØØ1111111 binary (an address)

Conversion: Binary to Hex

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

So Ø11Ø1Ø11 is 6BH

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

Group into fours, starting with lsb:
ØØ1Ø is 2H
1ØØ1 is 9H
Ø11Ø is 6H and
the remaining 1 is 1, making 1692H

Figure 2.3 Converting between hex and binary. This amounts to grouping digits in fours and using the table of Figure 2.3.

10011111 in binary – you simply write down the binary digits that correspond to the hex digits. Taking another example, the hex byte #B8 is 10111000, because #B is 1011 and #8 is 1000. 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 2.3 shows examples of the conversion in each direction so that you can see how easy it is.

Negative numbers

Negative numbers are not very important in CP/M programming, mainly because the old 8080 chip didn't need to use them. On the Z80, you sometimes want the MPU to perform the equivalent of a GOTO, perhaps jumping to a step which is 30 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. On the Z80, therefore, you need to know how to write a negative number in hex and how to recognise one. If, for example, you use the **H** command of SID, followed by two numbers, you'll get the sum of the numbers, and their difference, printed. As often as not, one of these numbers will be negative – but will you know?

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 achieved by a method called complementing, and Figure 2.4 shows how this is done. At first sight, and very often at second, third, and fourth, it looks entirely crazy. For example, when you are dealing with a single byte number, 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

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

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 0. The MPU can find out which is which just by testing the left-hand bit, the most significant bit.

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, −40 uses the same digits as +40. In hex, −40 becomes D8H and +40 becomes 28H. The denary number −85 becomes ABH and +85 becomes 55H. 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 9FH 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 0 to 7FH are positive, and values of 80H to FFH are negative. This corresponds to denary numbers 0 to 127 for positive values and 128 to 255 for negative. Unsigned numbers are always taken as positive. If you find the number 9CH described as signed, then, you know it's treated as a negative number (it's more than 80H). 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 the **H** command of SID, it will not deal with signs in single bytes. If, for example, you type: **H 2A 2B** then what you will see underneath is **0055**, the sum, and **FFFF**, which is the difference. You get FFFF rather than FF, because the **H** command works with hex numbers of four digits. It's not a real problem, because you simply ignore the first two digits when working with single bytes. There's nothing in SID, however, that will convert a denary number into a hex number for you. For that, you have to make use of the HEX$ function in BASIC, and you should know how to use that by this time!

The program in memory

Suppose that you have a CP/M program sitting in the memory of your computer. Just which addresses are used, and for what purposes? To start with, the second bank of RAM has been switched in, and all of it is in use. At the bottom end of RAM, there are some important pieces of code and a few odds and ends in the region from 0000H to 0100H. Take a look at these on your own machine, using SID with **D0000 0100**. You'll see several three-

byte sequences starting with C3 at addresses 0H, 05H, 30H and 38H. If anything happens to these addresses, you can expect trouble. You'll also find interesting things around 0082H, if you have a SETKEYS program arranged so that it auto-boots. If you haven't used this feature of CP/M, and you don't know about it, perhaps I can recommend my book *Advanced Amstrad CPC6128 Computing*, published by Collins. In any case, the SUBMIT file for auto-booting will appear around here. The copyright notice for CP/M appears at around 0140H and is followed immediately by the copyright notice for SID. The SID message doesn't normally appear on your screen – it's just stored on the disc and (when you load it) in the memory. The SID program actually starts high up in the memory, around #E186 – and this address is, in turn, stored at locations #0030 and #0031. They are stored, like all addresses, in reverse order, with the higher byte second. For example, the address E186H is stored as **86 E1** when you see it on the screen. SID will execute a program for you when you type **G** followed by the starting address (then RETURN). You could therefore type **GE186**, press RETURN, and see SID return to its # prompt.

At the upper end of the memory are more bytes planted for special

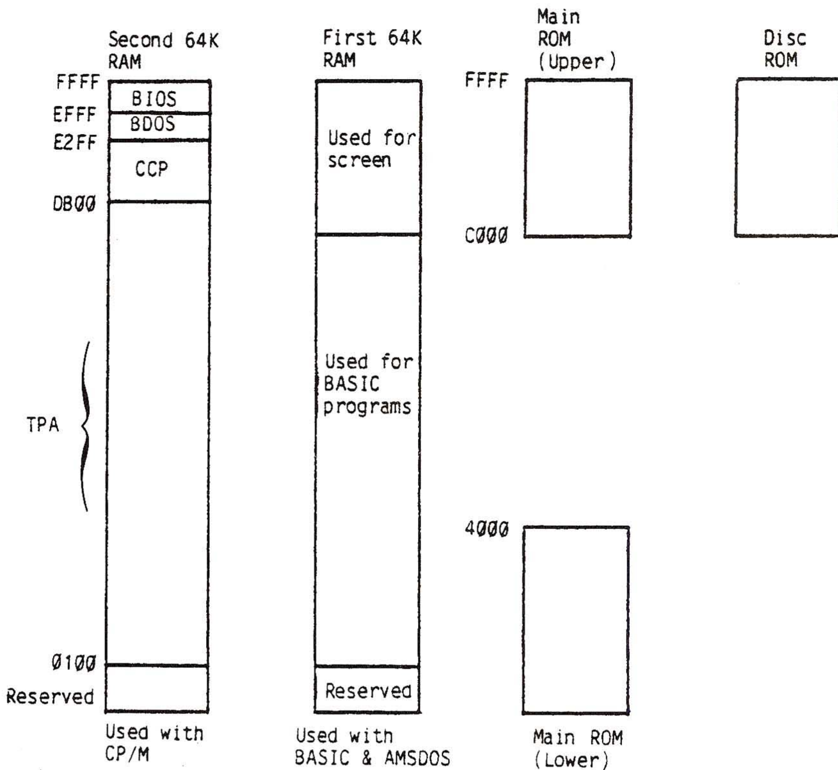


Figure 2.5 The arrangement of ROM and RAM for using CP/M. Practically everything is done using the second set of RAM.

purposes. Everything from address DB00H to FFFFH is forbidden territory unless you know the CP/M system in considerable detail. By the time you have read this book, you'll certainly know a lot more about it, but for real detail, you will need to turn to Digital Research's own manuals, which are weighty, expensive, and very comprehensive. If, however, you aren't going to write programs in CP/M professionally, you won't really need these.

The arrangement of the RAM and ROM for CP/M is shown in Figure 2.5. The region from 0000H to 00FFH is known as Page 0 (no, there are no thrills in addresses 0200H to 02FFH), and from 0100H to around DAFFH is the TPA, the *transient program area.* This is where CP/M program bytes are stored when a program is loaded in, and where any RAM that these programs might need is located. Above the TPA come the fixed programs of CP/M. The part which is labelled CCP means *console command processor*, and it contains addresses for all the routines that relate to the keyboard and the screen. This is the first portion of the reserved part of the RAM, starting at address DB00H (immediately following DAFF, remember), and it extends up to address E2FFH. The next section is called BDOS, and is concerned with disc operations. It extends from E300H to EFFFH. The last section of the memory is occupied with several parts of BIOS, the *basic input/output system*, starting at F000H and going all the way to the end of RAM memory at FFFFH. Consider all of these reserved addresses as being surrounded by barbed wire!

Running a program

Now that we have looked at the way in which a program and the routines of CP/M are stored in the memory of the CPC6128 and PCW 8256 computer, we can give a little thought as to how the program runs. This action is carried out directly by the microprocessor chip, because the whole CP/M program is written in machine code for the microprocessor. Any machine code program must be run by giving a starting address to the microprocessor. This address is, except for SID, the usual 0100 for all computers that run CP/M. Any program that is read from the disc as a COM file will locate itself in memory starting at 0100H, the bottom end of the TPA. Even SID does this, but then gallops up into addresses right at the top of the memory, replacing some of the routines above #DB00, immediately it starts running.

Try, for example, this exercise. With SID in place, type **ELANGUAGE. COM**, and place side 1 of the System discs in the drive. The **E** is the SID command to load a file, and the filename must follow it immediately, as in all the SID commands. When you press RETURN, SID will read the size of the new file **LANGUAGE.COM** from the disc, and then place the new file into the memory starting at 0100. This will wipe out the CP/M and SID copyright notices, and replace them with the bytes of the new file,

LANGUAGE.COM in this case. If you're wondering why I always use **LANGUAGE.COM** as an example, by the way, it's because it's a short file which loads quickly, and doesn't take up much space in the memory. It's not, for some other reasons, the best example to use because it doesn't start in a way that is strictly in CP/M code. This is a problem that you may find with some of the utilities that are specific to the CPC6128, as distinct from the CP/M utilities that have been written by Digital Research for any CP/M machines.

The first two bytes of **LANGUAGE.COM** are 18 6A, and these are Z80 instructions which amount to a direction to go to the address 016C (obtained by adding 6A to 0100 + 2). This is a Z80 instruction which does not appear on the 8080, so if you try to get SID to trace what is happening, the result will be a return to BASIC, or a complete lockup, with none of the keys having any effect. In the second case, you'll have to switch off and then on again. In either case, you'll have to load CP/M all over again. This is just one of the pitfalls that await the unwary CP/M user who is trying to get to grips with this system, and it's another reason why some books that were written in the early days of CP/M are so useless for exploring this new version in its Amstrad form.

Finally, before we really get to grips with the mysteries, another point about programming for CP/M. The important thing about a CP/M program is that it should start at address 0100H. If you want a CP/M program to run on any CP/M machine, including the older machines that used the 8080 and 8085 chips, then you have to write in 8080 code. Tools for writing 8080 code are part of the package of CP/M utilities that you have on your System discs. Nowadays, however, all of the 8-bit computers that run CP/M use the Z80, and the 8080 and 8085 are almost forgotten. This allows you to use Z80 programming aids, like the ZEN program, to write Z80 code, provided you write such programs into the form of a **.COM** file. You can also program in BASIC, provided you buy a BASIC compiler which will run under CP/M and make COM files. This allows you to create programs using BASIC, and then convert them into machine code files of the **.COM** type which will run like any other CP/M file. This is the simplest way of creating large programs to run under CP/M. It may not make the most compact or fastest-running CP/M programs, but it's very much easier than trying to create a large program using only the software tools in the System discs. You can also get compilers which will run under CP/M for the language which is called 'C', also for the very popular language Pascal. Once again, the use of suitable compilers will result in CP/M programs which run fast, will be more efficient than BASIC of any type, and which are much easier to write than assembly language. This is why I shall concentrate in this book on using assembly language for short routines, and for things that aren't provided for in compilers. In addition, though, this book concentrates on 8080 code, whereas the compilers will all generate Z80 code. You can't win 'em all!

Chapter Three
The Miracle Microprocessor

In this chapter, we'll start to get to grips with the way that the 8080 microprocessor works. 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 0 signal at each of the eight 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 CPC6128 computer is very fast indeed. You may be used to clocks that tick once a second, but the 'clock' of the CPC6128 computer ticks 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 them 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 0 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 8080 (and Z80) instructions consist of just one byte, and need no data. Others may be followed by one or two bytes of data, and some (Z80) instructions need two bytes (or more) rather than one, and are also followed by data. When the MPU reads an instruction byte, 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 fed to it as being the data bytes for that instruction. This action of the MPU is completely automatic, and is built into it. 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 CPC6128 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 anything that will cause machine code to be used, on disc 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 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, the type of program that is called an 'assembler' helps considerably. We'll look again at this point shortly. In the meantime, the best way to learn about machine code is to write it, use it, and make your own mistakes.

Registers – PC and accumulator

A microprocessor consists of sets of memories, of a rather different type

from ROM or RAM, which are called *registers*. These registers can be connected to each other and to the pins on the body of the MPU by the circuits that are called gates. All 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 8080 and how they are used. These register types are also used in the Z80, though the Z80 has several additional registers that are not present in the 8080. A good starting point is the register called the *PC* – short for *program counter*.

No, it doesn't count programs – what is does is 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 FFFFH (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 1F3AH, and this address contains an instruction byte, then the PC will increment to 1F3BH 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 0s 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 0s and 1s 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 8080 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 8080 accumulator can't store a number greater than 255 (denary).

The 8080 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 8080 had 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 instructions like PEEK or POKE. The task of finding where bytes are stored is then dealt with by the operating system of the machine. Similarly, when you run and use a CP/M program you need not worry about where the program stores the quantities it is working on, because all that has been take care of. Remembering our comparison with wall-building, though, we can expect that when we carry out machine code programming for ourselves, we shall have to specify each number 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 8080 addressing methods at this stage would be rather baffling, and for that reason has been consigned to Appendix B. What we shall do here is look at some examples of selected addressing methods and the way we write them in 8080 assembly language.

Assembly language

Trying to write machine code directly as a set of numbers is a very difficult

process which is liable to certain 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 must be in hex form for the 8080 assembler program on your System disc. 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 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 (with 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:

```
MVI A,12
```

The operator is MVI, a shortened version of 'move immediate', meaning that a byte is to be copied from the following place in memory into a register. 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, which is always 12 hexadecimal, rather than twelve denary. The use of a MVI instruction rather than the alternative MOV shows that the addressing method 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 12H 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 MVI 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 MVI A, part of the whole instruction, and this byte is 3EH, so that the hex sequence in memory of 3E 12 will

represent the entire command MVI A,12. It's a lot easier to remember what MVI A,12 means than to interpret the numbers 3EH and 12H stored in the memory, however, which is why we use assembly language as much as possible. In Z80 code, incidentally, this instruction uses different mnemonics (LD A,#12), but has the same code, because the code is what makes the action happen in either type of microprocessor.

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, 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

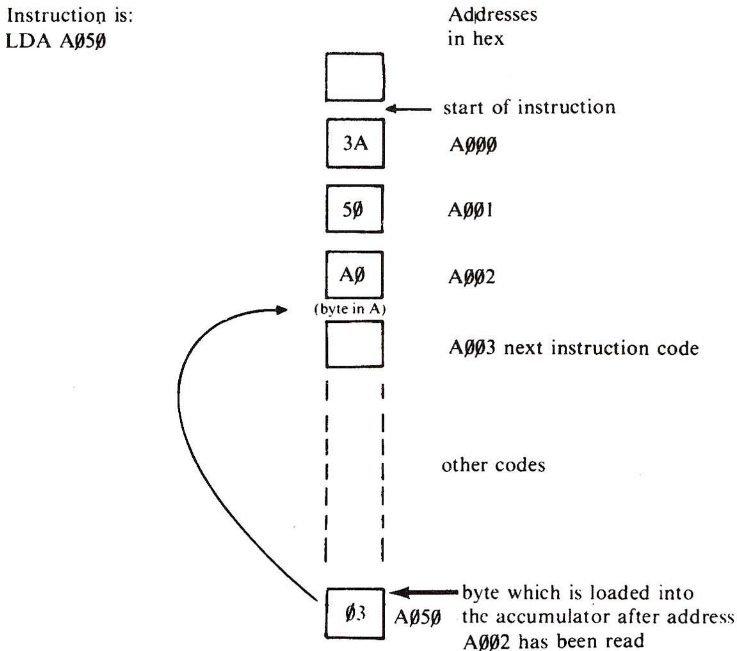


Figure 3.1 How the extended (absolute) addressing method works.

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 microprocessor, 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 3.1 shows in diagram form what has to be done. An extended addressed operation is therefore much 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!

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

```
LDA 7FFE
```

In this slice of assembly language, the operator is LDA, meaning that a load is to be carried out using the accumulator, and from the address 7FFEH. The effect of the action will be to take a copy of the byte stored at 7FFEH and put it into the accumulator. What you have to remember is that when you use LDA 7FFE, what is put into the accumulator A is not 7FFEH, 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 7FFEH into the accumulator A of the 8080. When the instruction has been completed, the address 7FFEH 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 same as the byte that was held in address 7FFEH. In a listing of 8080 instructions, this type of addressing would appear as LDA nnnn.

We can also use the extended addressing method in a command which will store a byte into the memory. The command:

```
STA 7FFF
```

means that the byte that is stored in the accumulator A is to be copied to memory at address 7FFFH. 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.

Register indexed addressing

Immediate addressing and extended addressing (also called direct addressing) are both useful, but the 8080 also permits another type of addressing called *register indexed addressing*. The register indexed

addressing method is so called 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 8080 and is found mainly on this microprocessor and others that have developed from it, like the Z80 and the 8085. In the 8080, the method depends on being able to combine certain pairs of eight-bit registers and use them as if each pair were one sixteen-bit register. This ability to use registers singly or in pairs, as you choose, is just one of the features which made the 8080 such an important microprocessor. The design principles of the 8080, unlike some other early types of microprocessors, were used in many later designs, and have now been carried over into newer sixteen-bit micros. By learning 8080 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 8080, and for convenience, they are labelled 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 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 for register indexed addressing. 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:

```
LXI H,7FFF
```

– taking an example of an address to use. This means that the high byte of the address, 7FH in this example, will be held in the H register (the H should remind you of *H*igh) and the low-byte of FFH 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 the address 7FFFH by a command such as:

```
MOV A,M
```

which means that the accumulator is to be loaded from the byte whose memory address is stored in HL. In our example, this means the byte which is stored at the address 7FFFH, and the letter **M** is used to remind us that we are loading from memory. 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:

```
MOV M,A
```

then the byte in the accumulator is copied to address 7FFFH, assuming that this is still the address that is stored in HL. Remember that the letter **M** used in this way means the memory address that is stored in HL, 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 LDA,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 7FFFH, then the instruction DCX H will make the address number which is stored in HL equal to 7FFEH, one less. If we now use MOV A,M again, then the load will this time be from the address 7FFEH, not 7FFFH 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 MOV A,M instruction and one DCX H instruction in the loop. You could, of course, just as easily use INX H 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 INR M and DCR M 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.

The other 8080 registers

We've mentioned a number of the 8080 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 8080, keeping track of the memory location which is being used. 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. For any CP/M program, this first address will always be 0100H. 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 this is part of the action that is carried out when you type the name of a **.COM** file in CP/M and press RETURN.

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 B, C, D, E, H and L which can be used in the way that we use the accumulator itself, though none 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. These register pairs can then be used much as we used HL, but in place of commands like MOV A,M, we use commands such as LDAX B and LDAX D. A limited number of sixteen-bit arithmetic operations are also possible in the HL register pair. 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 more detail later. One important register remains, however, the flag register.

The flag register

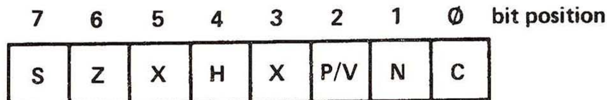
The *flag register*, usually referred to as the *F register* but sometimes called the *status register*, isn't really a register like the others. You can't *do* anything with the bits in this register apart from test them, and they don't even fit together as a number. What the flag register is used for 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 0 to value 1 to bring this to the attention of the MPU. If you add a number taken from memory to the number in an accumulator, and the result consists of nine bits instead of eight (Figure 3.2) then another of the bits in the flag register is 'set', meaning that it goes from 0 to 1. If the most significant bit in a register goes from 0 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

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 3.2 Why the carry bit is needed.



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 3.3 The bits of the flag, or processor status, register. Only three of these, N,Z and C, are extensively used in most programs. The diagram applies to both 8080 and Z80, but flag 2 is used only for parity detection in the 8080.

that you can make branch commands depend on whether a flag bit is set (to 1) or reset (to 0).

Figure 3.3 shows how the bits of the flag register of the 8080 are arranged. Of these bits, numbers 0, 6 and 7 are the ones we are most likely to use at the start of a machine code career. The use of the others is rather more specialised than we need at the moment. Bit 0 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' was needed. It can for some purposes be used as a ninth bit for the accumulator, particularly for rotate operations in which the bits in a byte are all shifted by one place. The carry bit is used by all of the addition and subtraction operations of the 8080, 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 (or store) will set flags, and many operations in other registers do also. You have to be careful, however, to make sure when you are using flags, that they will have been affected in the way that you expect by the previous operation.

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 (0) 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 CMP (*compare*) action will set or reset this flag *without* actually carrying out the subtraction action. The CMP 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 8080 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 STC instruction byte, and you can ensure that the carry flag is reset by using XRA A command, 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, though, you won't ever want to alter the contents of the flag register. You very seldom know or care about what is stored in it, and its main importance to you is that it controls jumps. To make another comparison with BASIC, suppose 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 is 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:

```
JZ nnnn
```

where JZ is the operator, meaning jump if zero, and the operand consists of the address nnnn, four hex digits. If the accumulator contains zero, then the new address which has been shown as nnnn above will be put into the PC, and execution will then start from that point. If Z=0, meaning that the accumulator does *not* store a zero, then the jump instruction 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 JZ part will cause the 8080 to check the Z flag in the flag register.

One peculiarity of the 8080, which was carried over to 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 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:

```
MOV A,M
DCR A
LDAX D
JZ nnnn
```

then the flags are unaffected by the LDAX D step. If the DCR 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 LDAX D step, even though this step places a new byte into the accumulator so that it no longer contains zero. The jump at the JZ nnnn 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 3.4 shows a map of all the 8080 registers. The registers are shown in pairs to indicate the grouping for sixteen-bit use. The groupings HL, BC and DE are already familiar to you, but you will see that the accumulator A and flag register F are also grouped as AF, but abbreviated to PSW (processor status word). 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 point later when we look at stack use.

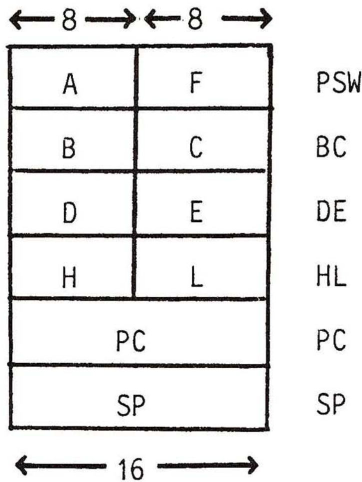


Figure 3.4 A 'map' of the 8080 registers, all of which are present in the Z80. The Z80 has additional registers which we ignore when programming for CP/M.

Chapter Four
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 8080, which means also in some of 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, and particularly CP/M 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 8080 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 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 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:

```
LDA SOURCE
STA DEST
```

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. 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* valid 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 a language like BASIC. Suppose, for example, that you have a BASIC program that carries out 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 important reason for doing this is that it needs to be assigned just once. If the rate of VAT changes to .18, all you have to do is 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 example shows the accumulator being loaded using extended addressing, but you could, of course, load the accumulator *immediately*, and then store the byte using extended addressing. You could also load or store using an address in the HL registers, using MOV A,M and MOV M,A, or you could use addresses that were held in the BC or DE register pairs, using LDAX and STAX commands. There are always many options for addressing methods when you carry out any action that involves loading the accumulator. Examples in this book will not show every possible addressing method for each example, and you will simply have to get used to the options gradually by experience.

The next most important group of accumulator actions is the arithmetic and logic group, which contains addition, subtraction, increment, decrement, AND, OR and XOR. We can add to it the rotate action which we looked at briefly in the previous chapter. What we decide to place in this group is rather arbitrary, and many books place the CMP command also in this group. We'll start with the add and subtract operations. The 8080 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 ADI 0A (add immediate to accumulator) will be to add the number ten (0AH) to the contents of the accumulator. The carry flag is ignored when the addition is carried out, but it might be set *following the* addition. For example, if the accumulator register contained 2BH (denary 43) before ADI 0A, then the result will be that the accumulator contains 35H (denary 53), and the carry bit is reset (to 0) because there is no carry. On the other hand, if the accumulator had contained the number F9H (denary 249), then the effect of ADI 0A would be to leave the accumulator storing #03, and the carry bit set. Why? Because F9H+0AH=103H, and since the accumulator can store only one byte (the lower two digits of this hex number), then it stores the 03H, 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, ACI. ACI means 'add immediate with carry', and it will *always* add the carry flag number to the other number. If, for example, we have the number 2BH in the accumulator and then use ACI 0A, then the result will be 35H as before if the carry bit was reset (0) before the addition. If the carry bit was *set* however (equal to 1),

then the result of the addition would be 36H, 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 more common forms of these two commands are ADC for add with carry, and ADD for add, no carry. The subtraction commands SUI and SUB (ignore carry) and SBI and SBB (use carry) perform in the same way. When SBB or SBI 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. Addition and subtraction can also be carried out with some sixteen-bit registers, notably HL. When this is done, the result is also stored in the same sixteen-bit register.

The INR and DCR commands are straightforward by comparison. INR and DCR can be used to increment or decrement any of the single-byte registers, so we can use commands such as INR A, DCR C, INR 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. INR and DCR can also be used with M (the byte whose address is stored in HL). The Z and S flags will be affected by these commands also, but the C flag will not. There are also INX and DCX commands for incrementing and decrementing the double registers (HL, BC, DE, and SP). These INX and DCX steps do *not* affect any of the flags in the status register. As we'll see later, this is something that can be a nuisance, but we can get around it.

The logic commands ANA, ORA, XRA *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 the accumulator contains the byte 3EH, which in binary is 00111110, and that the C register contains the byte ABH, binary 10101011. The result of ANA C will therefore be the AND of 3EH and ABH which is 2AH, binary 00101010, 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, with command words ORA and XRA. 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.

The rotate action

The effects of the 8080's rotate commands, with their assembly language mnemonics, are shown in Figure 4.1. The Z80 allows a range of shift

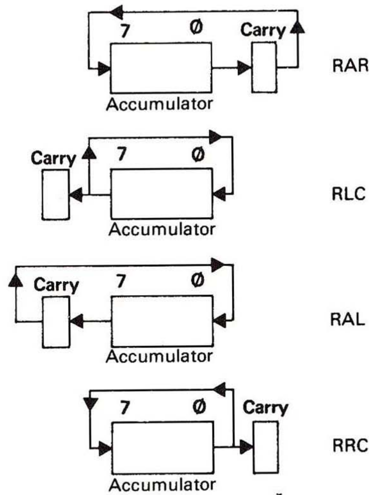


Figure 4.1 The 8080 rotate instructions.

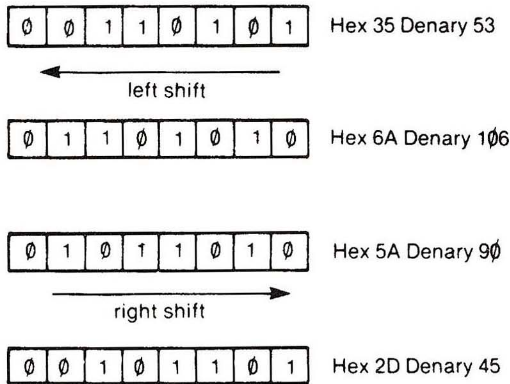


Figure 4.2 The effect of a shift on a number. This is also the effect of RAL and RAR when the carry is cleared before each rotation step.

commands, but these were not implemented on the 8080, which uses only rotation. The difference is that almost every type of Z80 shift results in a register losing one of its stored bits (the one at the end which is shifted out) and gaining a zero at the opposite end. The carry bit is used as a ninth bit of the accumulator in all of the Z80 shifts. The shift action can be carried out on the accumulator, on a byte in any of the other eight-bit general-purpose registers, or on a byte that is stored in the memory and addressed by (HL). The effect of a shift on a binary 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 4.2). A rotation, by contrast, always keeps the same bits stored in the

register, but the *positions* of the bits are changed. The 8080 has two directions of rotation, left and right, and two ways of using the carry flag – as part of the rotation or not. There are four possible rotate commands in all, of which you are likely to use no more than two in the course of most of your programs. All of the rotations are carried out exclusively in the accumulator.

The CMP (compare) instruction is a particularly useful one which appears in almost every program. The command applies *only* to a byte stored in the accumulator. It can use three of the standard memory addressing methods, and its effect is to compare the byte copied from the memory with the byte 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 the accumulator contained the byte 4FH, and we happen to have the same size of byte stored at an address 727FH which is held in the HL register pair. If we use the command:

```
CMP M
```

then the zero flag in the F register will be set (to 1), but the byte in the accumulator will still be 4FH, and the byte in the memory will still be 4FH. 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 4EH (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 59H, 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 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 JZ nnnn and JNZ nnnn. JZ nnnn means 'if zero flag set, then jump to address nnnn'. As this suggests, it will cause a jump if the result of a subtraction or comparison is zero. Its 'opposite number', JNZ nnnn means 'if the zero flag is *not* set, then jump to address nnnn'. There are, therefore, two branch instructions which test the zero flag, but in opposite ways. The

Mnemonic	Effect
JC a	Jump to address 'a' if carry flag set.
JM a	Jump to address 'a' if sign flag indicates negative.
JMP a	Jump to address 'a' unconditionally.
JNC a	Jump to address 'a' if carry flag not set.
JNZ a	Jump to address 'a' if zero flag clear.
JPE a	Jump to address 'a' if parity even.
JPO a	Jump to address 'a' if parity odd.
JZ a	Jump to address 'a' if zero flag set.
PCHL	Jump unconditionally to address held in HL.

There is one difference between the 8080 chip and the Z80 chip as regards the use of flags. The P flag is used by the 8080 only to indicate parity, but the Z80 uses it also to indicate an overflow in arithmetic. If you are writing programs which need to use this flag, you should make sure that it is being set only by the action that you want.

Figure 4.3 The complete list of 8080 jump instructions. The Z80 uses an additional set of 'jump-relative' instructions.

same sort of thing goes for most of the other flags. There are two jump instructions which make no test of flags, and these are called the unconditional jumps. One of them is JMP nnnn, and the other is PCHL, which will take a jump to the address that is held in the HL register pair.

The complete list of all the available jump instructions is shown in Figure 4.3. Many of these are instructions that you'll probably never use, and the really important ones use the zero, carry and negative flags. The Z80 allows a larger range of jumps, and in particular, a type known as *relative* jumps. These allow a change of address by up to 127 steps forward or back from the address at which the jump is taken, and they have considerable advantages for writing 'position-independent' code. This means that it's possible to write machine code that can be placed anywhere in memory and will run, unlike CP/M, which must be placed in the addresses which start at 0100H. Since we are concerned purely with CP/M in this book, we shan't waste any time regretting the relative jumps.

Interacting with the CPC6128 computer

The time has come at last to start some very elementary practical machine code programming of your CPC6128 and PCW 8256. This is not simply a matter of typing the assembly language lines as if they were lines of BASIC. Unless you happen to have an assembler program in operation, CP/M will

simply give you '?' messages when you try to type these program instructions. Since we want to start on a small scale, we'll forget about large-scale assemblers at the moment, and make some more use of that very valuable utility, SID. Make sure that you have a copy of a disc formatted as a CP/M System disc, unprotected, with SID in place. What we need now is a piece of memory that is safely roped off for our use only, one that won't be used for any other purpose by the computer. This is comparatively simple with the aid of SID. As you know, SID loads in initially at the usual address of 0100H, but then immediately 'relocates' itself higher in the memory, beyond the end of the TPA, and taking the place of the CCP. This allows space for another program to be loaded into 0100 using the **E** command of SID, or if the other filename is used following SID when SID is called. The relocation of SID leaves the bytes of the original copy in the memory, however, and we need to clear this so that we can see what we are doing in the memory, without the complication of trying to figure out which bytes we have put in, and which are left over from SID's brief stay. What we shall do to create a convenient piece of memory, then, is make a blank program, consisting only of zeros, which will occupy memory addresses 0100 to 01FF. This is a comparatively tiny piece of memory, but it will be quite enough for all our early efforts. As it happens, the simplest way of clearing this space so that we can use it over and over again is to make a file of it. We start by loading SID in the usual way, and then looking at what's in memory by using **d0100**. Now type:

F0100,01FF,0

and press RETURN, which will have the effect of making every byte stored from 0100 to 01FF a zero. This clears the piece of memory that we want. Now make this into a disc file that you can load again by typing:

WSPACE.COM,0100,01FF

and press RETURN again. Now you will have a blank file called SPACE.COM on the disc. Because this is a .COM file, it will load into the correct part of memory when you call it with SID, and it also starts at the correct address for creating CP/M programs. Now leave SID, and use **DIR** to check that you have the SPACE.COM file on the disc. It's also a good idea to remove the disc, switch off, then on again, and check that you can boot up CP/M, then SID, and then SPACE.COM. You can check that you have the spaces present by using the **D** command of SID.

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 8080. Fortunately, the use of SID along with this blank space does it all for you. Since the blank space starts at 0100, and SID will allow you to put code into it, all the needs for a CP/M program are fulfilled, but you now 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 CPC6128 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 CP/M operating system of the CPC6128 computer, you need to end each machine code program with a JMP 0000 instruction. This will allow a return to CP/M without upsetting anything. For all our early efforts, however, we *don't* want to return to the main part of CP/M. The reason is simply that we shall want to find out what the program has done, and for that we need SID. We need to end our first set of programs, then, with a return to SID rather than to the A> prompt of CP/M. To do this, we must end each program with the command **RST 6.** Most other versions of CP/M on other machines use RST 7 for this, but for your Amstrad is has to be RST 6. Later on, we'll look at other ways of returning, not all of which can be used with equal certainty of success unless we exercise a little care along with them.

When we make use of SID for programming and for checking programs, we are relieved of several worries about what our program does to any other CP/M program which may be in the memory. If, for example, we were 'patching' (adding a chunk of code to an existing program) we would have to take a great deal of care about what our program did. If, for example, our program used the A and BC registers, it would replace anything that happened to be stored in these registers by the main program. Unless we replaced that data after running our piece of program, the results would be, at the least, unfortunate. One way to deal with this is to place such essential quantities into a part of the RAM memory called the *stack*. This, incidentally, is another good reason for being careful about where you place your machine code in the memory. If you wipe out the stack, neither CP/M nor the CPC6128 computer will like it! When you use SID for running small pieces of program, however, and end with the RST 6 command, all of this 'housekeeping' is taken care of for you. If, however, you use a piece of program for patching, or even by itself, then you may have to attend to these salvage operations for yourself as part of your machine code program. This is rather more advanced programming than we want to get into at this stage, 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 make you familiar with the way in which programs are placed into the memory of the CPC6128 and PCW 8256. You will also gain some experience in writing assembly language and placing it into memory with SID, and in 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. As written on paper in assembly language, it reads:

```
ORG 0100     ;start placing bytes here
MVI A,53     ;place hex 53 in accumulator
STA 0120     ;store it at 0120
RST 6        ;go back to SID
```

The first line contains a mnemonic, ORG, which you haven't seen before. It *isn't* part of the instructions of the 8080 or 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 the start address for CP/M here, the address which you would use if you were writing a program that would exist by itself. As often as not, though, you will be adding to an existing program, and so using quite different addresses. When, later in this book, you start to program using an editor and separate 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 SID, the line is not typed, and it simply acts as a reminder of what addresses to use. In particular, you have to remember to create the SPACE.COM file in order to rope off the piece of memory from 0100 to 01FF which we shall partly use. 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. For the moment, keep these comments on paper only, because there is no point in typing them into SID's assembler.

Now we need to look at what the program is doing. The first real instruction is to load the number #53 into the accumulator. This uses immediate addressing, so the number #53 will be placed by the assembler immediately following the command code. The next line commands the byte in the accumulator (now #53) to be stored at address 0120. This is an address in hex, remember, and is well above the few addresses that we shall use for the program. Obviously, we wouldn't normally want to use an address which was also going to be used by the bytes of the program. This instruction uses direct addressing. Finally, the program ends with the RST 6 instruction, essential for ensuring that life with SID continues normally after our program ends.

The next step is to put this lot into memory as code. To do this, we use the assembler command of SID. With the # prompt of SID showing on the screen, type **A0100**, and press RETURN. The **A** means assemble, and the address which follows is our origin address at which we want assembly to start. When this command is obeyed, SID shows the starting address of

```
0100: 3E 53 32 20 01 F7 00 00 00 00 00 00 00 00 00 00 >S2.............
0110: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
0122: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
0130: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
0140: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
0150: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
0160: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
0170: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
0180: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
0190: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
01A0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
01B0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
```

Figure 4.4 The coded bytes in memory, revealed by the **D** command of SID.

0100, and you can then type the first instruction of the program, MVI A,53 and then press RETURN. All you'll see is that SID takes a new line, and shows the address 0102. You can now type the second instruction, and SID will show the next free address 0105. You type the RST 6 into this one, and SID shows 0106. At this point, pressing RETURN without typing anything will cause SID to return to its # prompt, to await another order. The address numbers that appear at the left-hand side are the address for the next free memory space. The first instruction takes two bytes, in addresses 0100 and 0101, so the next free space is 0102, and so on. You can now take a look at the code in the memory by using **D0100** to examine a block of memory from 0100 onwards, and you'll see a display rather like the example in Figure 4.4.

The next step is to run the program and see if it does what it should. Since it will return to SID after running, it should be safe, but it's better to record *any* program of this kind before you make any attempt to run it. To record this, you use the **W** command of SID to record the whole block of memory. By typing **WTEST,0100,01FF**, and RETURN, the block of memory will be recorded under the filename TEST (obviously you can use whatever filename you like). This ensures that the program will be available for correction if by any chance something goes wrong. You can now run the program by using the **G** for go command. Type **G0100** and SID will execute your program. The message which appears on the screen will be ***0105**, marking the place where the RST 6 command was put, the end of the program. You can now use **D0100** to see that the byte 53H has been put into memory address 0120.

The next thing to try is a single step of the program. This is a SID facility that we normally use when a program is misbehaving, but it's better to get the hang of using it on a program that is working correctly. Like practically everything else in CP/M, it isn't as straightforward as it looks, and this is a good time to start sorting out its peculiarities. Single stepping means that you trace the action of the program one step at a time, using the **T** command of SID. You can't just rush into this, though. The **T** command starts its work from whatever address is in the program counter. If you have just run the program, the PC will be at address 0105, the point at which the program ended with a return to SID. To check this, type **XP** and RETURN. The **X** means examine, and the **P** means the PC. If the address which appears is 0105, you can change it back to 0100 simply by typing 0100 and then RETURN. Once you have 0100 in the program counter, you can press **T** to make the first instruction of the program run. Pressing **T** again will give the next step, and so on. You cannot step beyond address 0105, however, where the RST 6 has been placed, because **T** will not dive into the depths of SID in this way!

The sort of lines that you will see on the screen by using **T** are illustrated in Figure 4.5. For each line, you get a display of flags, registers, and instruction, and underneath this line, the address at which the next instruction starts. The flags appear at the left-hand side. The example

```
#
-------- A=53 B=F705 D=0105 H=0100 S=F55E P=0100 MVI  A,53
*0102
#t
-------- A=53 B=F705 D=0105 H=0100 S=F55E P=0102 STA  0120
*0105
#t
-------- A=53 B=F705 D=0105 H=0100 S=F55E P=0105 RST  06
*0105
```

Figure 4.5 The information you get from the single-stepping **T** command of SID.

program does not affect flags, so nothing significant will appear here. Of the registers, only A and P are of importance to us at present, but note that the other registers are being used by SID. The instruction that has been carried out in the line appears at the right-hand side. This trace facility is very useful, and it allows many variations, such as tracing several steps. Try, for example, setting the PC back to 0100, and then typing **T3** to trace all three lines of the program.

Now this first example of machine code isn't an ambitious piece of work; it does no more than a POKE 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 assembly

```
0100  LXI  H,0120
0103  MVI  M,A5
0105  MOV  A,M
0106  INX  H
0107  RAL
0108  MOV  M,A
0109  RST  06
```

Figure 4.6 An assembly language program which loads a byte, rotates it, and saves it in memory again.

language and how you place it into memory, run it, and trace 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 4.6 shows the assembly language version of the program. What we are going to do is load a byte into the accumulator, rotate 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 addressing with the HL register pair, so we shall start by placing an address into the HL registers. This is the LXI H,0120 step, and its effect will be to place the address #0120 into the HL register pair. The next line, MVI M,A5 means that the memory at address 0120 (held in HL) is to be loaded immediately with the byte A5, which in binary is 10100101. The next step is to load the accumulator from this address, using MOV A,M. Normally, we would just have loaded A5 directly into the A register, but I've chosen this way to illustrate the use of the HL register pair. The next step is INX H, incrementing the address stored in HL to 0121, because this is where we shall put the byte after processing. Now comes the processing, in the shape of RAL, a rotate left, followed by a MOV M,A to put the result back into memory at the address held in HL, which is now 0121. We end, as before, with the RST 6 instruction.

Now we can assemble all this into code form, using SID in the same way as before. Once this has been done, ensure that the PC is at 0100, and trace your way through it. If you have started with everything fresh, the first steps should show no flags set, but you will start to see flags being set as you work your way through. The one to look for is the C flag, which appears at the extreme left-hand side when the rotation is carried out. This is because the byte that we chose to rotate started with a 1, and this 1 has been shifted into the carry flag position, setting the flag. Otherwise, the effect of RAL is just as you would expect for a left shift. If you run the program again, however by putting the PC back to 0100 and tracing again, you'll find a different effect! This is because the carry flag is still set, and it is shifted by RAL into the least significant position of the byte, making the result one more than it was the previous time. If you want to use the RAL as a simple shift, then you need to clear the carry bit before you start by using the sequence STC CMC (set carry, then complement carry). You can try putting different numbers into this program, and you will find that if the carry flag is clear, and the number

is less than 128, then the action of the shift is to double the value of the number. If a carry bit is present, however, the answer will be one unit more. The answers will be correct for numbers up to 127 (denary), which is 7F hex, but from 128 up, hex 80 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.

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 4.7 shows the assembly language version, which starts by loading an address, #0120, into the HL register pair just as we did previously. The next step is load the accumulator from this address, using MOV A,M. What we want to do is to load the byte that is stored at address #0120 into the accumulator. Having done that, the next step is INX H. This will increment the address to #0121. With the HL address incremented, the byte in the new address #0121 is added to the byte in the accumulator. The address 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 #0123, after which the program returns.

Type this in, using SID, and check that it is correct. If you have scrolled it off the screen, you can get a new version by typing **L0100,101A**, which is a command for a disassembly. Before you can run the program, you need to

```
0100  LXI  H,0120
0103  MOV  A,M
0104  INX  H
0105  ADD  M
0106  INX  H
0107  ADD  M
0108  INX  H
0109  MOV  M,A
010A  RST  06
```

Figure 4.7 An assembly language program which uses register-indirect loading to perform some additions.

put in the numbers, and this can be done by using the S command of SID. Type **S0120**, and you will see the address 0120 appear on the left-hand side of the next line, with the content of this address, 00, next to it. Now type a number to put in here, say 15. Remember that this, like all other numbers in this system, is in hex. When you press RETURN, the number will be entered, and the next address is displayed, ready for another number. Try 12 here, and 27 in the third address. When you have entered these three numbers, press ESC and RETURN to escape from the number entry routine. You can now use SID to single step its way through the program, watching the numbers accumulate in the accumulator. As an alternative, you can just use **G0100**, and watch it all happen a lot faster. The end result will be the same – if you have used the suggested numbers it will give **4EH** in memory address **0123**. You can try changing the numbers for yourself, but remember that if the sum exceeds FF hex (255 denary), then what is printed will be only the remainder, the lowest eight bits, remaining in the accumulator. If the sum is 12CH (300 denary), for example, then what remains will be **2CH**. If the sum is 2BCH, then what remains will be **BCH**.

For a last example in this chapter, let's look at something a little 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. Figure 4.8 shows the assembly language version of what we're doing in the program. The HL register pair is loaded once again with the address #0120. This address will be used to store an ASCII code number for a letter. The accumulator is loaded from this address, and the HL address is incremented. The accumulator is then ORd with #20. The principle here is that if the ASCII code for an upper-case (capital) letter is ORd with #20, the result is the ASCII code for the same letter in lower-case. The same result can be achieved by adding #20, but this seems a good chance to see the OR action working. The next line of the assembly language program stores the accumulator content back into memory at address #0121 and we return to SID. Now place a byte such as **56**, the ASCII code for V, in the memory at 0120, run the program, and look at the memory contents using **D0100**. You'll see on the ASCII printouts at the right-hand side that V and v are placed next to each other, indicating that the conversion has been effective. The next thing to do is make a loop of this action – but that's going too far for the moment!

```
0100  LXI  H,0120
0103  MOV  A,M
0104  INX  H
0105  ORI  20
0107  MOV  M,A
0108  RST  06
```

Figure 4.8 An assembly language program for converting from upper-case to lower-case letters.

Chapter Five
Taking a Bigger Byte

The simple programs that we looked at in Chapter 4 don't do much, though they are useful as practice in the way that CP/M assembly language programs are written. Practising assembly language writing, seeing it converted into machine code, and tracing its action 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 assembly language program, particularly when you are still struggling to learn the language! Most beginners' difficulties arise, oddly enough, because assembly language is so simple, rather than because it is difficult. Because assembly language is 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 assembly language.

Flowcharts

Flowcharts are to programs that 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 a type of action. Figure 5.1 shows some of the most important flowchart shapes for our purposes (taken from the British Standard 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, or next to them, we can write brief notes of the action we want, but once again without details.

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

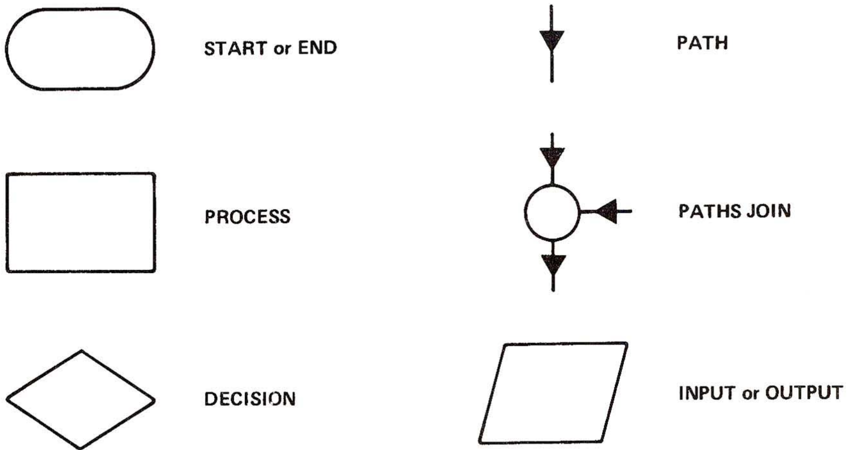


Figure 5.1 The main flowchart shapes.

Figure 5.2. 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 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

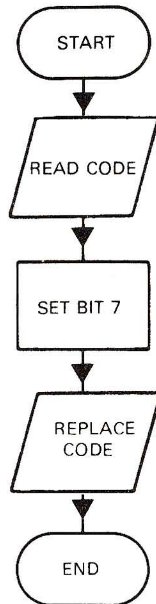


Figure 5.2 A flowchart for a program which will read a byte, set bit 7, and store the byte again.

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 inside. 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 – 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 assembly language – or in any other computer 'language', such as BASIC, C, Pascal and so on. Many flowcharts, alas, are constructed after the program has been written (usually by a great deal 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 assembly language 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 assembly language. 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 CPC6128 BASIC, you will already know how to plan a program by breaking it into smaller and smaller pieces. The main difference about assembly language 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 method 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 than symbols. Its great advantage compared with 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.

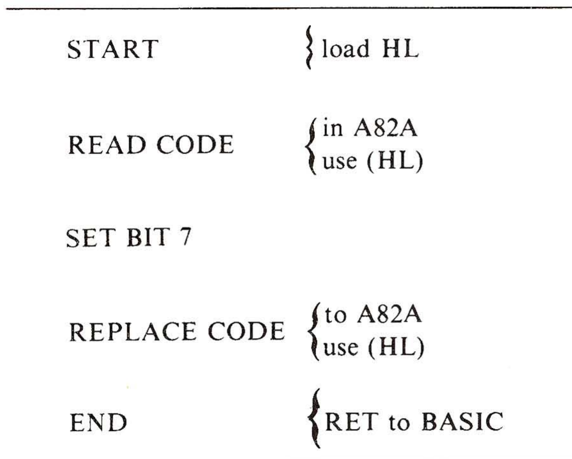


Figure 5.3 A simplified version of a Warnier-Orr diagram for the program which sets bit 7.

Figure 5.3 shows a very simple example of a form of Warnier-Orr diagram for the same problem we looked at in Figure 5.2. 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 a simplified *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 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 5.4. This puts the address #0120 into HL as usual, then loads the accumulator from this address, using MOV A,M. Setting the number 7

```
0100   LXI   H,0120
0103   MOV   A,M
0104   ORI   80
0106   MOV   M,A
0107   RST   06
```

Figure 5.4 Implementing the plan in assembly language.

bit is done by the ORI 80 command, and the byte is then put back into memory using MOV M,A. This time, the byte is put back into the same piece of memory, because no incrementing actions have been used on HL. If you put a number such as **4C** into address **0120**, then after running the program, the number will be **CC**, with bit 7 set. This type of action can be done in a different way with the Z80, using a SET command which is not available in the 8080 command set.

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 assembly language programs, as distinct from small sections, 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 assembly language. 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:

```
200 IF A=0 THEN GOTO 100
```

This contains a test (is A=0?), and if the test succeeds (yes, A is 0), then the program goes back to line 100 and repeats all the steps from there to line 200 again. That sort of loop in BASIC corresponds very closely to how we create a loop in assembly language. Instead of using line numbers, however, we use 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 5.5 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

START	
LOAD REGISTER	FFH into B
A: DECREMENT	use DCR B
IF NOT ZERO, GOTO A	use JNZ address
END	RST 6

Figure 5.5 A plan for a one-byte counting loop.

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 DCR B, and the test will use JNZ, and will return to the point A, the decrement step.

The action, then, will be that the B register is loaded immediately 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 SID. 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. Knowing what we know so far, we can write this in assembly language as:

```
      MVI B,FF
Loop: DCR B
      JNZ,Loop
      RST 6
```

and this would carry out the action we want. We can't type it in this form using SID, however. Later, when we work with ED and ASM, we can use labels in this way, but the very simple (and useful) assembler of SID doesn't allow such frills. For such small programs, it doesn't matter. We know, when we enter the program, that the address marked by Loop is 0102, and the JNZ address must therefore use 0102.

Now it's time to try out this looping business. Type the program into SID, using the form shown in Figure 5.6. Following the JNZ step, we put the address to which the program has to return, which we can see is 0102. This is relatively easy when you are entering short programs, because the display that SID lays on for you shows the addresses. For longer programs, it's not so easy to use, and we have other ways of writing really long programs. Patience for the moment. Try this program, single stepping it for a few times through the loop to show what is happening. Then try it at full speed, using **G0100**. Now when you do this, you might get the impression that the assembly language doesn't seem to have a very noticeable advantage of speed compared with a BASIC loop that counts down from 255 to zero.

```
0100   MVI   B,FF
0102   DCR   B
0103   JNZ   0102
0106   RST   06
```

Figure 5.6 The assembly language for a one-byte counting loop.

Both versions, in fact, deal with the count from 255 (hex #FF) to zero pretty quickly, but in fact most of the time 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 with BASIC we must use much longer counts, so that the time delays 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 assembly language.

A look at longer loops

The most obvious way of carrying out a longer countdown in assembly language is to load one of the double registers and count that down. Unfortunately, although we can decrement the double registers by using DCX (with H, B, or D), the DCX action on a double register does *not* affect the flags in the status register, so we can't use JNZ following DCX to loop back until the count is finished. This lapse on the part of the designers can be remedied with a little 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 MOV A,B , then the accumulator will be loaded from B. If we then ORA 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 ORA C *will* set flags. We can therefore follow the ORA C by a JNZ to perform the looping action.

Time to look at it, in Figure 5.7. The first action is to load the BC register pair with the number #FFFF, 65535 in denary. The label word 'Loop' in the 'written' version marks where the loop starts, and in this line the BC register pair is decremented. The result is then tested by using MOV A,B and ORA C, as described, and this is followed by JNZ LOOP to make the program loop back if the count has not reached zero. The program returns to SID after the end of the countdown. If you compare this for speed with a BASIC countdown from 65536 to 0, you'll notice a very marked difference between the machine code version and the BASIC version. The machine code version appears to end in about half a second – it's still almost too fast to time. A BASIC version takes about 73 seconds. In this example, then, assembly

```
0100  LXI  B,FFFF
0103  DCX  B
0104  MOV  A,B
0105  ORA  C
0106  JNZ  0103
0109  RST  06
```

Figure 5.7 A counting program which uses a register pair, BC. This takes considerably longer to execute.

language 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 dealing with a program which is purely machine code.

Trying to time the assembly language version with a stopwatch is very inaccurate because of the time 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 CPC6128 computer is given as 4.00 MHz. This means 4.00 *million* clock pulses pėr second, so that the time between pulses is approximately 0.25 *millionths* of a second. Now for each instruction, there is a time which is measured in terms of the number of clock cycles. These times are shown in detail in Appendix C, and Figure 5.8 shows how much time is needed for each instruction in the machine code loop. Twenty-four clock cycles, with each clock cycle taking 0.25 millionths of a second, gives us a time of 6.0 millionths of a second for each loop, so that 65536 loops take a time of 0.393 seconds, well 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, disc input and output and printer routines, just to name a few, make use of

Command	*Time*
DCX B	6
MOV A,B	4
ORI C	4
JNZ addr	10
	24

Since each clock cycle takes 0.25 microseconds, 24 clock cycles take $24 \times 0.25 = 6$ microseconds.
Time for 65536 loops = $65536 \times 6/1000000 = 0.393216$ seconds.

Figure 5.8 The times for each instruction can be summed to find how much time is needed for each loop. Times are expressed in clock cycles.

precise timing to achieve their actions, and loops such as the ones used here are the basis of that accuracy.

Time delays

One of the main uses for a countdown loop of the kind we have been testing is for time delays. Even if we use a countdown from a double register, however, the time delays we get are very short. For most practical purposes, we would probably need rather longer delays. The alternative is a holding loop of the 'press any key' type, but that's something we'll look at later. The next step at the moment is to develop a more useful routine for time delays of practical length. Since a countdown of the number FFFF in a double register gives a delay of under half a second, we need to use another countdown *of this countdown* to extend the time. This method will be more useful if we can make it a fairly precise amount, say 0.1 seconds for each time the double register countdown is used. To start with, we have to determine what number we need in the BC register pair for a 0.1 second countdown. Using the delay figures above, it works out at 16667 in denary, hex 411B. What we need to do, then, is have an inner loop in which the number 411B will be loaded in place and counted down, and an outer loop which will repeat this action several times. If we use a single register for this action, then the longest delay we can get is 255×0.1 seconds, which is 25.5 seconds, long enough for most purposes. If you need longer counts, then the outer loop can use a double register in place of a single one.

The scheme for a 'universal' time delay is shown in Figure 5.9. The routine starts by loading a delay byte into register E. Since the main time delay will be 0.1 second, the number that is loaded into E should be ten times the delay, in seconds, that you want. In this version, it has been represented by the word DELY. Following this load, the BC registers are loaded with the number 411B which will give a 0.1 second delay. This is exactly the same as the routine we looked at previously, so we needn't spend any more time with it. Following this 0.1 second delay, the E register is decremented (which will affect flags, since this is a single register) and tested with JNZ START. In

```
START
LOAD E WITH DELY
RUN DELAY ROUTINE <-+
DECREMENT E         |
REPEAT IF NOT ZERO -+
END
```

Figure 5.9 A plan for a 'universal' time delay.

```
0100  MVI  E,14
0102  LXI  B,411B
0105  DCX  B
0106  MOV  A,B
0107  ORA  C
0108  JNZ  0105
010B  DCR  E
010C  JNZ  0102
010F  RST  06
```

Figure 5.10 The assembly language for a two second delay, using the plan of Figure 5.9.

this way, the inner loop is repeated for as many times as is needed to count down the number DELY.

Figure 5.10 shows an example of the routine arranged for a two second delay. The number loaded into register E is 14H, 20 denary, and the whole routine takes only 16 bytes to store. Try single-stepping this one and you'll see the BC register pair being loaded and decremented. Once you have convinced yourself that this part is working correctly, see what happens when the BC register is empty. You don't have to single-step it 16667 times in order to see this happen! If, during the single-stepping, you type (when the prompt appears) **XB**, then the number in the BC register pair will be revealed. You can now alter this to 1 just by typing **0001** (or just 1, if you feel lazy). Using **T** will give the next single step, but with the new contents of BC. Now you can see what happens when BC is brought to zero, and the outer loop is resumed, decrementing E. This action of altering what is stored in the registers of the microprocessor is a very powerful way of checking the action of a counting loop, or any point in a program where there is a test of register contents like this. Any program that you develop should, if at all possible, be tested in this way, because it takes less time in the long run than having to reboot everything because of an error somewhere.

The routine is a useful sixteen-bytes worth if you need several time delays in the course of a program, but how would you use it? In its present form you would have to put the routine into memory wherever you needed it, but there is an alternative. The alternative, as you will know if you have programmed in BASIC, is the use of a subroutine. By converting this time delay routine into subroutine form, we can put it in a fixed place, then call it up and use it from any other part of a program. Before we can do this, though, we need to know how to organise subroutines in assembly language.

Subroutine delay

A subroutine in assembly language, is any piece of routine which can be written once and called from any part of a program. The subroutine is called

```
0100  MVI   E,14
0102  CALL  010B
0105  MVI   E,32
0107  CALL  010B
010A  RST   06
010B  LXI   B,411B
010E  DCX   B
010F  MOV   A,B
0110  ORA   C
0111  JNZ   010E
0114  DCR   E
0115  JNZ   010B
0118  RET
```

Figure 5.11 The timer arranged as a subroutine which can be called when needed.

by using the word CALL, followed by the starting address of the subroutine. To make the subroutine return correctly, it must end with the instruction word RET. The advantage of using subroutine form for a time delay is that it allows the same routine to be used with different delays. Figure 5.11 shows the subroutine form with a call from a 'main' program. The subroutine contains all the steps of the routine with the exception of the loading of register E, which is done in the calling program. Whenever E has been loaded, the delay subroutine is called by using CALL 010B (in this example) to run the time delay. The illustration uses two calls, one of two seconds and one of five seconds, to give a total of seven seconds delay. In reality, there would be other program items between the calls to the time delays, but I want to avoid long-winded examples. One point you may need to remember is that if register E is not loaded before this runs, and contains zero, the routine will run for 256 cycles, giving a time delay of 25.6 seconds. This is because decrementing zero in a register produces FF, and this will then be counted down. Don't panic, then, if you haven't loaded E, and find the cursor apparently stuck – it will all come right in about 25 seconds!

Another point that this raises is how we get quantities, like the number in register E, to be used in a subroutine. The example shows one way, which is to load E before the subroutine is called. This corresponds to the use of a 'global variable' in some programming languages. Sometimes, however, you will want the subroutine to be much more self-contained, with all of the registers loaded from inside the subroutine. This can be done if the byte needed by this subroutine, to take an example, is stored in memory. The subroutine can then load E from this memory address, perform the delay routine, and then return. You arrange for different delays by arranging for different numbers to be stored in this memory address. It looks more clumsy, and you wouldn't want to use it for short routines, but it can be an advantage for longer programs. Another method, one that we'll look at in more detail later, is the use of the stack for holding and releasing quantities

that need to be passed to subroutines. For the moment, though, we need to turn our attention to other aspects of subroutines and loops.

Chapter Six
CP/M Interactions

Other 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. Consider, for example, a BASIC program which fills the screen with the letter 'A', and which takes about six seconds. We could expect that a machine code version of this program would take rather less time. The problem is, how do we go about it?

The answer is important, because there is a way of going about all of these things which is peculiar to CP/M, and which you must follow very carefully if your programs are to be trouble-free. The whole principle of CP/M is that routines are provided for all inputs and outputs, and they demand the use of specified registers, plus a call to a special location in memory, #0005. If you use **D0000** to look at this piece of memory, you'll find that what is stored at address 0005 is the set of bytes **C3 00 DB**, which is the code for JMP DB00. The address DB00 is the start of a very important set of routines, called BDOS, the basic disc operating system. 'Basic' in this sense has nothing to do with the BASIC programming language. The important point about the BDOS address is that all kinds of inputs and outputs can be achieved by loading bytes into the C and E registers, then calling the address 0005, which performs the jump to BDOS. Now if you were writing only for your Amstrad machine, you could just as easily use CALL DB00, cutting out the JMP step. Unless, however, you are convinced that you'll never use another machine (a foolish belief, if you remember the CPC664 affair), it's better to use CALL 0005. The reason is that CALL 0005 will work on *any CP/M system*, which means on any machine that runs CP/M. This is the whole point of using CP/M, that the programs can be made so that they will run on any machine that uses the 8080, 8085, or Z80 microprocessors. The sooner we get to grips with the use of BDOS, then, the better.

With all of that to digest, let's plan a program which will fill the screen with the letter 'A'. Figure 6.1 shows the planning, with the outline, as usual, on the left-hand side. We want to clear the screen, write 2000 ASCII codes for 'A' to the screen, and then return to BASIC. The figure of 2000 comes from the rows and columns that we use. There are 25 rows, each of 80

```
START
CLEAR          { write 1B 45
                 *               { use HL for count
WRITE# 41      ( E = 41
2000 TIMES     { C = 2
               ( CALL 005        ( decrement HL
                                 { return to * if not zero
END            { RST 6
```

Figure 6.1 A plan for a program that will use machine code to fill the screen with the letter 'A'.

characters in the normal text screen and 25×80 is 2000. In the CPC6128 only 24 rows are normally used, but the 25th row can be obtained, as the manual shows. The next stage in planning shows more detail. All printing on the screen uses a call to BDOS, with the number **2** placed in the **C** register, and the byte which we want to put on the screen in the **E** register. The screen will be cleared by using the form-feed codes of **1B 45**. These codes are shown in the Amstrad CPC6128 manual, in Chapter 7:15, and in Chapter 4, page 7 of the manual. The full code list allows a lot of control over the workings of the CPC6128 from inside CP/M, and it's something that we shall come back to. Getting back to the problem in hand, the action of placing letter 'A' on the screen will be achieved by loading the **E** register with the number #41 (ASCII 'A'), setting a counter to 1999, printing to the screen with CALL 0005, and then decrementing and testing the counter. If the counter has not reached zero, the program returns to the point that is labelled *, the screen print CALL. The further detail is then shown in the next set of brackets, which shows that we need to use HL for counting, since C and E are in use. The alternative method, using the stack, will be discussed later. Now we can write the assembly language program.

Or should we? Never take anything for granted, especially when it involves using a built-in subroutine for the first time. You see, we have assumed so far that when we load up registers and call 0005, that everything will be unchanged after the CALL has been done. Perhaps we should just try things out, and see what happens. Figure 6.2 shows a simple load and call arrangement which we can single-step to find what happens. The single-stepping in this case has to be slightly differently arranged. If we set the PC to 0200 and use **T**, then the stepping will go through all the steps of the CALL 0005, and it will take a long time and be very puzzling. If we modify the command to **TW**, however, SID traces only the main steps, and executes the calls at normal speed. In this way, we see the result of a call without having to go tediously through all its steps. Figure 6.3 shows the result, and

```
0100  MVI  C,02
0102  MVI  E,41
0104  CALL 0005
0107  RST  06
```

Figure 6.2 Trying out the effect of the CALL to 0005 on the registers.

the line to look at very closely is the one following the CALL 0005 step. This shows that both the C and the E register contents have been changed by the CALL action. In fact, the number in the C register after the call is the same as the number that was in the E register before the call. Incidentally, the 'A' which appears following the CALL 0005 instruction in this listing is the

```
--M-- A=00 B=0041 D=174F H=0000 S=00FC P=0100 MVI  C,02
--M-- A=00 B=0002 D=174F H=0000 S=00FC P=0102 MVI  E,41
--M-- A=00 B=0002 D=1741 H=0000 S=00FC P=0104 CALL 0005A
--M-- A=00 B=0041 D=FE16 H=0000 S=00FC P=0107 RST  06
*0107
```

Figure 6.3 Tracing the action with SID, showing how the registers are affected.

result of the call being obeyed – the A is printed on the next available space.

What all this research (it sounds better than 'messing about') shows is that using CALL 0005 will have an effect on registers, and we can't be sure if any registers will escape. This makes it a dodgy business to try, for example, to set up a counting loop which contains CALL 0005, because if the register that keeps the count has its contents corrupted, then the count may never end! The answer to these problems is to store the contents of registers in the memory before using CALL 0005, and then restore the register contents afterwards. This is required so often that special commands, and a specially selected piece of memory, are used for the purpose. The piece of memory that is reserved for this purpose is the stack.

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. The starting address of the stack is fixed unless you alter it by a command, and the stack memory is from that address *downwards*. That doesn't mean that the stack will always be in the same place. While SID is running, for example, the stack memory starts immediately below the start address of CP/M, at #00FF. Other programs place the stack in other parts of the memory, but whatever is done, the stack must take up a set of addresses that will not be used for anything else. If SID used the stack to such an extent that it altered the RST 6 address at 0030, for example, SID would crash. As it happens, the way that SID is designed makes such intensive use of the stack practically impossible, but this doesn't mean that it will be impossible to overfill this stack if you also make use of it. The starting position of the stack can be moved by assembly language commands. For many purposes, it may be useful to set up a stack address early on in a program, but if you are simply adding code to an existing program, you will make use of the stack setting for that program, rather than set up a new stack position simply for your piece of program. The position of the stack, in any case, *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 languages like BASIC, you can be prevented from doing anything like this, because the program will stop with an error message if 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 #0100 upwards, with SID setting the stack to start at 00FF. 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 #00FF and a pair of bytes is stored, then the address changes to #00FD, 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 SID can normally be used with no problems in your machine code programs written using SID. When you start to write programs that will not be run with SID in attendance, you may have to set up a separate stack for use by that program.

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 are writing a program which is intended to run on its own, and it occupies addresses 0100 to 03FC, then you might want to set up a stack starting at 0500, leaving plenty of stack space. With this set up, you are not likely to corrupt the stack, and you can use it as you please.

The Z80 uses the stack in two ways, automatic and manual. The automatic use is by commands such as CALL #0005. Just before this is executed, the PC will have an address 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 to be 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 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 to mark the position of the top of the stack memory. Figure 6.4 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 PSW (meaning A and F), B (meaning B and C), D (meaning D and E), and H (meaning H and L), so that all 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.

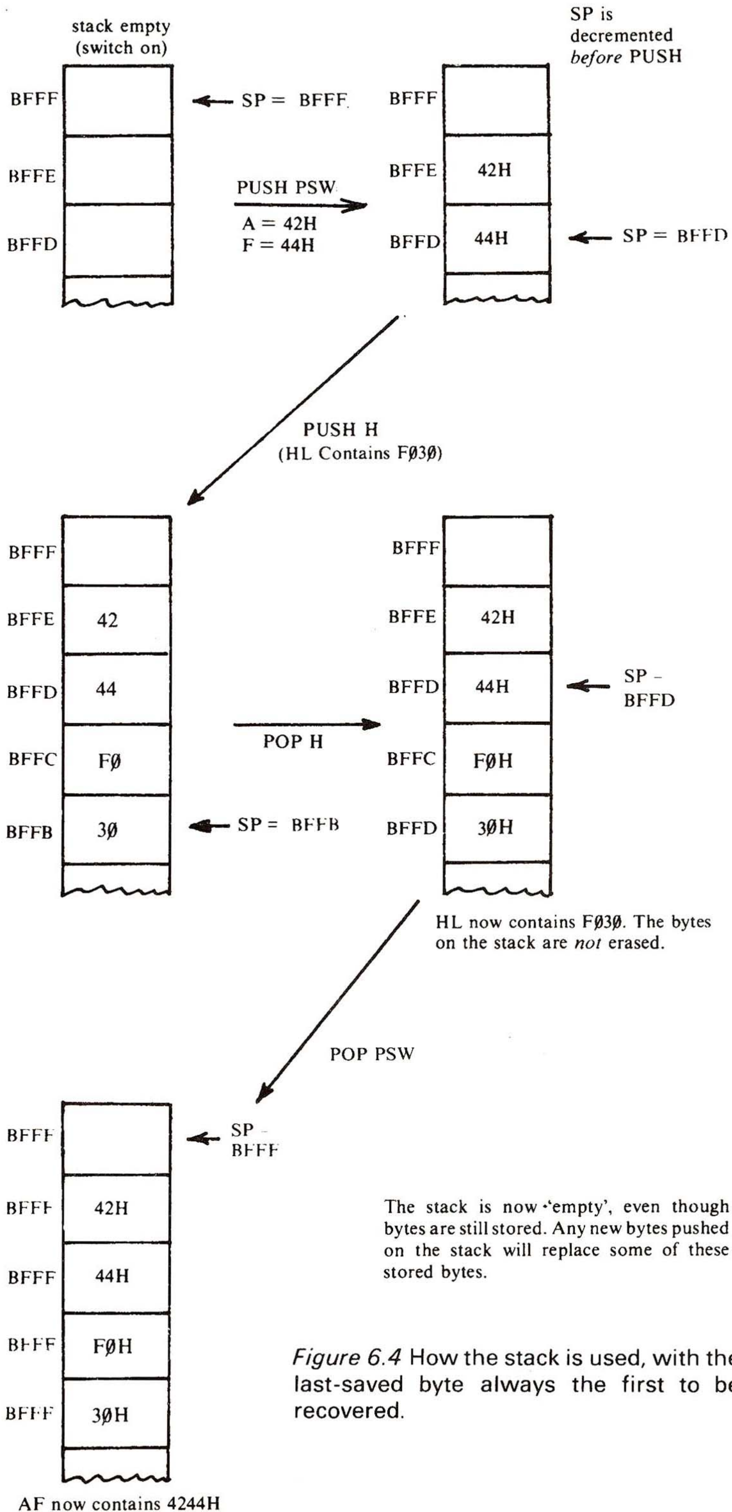


Figure 6.4 How the stack is used, with the last-saved byte always the first to be recovered.

(a)

```
        LXI     B,F100H
        MVI     B,FFH       this replaces byte F1H in B
LOOP:

        (instructions inside loop)

        JNZ     LOOP        end of loop
        LDAX    B           next instruction
```

The accumulator will be loaded from the incorrect address. It should be loaded from F100H, but the F1 byte has been replaced in the course of the steps of the counting action.

(b)

```
        LXI     B,F100H
        PUSH    B           save on stack
        MVI     B,FFH
LOOP:

        (instructions in loop)

        JNZ     LOOP        end of loop
        POP     B           get correct BC contents back
        LDAX    B           load accumulator
```

Because the contents of BC have been saved on the stack, the correct address exists in BC when the LDAX instruction is carried out.

Figure 6.5 (a)Incorrect programming, because the byte in B is reloaded. (b) How the stack can be used to save the correct values and restore them later.

One very common requirement is to save BC because of a count. Suppose we have a loop which has been programmed by loading the B register with #FF, and using the usual decrement and jump-if-not-zero count system. 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 6.5(a), where an address is loaded into BC just before a loop, and then used afterwards. Using DCR B 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 #0000 because B has been decremented to zero. The program sample of Figure 6.5(b) gets around this by pushing BC on to the stack just before the loop, and pulling it off the stack immediately afterwards.

(a)

```
        PUSH    B       BC contents on stack
        PUSH    D       DE contents on stack
        ...
        ...
        ...
        ...
        POP     D       from stack to DE
        POP     B       from stack to BC
```

Bytes are now back in correct registers.

(b)

```
        PUSH    B       BC on stack
        PUSH    D       DE on stack
        ...
        ...
        ...
        POP     B       BC loaded with byte from DE
        POP     D       DE loaded with byte from BC
```

Bytes exchanged between registers.

Figure 6.6 (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.

Another use for PUSH and POP is in preserving flags. Suppose you have just carried out a CMP action that set flags, but 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 PSW just after the CMP, and POP PSW just before the JP step, you can restore the values in A and F that were present immediately after the CMP, *no matter how much has happened between the PUSH and the POP*. In this context, incidentally, PSW means 'processor status word', the two bytes that contain the important accumulator and flag contents.

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 PSW, 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 you have a piece of program which looks like Figure 6.6(a). This pushes 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 D and then POP B, the bytes will be restored to the register they came from. If you

```
0100  MVI   C,02
0102  MVI   E,1B
0104  CALL  0005
0107  MVI   C,02
0109  MVI   E,45
010B  CALL  0005
010E  LXI   H,07CF
0111  PUSH  H
0112  MVI   C,02
0114  MVI   E,41
0116  CALL  0005
0119  POP   H
011A  DCX   H
011B  MOV   A,H
011C  ORA   L
011D  JNZ   0111
0120  RST   06
```

Figure 6.7 The assembly language program for the screen-fill plan.

used the sequence which is shown in Figure 6.6(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 neat and simple way of exchanging register contents.

One very common use of the stack is connected with CALL instructions, which is where we came into all this. When you use a CALL to 0005 or to your own subroutines in RAM, these subroutines may corrupt the registers 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 H just before the CALL and POP H 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 PSW and POP PSW, because a lot of subroutines will alter both the A and F registers. For our purposes, when we make use of any call to 0005, it's as well to assume that all of the registers are likely to be corrupted by it. Having taken that lot on board, we can now carry on at last with our proposals to fill the screen with letters 'A'.

The screen-write loop

What we shall have to do, then, is to ensure that registers are either saved on the stack each time 0005 is called, or reloaded before they are used. Often a combination of the two is simpler than the use of just one method, and this is illustrated in Figure 6.7, which is the outcome of the plan in Figure 6.1. To clear the screen requires two codes, **1B** and **45** to be sent in sequence. This means that the C and E registers have to be set up twice, and 0005 called

twice. If a program is likely to need a lot of screen clearing, then it's a good idea to make this into a subroutine that can be called at any point in the program. You can, incidentally, put all of your subroutines at the start of a program if you like, and have the first three bytes, at addresses 0100 to 0112, consist of a jump to the real start. Another option is to put all the subroutines at the end. If you scatter them about the main body of the program you will have to make sure that they can't be executed accidentally. You will normally do this by placing a jump just ahead of each of them.

Continuing with the 'A' writing program, the next part is the main loop. To prepare for this, the HL pair are loaded with **07CF**, the byte count. This is actually more than we need, because the bottom line of the screen is

```
--ME- A=CF B=0041 D=174F H=07CE S=00FA P=0100 MVI  C,02
--ME- A=CF B=0002 D=174F H=07CE S=00FA P=0102 MVI  E,1B
--ME- A=CF B=0002 D=171B H=07CE S=00FA P=0104 CALL 0005
--M-- A=00 B=001B D=FE16 H=0000 S=00FA P=0107 MVI  C,02
--M-- A=00 B=0002 D=FE16 H=0000 S=00FA P=0109 MVI  E,45
--M-- A=00 B=0002 D=FE45 H=0000 S=00FA P=010B CALL 0005E
--M-- A=00 B=0045 D=FE16 H=0000 S=00FA P=010E LXI  H,07CF
--M-- A=00 B=0045 D=FE16 H=07CF S=00FA P=0111 PUSH H
--M-- A=00 B=0045 D=FE16 H=07CF S=00F8 P=0112 MVI  C,02
--M-- A=00 B=0002 D=FE16 H=07CF S=00F8 P=0114 MVI  E,41
--M-- A=00 B=0002 D=FE41 H=07CF S=00F8 P=0116 CALL 0005A
--M-- A=00 B=0041 D=FE16 H=0000 S=00F8 P=0119 POP  H
--M-- A=00 B=0041 D=FE16 H=07CF S=00FA P=011A DCX  H
--M-- A=00 B=0041 D=FE16 H=07CE S=00FA P=011B MOV  A,H
--M-- A=07 B=0041 D=FE16 H=07CE S=00FA P=011C ORA  L
--ME- A=CF B=0041 D=FE16 H=07CE S=00FA P=011D JNZ  0111
--ME- A=CF B=0041 D=FE16 H=07CE S=00FA P=0111 PUSH H
--ME- A=CF B=0041 D=FE16 H=07CE S=00F8 P=0112 MVI  C,02
*0114
```

Figure 6.8 The result of single-stepping the program.

normally reserved for use with CP/M messages. We'll keep things as they are, however, to see what happens. In the loop, which starts at address 0111, the HL registers are pushed on to the stack. The C and E registers are then loaded for printing a letter 'A', and the BDOS call is made. Following the call, the HL register contents are popped from the stack back into HL, and the number decremented and tested so that the JNZ can form the loop. When the countdown is completed, the routine returns to the waiting arms of SID. Having entered the program, try single-stepping it with TW12. This produces the listing shown in Figure 6.8. You can see very clearly here the effect of the CALL 0005 on the registers, in particular how it clears the HL registers. Note, too, how the S (stack pointer) register changes at each PUSH and POP. The stack pointer shows the address of the next free place on the stack, and it will go down by two units each time a register pair is pushed, and rise by two units when a register pair is popped. You can see from this section of listing that we are in no danger of misusing SID's stack by forcing it down to 0030. This, incidentally, is another good reason for checking with single-stepping, because if you have one more PUSH than POP, you will find the stack steadily growing downwards until disaster strikes.

Having gone through all this work for the sake of the program, we had better run it at normal speed now, by using **G0100**. The result is rather unexpected. The line of A's starts at the bottom of the screen and scrolls upwards! The reason is that the sequence **1B 45** clears the screen, but it leaves the cursor at the bottom of the screen. To home the cursor to the top of the screen, we need to add the sequence **1B 48**. Everything in CP/M assembly language is hard work for the typing fingers! This screen-filling action 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 the purposes for which we write CP/M programs, this rate of screen filling would be fast enough. We can, in fact, speed it up by homing the cursor, because the scrolling action takes time to execute. We could simply add six lines to the program to add the home cursor bytes, but this looks like a good place to look at a more useful clear-and-home routine which can be used more generally.

This is illustrated in the program of Figure 6.9, which is another version of the As program. The program starts with a jump to the main program, because in this example the subroutine has been written at the start. The subroutine is long but straightforward, and does not take all that many bytes of machine code. It simply puts out the four codes that clear the screen and home the cursor. The main program then calls this subroutine, and does its act of filling the screen with As. There are, however, two new points involved here. One is that SID starts to come near the end of its usefulness when programs get long. If, for example, you have made a mistake early in this program, and need to insert another instruction, the only way open to you, using SID, is to type the new instruction into the correct place, and *retype*

```
0100  JMP  0126
0103  PUSH B
0104  PUSH D
0105  PUSH H
0106  MVI  C,02
0108  MVI  E,1B
010A  CALL 0005
010D  MVI  C,02
010F  MVI  E,45
0111  CALL 0005
0114  MVI  C,02
0116  MVI  E,1B
0118  CALL 0005
011B  MVI  C,02
011D  MVI  E,48
011F  CALL 0005
0122  POP  H
0123  POP  D
0124  POP  B
0125  RET
0126  CALL 0103
0129  LXI  H,07CF
012C  PUSH H
012D  MVI  C,02
012F  MVI  E,41
0131  CALL 0005
0134  POP  H
0135  DCX  H
0136  MOV  A,H
0137  ORA  L
0138  JNZ  012C
013B  RST  06
```

Figure 6.9 An improved program, which starts with a clear screen and the cursor in the 'home' position.

each following instruction also. That's hard work. Unfortunately, because of the history of CP/M, the alternative, using ED and the ASM family, is also hard work. There are better methods, but you have to pay for them – they don't come free with the CP/M package. You may, however, be able to get good software of this kind for the cost of the discs and a transfer fee from the CP/M User Group, who perform miracles of kindness to keep CP/M users sane and fed with good information and ideas – but more of that later. The second point about the program of Figure 6.9 is that writing the subroutine at the start is not just another Sinclair aberration; it can be useful. The reason is that if you want to write another main program, you only have to delete the existing main program, and you can keep the subroutine and the JMP which points to the main program. If you are developing a larger program with SID, and you may be adding subroutines,

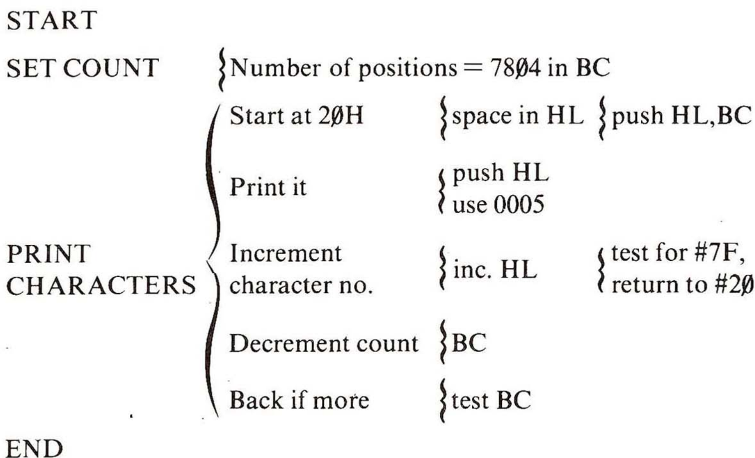


Figure 6.10 A plan for printing out the entire character set of the computer.

it's not a bad idea to leave more space deliberately, perhaps starting the main routine at 0500 or higher so as to leave lots of space for the subroutines. It may sound wasteful, but there's plenty of space in the memory of the CPC6128 and PCW 8256, and a few hundred bytes more or less make very little difference to the time that is needed to load or save a routine on disc.

Having said that, 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. Figure 6.10 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 characters that extend from 20H (32 denary) to 7FH, 127 denary. There are more characters available in the Amstrad machine, using codes 128 to 255, and some for codes below 20H, but we'll stick to the standard CP/M set for the present. The first character code number is 20H, the spacebar character. This is loaded into HL, and the BC register is loaded with the number of character positions on the screen, taking 24 lines of 80 characters this time. The printing can be done as usual by CALL #0005 when the C and E registers have been loaded, and on each loop we shall have to increment the character number and decrement the counter. We shall use a test for L holding the number 7F to determine when HL has to be reloaded with 20H. All of this planning leads to the program in Figure 6.11.

```
0100  JMP  0126
0103  PUSH B
0104  PUSH D
0105  PUSH H
0106  MVI  C,02
0108  MVI  E,1B
010A  CALL 0005
010D  MVI  C,02
010F  MVI  E,45
0111  CALL 0005
0114  MVI  C,02
0116  MVI  E,1B
0118  CALL 0005
011B  MVI  C,02
011D  MVI  E,48
011F  CALL 0005
0122  POP  H
0123  POP  D
0124  POP  B
0125  RET
0126  CALL 0103
0129  CALL 0147
012C  LXI  B,0780
012F  PUSH B
0130  PUSH H
0131  MOV  E,L
0132  MVI  C,02
0134  CALL 0005
0137  POP  H
0138  INX  H
0139  MOV  A,L
013A  CPI  7F
013C  CZ   0147
013F  POP  B
0140  DCX  B
0141  MOV  A,B
0142  ORA  C
0143  JNZ  012F
0146  RST  06
0147  LXI  H,0020
014A  RET
```

Figure 6.11 The assembly language for the plan of Figure 6.10.

In this routine, the clear-screen subroutine is placed at the start again, so that you don't have to type it all over again. The character printing routine starts at address 0129 with a call to 0147. This is the HL reload routine, and with HL loaded with 20H, the BC registers are loaded with the character count number 0780. The loop then starts by pushing BC and HL on to the

stack. The character whose ASCII code number is in L is then printed by moving it into E, and with C loaded with 2, calling 0005. This action will print the character and corrupt the contents of the registers. The HL registers are then popped (since HL was last pushed), then incremented, and the content of L is tested to find if it is equal to 7F. The test is done by moving the byte into A, and then using CPI 7F. If the byte is equal to 7F, the conditional call CZ is used to reload the registers. The action of the conditional call is rather like the action of the conditional jump, with the advantage that the call will return to the next address automatically. Whether a reload is needed or not, the next action is to pop BC so that the number of character positions can be checked. This is done in the usual way, and the program loops back if the required number has not been reached. If you want to see the characters in the range 80 to FF, then replace the steps in 0139 to 013E with NOP instructions.

Reading the keyboard

The routine at #0005, used with 02 loaded into the C register, 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 you would expect, there is provision in CP/M for doing just this, by loading 01 into the C register before calling 0005. The routine which is called up in this way carries out a complete INKEY$ routine. In other words, when you call this routine, 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. Note that this routine uses the accumulator, not the E register. 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, press a key.

Let's start the proper way with a bit of planning. Figure 6.12 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 #0005 with C=1 to do all of this. The 'print character' part is done by the routine at #0005 with C=2 as before. Figure 6.13 shows the assembly language version of all this. The subroutine at the start of the program 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 returns, and the byte is printed by the next CALL to 0005. When you try it, you'll see that whatever key you pressed has its letter displayed twice. This is because the call which gets the

START	
CLEAR	use subroutine
GET CHAR	C = Ø1 CALL 0005
PRINT IT	C = Ø2 move A to E CALL 0005
END	RST 6

Figure 6.12 Planning for a program which will read the keyboard and print to the screen.

```
0100  JMP  0126
0103  PUSH B
0104  PUSH D
0105  PUSH H
0106  MVI  C,02
0108  MVI  E,1B
010A  CALL 0005
010D  MVI  C,02
010F  MVI  E,45
0111  CALL 0005
0114  MVI  C,02
0116  MVI  E,1B
0118  CALL 0005
011B  MVI  C,02
011D  MVI  E,48
011F  CALL 0005
0122  POP  H
0123  POP  D
0124  POP  B
0125  RET
0126  CALL 0103
0129  MVI  C,01
012B  CALL 0005
012E  MVI  C,02
0130  MOV  E,A
0131  CALL 0005
0134  RST  06
```

Figure 6.13 The assembly language to implement the plan of Figure 6.12.

START		
CLEAR	use subroutine	
GET LINE	set up registers set up buffer CALL 0005	C = ØAH DE = address
END		

Figure 6.14 Planning to input a complete line and then print it.

```
0100  JMP   0126
0103  PUSH  B
0104  PUSH  D
0105  PUSH  H
0106  MVI   C,02
0108  MVI   E,1B
010A  CALL  0005
010D  MVI   C,02
010F  MVI   E,45
0111  CALL  0005
0114  MVI   C,02
0116  MVI   E,1B
0118  CALL  0005
011B  MVI   C,02
011D  MVI   E,48
011F  CALL  0005
0122  POP   H
0123  POP   D
0124  POP   B
0125  RET
0126  CALL  0103
0129  LXI   D,0140
012C  LXI   H,0140
012F  MVI   M,50
0131  INX   H
0132  MVI   M,00
0134  MVI   C,0A
0136  CALL  0005
0139  RST   06
```

Figure 6.15 The assembly language for the line-print program.

key code also echoes it to the screen, making it unnecessary for us to print it again. Don't assume that all CP/M machines will be identical in this respect. While you have this program running, it's interesting to see which keys will produce a result (find out which ones don't) and which keys produce

unexpected effects. Try the ESC, DEL, CLR and COPY keys, for example, and try the effect of pressing the SHIFT key along with other keys.

Meantime, how about developing this fragment of program action into something more useful? Suppose we wanted to print on to the screen everything that was entered until the RETURN key was pressed? This means using a loop, to test the key character that has been obtained to find if it is a RETURN code (0DH, 13 denary). Figure 6.14 shows the plan for this action, which is a lot simpler than it looks. You might expect that you would have to carry out a loop action here, testing for a carriage return character. In fact, CP/M contains this routine, and all you have to do is call it. The routine is called by loading C with **0A**, and calling good old 0005 once again. You can then type whatever you like, using the DEL key to amend mistakes, until you press the RETURN key, which terminates the action. All of the text appears on the screen as you type it. The actual assembly language, however, in Figure 6.15, shows a few steps that are not catered for in the plan. The reason is that a line reading routine would be rather useless if it did nothing more than print a line of text on the screen. The routine which is called up by using C=0A and CALL 0005 will, in fact, do quite a lot more. It will store the text in a buffer section of memory, and keep a count of the number of characters up to the RETURN key. To do this correctly means that some setting up has to be carried out. For one thing, a section of memory has to be set aside, and its starting address put into the DE register pair. In addition, the buffer must contain as its first byte the number of characters that will be permitted. This is usually 80 for CP/M use, allowing a line of reply to a question. The second byte in the buffer will eventually contain the number of characters actually used, and we can set this to zero as a start. The assembly language program as listed does all this setting up by using HL to put in the characters, and loading DE with the starting address. More elegant methods are available, but for the moment it's results we want, not elegance. When you run this routine, the screen clears, and you can type characters, ending with a RETURN. Now if you look at the memory, you'll see that the characters are stored starting at address 0142, the third byte of the buffer. The first byte of the buffer is the usual 80 character limit, and the second shows how many characters were actually typed. Since one byte is used for each number, this indicates that you should not try to accept line lengths of more than 255 characters. There aren't many people who write lines of that length, and they all work for *The Times*.

What happens if you exceed the limit of length? Try it by making line 012F read MVI 0A, limiting you to ten characters. When you try to type an eleventh character now, you'll hear the loudspeaker beep, and the computer will refuse to accept the character. The beep is not something that all computers will do, it's an Amstrad special, but the refusal to accept more characters is standard. This routine is the CP/M equivalent of BASIC's INPUT. The count number for the characters entered is very useful in a lot of applications, because when we put text into memory, we often want to

terminate it with a zero so that it can be read by a looping routine. The count number added to the start address for the memory used will give the position for the zero which will indicate termination.

Chapter Seven
More Routines

Take a message...

In Chapter 6, you remember, we looked at the method by which we could write characters on the screen. It's time now that we looked at ways of putting something more interesting onto 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. This is, in fact, the reverse of the process which is carried out when you load **#0A** into register C and call 0005. That routine placed the starting address of the block of text into the memory whose starting address was held in the DE register pair. It also required that the block should be set up with a maximum line length number, and a zero byte which would later be replaced with the correct character count. We can use this in two ways, as we'll see later.

We start, as always, with a program plan. It's not so easy this time, because we need to decide how to end 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 and simpler technique first of all - 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 any kind of data in ASCII codes is 0, because the 0 byte (*not* the ASCII number 0, which is coded as ASCII #30) never occurs in ASCII text. For text, the 'carriage return' code of &0D is also quite a useful terminator, but for the moment we'll stick with the zero byte. We can do this by testing for the presence of a zero *before* the byte is used in the printing routine. Before, you note, not after, because we must not attempt to print codes like this either on the screen or on the printer. If, on the other hand, we were using the **0D** character as a terminator, we would want to make the test *after* printing, because the carriage return is a valid character whose effect we need in the string of characters, placing it into the writing routine. We would also need the 'line feed' character, 0AH, to ensure that the cursor will move down by

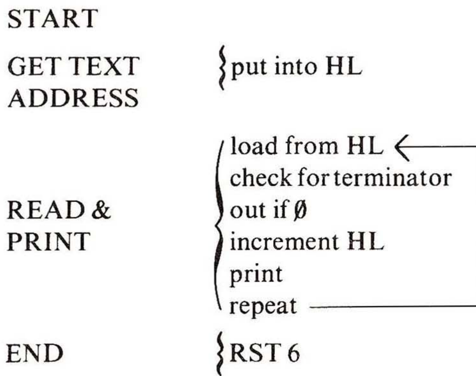


Figure 7.1 A plan for printing a message on to the screen, using a terminator byte of zero.

one line after printing the phrase.

The plan that we need for using a zero terminator is shown in Figure 7.1. What we have to do is store an address for the start of 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 TEXT as the label name for this piece of memory. If we were using a more elaborate assembler, as we shall soon, then this name would be placed into the assembly language as it is. The main sequence is straightforward, and the only detail that is really needed is in the READ & PRINT section. The byte to be printed is loaded from the address stored in HL, and tested for the zero terminator. If the byte is a zero, the loop ends; otherwise, the byte is printed, HL is incremented, and the loop continues.

Figure 7.2 shows what we need. The HL register pair is loaded with the

```
0100  LXI   H,0150
0103  PUSH  H
0104  MOV   A,M
0105  ORA   A
0106  JZ    0114
0109  MVI   C,02
010B  MOV   E,A
010C  CALL  0005
010F  POP   H
0110  INX   H
0111  JMP   0103
0114  RST   06
0150: 54 68 69 73 20 69 73 20 69 74 0D 0A
```

Figure 7.2 The assembly language program for printing the message. The bytes of the message will have to be put into place using SID's **S** command.

address 0150 at which the text will be stored. In the loop, the accumulator is loaded from M, the memory addressed by HL, which means that it copies the first ASCII code in the text. This byte is tested, and if it is zero, the jump to the RST 6 address is taken. If the code is acceptable, the printing subroutine is called to print the character in the usual way. The HL address is then incremented, and the program loops back to get another character. Note that the HL register has to be saved on the stack as usual to prevent corruption by the CALL 5 routine. When the 0 character is read, the program then ends by returning to SID. Once the program has been put in place, you need to put in the bytes of the message separately. A more elaborate assembler would allow you to put in the message using letters preceded by DB (define byte), in the form:

0150 DB'This is a message, 0D,0A,00

– but the simple assembler of SID does not allow for such frills. We will need to put in the bytes of the message by using the **S0150** command, looking up the ASCII codes for the letters, and adding the 0D,0A at the end to ensure a carriage return and line feed.

Well, it works, and we can use it for greater things. Suppose, for example, that you wanted to use this routine with a screen-clear and home-cursor operation. You do *not* need to put in the subroutine that we looked at in the previous chapter, because all it does is print the codes 1B 45 1B 48. If we simply add these codes to our message, at the start, the screen will be cleared and the cursor homed. Rather than alter all the bytes starting from 0150, you can put in these codes at 014C to 014F, and alter the address for the LXI H command to 014C. When you run this one now, you'll find that the screen is cleared and the cursor homed before the piece of text is printed.

Now we can look at the other problem, of using a piece of text which has been prepared in the way that the line input routine of CP/M leaves it. This time, the text is pointed to by the DE register, and it starts with the maximum length code and has as its next byte the actual length. Using text in this form is rather more difficult than using a terminator of zero, but it's worth looking at simply because we have such a convenient way of creating it from the keyboard. As we proceed, you'll probably conclude that most of CP/M is organised to make life easy for the writers of word processors. This makes it all the more difficult to understand why ED is still included in the package! The outline plan is shown in Figure 7.3. The second layer of detail is the part to look at, because the main scheme is the same – it still concerns printing a message. This time, the address of the start of text space will be transferred from DE to HL, assuming that we have just read it in from the keyboard, using the line read routine. The character count will be placed into register C, since only one byte is used. In the loop, the byte is read from the HL address, printed, and the character count decremented and checked. The program will loop until the decremented character count reaches zero. Looking at the third layer of detail, we shall have to make sure that the registers are kept on the stack to avoid corruption.

```
START

TEXT SPACE    { from DE to NL
              { put length in C

              text in HL          { keep on stack
              length in C         { keep on stack
              * check count
PRINT TEXT    print character
              decrement count     { get C from stack    { replace on stack
              increment address   { get HL from stack   {
              loop to *

END
```

Figure 7.3 A plan for using the text that the line input routine places in memory.

```
0100  LXI   H,0161
0103  MVI   M,00
0105  DCX   H
0106  MVI   M,50
0108  PUSH  H
0109  XCHG
010A  MVI   C,0A
010C  CALL  0005
010F  POP   H
0110  INX   H
0111  MOV   C,M
0112  INX   H
0113  PUSH  H
0114  PUSH  B
0115  MOV   E,M
0116  MVI   C,02
0118  CALL  0005
011B  POP   B
011C  DCR   C
011D  POP   H
011E  JZ    0124
0121  JMP   0112
0124  RST   06
```

Figure 7.4 The assembly language for reading a phrase from the keyboard and then printing it.

All this leads to the routine in Figure 7.4. This places the phrase in memory using the keyboard (for the sake of demonstrating the principles), and then prints it using the new routine. At the same time, I have taken the opportunity to introduce some new ideas into the older section of the program, so we'll look at all of it. The program starts by loading 0161 into HL for a text line that will start at 0160. The zero byte is loaded here, and then HL is decremented so that the byte #50 (denary 80) can be loaded. This leaves HL containing the correct starting address. If we had started with 0160, we would have to have incremented to get to 0161, then decremented to get back again. Having put the first two bytes into the line of text as required, we then save HL on the stack, and get the address into DE by using XCHG. This is a command which swaps the contents of the HL and DE register pairs, so that the starting address of the text buffer is now in DE, and whatever was in DE is now in HL. Since we have the address on the stack, this doesn't matter. We can now use the line input routine, which will place the text that we enter into the DE address. The routine will corrupt the DE contents, but this is unimportant since we have the correct starting address stored on the stack. We pop this back into HL, and then increment so that we have the character count byte ready, which we put into register C. The HL register pair are then incremented again, so that they point to the first character of the text, and both HL and BC are pushed so that we can print this character. This is done in the usual way, and then registers BC are popped. Register C is decremented and tested to find if it has been reduced to zero. The HL pair are then popped, and the program jumps back to the INX H step, ready for another character. You have to be careful about the order of commands here. It might seem more logical to have the sequence:

```
DCR C
JZ 0124
POP H
JMP 0112
```

but this will not be good for your stack! The reason is that when C is decremented to zero, the program stops with the HL contents still on the stack. For a short example, with SID in attendance, this doesn't matter too much, but it could cause disasters in a longer routine. A lot of care is needed to ensure that there is a POP for each PUSH, especially when the program branches between the PUSH and its POP.

When you try this out, it doesn't look as if it works! You enter a phrase, press RETURN, and all you see is the prompt for SID appearing under the letters. In fact, the program *has* worked. The point is that the input routine restores the screen cursor to the start of the line again. This means that whatever you typed is printed again over the top of the original phrase, so you can't see it. To convince yourself, type another phrase, but this time start with ESC E ESC H (pressing the escape key, then E and so on) and then the letters of your phrase. This will put the screen-clear, home-cursor bytes into

the text memory, so that when you press RETURN you will see the screen clear, and the message will appear at the top of the screen. Now do you believe it?

Still more text

While we are on the subject of putting text into the memory and reading, we can develop these routines a little further. One of the main guiding principles of assembly language is that you always do a little at a time, and check it. Back in the Stone Age days, it used to be reckoned that a programmer would write about ten lines of good working code each day. Nowadays, a lot of people write faster than this, and sometimes it shows! For your own programs, it's a good idea to be very critical and to test exhaustively before you decide to expand a piece of program. By making your programs out of subroutines, with no subroutine longer than about 50 or 60 lines, you can be more certain of good results than if you blindly start writing great chunks of code. The point about using the assembler part of SID is that it allows you to write and test routines which you can later copy to work with ED and ASM. As you'll discover shortly, writing a program with ED and ASM takes time, and you will prefer the program to be one that you are sure will work first time, rather than one that needs a lot of correction.

Now to work. Figure 7.5 shows the outline of what we hope to achieve eventually. I stress eventually, because I want to build this up slowly, as an illustration of what can be done using only the comparatively limited facilities of SID. We want to be able to type in text lines, terminated by the RETURN key, so we shall be using the RETURN key like the carriage return of a typewriter. Each time the RETURN key is pressed, we need to issue a carriage return and line feed to the screen, so that we do not type one line on top of another. We also want to store the lines one after another in the memory, wait for a key to be pressed, and then print them on to a clear screen. Rather than develop this as one long program from the start, we

---

```
START
    DO -
            INPUT LINE UNTIL C/R
            PUT IN C/R L/F CODES
    UNTIL CTRL-Z
PRESS ANY KEY LOOP
PRINT ALL TEXT
```

Figure 7.5 An outline plan for a text storage and printing program.

```
0100  LXI   H,0201
0103  CALL  01F0
0106  PUSH  H
0107  XCHG
0108  MVI   C,0A
010A  CALL  0005
010D  POP   H
010E  INX   H
010F  MOV   E,M
0110  INX   H
0111  MOV   A,M
0112  CPI   1A
0114  JZ    0121
0117  MVI   D,00
0119  DAD   D
011A  INX   H
011B  CALL  01DF
011E  JMP   0103
0121  RST   06


01DF  PUSH  H
01E0  MVI   C,02
01E2  MVI   E,0D
01E4  CALL  0005
01E7  MVI   C,02
01E9  MVI   E,0A
01EB  CALL  0005
01EE  POP   H
01EF  RET
01F0  MVI   M,00
01F2  DCX   H
01F3  MVI   M,50
01F5  RET
```

Figure 7.6 The assembly language for getting the text on screen and into memory. The action is terminated by using CTRL-Z.

shall construct it in two parts, so that we can be certain that everything is working as it should. We can then add the printing part, and use SID to test that it will operate correctly.

The first part of the plan produces the program section of Figure 7.6, which gets the text and puts it on to the screen and into memory. This is made out of bits that we have used already, and the only novelty is the test in 0112, CPI 1A. This tests for the CTRL-Z keys, and is the way that the

```
0121  CALL 01B2
0124  PUSH H
0125  MVI  C,0B
0127  CALL 0005
012A  ORA  A
012B  JZ   0125
012E  POP  H
012F  RST  06

01B2  PUSH H
01B3  LXI  H,01CB
01B6  PUSH H
01B7  MOV  A,M
01B8  ORA  A
01B9  JZ   01C7
01BC  MOV  E,A
01BD  MVI  C,02
01BF  CALL 0005
01C2  POP  H
01C3  INX  H
01C4  JMP  01B6
01C7  POP  H
01C8  POP  H
01C9  RET

01CB: 1B 45 1B 48 50 52 45 53
      53 20 41 4E 59 20 4B 45
01DB: 59 0D 0A 00 E5
```

Figure 7.7 The 'press any key' steps.

program jumps out of its loop. The CALL 01F0 sets up the numbers at the start of each line of text, and the CALL 01DF is the subroutine for carriage return and line feed to prevent overwriting text on the screen. If you try out this lot, typing several phrases terminated by RETURN, and ending by typing CTRL-Z (RETURN), you can see the result by using **D0200**. The text for each phrase is stored, with no space between them, in the memory. Each phrase carries the bytes 50 (maximum line length) and the character count at the head, as you would expect. Now it may for some purposes be useful to have these counter numbers embedded in the text, but for most purposes it would be more useful to replace them with the bytes 0D and 0A, since we want a carriage return and line feed at each point where these items are placed. The **1A** byte is also placed in the memory, and this can be used as the end-of-text marker. Knowing that the first half is working, then, we can get to work on the second part.

We want to start with a 'press any key' step. This can be done by using the inevitable CALL 5 again, with 0B loaded into register C. This call is like INKEY$ in BASIC, however. It tests the keyboard, and moves on without waiting. If a key happens to be pressed while the instruction is being executed, the accumulator and the L registers contain 01, otherwise 00. To use the call, then, we need to put it into a loop, testing for A = 01 to indicate a key pressed. This is neater than using an input step, which requires the RETURN key to be pressed. We also want to print a message, because the user (you, probably) will need to be reminded of what to do. These, then, are the next items to attend to. These stages are illustrated in Figure 7.7, showing only the new parts. Each time the program is expanded like this, the RST 6 is overwritten, and put in again at the end of the new section. In this way, we always return to SID at the end of the routine. We can, however, break off in the middle of a routine, by using a 'breakpoint address' in the **G** command of SID. For example, **G0100,011F** would command SID to execute the program which starts at 0100, but to stop at 011F. The stop is done by putting a RST 6 code in the memory at the desired address, and replacing the correct byte after the program has returned to SID. By testing each piece as we add it in this way, we have made the use of a breakpoint unnecessary, but it's useful to bear in mind for later.

In the new chunk of routine, then, the call to 01B2 brings in the routine which prints a message. This is the standard routine that we used before, with a zero used to terminate the message. The bytes of the message are stored from 01CB onwards, and they start with the clear-screen and home-cursor codes. The carriage return (C/R) and line feed (L/F) codes are placed at the end of the message, and finally the zero terminator. Incidentally, you need to use **D** to show these codes, because if you use **L** they will appear as 8080 instructions rather than as hex numbers.

In the print routine, some care has to be taken with the stack. At the time when the routine is called, the HL registers are being used to hold a current address in memory, which we don't want to lose. As it happens, we don't really need to use it in this program, but who knows what else we might want it for. If we start the routine with PUSH H, then the address is safe. We can then load in the address of the start of the message, but we must then push HL again to prevent this address from being lost when CALL 5 runs. The loop that follows is a familiar one, but note that when the zero is found and the program jumps to 01C7, there are two POPs. This is because the test for a zero is made when HL has been pushed twice, so it must be popped twice at the end. Normally in the loop, the HL pair get popped and pushed an equal number of times, but things are different on the last loop. This is another example of how you have to watch the use of the stack when any jump is made.

Once the message has been printed, we need the 'press any key' action. This is placed in addresses 0124 to 012E. Once again the PUSH H has to be used to prevent corruption of HL, and by loading 0B into register C and calling 0005, we test the keyboard. The ORA A is needed because a load by

itself does not affect flags. If the accumulator contains zero, no key has been pressed, and the routine loops around. When a key is pressed, the RST 6 returns everything to SID so that you can check what has happened.

Now the next thing to add is the routine for printing the contents of the memory on the screen. We need to put in a C/R and L/F in place of the numbers, and to halt and return when we find the 1A byte which signals the end of the text. There two ways that we could go about this. One is to go through the text initially, stepping from one set of numbers to the next and replacing the numbers with C/R L/F bytes. The other possible method is to print directly, replacing the numbers as we find them. The first method is fast, because the length-of-text number allows us to find the next pair of numbers quickly and easily, whereas if we go byte by byte we have to keep a count, and check the count each time. Compared with the time needed to print a character on the screen both methods are acceptable as far as speed is concerned. The first method scores, however, because its routine does not use a CALL 5, and so no pushing of registers is needed. If you use a count in the course of printing, the register that holds the count will need to be pushed before each print action, and popped afterwards. The first method, then, is the one that we shall use. You might like to think about the possibility of a routine which dispensed with a character count, and simply printed the LF and CR in place of the character count numbers until it encountered the end-of-text character, 1A.

The assembly language listing is shown in Figure 7.8. We load in the text starting address of 0200 to HL, and we know that the first two bytes will be #50, the maximum length of text, and then the actual length byte. By using MVI D,00 we ensure that the D register is cleared. This is important, because the method of stepping from one line to another will be by adding the DE contents to the HL contents, and any byte in D will affect this. The

```
012F  LXI  H,0200
0132  MVI  D,00
0134  MVI  M,0D
0136  INX  H
0137  MOV  E,M
0138  MVI  M,0A
013A  INX  H
013B  MOV  A,M
013C  CPI  1A
013E  JZ   0145
0141  DAD  D
0142  JMP  0134
0145  RST  06
```

Figure 7.8 The program steps for replacing the length bytes by the line feed and carriage return bytes.

sixteen-bit addition has to be used because though the numbers that we add to HL are always single bytes, the result may not be. The next step is MVI M,0D which replaces the byte #50 by the carriage return byte. This is the start of the loop which will continue until all of the number bytes have been replaced. The HL register pair is incremented, and the length byte is loaded into register E, ready to be added to HL. The 0A byte is then put into the place of this length byte. We then have to increment HL again before adding the DE contents, and after this increment, the HL address will be the address of the first character following the new line. If we have reached the end of the text, this will be the 1A character, so we test for it here. If the character is not 1A, we add DE to HL to get the address of the next pair of number bytes, and then loop back to the replacement routine. Once again, you can add this to your program, and test it. If your program is identical to mine (which it will be unless you are using your own methods), then the start of this part of the routine is at 012F. You can therefore put in some text before you enter this routine, and then single-step this part after you have entered it. This is done by setting the PC address to 012F, and then using **T** for each step, or **T5** to step in fives, whichever you find more convenient.

Now all that is left is to print the memory contents. Once again, we could make use of the ending number in HL to count out the number of print steps, but this would mean saving the HL register each time we call the print routine. It's easier, as usual, simply to print each character, starting at 0200, until we reach the 1A terminator. This only requires a test when each character is read in. We shall still have to save HL at each step, but no other registers will be needed, and no number comparisons. Remember throughout this that you will need the routines that sit just under 0200, so when you record the bits of this program, always record from 0100 to 01FF to make sure that these routines are incorporated. The latest addition is shown in Figure 7.9. It's completely straightforward, and it just reads the

```
0145  LXI   H,0200
0148  PUSH  H
0149  MOV   A,M
014A  CPI   1A
014C  JZ    015A
014F  MVI   C,02
0151  MOV   E,A
0152  CALL  0005
0155  POP   H
0156  INX   H
0157  JMP   0148
015A  POP   H
015B  RST   06
```

Figure 7.9 The program portion for printing the text.

```
0100  LXI   H,0201
0103  CALL  019C
0106  PUSH  H
0107  XCHG
0108  MVI   C,0A
010A  CALL  0005
010D  POP   H
010E  INX   H
010F  MOV   E,M
0110  INX   H
0111  MOV   A,M
0112  CPI   1A
0114  JZ    0121
0117  MVI   D,00
0119  DAD   D
011A  INX   H
011B  CALL  013B
011E  JMP   0103
0121  CALL  015E
0124  PUSH  H
0125  MVI   C,0B
0127  CALL  0005
012A  ORA   A
012B  JZ    0125
012E  POP   H
012F  LXI   H,0200
0132  MVI   D,00
0134  MVI   M,0D
0136  INX   H
0137  MOV   E,M
0138  MVI   M,0A
013A  INX   H
013B  MOV   A,M
013C  CPI   1A
013E  JZ    0145
0141  DAD   D
0142  JMP   0134
0145  LXI   H,0200
0148  PUSH  H
0149  MOV   A,M
014A  CPI   1A
014C  JZ    015A
014F  MVI   C,02
0151  MOV   E,A
0152  CALL  0005
0155  POP   H
0156  INX   H
0157  JMP   0148
```

Figure 7.10 The complete listing for the program in Figure 7.9 with sections relocated.

```
015A  POP   H
015B  RST   06
015C  NOP
015D  NOP
015E  PUSH  H
015F  LXI   H,0177
0162  PUSH  H
0163  MOV   A,M
0164  ORA   A
0165  JZ    0173
0168  MOV   E,A
0169  MVI   C,02
016B  CALL  0005
016E  POP   H
016F  INX   H
0170  JMP   0162
0173  POP   H
0174  POP   H
0175  RET
0176: 00 1B 45 1B 48 50 52 45 53
      53 20 41 4E 59 20 4B
0186: 45 59 0D 0A 00

018B  PUSH  H
018C  MVI   C,02
018E  MVI   E,0D
0190  CALL  0005
0193  MVI   C,02
0195  MVI   E,0A
0197  CALL  0005
019A  POP   H
019B  RET
019C  MVI   M,00
019E  DCX   H
019F  MVI   M,50
01A1  RET
01A2
```

Figure 7.10 (contd)

bytes, looking for the 1A character as its terminator.

Now we can start to use SID to good purpose. The point of putting the subroutines just under 0200 was to keep them out of the way. Now that we know where the end of the code is, we could move these routines down to fit at the end. At the same time, however, we'll leave two bytes at the end in case we want to replace RST 6 by JMP 0000 to make this routine one that will run independently of SID. The moving is done using the **M** command of SID. This has to be followed by three numbers in hex. The first two are the start address and the end address of the block that you want to move, and the last number is the start address of where you want to move code to. To

shift down the subroutines and the codes for the 'press any key' message, use **M01B2,01F6,015E**. Then you will have to change the addresses in several jumps and LXI steps. You will have to change the jumps at 0103, 011B and 0121, and the LXI at 015F to suit the new addresses. In addition, all the jump steps in the subroutines will need to be changed. If it all proves to be too much, Figure 7.10 shows the complete listing, including the data bytes. If you now want this routine to stand on its own, without using SID, then the final RST 6 can be replaced with JMP 0000, so that control returns to CP/M after the program has run. Since there is very little use of the stack, the program should return safely and run normally.

The typewriter program

It's time now to illustrate another call to the CP/M system and to combine this with something which is quite useful, a 'typewriter function'. A normal typewriter types one letter at a time, allows no correction, and stores no data. With a very simple program in CP/M, we can use a printer attached to the computer as if it were a typewriter, but with a few important advantages. One is that it's easy to deal with a line at a time, making corrections before printing. The other is that the program can be very simple indeed, so much so that it can save time compared with a word processor. I use a word processor for anything more than a page or so in length, but for short letters or notes, a typewriter action is much handier because there is only the minimum of setting up to do. In this book, of course, there is an ulterior motive – the program introduces the printer call to CP/M and it also gives you a basis to develop a program for yourself which will be more to your own taste.

Figure 7.11 shows the listing. This time, the program is one that returns to CP/M directly rather than to SID, so that you can record the program directly under the name that you intend to use. I used the filename TIPPEX.COM, with apologies to the fluid which it renders redundant. In this way, typing TIPPEX is enough to start the program running, and I have the use of my printer as a typewriter controlled by the keyboard of the CPC6128 or PCW 8256. The start should be familiar from what we have done before, selecting a buffer area for entry of a line of text. What is less familiar is that the HL registers are pushed while at address 0201. The line input routine is called in the same way as previously, and at address 010F the HL registers are popped again to get back to the 0201 address which existed originally in these registers. What we want to do now, to avoid difficulties later, is to put a zero byte at the end of the line of text that we type into the buffer. The printer routine can use this to halt printing, rather than depend on a character count, which is the usual alternative. The character count byte is put into register E, and register D is zerod so that with HL incremented, adding DE to HL gives the address which is one place beyond

```
0100  LXI   H,0201
0103  MVI   M 00
0105  PUSH  H
0106  DCX   H
0107  MVI   M,50
0109  XCHG
010A  MVI   C,0A
010C  CALL  0005
010F  POP   H
0110  MOV   E,M
0111  INX   H
0112  MVI   D,00
0114  DAD   D
0115  MVI   M,00
0117  LXI   H,0202
011A  MOV   A,M
011B  CPI   1A
011D  JZ    0132
0120  MOV   A,M
0121  CPI   00
0123  JZ    0135
0126  PUSH  H
0127  MOV   E,A
0128  MVI   C,05
012A  CALL  0005
012D  POP   H
012E  INX   H
012F  JMP   0120
0132  JMP   0000
0135  MVI   E,0D
0137  MVI   C,02
0139  CALL  0005
013C  MVI   E,0A
013E  MVI   C,02
0140  CALL  0005
0143  MVI   E,0D
0145  MVI   C,05
0147  CALL  0005
014A  MVI   E,0A
014C  MVI   C,05
014E  CALL  0005
0151  JMP   0100
0154  CALL  0005
0157  MVI   E,0A
0159  MVI   C,02
015B  CALL  0005
015E  MVI   E,0D
0160  MVI   C,05
0162  CALL  0005
0165  MVI   E,0A
```

Figure 7.11 The listing for the typewriter program.

```
0167  MVI   C,05
0169  CALL  0005
016C  JMP   0100
```

Figure 7.11 (Contd)

the last character in the text. A zero is then loaded into this memory address by the command in address 0115. This concludes the writing part of one line. We now have to deal with printing this, assuming that it is suitable – and by this stage it will have to be! The HL registers are reloaded with 0202, the address of the first character in the buffer. What we want do now is to go through a loop which will look at each character and test it. There are two tests, one for code 1A, which is the CTRL-Z character that marks the end of all typing; the other is for code 0, which marks the end of a line. After a line end, we want to put out the carriage return and line feed codes both to the screen and to the printer.

The first test, at 011B, is for the CTRL-Z character. If this is found, the program jumps to the exit, which in this case is a JMP 0000 command that returns to CP/M. The second test of the character code is for 00, and if this is found, the program jumps to a routine at 0135 which will put out these codes. If neither the 1A nor the 00 code has been found, then the character must be one that is to be printed. This, incidentally, includes ESC codes, so that you can do things like pressing ESC E for emphasised mode on an Epson, or whatever you want to use. It's not so easy if you want to send codes like ASCII 0FH for condensed print. Theoretically, this is CTRL-O, but this code is filtered out by CP/M, like most other CTRL codes. In any case, the character is printed by putting its ASCII code into register E, putting 05 into register C, and calling address 0005 as usual. When a character has been printed, the routine jumps back to 0120 to get another character. When the zero character is reached, this is not sent to the printer, but the CR/LF routine runs, and ends with a jump to 0100 to start it all off again for another line of text.

It is, of course, very short and simple, but it works, and can be useful. I usually set my printer to a five-space margin, so that the maximum number of characters per line is five less, only 4B in hex rather than 50. As an example of what can be done using only SID, it is as far as we go in this respect, because in the following chapter we are about to look at the complications of using ED and ASM for real CP/M assembly. SID remains unrivalled as a way of finding out what your program is doing, and for writing these comparatively short routines. The point is that if you are dealing with a short routine, and you need to insert a new command, SID's assembler forces you to rewrite everything from the point where the new command is inserted. As you'll see, a real editor/assembler package does not force you to do this kind of thing, and for that reason, along with many others, it's more satisfactory to use for really ambitious work. If what you have done so far has acted to whet your appetite, then read on!

Chapter Eight
SID,ED and Family

Debugging delights

Now that you have experienced some of the delights of machine code programming, using the assembler in SID, 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. It all sounds rather insecticidal (the old name for SID was DDT), 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 all the addresses that you have put in place are correct. This particularly applies to anything that follows a jump instruction. When you use the simple assembler of SID, you often have to type JMP FFFF (or any other 'noticeable' address) and change this address to the correct one when the program is more completely assembled. You might, for example, want to jump to the end of the program, but until you have finished entering it you don't know the address that will need to be used. All of these addresses must be correctly filled in if the program is to run correctly. 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 with experience that you can get to the stage where the bugs will be few in number and easy to find.

If you use any kind of 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. By using the assembler of SID throughout this book, we have therefore eliminated the problems that less fortunate machine code programmers have when they start first. In this chapter, however, we are going to look at

the ASM.COM assembler of CP/M, and very briefly at its more modern counterpart, MAC.COM. 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. The weak point among the programs that come with CP/M is the ED editor, and you may feel at a later date that you would like to make use of the Microsoft M80 assembler in place of the built-in CP/M utilities. If, however, you intend to be just a dabbler, spawning the odd drop of machine code now and again, then the methods based on SID 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. When you are developing a program, you often find that you want to insert another instruction somewhere, and the assembler of SID does not allow for this. If you put in a new instruction, then you have to retype all the following instructions. This is tedious, and all tedious jobs result in mistakes.

A lot of problems, as I have already said, can be eliminated by meticulous checking, and it pays to be extra careful about which jump command you use. It's remarkably easy to type in JZ when you mean JNZ, or JC when you mean JNC, simply because you have not thought enough about which flags will have been affected. 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. The stack is always a potent source of problems, and if your program is likely to contain a lot of pushing and popping, it's wise to start with a stack location (loading the stack pointer) at a part of the memory which leaves a reasonable amount of space. In the examples so far, we have either used the stack location of SID, or we have made use of the CP/M system stack. You can't always rely on the system stack having enough space if you intend to make a great deal of use of the stack. 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've got what you deserve. It may be that you are trying to make use of a CP/M 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. Many will change the contents of almost every register you are using, so that a number of PUSH and POP actions will need to be included in your program -- and that's where you can run into stack trouble! 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 disc, each of which you have thoroughly tested, using SID, before you assembled them into a long program. In real life, this is not always so easy, particularly when the subroutines need to be retyped, because typing errors get in the way. Another point is that the addresses which are contained in such subroutines may not suit the new range of addresses which will be used. As usual, users of a more elaborate assembler have the best of it, because they can keep assembly language instructions, called 'source code', stored like BASIC programs, and merge and edit them as they choose. The next best thing to keeping a subroutine library on disc 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 splendid 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 source, better for 8080 routines, is the magazine of the CP/M User Group, which must be the best financial bargain that exists for the serious CP/M user. 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 what has to be in each register before the routine is called, and what will be in the registers afterwards. 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 SID. We have already looked at the single-step action of SID, and how the contents of any register can be altered. These features make SID a very powerful weapon in the fight against bugs, and there's a supplement in the form of breakpoints. A breakpoint, as far as the CP/M operating system is concerned, is the byte #F7 (denary 247) replacing another instruction byte. This is the RST 6 byte, and its effect is to return to SID. When you are back in SID, you can examine the contents of memory by using the **D** instruction. The principle is to pick a point in the program at which something is put into memory. If you place a breakpoint byte following this, then when the program runs, it will return to SID immediately at the breakpoint. This is done as part of the **G** instruction. For example, if you type **G0100,011F**, then the program will run from 0100 and there will be an RST 6 instruction put in at 011F which will return you to SID at that place. You can then examine the contents of registers using **X**, or of memory using **D**, so checking what the instructions up to that point have done to registers and memory. If this isn't what you expected, then you should know where to look for the fault. The system automatically removes the RST 6 breakpoint after it has been used, and replaces the correct code into memory, so that using the breakpoint like this does not corrupt your program. If all is well at this breakpoint, then you can put a breakpoint further along the program, and try again. What you must be careful about when you do this is that a breakpoint must replace an

instruction byte, not a data byte. This means that you need to keep a note of what addresses are being used. This is easy enough if you print a listing on paper at intervals, but it can be awkward if you have to work directly from the screen.

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 using the usual CTRL SHIFT ESC sequence, or even switching off and reloading. 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:

```
     MVI B,FF
LOOP:DCR B
     JNZ LOOP
     LD A,#A650
```

we could encounter problems. Suppose that this was assembled with SID, and we made the branch back to the MVI B,FF instruction rather than to the DCR B 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 ordinary 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 addresses to look at, as when you use SID's simple assembler. As always, taking care over loops is the only answer, and the method that has been mentioned in this book of putting in FFFF as the jump address at first and then replacing it later is a good precaution.

Using ED, ASM and HEXCOMDE

Using full assembler action for CP/M programs involves the use of three programs, named ED, ASM and HEXCOM. Much of this is due to historical reasons, and the job could be done with just two programs, but the reasons were sound ones, and the split-up of tasks among three separate programs can be useful at times. Of the three, ED is the one that you have to take time with, and learn. ED is the editor part of the package, the part which allows you to type and enter assembly language statements. Note that I said statements. Unlike the assembler of SID, ED deals only with text in ASCII codes, it is a (primitive) form of word processor. Like any program of its kind, ED is full of 'bells-and-whistle' features that you may never use, and in this book, I'll concentrate on the main features. I have dealt with other uses of ED more fully in my book *Advanced Amstrad CPC6128 Computing*, and in this book I shall deal only with the features that are essential to the use of ED in typing assembly language text.

To start with, we need to look at what the three programs do. ED is used to create text, meaning assembly language and comments. The assembler

program, ASM (or the later MAC) is then used to convert or assemble the assembly language into code, but the code is in ASCII form which uses seven bit bytes only, suitable for being passed down a line in serial form. The HEXCOM program then transforms this seven-bit ASCII code version into the normal form of a COM file which can be run in the usual way by typing its name. For short program routines, of course, you don't need all of these features, and you may prefer to stick with SID. The use of the full assembler system, however, has many advantages, not least of which is that you always have a file consisting of a fully commented assembly language, something that you would have only in handwritten form if you use SID. It would certainly be simpler and would make assembly faster if the output of the assembler were a COM file, but for historical reasons it isn't, and unless you turn to the use of another assembler, you can't do much about it. Since this book is devoted to the use of the program utilities that come on the System discs, we'll stick with these programs.

To illustrate the process, then, we'll take a fairly short routine which can be assembled in this way. This is a routine to insert ESC code letters into the system, so that features of the display can be changed, and it offers a useful way of checking what some of these ESC codes do. The escape codes are listed in your 6128 manual, on pages 7/49 to 7/52. The principle is that pressing the ESC key followed by another key will produce some action, such as clearing the screen, inverse video, changes in background or foreground colour, etc. Some of these actions can be carried out even without a program, but a short routine to print the ESC character (1B) followed by a character from the keyboard will allow any of these codes to be entered, so that you can carry out a set of alterations from the keyboard. The program consists of a section which prints the ESC character, then a loop which gets a character from the keyboard and prints it unless the character is the RETURN character of 0D.

The first step is to invoke ED by typing this name, followed by RETURN. When ED has loaded, you will get the message:

Enter Input File:

and you should answer this with a valid filename which has the extension **.ASM**. In this example, I used ESCHAR.ASM. This name is entered by pressing the RETURN key, and you then get the prompt:

Enter Output File:

which in this case you ignore by pressing RETURN. An alternative to this method is to start from scratch with:

ED ESCHAR.ASM (RETURN)

– but the result in each case is to bring up the message **NEW FILE** and the usual ED prompt of **:***, awaiting instructions. Since you want to input text, you type **I** (RETURN), and this in turn brings up the display **1:** followed by a

```
READ    EQU     01H
CPM     EQU     05H
ESC     EQU     1BH
WRITE   EQU     02H

BOOT    EQU     0000H
        ORG     0100H
        MVI     E,ESC           ;PUT IN ESC CHAR.
        MVI     C,WRITE         ;TO SYSTEM
        CALL    CPM             ;THROUGH 0005
GETIT:  MVI     C,READ          ;READ KEYBOARD
        CALL    CPM             ;FOR CHARACTER
        CPI     0DH             ;CHECK FOR RETURN
        JZ      EXIT            ;OUT IF SO
        MOV     E,A             ;OTHERWISE WRITE
        MVI     C,WRITE         ;CHARACTER
        CALL    CPM             ;TO SYSTEM
        JMP     GETIT           ;BACK FOR NEXT
EXIT:   JMP     BOOT            ;OUT
        END
```

Figure 8.1 The ESCHAR.ASM file, produced using ED.

block cursor. You are now in the text entry mode of ED, ready to type the program listing. This time, however, you need several changes in the way that a program is entered, and these will be obvious on the listing. The most obvious point is that ED, unlike SID, assumes that numbers are in *denary* unless you follow them with H to mean hex. This is awkward, because you will quite often be flitting from one program to the other, but it's just one of the awkward aspects you have to put up with.

Figure 8.1 shows the result, printed from a finished listing, but this does not show what had to be done to achieve the text as shown. To start with, each part of a listing must be in a field of its own. 'Field' in this sense means a set of columns that are selected by using the TAB key on your CPC6128 or PCW 8256. For example, the first line, which appears as:

```
READ        EQU        01H
```

is obtained by typing READ, then pressing TAB, typing EQU, pressing TAB then typing 01H. The first field is devoted to label names which will stand in for numbers or addresses. The second field is for an instruction operating name, and the third field is for the operand, the numbers or register names which go with the instruction. You can use a fourth field for comments, and we'll look at this later. Meantime, the next four lines are entered in the same way as the first. If you make a mistake, remember to use the CTRL-H keys rather than the DELETE key. The DELETE key will

delete the text, but it places characters on the screen, whereas the CTRL-H keys show the deletion actually carried out on the screen. This is just one of the features of ED that makes you feel that there must be better ways of entering text! When you come to the line that states:

```
ORG      0100H
```

you need to precede the ORG with a TAB to place ORG under the previous EQU. This is because ORG is a form of instruction word, not a label name, and it belongs in the second field. If you mix up the positions of these words, you will confuse the assembler, which is not one that can make use of 'free-form' text entry, unlike the Z80 assembler, ZEN, which is *not* a CP/M utility.

So far, what we have entered is no part of the program, simply preparation. The words EQU and ORG are called 'pseudo-instructions'. They look like instruction words, and they are placed in the field that is allocated to 8080 instruction words, but they are not 8080 mnemonics and are not understood by SID. These words are instructions to the assembler program which will make sure that things are done correctly. The EQU word, for example, specifies what a label word means, so that wherever the word READ occurs in the text, the assembler will substitute the number 01H, and when the assembler finds CPM it will put in 05H. This scheme allows us to use these label words in place of numbers. This is an advantage when you are reading a listing, because it makes the purpose of an instruction much easier. If the instruction uses the word WRITE, for example, this makes the purpose of a command much easier than simply using the number 02H. Similarly, using ESC for 1BH is a good way of showing that you are using the code for the ESC key, and when you read a listing of this type you should not have to be constantly checking what numbers mean. The ORG pseudo-instruction, by contrast, shows where the program must start – at the usual address of 0100H. Once again, remember to use the H ending, or the numbers will be entered in denary. A number such as 1B, for example, will be entered as 01, since B is not part of a denary number. This is something you must check carefully when you see the listing.

Only when these preliminary pseudo-instructions have been typed can we now start the program itself. This is typed in fields as before, keeping the first field clear unless a label name is needed. This needs some determined practice, because it's very easy to relapse into typing in the ordinary way, forgetting the first TAB. If you do this, you may have to kill some lines by using the **K** command of ED. The last line consists of the word **END** in the second field, with nothing else in this line. When the whole program has been entered, press CTRL-Z to go back to the control mode of ED. To look at the result of your work, you now need to press **B** to make ED go back to the start, then **#T** to type all the lines of your text. Appendix D summarises the commands of ED in case you do not have a copy of my book *Advanced*

Amstrad CPC6128 Computing. Remember that ED places text into memory rather as we did in the examples of the previous chapter, keeping a 'pointer' address for the current position in memory. Unless you alter this pointer address and specify how many characters or lines you want to work with, ED will show nothing useful on the screen. The fields, incidentally, are not wide, only eight characters in all, and you need to make your label names short, six characters at the most. Note how the label names in the first field need to be followed by a colon unless they are used with EQU. The comment line can be separated by more than one TAB if you like, and you can have a line which starts with a semicolon in the first field and which consists entirely of a comment. This is one of the few exceptions to the rule that everything goes into its own field at all times. If you need to edit a line at this stage, the best method is to shift the pointer to the line by using **B** (RETURN) followed by a number which is *one less than the line number*. For example, if you want to look at line 6, type **B** (RETURN), then **5** (RETURN). This displays the line, and the asterisk indicates that you are still in command mode. You can then kill that line by using **K** (RETURN), and type in a new line by using **I** (RETURN), your new text, RETURN, then CTRL-Z. It's all very clumsy compared with modern text editors, but remember it came free!

When you have the text in the correct form, you can record it under the filename you originally provided by using **H** (RETURN). This spins the disc, and leaves you in the command mode of ED. The **H** command leaves the text in the memory, but you can't see it again unless you use the **#A** command, followed by **B** and **#T** if you want to type out the entire file again. It's things like this that can cause you to wonder if the description of an action in the HELP pages is really accurate! Once you have the file recorded, it is safe, and you can leave ED and look at the disc directory. You'll find that the ESCHAR.ASM file is in place, and there is also a ESCHAR.$$$ file. This latter file is a temporary file which was used during the creation of the ASM file, and which is not used subsquently. You cannot find what this file contains by typing TYPE ESCHAR.$$$, for example, because the file will be empty by this time. If you have made more than one version of the file, there will be an ESCHAR.BAK file also, and this one will be a normal text file. Once the ASM file has been recorded, you can make use of the next step, assembly of the code into a hex file, using the assembler program ASM or MAC.

Using ASM

To use ASM on your ESCHAR.ASM file, simply type ASM ESCHAR (RETURN), and the process will be automatic provided you have ASM and ESCHAR.ASM on the same disc side. You will get the message:

CP/M ASSEMBLER - VER 2.0

followed by a number. This number is the address of the end of the program in the memory, the address of the first unused byte. You will also get a report on 'USE FACTOR', usually 000H, which is not important for small programs. The really important point is that the message END OF ASSEMBLY means that you have returned to CP/M, and you can now use DIR to find what is on the disc. This reveals two new files, ESCHAR.PRN and ESCHAR.HEX. ESCHAR.PRN is an expanded version of the text file. It shows the addresses that are used and the hexadecimal codes that are placed into these addresses, and you can look at this file by using TYPE ESCHAR.PRN. Figure 8.2 shows the file that was created for the sample program.

The next step is to look at the **.HEX** file which is also on the disc. This can be examined by using the **DUMP.COM** utility, which you can put on to the disc with PIP. By typing **DUMP ESCHAR.HEX**, you will see the coded program on your screen. The codes in this program are not the same as in the version that appears when you use TYPE ESCHAR.PRN, because the hex codes for the program have been put into ASCII code for the .HEX version. In other words, the hex byte 1E, the first byte of the program, has been coded as 31 45, the ASCII codes for '1' and 'E' respectively. This makes it easy to transmit this version along lines and to record it with error-checking, because ASCII codes use only seven of the possible eight bits in a byte, leaving the most significant bit free to be used as a 'parity' bit for error-checking. Figure 8.3 shows the .HEX file that is obtained from the program. To obtain a **.COM** file that will run under CP/M, you now need to use the HEXCOM utility. This also is straightforward – you need only type **HEXCOM ESCHAR** and the task will be carried out unless for some reason there is no ESCHAR.HEX file on the disc. The HEXCOM program will work only if it can find a .HEX file of the correct name to operate on. When HEXCOM operates, it will print a message about the file, showing the first address and last address of the codes, the number of bytes read, and how many records have been used on the disc. Once the ESCHAR.COM file has been created, you can, at last, summon up this program by typing **ESCHAR**. The cursor will disappear, and you can then type the letter which will give the effect that you want, such as **H** for home cursor.

The ESCHAR program is not exactly a useful one, but it does serve to illustrate how ED, ASM and HEXCOM are used in a relatively painless way. As always, if you work with large programs when you are learning how to use a utility, you run the risk of getting bogged down with the complications of the example, let alone the utilities. Even this short example, however, will serve to show you how ED, ASM and HEXCOM are used, and that's the important point at the moment. The file that is created by ED can contain comments, and can be edited. This means that if you want to insert more lines of instructions into a program, then the changes are made to the text version, which can then be recorded again. This is the reason for allowing both an input filename and an output filename for

```
0001 =          READ    EQU     01H
0005 =          CPM     EQU     05H
001B =          ESC     EQU     1BH
0002 =          WRITE   EQU     02H

0000 =          BOOT    EQU     0000H
0100                    ORG     0100H
0100 1E1B               MVI     E,ESC           ;PUT IN ESC CHAR.
0102 0E02               MVI     C,WRITE         ;TO SYSTEM
0104 CD0500             CALL    CPM             ;THROUGH 0005
0107 0E01       GETIT:  MVI     C,READ          ;READ KEYBOARD
0109 CD0500             CALL    CPM             ;FOR CHARACTER
010C FE0D               CPI     0DH             ;CHECK FOR RETURN
010E CA1A01             JZ      EXIT            ;OUT IF SO
0111 5F                 MOV     E,A             ;OTHERWISE WRITE
0112 0E02               MVI     C,WRITE         ;CHARACTER
0114 CD0500             CALL    CPM             ;TO SYSTEM
0117 C30701             JMP     GETIT           ;BACK FOR NEXT
011A C30000     EXIT:   JMP     BOOT            ;OUT
```

Figure 8.2 The ESCHAR.PRN file which has been produced by using **ASM** on the ESCHAR.ASM file.

```
CP/M 3 DUMP - Version 3.0
0000: 3A 31 30 30 31 30 30 30 30 31 45 31 42 30 45 30  :100100001E1B0E0
0010: 32 43 44 30 35 30 30 30 45 30 31 43 44 30 35 30  2CD05000E01CD050
0020: 30 46 45 30 44 43 41 31 41 30 34 0D 0A 3A 30 44  0FE0DCA1A04..:0D
0030: 30 31 31 30 30 30 30 31 35 46 30 45 30 32 43 44  011000015F0E02CD
0040: 30 35 30 30 43 33 30 37 30 31 43 33 30 30 30 30  0500C30701C30000
0050: 31 32 0D 0A 3A 30 30 30 30 30 30 30 30 30 30 0D  12..:0000000000.
0060: 0A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A  ................
0070: 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A  ................
```

Figure 8.3 The ESCHAR.HEX file. This contains the hex characters expressed in ASCII format.

ED. Once an amended file has been created, this can be subjected to ASM and HEXCOM as before. In most cases, you will want to put both ASM and HEXcom into a SUBMIT file, since you will almost invariably use them both in sequence.

Another example will help you to feel your way into the enormous expansion of computing power that the use of these utilities presents you with. This time, we'll remedy some of the deficiencies of the earlier ESCHAR program. The problem with ESCHAR is that it works only on single character entries, so that sequences such as ESC 3 0 (switch to Mode 0 screen) do not work – though these work when you press the appropriate keys in normal service. The problem with ESCHAR is that it gets a character from the keyboard between the actions of printing the ESC and printing the following character, and the gap is not desirable – it prevents the correct operation of the ESC action. In this new version, then, we shall put the characters into a buffer, terminated with the return character, and then

```
 1:  READ   EQU   Ø1H
 2:  CPM    EQU   Ø5H
 3:  ESC    EQU   1BH
 4:  WRITE  EQU   Ø2H
 5:  BOOT   EQU   ØØØØH
 6:         ORG   Ø1ØØH
 7:         LXI   H,BUFR    ;BUF START IN HL
 8:         PUSH  H         ;AND SAVE IT
 9:         MVI   M,ESC     ;PUT IN ESC
10:         INX   H         ;INCREMENT POINTER
11:  INP:   PUSH  H         ;SAVE POINTER
12:         MVI   C,READ    ;READ KBD.
13:         CALL  CPM       ;GET CHAR.
14:         POP   H         ;GET POINTER
15:         MOV   M,A       ;PUT CHAR. IN
16:         INX   H         ;INCREMENT POINTER
17:         CPI   ØDH       ;IS IT RETURN?
18:         JNZ   INP       ;BACK IF NOT
19:  EXEC:  POP   H         ;GET START OF BUF.
20:         MOV   A,M       ;GET IN A
21:         CPI   ØDH       ;IS IT RET?
22:         INX   H         ;INCREMENT POINTER
23:         JZ    EXIT      ;OUT IF SO
24:         PUSH  H         ;SAVE POINTER
25:         MOV   E,A       ;GET INTO E
26:         MVI   C,WRITE   ;TO WRITE
27:         CALL  CPM       ;TO SYSTEM
28:         JMP   EXEC      ;AND REPEAT
29:  EXIT:  JMP   BOOT      ;OUT
30:  BUFR:  DS    5         ;BUFFER SPACE
31:         END
```

Figure 8.4 The modified program, in which characters are placed first into a buffer.

write them one by one to the system. In this way, we can enter character sequences, and have them correctly obeyed. Since these characters sequences are needed for such actions as changing modes, positioning the cursor, changing foreground and background colours, etc., this action can be useful, and the ESC key does not have to be used.

The start of the program, in its ED form, is shown in Figure 8.4. The first six lines are the same as the first six of ESCHAR, so that the listing is started by using ED ESCHAR.ASM to get the text of ESCHAR into the machine. By typing 7, and then **#K**, the rest of the lines of the program are deleted, and the **I** command will then allow you to insert the remaining lines of the new program. This has been fully annotated to make it easy to follow, so that only the outline needs to be stated. The principle is to use the reading action in a loop, placing characters into a buffer until the return character of 0DH is read. The first character in the buffer is the ESC character of 1BH, and the stack contains both the starting address of the buffer and the updated pointer address which will be used for the current character. When the return character has been entered, the program continues into a new loop. The start of buffer address is obtained from the stack, and characters are printed out from the buffer until the return character is found. The important point to note in this listing is how the buffer space is allocated. The label name BUFR locates the starting address, and by using the pseudo-instruction DS5, we allocate five bytes of space for the buffer following the last instruction of the program. As before, this text from ED is saved, assembled with ASM, and then converted into correct form using HEXCOM. You can, incidentally, use the more modern assemblers MAC or RMAC, but unless you are working with particularly large and complicated programs, these assemblers are not really much more useful. MAC and RMAC each produce an additional disc file of the symbols (such as WRITE, BUFR, etc.) that have been used in your program.

Chapter Nine
Disc Use and Utility Programs

A disc utility program is one that is intended to make your use of discs easier, particularly if you want to do rather more than just load and save. In particular, disc utilities allow you to see what is stored on the discs, including information which is not normally available to you and which cannot be obtained by normal load operations. The most important disc utility is one which allows you to look at any sector of any track. This allows you to see what is stored on the CP/M reserved tracks, for example, and to find where programs are stored. It can also be used to read a disc which, because of partial demagnetisation, for example, will no longer load correctly. If a disc editor program is also available, the correct bytes can *sometimes* be replaced so as to make the disc usable again. I must stress, though, that this is a desperate measure. You would normally have a backup of any disc, and if a disc became unusable, you would normally re-format it, back up the other copy on to it, and then continue. Human nature being what it is, however, a disc editor is still useful at times, and an editing program has been included in this chapter. Another function is that of deliberately making a disc difficult to copy, and for such protection you need to be able to carry out disc editing, such as replacing upper-case titles of CP/M programs by lower-case letters.

This chapter, then, is concerned with various disc utilities, and how CP/M carries out the management of discs. Most of these utilities make use of a mixture of BASIC and machine code to achieve disc reading or writing, but the programs themselves are in BASIC, with the machine code poked into memory. You need no knowledge of machine code either to enter the programs or to use them. This means that you will have to leave CP/M and return to AMSDOS to enter and use these programs. This has been done deliberately to avoid the long listing that such a program would need if it were written exclusively in CP/M machine code. I have added a listing and a short explanation of the machine code part of each program, but these are in Z80 machine code rather than in 8080, so it's the broad outline of the action that is of interest. The routines call on other routines in the ROM of the CPC6128, and these are much easier to access from BASIC than from CP/M.

```
10 CLS:GOSUB 250:GOSUB 350
20 PRINT#1,TAB(13)"Track & Sector.":PRIN
T#1,TAB(13);STRING$(14,"_")
30 PRINT#0,"SYSTEM OR DATA FORMAT - "
40 PRINT#0,"Please answer S or D":PRINT#
0,"The default is D."
50 INPUT DS$
60 PRINT#0:PRINT#0,"TRACK No.- (0 to 39)
":INPUT T%
70 IF T%<0 OR T%>39 THEN K%=39:GOSUB 400
:GOTO 60
80 PRINT#0:PRINT#0,"SECTOR No. (0 to 8)"
:INPUT S%
90 IF S%<0 OR S%>8 THEN K%=8:GOSUB 400:G
OTO 80
100 IF DS$="S" THEN S%=S%+1+&40 ELSE S%=
S%+1+&C0
110 D%=T%*256+S%
120 CALL &A000,D%
130 CLS#0:CLS#1
140 PRINT#1:PRINT#1,TAB(8)"Byte No.";TAB
(20)"Hex";TAB(27)"Char"
150 PRINT#2:PRINT#2,TAB(6)"Press SPACEBA
R for next byte"
160 FOR N%=0 TO 511:K%=PEEK(B%+N%)
170 PRINT#0,TAB(7);N%;TAB(16)HEX$(K%,2);
180 IF K%<32 THEN K%=127
190 PRINT#0,TAB(24);CHR$(K%)
200 WHILE INKEY(47)=-1:WEND
210 NEXT
220 PRINT#0:PRINT#0,"Another one - Y or
N?"
230 INPUT A$:IF UPPER$(A$)="Y" THEN 20
240 END
250 MEMORY &9FFF:B%=&A000
260 INK 0,0:INK 2,26:INK 3,1
270 CLS:WINDOW#1,1,40,1,3
280 WINDOW#2,1,40,23,25
290 WINDOW#0,5,35,4,22
300 BORDER 4
310 PAPER#1,3:PAPER #2,3
320 CLS#1:CLS#2
330 PEN#0,1:PEN#1,2:PEN#2,2
340 RETURN
350 SUM%=0
360 FOR N%=0 TO 27:READ D$
370 POKE B%+N%,VAL("&"+D$):SUM%=SUM%+VAL
("&"+D$):NEXT
```

Figure 9.1 The disc Track & Sector reader program. This program is in BASIC, so you will have to leave CP/M in order to write it and use it. *Note*: this program cannot be used on the PCW 8256.

```
380 IF SUM%<>3407 THEN PRINT"Error in da
ta - cannot continue":END
390 B%=&A01C:RETURN
400 PRINT#0:PRINT#0,"Range 0 to";K%" onl
y, try again":RETURN
410 DATA DD,7E,00,DD,56,01,1E
420 DATA 00,21,1C,A0,F5,0E,07
430 DATA CD,0F,B9,F1,C5,4F,CD
440 DATA 66,C6,C1,CD,18,B9,C9
```

Figure 9.1 (Contd)

Read track and sector

This utility program, listed in Figure 9.1, will allow you to read any sector of an Amstrad CP/M disc, in either System or Data format.(It cannot be used, however, on the PCW 8256.) You should type in the program as usual – the listing has been produced with a printer setting of forty characters per line so that the listing will look on the screen as it does here on paper. In addition, the lines are numbered in tens, so that you can use the AUTO facility of the CPC6128 to enter the lines without needing to type line numbers. Be very careful about the DATA lines, because these contain the machine code. An error in any one of these items will cause certain doom, so save the whole program before you attempt to run it. The 'checksum' in lines 370,380 will prevent any gross errors in DATA from affecting the program. This adds up the data bytes, and checks against the correct sum of 3407. If the sum is not correct, at least one data byte is incorrect, and the program is terminated. This is a useful safety precaution against a system crash. When the program runs, the screen divides into three windows, and you are asked whether the disc to be investigated is in System or Data format. You should answer this with S,s,D or d, though in fact, only S or s is checked for, and the program will assume Data format if S is not answered. You are then asked for a track number. The track numbers range from 0 to 39 and on CP/M System discs the first two tracks, 0 and 1, are reserved for CP/M use, with programs. Track 2 is used for directory entries, using 32 bytes for each entry. On Data discs, directory entries start on track 0, sector 0, and the programs start on sector 4 of this track. When you have entered an acceptable track number, you are then asked for a sector number. The manuals show the usual range of sector numbers as hex #41 to #49 for System discs, #C1 to #C9 for Data discs, but in this program the sectors are numbered from 0 to 8 only, to avoid complications. The conversion to the numbers that need to be passed to the ROM routines is carried out in the program in line 100. When you enter a valid sector number, you will hear the disc spin, and the display changes to show the byte number, its value in hexadecimal, and the character which corresponds to the byte, if any. Any byte in the range 0 to 31, is shown as a block character only. Showing the character is very useful when you are

looking at text files or directory entries, because it allows you to read the text, even though it is in the form of a vertical column. The display shows one byte, and waits for you to press the spacebar so that another byte can be displayed. You can hold down the spacebar if you want to see the bytes scrolling up the screen, which is useful if you are looking for something specific. If you don't want to see all of the bytes in a sector, you can use ESC ESC to leave the program in the usual way.

How it works

The BASIC part of the program is relatively simple. After clearing the screen, the subroutine at line 250 is called. This sets up the memory size, the address for the machine code, and the windows. The paper, border and pen colours are also selected. The next subroutine at line 350 then pokes in twenty-eight bytes of machine code which perform the disc access. The bytes are stored in hex codes, because these are easier to enter and check than ordinary (denary) numbers. The checksum in line 380 will ensure that the machine code is not permitted to run unless the bytes are correct. This is not infallible, because if you entered two error bytes which gave the same sum as the two correct bytes, this would not be picked up by the checksum, but for most purposes, the checksum is adequate. With this done, the program is ready to start, and the title is printed. Lines 60 and 80 request the track and sector numbers respectively. If an incorrect number is entered, the error is trapped, and a subroutine at line 400 prints a message, so that you can re-enter the data.

When the numbers have been entered, lines 100, 110 adjust them. For a CP/M System disc, the sector numbers must be in the range #41 to #49. Line 100 adds 1 and also #40 to accomplish this. If the disc is a data disc, then #C0 is added rather than #40. Line 110 then combines the track and sector numbers into a two-byte number, with the track number as the higher byte. In BASIC, this is done by multiplying the track number by 256 and then adding the sector number. By putting the number into this form, it is easy to pass its value to the machine code section.

The call to the machine code is made in line 120, and the value of track and sector number is passed. The machine code selects the disc operating ROM, reads the track and sector that has been specified, and then restores the BASIC ROM so that the program can resume. Line 130 then clears two of the windows, and line 140 prints the headings on the top window. Line 150 prints the message on the bottom window, then a loop starts in line 160. This will print the byte number (its position in the sector, ranging from 0 to 511), its value in hex, and its character shape. If the byte value is less than ASCII 32 (the space), then a chequer pattern is substituted. This is because printing some characters whose ASCII codes are less than 32 can cause odd effects, like screen clear, cursor movement and so on. Line 200 tests for the spacebar

being pressed, and loops continually until it is pressed. At the end of the loop, you are asked if you want another set of track and sector numbers to investigate, and if you answer with **Y** or y, the program repeats from line 20.

```
 1                          ORG   0A000H
 2                          LOAD  $
 3  A000 DD7E00             LD    A,(IX+0)
 4  A003 DD5601             LD    D,(IX+1)
 5  A006 1E00               LD    E,00
 6  A008 211CA0             LD    HL,BUF
 7  A00B F5                 PUSH  AF
 8  A00C 0E07               LD    C,07H
 9  A00E CD0FB9             CALL  0B90FH      ;select ROM
10  A011 F1                 POP   AF
11  A012 C5                 PUSH  BC
12  A013 4F                 LD    C,A
13  A014 CD66C6             CALL  0C666H      ; read sector
14  A017 C1                 POP   BC
15  A018 CD18B9             CALL  0B918H      ;deselect
16  A01B C9                 RET
17                  BUF:    DS    512
18                          END
```

Figure 9.2 The Z80 assembly language listing for the Track & Sector reader machine code.

The machine code

This is for Z80 machine code programmers only! The assembly listing is shown in Figure 9.2. The listing has been produced by the ZEN assembler which I use, rather than the Amsoft GENA3, but there is no difference as far as the assembly language itself is concerned. If you possess and want to use the GENA3 assembler, you will need to use ENT $ in place of LOAD $ in the second line. Lines 3 and 4 get the sector number byte into the A register, and the track number into the D register. The E register is loaded with 0, which is the number for drive A. The HL register pair is loaded with the address of a buffer in which 512 bytes can be stored after being read from the disc. The AF registers are then pushed on to the stack so as to store the sector number which was loaded into A. With the number 07 in register C, the call to B90F will select the disc drive ROM, which is coded as number 7. The same call will leave the BC registers loaded with identification numbers for the BASIC ROM which usually occupies the addresses #C000 to #FFFF. When this has been done, the AF pair are popped from the stack, and the BC number is pushed on. The sector number in A is then passed to C so that it can be used by the next routine. This call is to a routine in the disc ROM, and it will read the sector which has been specified by the track and sector numbers that have been passed in registers C and D. The buffer will then be filled by bytes from the selected sector, and the BC number is popped from the stack. This is then used by the call to B918 to restore the BASIC ROM so that the program can return to BASIC.

Note that this assembly language is not convertible directly into 8080 language because of the use of the index registers of the Z80, which are not present on the 8080. In any case, it is not possible to use this form of routine from CP/M because the addresses which are called are in the ROM, and CP/M always calls addresses in RAM. The principle of CP/M, remember, is that it should operate on any computer which uses the 8080, 8085 or Z80, so that no routines can ever be called from a ROM, only from the CP/M codes in RAM. By using the ROM routines which exist in the CPC6128 for this program, we make the program considerably shorter and simpler than would be possible if we had to rely on the CP/M routines entirely.

Disc editor

This is a program which allows you to *change what is stored on a disc*.(It cannot be used, however, on the PCW 8256.) Because of what it does, it has to be used with very great care. You must never alter a disc unless you have a backup copy. The only exception to this is if you are in the desperate state of having a valuable disc which has been magnetically damaged and cannot be loaded or copied. Do not attempt to use this program unless you know what you are doing. Neither I nor the publishers can be responsible for any loss of

```
10 CLS:GOSUB 330:GOSUB 440
20 PRINT#1,TAB(13)"Track & Sector.":PRIN
T#1,TAB(13);STRING$(14,"_")
30 PRINT#0,"SYSTEM OR DATA FORMAT - "
40 PRINT#0,"Please answer S or D":PRINT#
0,"The default is D."
50 INPUT DS$
60 PRINT#0:PRINT#0,"TRACK No.- (0 to 39)
":INPUT T%
70 IF T%<0 OR T%>39 THEN K%=39:GOSUB 480
:GOTO 60
80 PRINT#0:PRINT#0,"SECTOR No. (0 to 8)"
:INPUT S%
90 IF S%<0 OR S%>8 THEN K%=8:GOSUB 480:G
OTO 80
100 IF DS$="S" THEN S%=S%+1+&40 ELSE S%=
S%+1+&C0
110 D%=T%*256+S%
120 B%=&A041
130 CALL &A000,D%
140 CLS#0:CLS#1
150 PRINT#1:PRINT#1,TAB(8)"Byte No.";TAB
(20)"Hex";TAB(27)"Char"
160 PRINT#2,"Type hex. number to change,
 RETURN to":PRINT#2,"ignore, CTRL \ (RET
URN) to put":PRINT#2," back to disc."
170 FOR N%=0 TO 511:K%=PEEK(B%+N%)
180 PRINT#0,TAB(7);N%;TAB(16);HEX$(K%,2)
;
190 IF K%<32 THEN K%=127
200 PRINT#0,TAB(24);CHR$(K%)
210 INPUT#3,K$:IF K$=""THEN 270
220 IF K$=CHR$(28)THEN 310
230 IF LEN(K$)>2 THEN PRINT"Faulty numbe
r":GOTO 210
240 K%=VAL("&"+K$)
250 POKE B%+N%,K%
260 GOTO 180
270 NEXT
280 PRINT#0:PRINT#0,"Another one - Y or
N? "
290 INPUT A$:IF UPPER$(A$)="Y" THEN 20
300 END
310 CALL &A023:REM RECORD
320 END
330 MEMORY &9FFF:B%=&A000
340 INK 0,0:INK 2,26:INK 3,1
```

Figure 9.3 The disc editor program. Once again, this is in BASIC, and will have to be used after leaving CP/M. *Note*: this program cannot be used on the PCW 8256.

```
350 CLS:WINDOW#1,1,40,1,3
360 WINDOW#2,1,40,22,24
370 WINDOW#3,10,30,25,25
380 WINDOW#0,5,35,4,21
390 BORDER 4
400 PAPER#1,3:PAPER#2,3
410 CLS#1:CLS#2:CLS#3
420 PEN#0,1:PEN#1,2:PEN#2,2
430 RETURN
440 CS%=0:FOR N%=0 TO 61:READ D$:D%=VAL(
"&"+D$)
450 POKE B%+N%,D%:CS%=CS%+D%:NEXT
460 IF CS%<>7825 THEN PRINT"FAULTY DATA,
 PLEASE CHECK":END
470 RETURN
480 PRINT#0:PRINT#0,"Range 0 to";K%" onl
y, try again.":RETURN
490 DATA DD,7E,00,DD,56,01,32,3E,A0
500 DATA 1E,00,ED,53,3F,A0,21,41,A0
510 DATA F5,0E,07,CD,0F,B9,F1,C5,4F
520 DATA CD,66,C6,C1,CD,18,B9,C9,3A
530 DATA 3E,A0,ED,5B,3F,A0,21,41,A0
540 DATA F5,0E,07,CD,0F,B9,F1,C5,4F
550 DATA CD,4E,C6,C1,CD,18,B9,C9
```

Figure 9.3 (contd)

data due to the use of this program, because what you do with it is entirely up to you. If you know what you are doing, it can sometimes be a way of rescuing a valuable disc from a fate worse than reformatting. If you don't know what you are doing, experiment with a backup disc that you don't care too much about until you *do* know what you are doing.

With that very necessary warning over, let's see what the program of Figure 9.3 does. It is very similar in style to the Track & Sector reader and you can create one from the other by renumbering and editing. There is much more machine code data, however, so once again a checksum has been used to ensure that you can call the machine code only when the data is correct. If you get an error message about this, check your data lines carefully. Look in particular for a B used in place of 8, or the other way round, since this is the most common source of trouble. If the DATA lines are correct, the program will display its Track & Sector title, and you will be asked as before to supply track and sector numbers. If you are trying the program on a spare disc in System format which has some files recorded, use Track 2, sector 0. This will produce the directory entries.

You will now see the same display as for the Track & Sector program. This time, however, you need to use the RETURN key if you want to proceed from one byte to the next. If you want to change the byte at an address, you type the new byte, *in hex code*, and then press RETURN. This

```
 1                                ORG   0A000H
 2                                LOAD  $
 3 A000 DD7E00                    LD    A,(IX+0)
 4 A003 DD5601                    LD    D,(IX+1)
 5 A006 323EA0                    LD    (STOR),A
 6 A009 1E00                      LD    E,0
 7 A00B ED533FA0                  LD    (STOR+1),DE
 8 A00F 2141A0                    LD    HL,BUF
 9 A012 F5                        PUSH  AF
10 A013 0E07                      LD    C,07H
11 A015 CD0FB9                    CALL  0B90FH        ;select ROM
12 A018 F1                        POP   AF
13 A019 C5                        PUSH  BC
14 A01A 4F                        LD    C,A
15 A01B CD66C6                    CALL  0C666H        ; read sector
16 A01E C1                        POP   BC
17 A01F CD18B9                    CALL  0B918H        ;deselect
18 A022 C9                        RET
19 A023 3A3EA0                    LD    A,(STOR)
20 A026 ED5B3FA0                  LD    DE,(STOR+1)
21 A02A 2141A0                    LD    HL,BUF
22 A02D F5                        PUSH  AF
23 A02E 0E07                      LD    C,07H
24 A030 CD0FB9                    CALL  0B90FH
25 A033 F1                        POP   AF
26 A034 C5                        PUSH  BC
27 A035 4F                        LD    C,A
28 A036 CD4EC6                    CALL  0C64EH        ;write sector
29 A039 C1                        POP   BC
30 A03A CD18B9                    CALL  0B918H        ;deselect
31 A03D C9                        RET
32                   STOR:        DS    3
33                   BUF:         DS    512
34                                END
```

Figure 9.4 The Z80 assembly language for the disc editor program machine code.

will repeat the line on the screen, to ensure that you see the change, and you will need to press RETURN to get the next line. When you have made alterations, pressing CTRL \ will enter a 'face' shape in the entry strip at the bottom of the screen, and when you press RETURN on this, the sector will be re-recorded on the disc. The whole procedure has been made *deliberately* clumsy, so that you don't find it too easy to zap valuable bytes off a disc! If you want to test it, look through the directory sector until you find a filename. If necessary, press ESC twice and re-run the program when you have noted the start of a filename. When you know the byte number at which the filename starts, you can edit this name into something different, like ALTERIT – but not more than eight characters. You have to enter the

characters as hex codes, so you will need a hex-ASCII table. A suitable table is included as Appendix E.

How it works

The BASIC program follows the lines of the Track & Sector reader closely, and the main changes are in the entry of replies. The machine code is poked into memory, and the address is updated in line 120 to the start of the section that will be used as a buffer. A new window, #3, has been defined for this purpose, so that the prompt for the INPUT step in line 210 does not disturb the data display. Lines 210 to 230 test the entry. It is not easy to make a complete set of tests for a valid hex number, and no attempt has been made to trap anything other than the length of the number and the value. Line 240 obtains the value of the hex entry, and line 250 pokes this in place in the memory buffer that has been set aside. The GOTO 180 in line 260 then repeats the display step so that you can check that the change has been made. If you come to the end of the stored bytes without pressing the CTRL \ keys, then you will be asked for another T & S, and this will allow you to look at more data without altering the disc. You can also use ESC ESC at any point to get out of any disc alteration. If, on the other hand, you have altered bytes and you are determined to see the disc altered, then press CTRL \. The disc will spin briefly, and the deed is done. Use CAT now to see if your disc still works! If you have only altered a filename, then you will see the new filename on the disc. If, however, you have altered the numbers that lie between the filenames, then you can expect anything to happen!

The machine code

The assembly language for the machine code is illustrated in Figure 9.4. Once again, this is in Z80 code, and you should ignore this section if you know no Z80 code. If you feel that this is something that you should know, then my book *Introducing Amstrad Machine Code*, also published by Collins, will be of interest. Getting back to the disc editor, this is a straightforward development of the Track & Sector machine code, and it starts by passing values from BASIC into the Z80 registers. In this case, however, the values are also stored at addresses in the program, labelled as STOR. This allows the program to pick up these values easily when the sector is rewritten to disc. The rest of the read section to line #A022 then follows familiar lines. The new part starts in line #A023. This loads the accumulator from memory, and the next line loads the DE register pair. This puts the correct track and sector numbers back into the registers for the write routine, and the disc ROM is then selected by loading the select number 7 into register C and calling #B90F. The sector number is then

transferred to the C register, and the sector write routine at #C64E is called. This writes bytes from the buffer to the disc, and then the ROM select bytes are popped into BC so that the ROM deselect call to #B918 can be made. Once this has been done, the program returns to BASIC.

CP/M disc files

CP/M controls its disc files in a way that is very different from the method used by AMSDOS for BASIC programs, and it's not easy to discover for yourself how this is done. The best introduction is a practical one, using SID once again. Load in SID, and then load in (using the **E** command) any of your CP/M programs, such as ESC2. Now note the end address of this program, which was 0129 in my version, and use SID to write a file to the disc. Name the file TEST, so that you use the **W** command of SID in the form **WTEST,0100,0129**. When you press RETURN on this, the program will be recorded under the name TEST. Now take a look at memory locations 005C to 007C, as illustrated in Figure 9.5. This part of the memory is referred to as the *transient file control block*, or *TFCB*. This is transient, because the contents of this part of memory will change each time a new file is being used. The block contains all the information on the file, and is a copy of the directory entry for the file. The first byte, at address 005C, is the disc drive number, which will always be 00 if you are using a single-drive CPC6128 or PCW 8256. The next eleven bytes are used for the filename, including the extension but omitting the dot, so that the name TEST appears with seven blanks (hex code 20) following it. Following the name bytes are several bytes that are used by CP/M. The first of these, at address 0068, indicates how many multiples of 16K will be needed to store the file. This is zero for this example, because it will locate in the lowest 16K of memory. The next two bytes, 04 C0, are used by CP/M for its own purposes, and the last byte in the first set, 01, is the record number. The real meat occurs in address 006C. This is the start of the allocation table, which shows where the program is to be found on the disc. This is not a straightforward track and sector count, and this example has been obtained from a disc in Data format. The number is 3A, which is 58 in denary. The principle is that this is the number of 1K blocks on the disc, counting from the end of the directory track to the start of the file that we call TEST. Since there are 9 sectors per track, each of 0.5K, this gives 4.5K per sector. Dividing 58 by 4.5

```
005C: 00 54 45 53 54 20 20 20 20 20 20 20 00
04 C0 01 .TEST        ....
006C: 3A 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 :................
007C: 01
```

Figure 9.5 A printout of the transient file control block addresses.

gives 12 and leaves 4. The figure of 12 is the track number, and 4 is the number of 1K blocks used in this track, which is 8 sectors. The program TEST will therefore start on the ninth sector of track 12, sector 8. This, of course, was the allocation for the disc that I was using, and you can expect it to be entirely different for yours. If the file that is written consists of several 1K blocks, then the byte at #006C will be followed by others, up to a total of 16K. If the file consists of more than 16K, then the numbers will have to be changed after each 16K of file, and this is organised by the file extent number which is stored at #0068.

Since the relationship between allocation numbers and the disc track/sector structure is not exactly straightforward, it's as well that we don't normally have to worry about these numbers. By using the file control block with calls to CP/M, we can control the reading and writing of files without reference to where the files happen to be stored in the disc. The way in which a file control block is established is rather neat, and is a feature of CP/M. When you type the name of a file, that file will be loaded into the transient program area, the part of memory that starts at address #0100, and execution will start immediately at this address of #0100. In the course of this action, a file control block is set up momentarily. If this program, however, is to make use of any files for reading or storing data, or for reading another program file, then another set of data must be put into the file control block. This is organised when the name of the second file follows the first name. For example, suppose you had a program called FILIT.COM, and it needed to access a file called TEST. By typing:

FILIT.COM TEST

you would load FILIT.COM into the TPA starting at #0100, and then replace its file control data by some of the file control data for TEST. By the time that FILIT.COM runs, then, the file control block will contain the name of the file TEST, with its drive letter/number and extension. The rest of the data on this file is put into the file control block when the file is opened, using another CPM call. The system allows for error messages to be printed if the named file does not exist or if it subsequently cannot be read. The creation of these file control blocks therefore allows us to control the opening, closing, reading and writing of files of any type, as you would expect of an operating system which is mainly used for business programs.

We shall round off this book, then, with one simple example of the file control block in use. You are not likely to be deeply concerned with designing random access filing database programs at this stage in learning CP/M assembly language, but it's important to have had some experience with the use of the file control block. If you pursue CP/M assembly language programming further, then what you have done in this section will be a very useful foundation for later work. An illustration of opening a file and getting information from a file control block provides useful experience. This is shown in Figure 9.6 in a program which prints out the file

```
005C =          FCB     EQU     05CH
006C =          LOC     EQU     FCB+10H
0000 =          BOOT    EQU     0000H
0002 =          WRITE   EQU     02H
0005 =          CPM     EQU     05H
000F            OPNF    EQU     0FH
0010 =          CLSF    EQU     10H
0100                    ORG     0100H
                ;SHOW FILE LOCATION
                ;NUMBER(S)
0100 115C00             LXI     D,FCB           ;OPEN
0103 0E0F               MVI     C,OPNF          ;FILE
0105 CD0500             CALL    CPM             ;TO GET VALUE
0108 216C00             LXI     H,LOC           ;HL POINTS TO FCB
010B 7E         DISP:   MOV     A,M             ;GET BYTE
010C B7                 ORA     A               ;SET FLAGS
010D CA1701             JZ      EXIT            ;OUT IF ZERO
0110 CD2201             CALL    PRNT            ;OTHERWISE PRINT
0113 23                 INX     H               ;ADVANCE POINTER
0114 C30B01             JMP     DISP            ;NEXT ONE
0117 115C00     EXIT:   LXI     D,FCB           ;CLOSE
011A 0E10               MVI     C,CLSF          ;FILE
011C CD0500             CALL    CPM             ;AND OUT
011F C30000             JMP     BOOT            ;OUT
                ;PRINT HEX BYTE
0122 F5         PRNT:   PUSH    PSW             ;PRESERVE AF
0123 CD4D01             CALL    CRLF            ;NEW LINE
0126 F1                 POP     PSW             ;RESTORE
0127 4F                 MOV     C,A             ;STORE BYTE TEMPORARILY
0128 0F                 RRC                     ;SHIFT
0129 0F                 RRC                     ;RIGHT
012A 0F                 RRC                     ;FOUR
```

Figure 9.6 A program which prints out the file location numbers for a named file. Note that there are no error messages.

```
012B 0F                 RRC                ;PLACES
012C CD3701             CALL    OPT        ;PRINT BYTE
012F 79                 MOV     A,C        ;GET AGAIN
0130 CD3701             CALL    OPT        ;PRINT BYTE
0133 CD4D01             CALL    CRLF       ;NEW LINE
0136 C9                 RET
              ;SELECT NIBBLE AND PRINT
0137 E60F     OPT:      ANI     0FH        ;SELECT LOWER
0139 FE0A               CPI     0AH        ;DECIMAL?
013B DA4001             JC      DEC        ;JUMP IF SO
013E C607               ADI     07H        ;ADD FOR LETTER
0140 C630     DEC:      ADI     30H        ;CONVERT
0142 E5                 PUSH    H          ;SAVE
0143 C5                 PUSH    B          ;REGISTERS
0144 5F                 MOV     E,A        ;TRANSFER BYTE
0145 0E02               MVI     C,WRITE    ;AND
0147 CD0500             CALL    CPM        ;WRITE IT
014A C1                 POP     B          ;RESTORE
014B E1                 POP     H          ;REGISTERS
014C C9                 RET                ;RETURN
014D E5       CRLF      PUSH    H          ;SAVE
014E 1E0D               MVI     E,0DH      ;CR
0150 0E02               MVI     C,WRITE    ;TO
0152 CD0500             CALL    CPM        ;SYSTEM
0155 1E0A               MVI     E,0AH      ;LF
0157 0E02               MVI     C,WRITE    ;TO
0159 CD0500             CALL    CPM        ;SYSTEM
015C E1                 POP     H          ;RESTORE
015D C9                 RET
015E                    END
```

Figure 9.6 (Contd)

block numbers for a named file. This program was stored as SHOLOC.COM, and by typing (for example) **SHOLOC TEST**, it is possible to find the locations on the disc for file TEST, assuming that such a file exists. Note that in this simple example, no provision has been made for detecting any error, such as a 'no such file' error, and when you try this, you should make sure that the file you request does exist. The error traps have been omitted simply to reduce the amount of typing that you need to do.

As it is, this example requires a lot of work, and has been prepared with ED, ASM and HEXCOM. The listing starts with the labelwords that will be used. Many of these are familiar by now, but the new ones are: FCB, used to mean the first address of the file control block; LOC, used to mean the location of the first 1K record of the file, along with OPNF and CLSF. The OPNF label is 0FH, and CLSF is 10H, both being the 'action' numbers that need to be loaded into the C register so that a CALL 0005H can carry out a disc operation. The principle is that opening a file, with a name in the file control block, will read the data for that file from the disc. The file should subsequently be closed, though this is not strictly necessary when a file is being read as distinct from being written. Once the file has been opened, the data can be read from the file control block and printed on the screen. This involves a new routine, for converting each nibble of a hex byte into ASCII code so that it can be viewed on the screen.

Down to detail, then. The file control commands are carried out by loading the file control block starting address into DE, loading in the appropriate command code into C, and calling 0005H as usual. In this case, the command code is the OPNF command, so that the file which has already been named when the program was called will have its directory entry read into the control block, so opening the file to be used. If the file cannot be opened, the accumulator will contain the byte FFH after the CALL CPM step, and this is how the error is detected so as to trigger an error message. We have dispensed with such a luxury, and assumed that the file will always exist. The next step, then, is to load the HL registers with the address LOC, at which the first of the record address numbers is placed. There now follows a loop which will copy the byte from the control block and display it on the screen. The byte is copied by using MOV A,M, and we then have to test for the byte being zero, which in this simple example is used to indicate the end of data. Note that it would not necessarily work correctly with a very long file, because there might be no zero bytes. Loading a byte from memory into the accumulator does not set flags, and the ORA A step is needed to set the flags, without altering the value of the byte in the accumulator. If the byte is zero, then the program is ended by closing the file and jumping to the address 0000. If a byte exists at the HL address, then its value is printed by converting to ASCII codes and sending these codes to the screen.

That concludes the main part of the program, and illustrates all that is needed to gain access to a file directory entry. Note that this program does not read or write the file, only its control block entry. The remainder of the

Byte A7 in binary is 1Ø1ØØ111.

The RRC command rotates right, 8 bits only.

First RRC on 1Ø1ØØ111 gives 11Ø1ØØ11 D3

Second RRC on 11Ø1ØØ11 gives 111Ø1ØØ1 E1

Third RRC on 111Ø1ØØ1 gives 1111Ø1ØØ FH

Fourth RRC on 1111Ø1ØØ gives Ø1111Ø1Ø 7A

And Ø1111Ø1Ø 7A
with ØØØØ1111 ØF

gives ØØØØ1Ø1ØØA – which isolates the nibble A in the lower half of a byte

Figure 9.7 How the nibble positions are interchanged by four rotations.

program concerns the display of the number which has been copied into the accumulator from the file control block. This involves some steps that we have not looked at so far, and which you are very likely to use in your own programs. The main portion concerns how the byte is displayed. A hex byte contains two digits, the upper and lower nibble. Each nibble may consist of a digit 0 to 9, or a letter A to F. If the nibble is a digit, its ASCII code is obtained by adding 30H, so that the ASCII code for 5, for example, is 35H. If the code is one which will provide a letter, like A, then the addition must be 37H. The procedure for printing a byte must therefore separate the byte into two nibbles, and add the correct amount to the number in each nibble to convert to ASCII. Since the printing on the screen will be in the order of high nibble then low nibble, we have to select the nibbles in this same order.

This process starts at the PRNT label. To make sure that the first byte appears on the screen, a new line is taken by a call to CRLF. This subroutine is one that should by now be familiar, and no more time will be spent on it. Calling this routine, however, will corrupt the contents of the accumulator, and since the accumulator contains the byte that we want to print, we need to push it on to the stack before calling CRLF, and recover it afterwards. Note that the appropriate command is PUSH or POP PSW (processor status word), not PUSH A. When this has been done, the byte in its present form is stored also in C, because we will now select the upper nibble. This is done by rotating the byte by four places right, and then ANDing with 0FH (at the start of the OPT routine). Figure 9.7 shows the effect of this on a byte to illustrate the action. The net result of all this is to place the former upper nibble of the byte into the lower nibble of a byte whose upper nibble is zero. If, for example, the original byte was 3A, the new byte is now 03. We then convert this into ASCII code as part of the OPT subroutine. The conversion compares the nibble with 0AH, the number 10 in denary. If the nibble is less

than this, it is a digit, if equal to 0AH or more, it is a letter. If CPI 0AH results in the carry bit being set, this will be because the nibble is less than 10 denary, and the jump ensures that only 30H is added to convert to ASCII. If the byte is 0A or greater, then the 07H is added as well to give the correct conversion for a letter. This conversion routine will probably be one that you will use extensively in your own routines. The new byte, which is now an ASCII code, is then written to the screen in the usual way, pushing the HL and BC registers on to the stack so as to preserve the pointer address and the copy of the original byte in C.

At address 012F, the accumulator is reloaded with the original byte from C, and this type OPT is called directly to blank out the upper nibble, leaving the lower nibble to be converted and printed in the same way as before. Once a byte has been read from the file control block and printed, the program returns to the main routine at 0113 to increment the pointer address in HL, and loop back for another byte until a zero byte is read. The whole listing looks rather intimidating, but like all programs in CP/M assembly language, it consists of self-contained sections which can be taken piece by piece. When you try the program you do not need to type the comments, since you already have them in the figure, but you may want to include them if you intend to build on this example for your own purposes.

At this point, further examples start to be rather long and time-consuming, and I'll summarise the disc commands for writing and reading files. Each command is carried out by loading a command code into register C, some address or data into DE, and then calling 0005H. Taking a write action first, your data, which will normally consist of ASCII codes, must be contained in a buffer part of memory, for which you keep a starting address BUFR. You also need to keep a record count byte, which is the number of 128 byte units of buffer that will be used. The routine starts by enabling writing by using the control code 0DH – nothing is needed in DE for this action. The disc can be checked for an existing file of the same name by putting the file control block address into DE, and using the control code 11H – if the accumulator contains FFH following the call to 0005H, then the file already exists on disc. You can then arrange a message which will allow you to leave the program if you do not want to overwrite the file, or continue if you do. The existing file can be erased by loading DE with the file control block address, and using control code 13H. Once all this has been attended to, you can open the file for writing by using DE as before, with code 16H in C. This creates a directory entry – once again, the byte FFH in the accumulator shows that this could not be done for some reason, and allows you to generate an error message. The writing is done first by loading the buffer address into DE, and 1AH into C, then calling 0005H. This establishes the position of the data. The data is then written by putting the file control block address into DE, code 15H into register C, then calling CPM. As usual, FFH in the accumulator after the call means something is wrong. The buffer address then has to be incremented by 80H (128 denary)

in readiness for the next record, and the record count byte can be decremented and tested. If there are more records to write, the program loops back at this point.

Reading a file is done by allocating a buffer address and then opening the file, using the file control block address in DE and code 0FH in register C. The next step is to put the buffer address in place, and this involves putting the address into DE, and code 1AH into C, then calling 0005H. If the accumulator contains zero after the call to CPM, there is another record to read, and if the number is 01, then the end of the file has been reached – any other number indicates a read error. The records can be counted by incrementing a number in the memory, the buffer address will have to be incremented by 80H, and the program can loop until all the records of a file have been read. A limit here is that there may not be enough space in the memory for a long file, though this is unlikely with the Amstrad machine, which has more CP/M space than normal. The memory space can be checked by comparing the most significant byte of the buffer address with the most significant byte of the top-of-memory address, which is held in address 0007. This is normally DBH on the Amstrad, but the comparison should decrement this byte before comparing with the content of register H (assuming that the buffer address is held in HL) to avoid wiping any part of the CCP codes. The file can be closed after reading, but this is not necessary.

Tailpiece

It's impossible to end a book on a topic like this in a tidy way. There is always more to do, much left uncovered, loose ends to gather. In any book shorter than an encyclopedia, this is inevitable. Only a book which is aimed at an experienced programmer can contain a large amount of information in a small space, and such a book is meaningless to a beginner. The purpose of this book has been to introduce you to the methods of CP/M programming, along with some routines that will be useful. It inevitably leaves you part-way along a journey which can be a very long one, but this is the point at which you have learned enough to cope with more advanced texts. An advanced book on Amstrad CP/M will take you the rest of the way. It's all up to you now!

Appendix A
Languages Under CP/M

When you use CP/M on your Amstrad CPC6128 or PCW 8256, you cannot write in the normal Locomotive BASIC that is provided in the ROM of the machine, because this is incompatible with CP/M. You can, however, load in a language like BASIC as a CP/M file, providing that you can get hold of a copy on the Amstrad 3 inch disc. There is little point in buying the type of BASIC that is described as interpreted, because this type of BASIC must be present in memory in order to run any program it has created. By contrast, a BASIC that is described as 'compiled', and which is specifically described as creating CP/M COM files, can create programs which you can later run without the language file being present, just like any other COM file. At the time of writing, no BASIC of this type was available on the Amstrad disc size, though many versions are available if you are one of the fortunate users who have 5¼ inch discs connected to your CPC6128 or PCW 8256.

There is one very good higher level language, called 'C', which is available in a form which will generate CP/M COM files. This C Compiler is obtainable from Hisoft of Dunstable (look for their advertisements), and if you have already bought their earlier C Compiler for AMSDOS, an upgraded version is available. The use of this C Compiler is a much easier path to generating really large CP/M programs than programming directly in assembly language. If you need to use pieces of assembly language for specific purposes (such as random access filing), the Hisoft C Compiler allows for machine code to be placed in with the compiled codes.

Appendix B

Addressing Methods of the 8080

(1) Immediate addressing

The byte to be loaded immediately follows the instruction byte in the memory. Used only for loading.

Examples

LXI , MVI, CPI, ANI

(2) Absolute addressing

The full two-byte address for the data follows the instruction byte, so that three bytes in all are needed. Used for loading or storing commands.

Examples

JMP nnnn, LDA nnnn, LXI H,nnnn, CALL nnnn.
In these examples, nnnn means a two-byte address.

(3) Register indirect addressing.

A double register, such as HL, DE or BC is loaded with an address. The byte to be obtained at this address is then loaded or stored by using a single-byte instruction. When the HL register pair is used, the letter M describes the memory address in the HL registers, and commands such as MOV A,M or MOV M,A can be used. Other examples are LDAX B, STAX D.

(4) Implied addressing.

The address for the operation is implied in the instruction, and is usually a register. Examples are the RST commands, PUSH and POP, RLC, DCR, etc.

Appendix C
8080 Instructions

The following is a list of all the instructions of the 8080 in alphabetical order, with execution times and abbreviated actions. The times are given in terms of machine-cycle time, which is the time between clock pulses. For the Amstrad CPC6128 and PCW 8256 this time is 0.25 millionths of a second.

Abbreviations

d – byte of data or single-byte port address
a – two-byte address
r – single byte register, or byte in address held in HL
rp – register pair, HL, DE, BC

Mnemonic	Time	Effect
ACI d	7	Add immediate to A, with carry in
ADC r	4	Add to A from register, with carry in
ADD r	4	Add to A from register, with no carry in
ADI d	7	Add to A immediate, with no carry in
ANA r	4	AND A with register contents, result in A
ANI d	7	AND immediate with A, result in A
CALL a	17	Call subroutine at address
CC a	17/10	Call address if carry set
CM a	17/10	Call address if sign negative
CMA	4	Complement number in A
CMC	4	Complement (invert) carry flag
CMP r	4	Compare accumulator with register content
CNC a	17/10	Call address if carry flag clear
CNZ a	17/10	Call address if zero flag clear
CP a	17/10	Call address if sign positive
CPE a	17/10	Call address if parity even
CPI d	7	Compare accumulator with immediate byte
CPO a	17/10	Call address if parity even
CZ a	17/10	Call address if zero flag set
DAA	4	Decimal adjust accumulator for BCD addition
DAD rp	11	Add contents of double register to HL, result in HL
DCR r	4	Decrement register contents

DCX rp	6	Decrement contents of register pair
DI	4	Disable interrupts to MPU
EI	4	Enable interrupts to MPU
HLT	4	Halt until reset or interrupt received
IN d	11	Load A from port number d
INR r	4	Increment register contents
INX rp	6	Increment register pair contents
JC a	10	Jump if carry set
JM a	10	Jump if sign negative
JMP a	10	Jump unconditionally
JNC a	10	Jump if carry clear
JNZ a	10	Jump if not zero
JPE a	10	Jump if parity even
JPO a	10	Jump if parity odd
JZ a	10	Jump if zero
LDA a	13	Load accumulator from address
LDAX rp	7	Load A from address in BC or DE
LHLD a	16	Load HL from address
LXI rp,a	10	Load register pair with address
MOV r,r	4	Load one register from another
MVI r,d	7	Load register immediate
NOP	4	No operation
ORA r	4	OR accumulator with register content
ORI d	7	OR accumulator immediate
OUT d	11	Send accumulator contents out at numbered port
PCHL	4	Jump to address in HL
POP rp	10	Load rp with two bytes from end of stack
PUSH rp	11	Copy two bytes in rp to end of stack
RAL	4	Rotate accumulator and carry left
RAR	4	Rotate accumulator and carry right
RC	11/5	Return from call if carry set
RET	10	Return from call unconditionally
RLC	4	Rotate A left, copy bit 7 to carry
RM	11/5	Return from call if sign negative
RNC	11/5	Return from call if carry clear
RNZ	11/5	Return from call if not zero
RP	11/5	Return from call if sign positive
RPE	11/5	Return from call if parity even
RPO	11/5	Return from call if parity odd
RRC	4	Rotate A right, copy bit 0 into carry
RST d	11	Restart at coded address in range 00 to 38H
RZ	11/5	Return from call if zero
SBB r	4	Subtract content of register and carry from A
SBI d	7	Subtract immediate data and carry from A
SHLD a	16	Load addresses with bytes from H and L

SPHL	6	Load SP from HL
STA a	13	Store byte from A to address
STAX rp	7	Load address in BC or DE with content of A
STC	4	Set carry flag (to 1)
SUB r	4	Subtract register content from A
SUI d	7	Subtract immediate byte from A
XCHG	4	Swap contents of DE with HL
XRA r	4	XOR byte in register with A
XRI d	7	XOR immediate byte with A
XTHL	19	Exchange end of stack address with HL

Notes

Where two times are shown, this is an instruction which is carried out only if a condition is true. The shorter time is the time that is used when the condition is false, the longer time applies when the condition is true. When the addressing method can use a register or M (the address in HL), then using M implies longer times, an extra 3 cycles. For example, SBB C will require 4 cycles, but SBB M will require 7 cycles.

Appendix D

The ED Commands

The command letters of ED. Where these are accompanied by a number, that number can be # to mean all, or number 'n'. Zero can sometimes be used to mean current line.

Command	Description
nA	Append 'n' lines from file into buffer
0A	Append from file until buffer is half full
B, −B	Move to start(B) or end(−B) of buffer
nC, −nC	Move 'n' characters forward(nC) or back(−nC)
nD, −nD	Delete 'n' characters forward(nD) or back(−nD)
E	Save file and return to CP/M Plus
Fstring↑Z	Find string in text
H	Save file, stay for editing
I	Enter insert mode
Istring↑Z	Insert string at current position
Jstring1↑Zstring2↑zstring3	Find string1, concatenate string2 on to it, then delete all characters up to start of string3
nK, −nK	Kill (delete) 'n' lines forward(nK) or back(−nK)
nL, −nL	Move 'n' lines forward(nL) or back(−nL)
0L	Move to start of current line
nMcommandlist	Execute command list 'n' times
n, −n	Move 'n' lines forward(n) or back(−n) and display that line
:ncommand	Execute from present line to line n
Nstring↑Z	Extend find string
O	Return to original file
nP, −nP	Move 23 lines forward(nP) or back(−nP) and display
Q	Abandon file, return to CP/M Plus; you will be asked to confirm Y/N
R↑Z	Read LIB file into buffer
Rfilename↑Z	Read named file into buffer
Sstring1↑Zstring2	Delete all string1, substitute string2
nT, −nT	Type 'n' lines forward or back
0T	Type current line

U, −U	Translate to upper-case
V, −V	Line numbering on/off
0V	Display buffer space figures
nW	Write 'n' lines to new file
0W	Write until buffer is half empty
nX	Write or append 'n' lines to LIB file
nXfilename↑Z	Write 'n' lines to named file; append lines if previous X command applied to this same file
0x↑Z	Delete LIB file
0xfilename↑Z	Delete named file
nZ	Wait for 'n' seconds

Appendix E

The ASCII Codes in Hex

No.	Hex.	Char.	No.	Hex.	Char.
32	20		80	50	P
33	21	!	81	51	Q
34	22	"	82	52	R
35	23	#	83	53	S
36	24	$	84	54	T
37	25	%	85	55	U
38	26	&	86	56	V
39	27	'	87	57	W
40	28	(	88	58	X
41	29	)	89	59	Y
42	2A	*	90	5A	Z
43	2B	+	91	5B	[
44	2C	,	92	5C	\
45	2D	-	93	5D	]
46	2E	.	94	5E	^
47	2F	/	95	5F	_
48	30	0	96	60	`
49	31	1	97	61	a
50	32	2	98	62	b
51	33	3	99	63	c
52	34	4	100	64	d
53	35	5	101	65	e
54	36	6	102	66	f
55	37	7	103	67	g
56	38	8	104	68	h
57	39	9	105	69	i
58	3A	:	106	6A	j
59	3B	;	107	6B	k
60	3C	<	108	6C	l

61	3D	=	109	6D	m
62	3E	>	110	6E	n
63	3F	?	111	6F	o
64	40	@	112	70	p
65	41	A	113	71	q
66	42	B	114	72	r
67	43	C	115	73	s
68	44	D	116	74	t
69	45	E	117	75	u
70	46	F	118	76	v
71	47	G	119	77	w
72	48	H	120	78	x
73	49	I	121	79	y
74	4A	J	122	7A	z
75	4B	K	123	7B	{
76	4C	L	124	7C	¦
77	4D	M	125	7D	}
78	4E	N	126	7E	~
79	4F	O	127	7F	

Appendix F

Effect on Flags

The list below indicates how the actions of commands may affect flags. Note that the arithmetic commands will always affect the condition of the flags following the command, but only the ADC, ADI, SBB, SBI commands are affected by the state of the carry flag *before* the command is executed. The abbreviations are as used in Appendix C.

(1) No flags affected

MOV r,r   POP   PUSH   RST n   RET

All conditional returns, such as RC, RZ

XTHL	PCHL	SPHL	XCHG	CMA
NOP	HLT	DI	EI	INX r
DCX r	LDAX rp	STAX rp	MVI r,a	OUT d
IN d	LXI rp,a	LHLD a	SHLD a	LDA a
STA a				

All jump instructions

All CALL instructions

(2) Only carry flag affected

RLC	RR	RAL	RAR	DAD rp
STC	CMC			

(3) Flags other than carry *affected*

INR r   DCR r

(4) All flags affected

ADD r	ADC r	ANA r	CMP r	DAA
ORA r	SBB r	SUB r	XRA r	ACI d
ADI d	ANI d	CPI d	ORI d	SBI d
SUI d	XRI d			

Appendix G
Calls to 0005H

The calls to address 0005H are all controlled by using a code in register C, and many require registers DE to be loaded, or will return a byte in A. In some cases, error signal bytes can be returned in A. The following are some of the most useful calls.

Byte in C	Action
1	Read ASCII code from keyboard.
2	Write ASCII code to screen.
5	Write character to printer.
0A	Read line from keyboard into buffer.
0B	Check keyboard for key pressed.
0D	Prepare BDOS, select drive A.
0E	Select drive.
0F	Open file for read or write.
10	Close a file.
11	Find a file in directory.
12	Find next occurrence of file.
13	Delete a file.
14	Read one record from disc into buffer.
15	Write one record from buffer to disc.
16	Create a disc directory entry.
1A	Set buffer address in RAM for read or write.

Index

absolute addressing, 32
accumulator, 28
accumulator actions, 39
address bus, 28
address numbers, 7
addressing methods, 29, 133
altering memory, 15
arithmetic, 40
arithmetic set, 10
ASCII codes, 5
ASCII codes in hex, 139
ASM filename, 105
ASM use, 109
assembler, 18, 27, 101
assembly language, 18, 29
assembly with SID, 45

BASIC, 1
BASIC keywords, 13
BDOS, 24, 66
BDOS calls, 142
binary code, 3
binary digit, 2
BIOS, 24
bit, 2
blank program, 45
block diagram, 8
breakpoint, 103
breakpoint address, 93
bug, 101
Byte, 3

C language, 25
calls to 0005H, 142
carriage return, 85
carry bit, 35, 40
case change, 53
CCP, 24
checksum, 116
clock, 26
CMP, 43
COM files, 25
compare action, 36
compiled program, 6
compilers, 25
complementing, 21
console command processor, 24
converting numbers, 20
copy protection, 114
copying action, 9
copyright notice, 23
CP/M user group, 77, 103
CPU, 9
crash, 46

data byte, 32
DDT, 7
debugging, 101
decrement, 41
direct addressing, 32
disc editor, 114
disc editor program, 119
disc utility, 114
double registers, 33
DUMP use, 7

ED summary, 137
ED use, 104
eight-line codes, 3
electrical programming, 13
endless loop, 27
entering line, 83
EQU use, 107
exchange register contents, 74
extended addressing, 32

faulty loop, 104
field, ED entry, 106
file directory, 128
fill screen action, 66

firmware, 4
flag register, 35
flags affected, 141
flowchart shapes, 55
flowcharts, 54

garbage, 17
gates, 26

hash sign, SID, 8
hex calculator, 19, 22
HEX file, 109
hex scale, 18
hexadecimal, 7
HEXCOM use, 109
holding loop, 62, 80

immediate addressing, 30
increment, 41
initialisation routine, 16
instruction byte, 27
instruction times, 134
instructions affecting flags, 141
invoking ED, 105

jump group, 43
jump set, 12
JZ action, 37

labels, 39
languages, 132
least significant digit, 3
line feed, 85
line length limiter, 83
load, 39
logic, 40
logic actions, 11
logic set, 10
longer loops, 60
loops, 58

MAC, 113
machine code, 1, 27
map of registers, 38
memory, 1
memory map, 23
message on screen, 85
Microsoft M80 assembler, 102
mnemonics, 30
monitor, 7
most significant digit, 3
MOV, 33
moving code, 97
MPU, 9

negative numbers, 21
number codes, 17

opening file, 125
operand, 30
operator, 30
ORG address, 107

parity bit, 109
patching, 46
PC, 28
permanent memory, 3
plan for count action, 59
POP, 70
port, 15
preserving flags, 73
press any key, 92
print character set, 78
printing text, 95
program counter, 28
pseudo-instructions, 107
PSW, 73
PUSH, 70

RAM, 4
read track and sector, 116
read-write memory, 4
reading keyboard, 80
record source code, 108
recording program, 49
register changes, 68
register indexed addressing, 32
register map, 38
register-indirect load, 52
registers, 28
RMAC, 113
ROM, 4, 16
rotate, 41
rotate program, 51
RST 6, 46

screen operations, 66
screen-write loop, 74
set flag, 35
SETKEYS, 23
shift action, 42
SID, 7
signalling, 2
signed number, 22
significance, 3
simple program, 47
single step, 49
source code, 103
stack, 69

stack pointer, 35, 70
stack use illustrated, 71
states, 2
status bits, 36
status register, 35
store, 39
subroutine, 63
Subset series, 103
syntax, 30
system use, 17

TAB use with ED, 106
terminator, 85
text entry mode, ED, 106
text space, 85
text storage and printing, 90
textfile, 5
TFCB, 124
time delays, 62
timer subroutine, 64
timing of loops, 61
top of stack, 70
top-down planning, 57
TPA, 24
track and sector reader, 116
transient file control block, 124
transient program area, 24
transient programs, 4
TYPE use, 4
typewriter program, 98

universal time delay, 62
unsigned number, 22
using ASM, 109
using HEXCOM, 109
using calls to BDOS, 67

Warnier-Orr diagrams, 56

XOR use, 12

Z80, 9
zero byte, 85
zero flag, 36

The new Amstrad CPC6128 and PCW8256 machines both use CP/M 3.1 — the first time that this operating system has been made available in computers in this price class — and as a result it has brought in a whole new group of users who have never come across the CP/M 3.1 system before.

This book has been written especially for these newcomers to CP/M. CP/M software never exactly fits your precise needs so some 'patching' is always desirable. You need, however, to have some knowledge of CP/M assembly language in order to adapt CP/M programs (including the large amount of free 'public domain' software available) to your needs. This book, therefore, deals with the CP/M structure and how simple CP/M machine code programs can be written with the assembler editor package supplied with the machine. Numerous useful routines and listings are provided, and the book is written in a straightforward style which you should find easy to follow.

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.

Other books for Amstrad users

ADVANCED AMSTRAD CPC6128 COMPUTING
Ian Sinclair
0 00 383300 3

AMSTRAD WORD PROCESSING on the PCW 8256
Ian Sinclair
0 00 383328 3

USING AMSTRAD CP/M BUSINESS SOFTWARE
Ian Sinclair
0 00 383359 3

ISBN 0-00-383309-7

COLLINS
Printed in Great Britain
£9.95 net