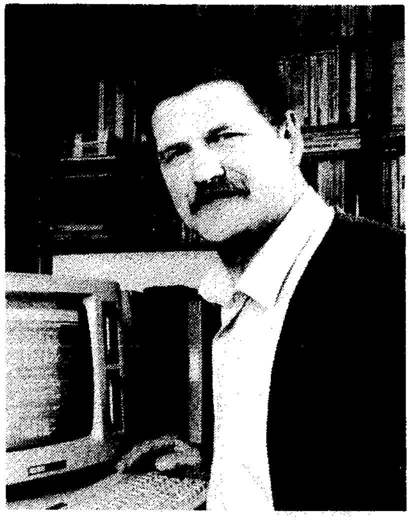
"PCW Machine Code"

The best book on programming the PCW.



How to control the screen, the printer, discs and the whole of the machine memory.

A full explanation of machine code with dozens of program examples.

With a special section on calculations such as Sin, Cos, Sq roots, Random Nos, 8-, 16-, and 32- bit arithmetic, etc., etc.

Full index and appendices.

"GOOD VALUE, VERY INFORMATIVE !" 'Amstrad PCW Magazine', Mar '89

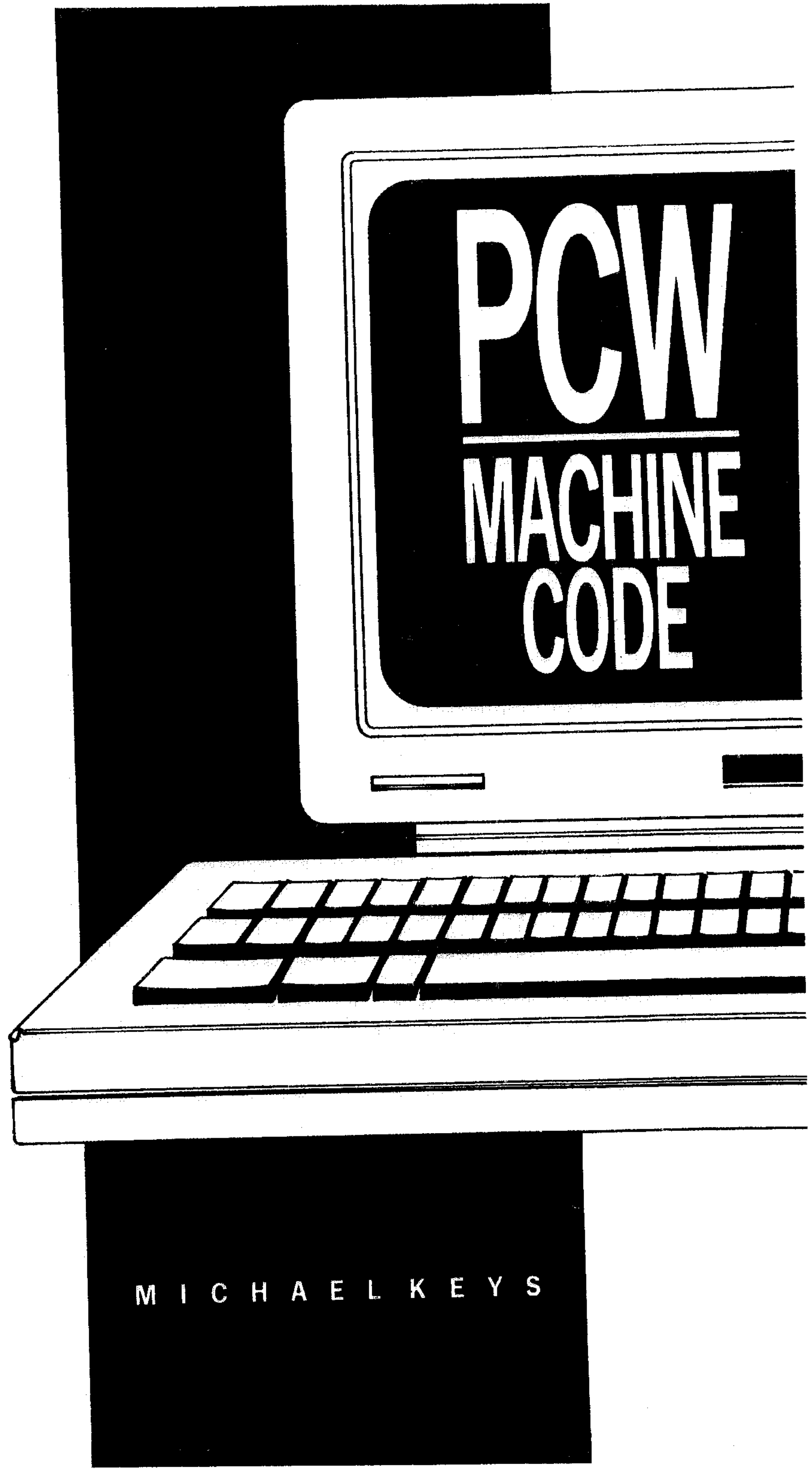
"WE STRONGLY RECOMMEND IT !" '8000 PLUS', Nov '89

"EXACTLY WHAT PCW OWNERS NEED !" 'Personal Computer World' Feb '89

£13.95 nett

ISBN 1 871892 00 7

SPA ASSOCIATES



MICHAEL KEYS

P C W

MACHINE CODE

ERRATA

page	position	correct version
99	"Deleting pixels"	'RUBOUT' is on page 97
110	near bottom	"See page 115 . . ."
121	second line	"page 62"
128	middle	"(see page 85)"
135	in first program	"(see page 62)"
137	16-bit div	"adc hl hl" (not hl de)
150	ninth line	"ld de, (Addr) . . ."
152	"Incr & Decr"	"inc de 19" (not '1')
165	second line	"out (240), a . . ."

P C W

MACHINE CODE

M. Keys

B.Sc.

Spa Associates
Spa Croft, Clifford Rd, Boston Spa, LS23 6DB

Acknowledgements

This book is based on my experience of using information derived over the years from books and periodicals, to the writers of which I would like to express my thanks. The information belongs to others, the mistakes are all mine.

Notice

'CP/M', 'CP/M Plus', 'Amstrad', 'PCW 8256', 'PCW 8512', and 'PCW 9512' are trade-marks.

First published December 1988
Second edition May 1989
Third Edition February 1990

ISBN 1 871892 00 7

CONTENTS

Chapter 1

Computing should be Sensible

If you believe that computing is fun, interesting, and useful then this book is for you. If you think it is a very serious pursuit that should be followed with great self-discipline according to laid down rules, then it may not be so suitable. In writing it I have attempted to share with the reader the pleasure I have derived from controlling my PCW from machine code programs. Apart from understanding rudimentary BASIC and having a healthy curiosity, you will need no other qualifications, even though all aspects of the machine's hardware are dealt with: as the man said, "Everything is simple once you know about it."

I have made Chapters 2 to 5 an introduction to machine code (for which I will often use the abreviation 'm/c') for those who have not met it before, though I have purposely kept this section short to leave more room for describing how to control discs, the screen, and the printer, which is the real purpose of the book. In referring to the Amstrad manuals I have given the page numbers in the books supplied with the '8256' and '8512' first, followed by the equivalent for the '9512' in square brackets, if any.

But, why bother with machine code at all ? Well, there are three reasons: it is fast, it uses very little memory, and it gives the programmer excellent control of the computer. Indeed, of all the available ways of programming the PCW, m/c runs the fastest, uses the least memory and gives the maximum level of control. And even if it is a little tedious to write, well, that's no price when set against the advantages.

Secondly, just a word about jargon. Jargon in private between consenting participants is fine. It is no more than a kind of verbal shorthand that enables people to communicate more freely; and who could object to that ? However, in computing it does get over-used and keeping that in check is a duty we owe to each other.

Not that jargonism is the sole prerogative of the world of computers; it seems to occur in every field of activity that has ever had a need for special words to describe its own peculiar objects and actions, though invariably that need has long been overlain by the tendency to wish to be seen to be a guy who knows what all the initials mean.

If a magazine or a book, which has a professional duty to communicate with its public, fogs you with pages of rubbish made up by ex-Pentagon stores clerks, then you should complain. For as long as computerites are allowed to self-stimulate in this way they will do so. It probably makes them feel better. My name for it is *w-language* where 'w' stands for a four-letter word ending in 'k'. In all cases it can either be ignored or replaced by a simple English word or phrase with the effect of improving the information content of the text. Maybe it is time for us all to oblige communicators to ensure that simple English phrases get a wider use.

I have honestly tried to exclude all *w-language* and jargon-berkery from this volume. If it turns out that I've failed, well, I will be duly humbled. Either way, I sincerely tells you what you wanted to know.

Chapter 2

The Basis of Computing

The Computer

As far as a programmer is concerned the computer consists of a **memory** and a **processor**. The PCW's processor is the Z80, which is made by the Zilog Corporation. It takes data from the memory, operates on it (ie. processes it), and then puts it back into memory where it is available to do something useful when required. Alternatively the processor can take in new data (from the keyboard, say) or convert existing data into the screen display, or into symbols for feeding to the printer.

Bits and Bytes

The absolutely smallest piece of data (ie. of information) that the computer can deal with is called a **bit**. A bit can be either switched on or switched off. A switched-on bit is said to be **set**, as opposed to a switched-off bit which is said to be **reset**. A set (on) bit corresponds to the number 1, and a reset (off) bit corresponds to zero. The arrangement of set and reset is as follows

SET = on = 1
RESET = off = 0

The PCW handles all its bits in groups of eight. A group of eight bits is called a **byte**. The combination of its set and reset bits in a byte determine the value of the byte. If all the bits have a value of zero, then the value of the byte will be zero. If some of them have a value of 1, then the value of the byte will be increased accordingly.

The bits have increasing rank from right to left. This corresponds to the way we write the numerals in conventional arithmetic; the figure on the right gives the number of units and figures to the left of it have increasingly greater significance (tens, hundreds, thousands, ten-thousands and so on). So it is with bits except that they can't represent numbers up to 9, they can represent only 0 or 1.

The bit on the extreme right is called the **least significant** bit, and the one on the extreme left is called the **most significant** bit.

Conventional arithmetic has ten numerals ("0" to "9") and we count in parcels of ten. (This is called counting 'to the base 10'.) I can specify increasing quantities up to 9 just by picking the next higher numeral. But, beyond 9, because there aren't any more numerals to pick from, I revert back to zero again, but I indicate that ten has been reached by writing a "1" to the left of the zero. After that I can keep increasing the units (the rightmost column) by picking higher and higher numerals until "9" has again been reached. I then have to revert the units to zero again, but I increase the tens to "2" to show that twenty has been reached, and this can be repeated *ad inf* to give numbers as large as we like.

Our normal system of counting is usually called the 'Decimal System' (because 'deci-' means "a tenth"). Purists usually suck their teeth and wag their heads at this, correctly pointing to the linguistic merits of "denary" over "decimal". ('Denary' means "of ten".) They are right of course, but we peasants now hold such sway that I don't think that 'denary' stands much chance, but it is definitely in in some circles so it is as well to be familiar with it. It is a feature of decimal (sorry, denary) arithmetic that a numeral acquires **ten times** its previous value if it is moved one column to the left.

Binary Arithmetic

The arithmetic that applies to counting with bits is called **Binary** arithmetic because only the two numerals "0" and "1" are available, and 'binary' means "having two parts". Counting in binary is the same as counting in decimal except that we run out of numerals much sooner; after the first increment in fact ('increment' means "add 1 to"). Starting at zero, the process of counting goes like this:

Start at zero	0 0 0 0 0 0 0 0	(=0)
Add 1 to give:	0 0 0 0 0 0 0 1	(=1)

Because we have now exhausted our list of numerals, we must revert the rightmost column to zero and increment the column to its lef

This gives	0 0 0 0 0 0 1 0	(=2)
Add another 1:	0 0 0 0 0 0 1 1	(=3)
Follow the rule:	0 0 0 0 0 1 0 0	(=4)
Etc	0 0 0 0 0 1 0 1	(=5)
Etc	0 0 0 0 0 1 1 0	(=6)
Etc	0 0 0 0 0 1 1 1	(=7)
Etc	0 0 0 0 1 0 0 0	(=8)
Etc	0 0 0 0 1 0 0 1	(=9)

You will notice that, analagously with decimal, a "1" acquires twice its previous value if it is moved one column to the left. This gives rise to the following important sequence in which the values are all powers of two.

00000001	= 1	00010000	= 16
00000010	= 2	00100000	= 32
00000100	= 4	01000000	= 64
00001000	= 8	10000000	= 128

If you add all these numbers up you will find that 11111111 in binary is equal to 255 in decimal, and hence 255 is the highest value that can be put into an 8-bit byte. Notice that, because '1' is the only odd number here, all even binary numbers have the least significant bit reset, whereas the odd ones have it set.

If we were unable to compute with numbers larger than 255 it's not likely that we'd bother to compute at all, but, as with the decimal system, there is no limit to the number of digits that may be used so a number of any size can be represented in binary, though for technical and economic reasons the Z80 never considers more than sixteen bits at a time (and even then it takes two bytes at the cherry). The additional eight bits make up what is called the **high byte**, and the original eight are referred to as, not surprisingly, the **low byte**.

Binary Multiplication and Division

These two operations are carried out as in decimal. Suppose I want to multiply 36 by 5. In binary these numbers are 0100100 and 00000101 and the multiplication goes :

```
       36            00100100
        5                 101
                     00100100
                 + 0010010000

   result            10110100  =  180
```

And to divide 188 by 5, ie. 10111100 by 00000101:

```
                 00100101      result = 37
         101    )10111100
             101
                 101
                  1000
                   101
      remainder     11   = 3
```

Using 16 bits

Suppose that in our counting we have reached 255. What happens if we add another '1' ? Well, if we have only one byte it will be reset to zero and the whole of our count will be lost, but with two bytes the count may proceed as if the two formed a single 16-bit number. All that is necessary is that any overflow from the low byte should be fed into the high byte and be preserved there. As follows:

	H.Byte	L.Byte
The count has reached 255:	00000000	11111111
Add another 1:	00000001	00000000
And another:	00000001	00000001
Etc	00000001	00000010
Etc	00000001	00000011

Notice that the high byte will not be incremented again until the low byte has again reached 255 and then another 1 is added. That is the same as saying that the high byte counts, not 1's, but 256's. Hence the

value of the high byte can be read as if it were an ordinary byte but with the result multiplied by 256. This gives us an easy way to calculate the maximum value that 16 bits can hold. The high byte can count up to 255 x 256 (ie. 65280), and the low byte may count a further 255. The total is therefore **65535.**

Numbering the bits

In computing the lowest number is considered to be 0, not 1. For this reason the least significant bit is called "bit No 0", the one on its left is called "bit No1", and so on, up to the most significant bit which is called "bit No 7". This is logical, but it gives rise to the apparent anomaly that the eighth bit is called "bit No 7" !

This can be confusing, but I suppose computerites will blame the confusion onto conversational speech for counting illogically not from the lowest number, but from only the second lowest, ie. from 1 ! The naming sequence is continued through the high byte, its least significant bit being called "bit No 8", and its most significant bit being called "bit No 15". In defence of the computerites, it is interesting that the bit numbers do correspond to the power of two that gives the value of each bit, as shown in the following table:

Bit values

bit No 0 has a value of 1, which equals 2^0
bit No 1 has a value of 2, which equals 2^1
bit No 2 has a value of 4, which equals 2^2
bit No 3 has a value of 8, which equals 2^3
etc., up to . . .
bit No 15 has a value 32768, which equals 2^{15}

Knowing that the values of all set bits are powers of 2, you may be interested to compute their individual values up to bit No 15, and obtain a check on the 16-bit total given earlier. Also notice that an individual bit value is always 1 more than the sum of all the bits to its right. For example bit No 7 has a value of 128, and the sum of bit Nos 0 to 6 is 127.

Negative numbers

With only 16 bits to work with, and each able to be only 0 or 1, how can we indicate that a number is less than zero ? Well to do so we have to reserve one of the bits as a **flag** to indicate the number's sign. If the flag is 'raised' (ie. if the sign-bit is set) then this indicates that the number is negative, and if the flag is 'down' (ie. if the sign-bit is reset) then we will take it to be positive. The sign-bit is invariably the most significant bit (bit No 7 for 1-byte numbers, or bit No 15 for 2-byte numbers).

Obviously the sign bit can't sometimes be used to indicate 'a value of 128' and at other times to indicate 'this number is negative' because then how could anyone distinguish between -128 and +128 ?

If we want it to be a sign flag we must make this clear at the start, and we must accept the penalty that 1-byte numbers can then be no larger than 127 (because bit No7 is reserved) and that 2-byte numbers can be no larger than 32767 (because bit No 15 is reserved). Naturally in 2-byte numbers you wouldn't reserve both bit No 7 and bit No 15 ; only one is necessary. Calculating with negative numbers is explained in more detail in Appendix 3.

Large and Small Numbers

Precise calculation with very large numbers is perhaps the hallmark of the computer. These are dealt with in a way that is quite unlike the one I have described so far. First the numbers are converted to their 'floating point forms' (which require 5 bytes each) and then the calculation is made. Floating point forms are reminiscent of the logarithmic form that was common before electronics took the drudgery out of calculation, but it might be as well for you not to wrestle with them yet; not many people do. Conveniently, the very small numbers that are used frequently in scientific and technical calculations can be handled in their floating point forms too.

A second way of making accurate calculations with large numers is called 'Binary Coded Decimal' (BCD). It is used pricipally in accountancy where it is important not to lose an odd digit or two, and the Z80 has a special facility devoted to it. We will look at it later.

The Alphabet

Because it finds numbers easy to handle, the computer adopts the simple expedient of giving each letter a number and then moves these about as if they were letters, and the more or less universally accepted

set of numbers which represent the letters, the numerals, the punctuation signs, and other useful symbols such as $, #, &, =, @, etc., are called the **ASCII Codes**. ASCII is an acronym for the American something or other connected with Information Interchange. The codes are listed on pages 113 to 118 [547 to 554] of the Amstrad manual.

A sequence of letters, numerals or similar symbols (ie. non-numbers) is called a string. Hence a string could be a single letter, a word, a sentence, a paragraph, a message, a set of numerals, or any other part or whole of a text item. (Note that a string of numerals is not a number; you can display it on the screen but you can't calculate with it. A number is the binary content of one or more bytes.) A sequence of ASCII codes may also be called a string. The end of a string is signalled by a **string-end marker** (which is called a 'delimiter' in *w-language*), which by convention is often the dollar sign ($), or its ASCII code.

The Hexadecimal System

If I hadn't told you, I bet you wouln't have guessed that there was any connection between 255 in decimal and 11111111 in binary. Still less does there appear to be a special significance to 65535; an arbitrary looking number if ever I saw one.

From the early days it was realised that the 'base-10' (decimal) is not a convenient base in which to express numerical values when dealing with electronic calculations. This is because ten is not a power of 2, but 2 is unavoidable because there are just two fundamental electrical states: 'on' and 'off' ; 'set' and 'reset'.

Counting to bases which are powers of two, ie to the 'base-4', and then to the 'base-8', were proposed as superior alternatives, but it is now universally agreed that the best one is the 'base-16' (though octal does have some modern uses). This gets rid of the terrible inconvenience of binary that it needs so many digits to express even quite small numbers, but at the same time is is easy to translate from one to the other if the need arises. The name given to counting in this base is Hexadecimal (literally 'six and ten') counting.

A big advantage of **hex** is that it expresses the values of bytes in a way that is easy to comprehend. The disadvantage to people unfamiliar with it is that it needs six extra symbols to supplement the usual "0" to "9", and their values take a while to sink in. Rather than make up six completely new symbols the first six capital letters were chosen (I think they should have made up new ones, and would have done it for them if they'd asked me.) Hence the numerals used in hex are as shown below.

decimal	hex	decimal	hex	decimal	hex
0	0	6	6	12	**C**
1	1	7	7	13	**D**
2	2	8	8	14	**E**
3	3	9	9	15	**F**
4	4	10	**A**	16	10
5	5	11	**B**	17	11

The sequence then continues in groups of sixteen so that 20h is equal to 32d , 30h is equal to 48d , etc.

Notice that to avoid misunderstandings over which base is being used, hex numbers invariably have a letter 'H' appended. You can add a 'd' to decimal numbers if you wish, but that is optional. Numbers without a following letter are assumed to be in decimal. Some writers use a small 'h', which can be easier to read.

Hex numbers are usually written with not less than two digits. Hence '1' would be written as 01h ; 13 as 0Dh ; 39 as 27h ; etc. The highest two-digit hex number is FFh , which is 255 in decimal. Thus the full content of an 8-bit byte can be given in two hex digits, which is very convenient, particularly as the right digit gives the value of the four rightmost bits, and the left one gives the value of the others.

Convenient or not, there is absolutely no need to learn hex if you don't want to. You already understand decimal, the keyboard already understands decimal, and provided that the two of you can handle the rudiments of binary, then you will have no trouble at all with machine-code programming on the PCW. However it is better to know it than not, and computer literature usually takes hex for granted.

The Memory

The computer's memory is where it stores the information given to it. The memory is arranged like a stack of boxes each of which contains one byte. The boxes are indelibly numbered so that we always know which is which, and the number of each is called its address. For the time being, assume that there are 65536 such boxes, ie. that the computer's memory consists of 65536 bytes and that these each consist of

8 bits. (In fact the '8256' has four lots of 65536 , and the '9512' and '8512' have eight such lots, but we won't be concerned with these additions until later.)

The address of the first 'box' is 0, which is written as 0000h in hex, and that of the last one is 65535 , which is FFFFh. Notice that it takes exactly four hex digits or two bytes to represent the highest address. If there had been even one more address then we would have needed six digits and three bytes to specify addresses unambiguously.

Take care over the distinction that an address points to a single byte of memory, but the value of address is made up of two bytes. Because there is an address No 0 there are 65536 addresses even though the highest one is only No 65535.

It is a peculiarity of the Z80 that when we are writing instructions for it we have to write two-byte numbers with the Low Byte first. This is just a convention adopted by the Zilog Corporation for their own good reasons some ten or twelve years ago, and there are times when it seems quite sensible. Sensible or not, we are stuck with it, and with a bit of practice it is easily remembered.

However this convention applies only when writing for the Z80 , so if you were to write out a list of addresses to show what was stored at each, then you would use normal arithmetical procedure and put the high byte first (to the left).

The table below gives a few addresses in decimal notation, in hex normal arithmetical notation, and in my own notation which shows the two bytes written separately in decimal ready for use by the Z80 (low byte to the left). It is a convention of my own that I write these always in brackets with a comma between using a red biro that I keep for the purpose.

Decimal	Hex	Red-Biro
0	0000	(0,0)
1	0001	(1,0)
10	000A	(10,0)
15	000F	(15,0)
16	0010	(16,0)
255	00FF	(255,0)
256	0100	(0,1)
1000	03E8	(232,3)
2560	0A00	(0,10)
32000	7D00	(0,125)
65535	FFFF	(255,255)

Calculating the two bytes

Starting from an address in decimal, first divide by 256. The value of the High Byte is equal to the result minus any fractional part, and the value of the low byte is given by multiplying the fractional part by 256. These two byte values (which are in decimal) can be converted to hex by a similar treatment of dividing by 16 instead of by 256, and bearing in mind that results over 9 are represented by capital letters not by numerals. The following examples convert the arbitrary address 39452 into its red-biro and hex equivalents:

39452 ÷256	=	154.10938	: the High Byte is 154
0.10938 x 256	=	28	: the Low Byte is 28
154 ÷ 16	=	9.625	: the 1st hex digit is 9
0.625 x 16	=	10	: 2nd hex digit is A
28 ÷ 16	=	1.75	: 3rd hex digit is 1
0.75 x 16	=	12	: last hex digit is C

Hence the red-biro version is (28,154) and the hex version is 9A1Ch. Alternatively, you could calculate the hex version direct from the decimal address by successive divisions by 4096, 256, and 16 on the same lines as shown for the 'red biro'. The significance of these numbers is: 4096 = 16 x 16 x 16 and 256 = 16 x 16. Occasionally you will get a low byte value that is not quite a whole number (although it will always be very nearly a whole number unless you have made a mistake). In such cases round it up or down.

Unless you like calculating you may as well give the task to BASIC when it is available. The following short program accepts an address in decimal and prints out the red-biro version.

```
100    INPUT; "Address ?      ", a
110    b = INT ( a/256 ) :  c = a - b * 256
120    PRINT  TAB (24) ; "(" ; c ; "," ; b ; ")"
130    GOTO  100
```


Calculating a decimal address

You can obtain the decimal address from Red-biro by

Address = (256 x High Byte) + Low Byte.

To convert a hex address to decimal, first re-write any letter digits as decimal numbers, and then multiply them by 4096, 256, 16, and 1, as shown below:

9 = 9	9	x 4096	=	36864
A = 10	10	x 256	=	2560
1 = 1	1	x 16	=	16
C = 12	12	x 1	=	12
		Total	=	39452

Chapter 3

The Z80 processor

Machine Code Instructions

Machine-code instructions are not like the instructions given in BASIC. A machine-code program, which is usually called a 'routine' or a 'sub-routine' (which latter I will abbreviate to "sub-r") consists of a sequence of numbers at consecutive addresses in memory. The program is run by telling the processor which address to start at. It runs through the numbers in turn treating each one as an instruction to do something specific. When BASIC is in place, the start instruction is *call, z*, where 'z' is a variable that has been given the value of the start address. (See page 44 for a program example.)

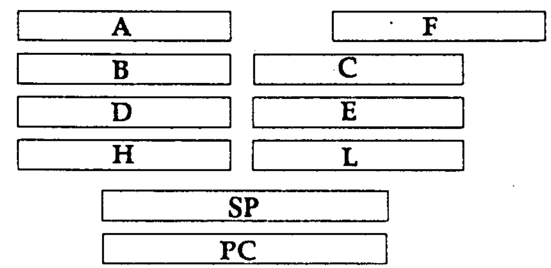
Because an address can hold only a single byte, only the numbers 0 to 255 can be used as m/c instructions, but the total number of them is not 256 but about 800 because some are two bytes long and hence more combinations are possible; but don't despair - you don't need to learn all 800 of them !

Ther e are several large groups in which the instructions are similar to each other; over 100 relate to loading the registers (see below), and about 200 relate to setting, resetting, and checking individual bits. All you need to do is to become familiar with the names of these groups and know what kind of effect they have. They are described in Chapter 4.

The Registers

There are only a few ways in which the Z80 can process data that is still in memory. For most purposes it has to take the data out of memory, put it into one or more of its **registers** and there process it according to the instructions it has been given. The registers are stores inside the Z80 each of which can hold one 8-bit byte, though some can act together as a register-pair for storing 16-bit numbers. The Z80 has 22 registers, though 12 are best left for the use of the PCW for its own housekeeping duties; but don't feel cheated, registers can be used very flexibly, and the remaining 10 will be enough for our requirements.

The registers, which are referred to by their letters, operate as if they were arranged as follows:



A schematic arrangement of the registers

The A register is the most versatile and probably the most used. It is also called the **accumulator.** It is the register in which many of the computations and all of the comparisons are made.

The F register is a special one called **flags.** It gets its name because it consists of a set of six indicators each of which is called a 'flag'. The flags indicate the effects of the last operation. Although the contents of 'flags' can be moved into and out of memory alongside the contents of the accumulator, the two are not a register-pair because they have independent roles.

The next six registers can each act as an independent 8-bit register, or with the register shown beside it as a 16-bit **register-pair,** in which case the one on the left (B, D, or H) takes the High Byte, and the one on the right (C, E, or L) takes the Low Byte. In the last pair the names were chosen to indicate this: H for 'high', and L for 'low'.

Although these six registers are interchangeable for many uses, they also have some specialisations. The HL pair is particularly useful in making additions and subtractions of 16-bit numbers, and also in 'pointing' to addresses. Because of this it is used more than any other pair. The DE pair has the specialised role of pointing to the addresses of strings and to other features required by the CP/M operating system. The HL and DE pairs can also exchange 16-bit numbers, which is very useful.

The B and C registers are a bit of an odd pair out, though B comes in for a special counting function, and C finds an application in specifying which CP/M function is required, and of course BC can hold a 16-bit number as and when required.

The Flags

Without an arrangement of flags computing would be a much more devious process than it is. The Z80 has six (leaving two bits in F unused), but we will be concerned with only the two most used ones. They operate as follows :

The Carry flag

This is abbreviated to C, or to Cy (to avoid confusion with the 'c' register), and is set by any operation that causes an overflow. Thus if the last operation was a subtraction that gave a negative result, or an addition gave a result that was too large for the totalising register(s), then this would set the Carry flag. Comparisons count as subtractions. An arithmetic operation that did not lead to either of these conditions would reset Cy. It can also be set and reset by 'shift' and and 'rotate' operations (see page 38).

The Zero flag

Abbreviated to Z, this is set by an addition, a subtraction, or a comparison that gives a zero result. It is reset by an operation of this type that does not give a zero result.

The two flags are not affected by loading operations, nor by many others of a non-arithmetical kind. They are affected by additions, subtractions, and by number- and bit-wise comparisons. They are affected by incrementing or decrementing 8-bit registers, but not by incrementing or decrementing the 16-bit register-pairs. (Incrementing means 'adding 1 to', and decrementing means 'subtracting 1 from'. There are special Z80 instructions for these actions.)

The Stack

The stack is a small area of memory given over to the Z80 as a 'scratch-pad' on which it records things that it needs to remember but doesn't want to devote valuable registers to. The latest address on the stack is pointed to by a 16-bit register called (what else?), the stack pointer, which is abbreviated to SP. When the machine is switched on CP/M provides a stack that is freely available, and this is the one to use until you are familiar with the ins and outs of stack operations.

Alternatively, you can decide on your own location for the stack if you like. You do this by loading SP with your chosen address, but if your program is not fully self-contained you must make provision to return to the old stack when it has finished. You must also give it enough room to 'grow' as more information is added to it. In short programs only a dozen or so bytes would be enough, in larger ones, to be on the safe side, you might allocate it as many as three dozen, which would be generous.

The stack grows downwards into successively lower addresses, so that the 'top of the stack' is at a lower address than the 'bottom of the stack'. It grows by a pair of bytes each time a 'call' is encountered (the m/c version of GOSUB), but retreats by the same two when the call has been completed. However if you have a set of deeply nested calls (several sub-routines called from within each other) then the stack can grow quite large before being returned to its former size.

Because the stack is also used a sort of fast retrieval storeroom for the contents of register pairs by means of the 'push' and 'pop' instructions, it might be better not to make any changes to the content of SP until you are have had plenty of m/c experience. A wrong address in SP is a tried and trusted way of making programs crash, but left alone the Z80 has the problem sussed.

The Program Counter

The program counter (referred to as 'PC') is the 16-bit register in which the Z80 keeps track of which address it should go to for its next instruction. It automatically updates PC according to whether it is now dealing with a 1-, 2-, 3- or 4-byte instruction, and thus is always able to move straight to the start of the next one when it has finished the last. It also modifies the content of PC when it encounters 'jump' instructions (like GOTO). It isn't possible for a programmer to change the content of PC; which is perhaps as well. The operation of the Stack and the Program Counter are described further in Appendix 6.

Assembly Language

If the Z80 were a person then we could say to it, "Put the value 100 into the A register, then transfer it to address No 12345, and stop". Because it is a microprocessor we actually have to feed it with the stream of bytes;

62 100 50 57 48 201

This gives rise to what we might call a communications gap. The sentence means nothing at all to the Z80, and the row of numbers means precious little to most of the rest of us, but fortunately there is an intermediate language that looks sufficiently like English to be meaningful once you are used to it, and is at the same time an economical and precise way of specifying the actions that we require of the processor. It is called **Assembly Language** because it is the language in which m/c programs are usually first assembled.

Assembly Language is written in abbreviations called **mnemonics**. A mnemonic is a 'reminder', ie. in abbreviated form it reminds you of the thing it represents; hence 'ld' is reminiscent of 'load', 'jr' of 'jump relative' and 'jp nz' of 'jump not zero' etc. The set of all the Z80 mnemonics is called the **Z80 Instruction Set.**

They are the names of the groups of actions that I referred to two pages ago. Once you are familiar with them, m/c is a piece of cake, though I suggest that you don't swot them; the easiest way is to write a few programs, because then your need to find easier methods of doing things will bring new ones to your attention.

M/c versus Assembly Language

Because it seems unclear to some people, it may be appropriate here to emphasise the difference between 'machine-code', which is a series of numbers in memory, and 'assembly language', which is a set of mnemonics. If, like me, you remember all the numbers, then you can insert them yourself directly into memory. This is 'machine-code programming' and is the lowest level of programming that is normally available. Alternatively, and to some folk more conveniently, you can feed a set of mnemonics into an **assembler** (which is a software program), and get it to insert the numbers into memory for you. This is one programming level up from m/c, ie. Assembly Language (often quite wrongly called "Assembler") is a 'higher' language than machine code is, but not much higher. The chapters that follow will help you to use either approach.

Chapter 4

The instruction set

The Z80 Assembly Language instructions are listed with their decimal codes in Appendix 1, but the following brief descriptions will help to explain their effects. If you are unfamiliar with m/c I suggest you read through this chapter to gain some impression of what kinds of instruction are available before moving on to look at the process of programming. No doubt you will return here from time to time for clarifications.

The load instructions

For the sake of clarity I have written each 16-bit number as a single decimal number rather than as two bytes. 'Load' is the instruction to copy a number into a register, into a register pair, or into a memory address. The mnemonic is **ld** followed by an indication of what should be loaded to where. The 'where' comes first and the 'what' second, eg.

```
ld a, 99      load A with the number 99
ld c, 101     load C with the number 101
ld a, h       load A with the content of H
ld b, c       load B with the content of C
```


When something is loaded into a location you don't need to clear the location first; anything in it is automatically obliterated. On the other hand, the location loaded from is left unaffected. Thus the following sequence would leave the registers A and D both containing 100 :

```
ld a, 100     load A with 100
ld d, a       load D with content of A
```

You can also directly load a register-pair with a 16-bit number as in the following three examples, but you can't load from one pair into another pair in one go (do it one register at a time)

```
ld bc, 65535     ld de ,1000     ld hl, 0
```

Loading into Memory

It is not possible to load a number directly into a memory address, but there are several ways of doing it indirectly. The most obvious is to put the number into A and from there transfer it into the chosen address. Thus the stream of bytes I mentioned above would be written in Assembly Language as :

```
ld a, 100         load A with 100
ld (12345), a     load addr 12345 with content of A
```

An alternative route would be to use HL as a pointer to the address into which the number is to be loaded

```
ld hl, 12345      load HL with the number 12345
ld (hl), 100      load the HL addr with 100
```

There are other routes. If you use HL as a pointer then the address pointed to can be loaded directly with a number (as above), or with the content of any register (including either H or L), but if either DE or BC is acting as the pointer, then only the content of A can be loaded to the address :

```
ld b, 100        ld a, 100        ld a, 100
ld hl, 12345     ld de ,12345     ld bc, 12345
ld (hl), b       ld (de), a       ld (bc), a
```

A very useful instruction is one that allows you to copy the two bytes in a register-pair into two consecutive addresses in memory. The following instruction would put the byte in L into address No 1000 and the byte in H into address No 1001 :

ld (1000), hl

Notice that the high byte goes into the higher of the two addresses, which makes more sense of Zilog's byte sequence. A similar instruction is available for both BC and DE. Notice that an address is always indicated by brackets; the instruction *ld 1000, hl* would be meaningless.

Loading from Memory

Bytes can also be copied from memory into registers in methods similar to, but the reverse of, the methods described above. The contents of memory are left unchanged by these operations. In general terms A can be loaded directly from memory, or by using any register-pair as a pointer, but the other registers can be loaded from memory only by means of HL acting as a pointer. To load from Address No 1000 the various instructions would be :

ld a (1000)	*load A from address No 1000*
ld hl 1000	*load HL with the number 1000*
ld e (hl)	*load E from addr pointed to by HL*
ld de 1000	*load DE with the number 1000*
ld a (de)	*load A from addr pointed to by DE*

Two bytes at consecutive addresses can be copied into a pair from memory, as by

ld hl (1000)	*load L from addr No 1000 and* *H from addr No 1001*
ld bc (24000)	*load C from addr No 24000 and* *B from addr No 24001*

Notice that again the address is in brackets. This is shorthand for 'the 16-bit value stored at this address and the address above'. Numbers not in brackets are just numbers. Suppose that address 1000 contains 10, and address 1001 contains 1.

	ld hl, (1000)	will put 266 into HL,
but	*ld hl, 1000*	will put 1000 into HL.

8-bit Additions and Subtractions

The Accumulator is the only register in which 8-bit additions and subtractions can be made. Whatever A contains you can add to it or subtract from it either a number, the content of a register, or the content of the memory address pointed to by HL. The result is always to be found in the Accumulator. You can also add the contents of A to itself thus doubling what was there.

If the result of an addition would be larger than 255 then A overflows thus setting the Carry flag and giving the arithmetic result minus 256. If the result of a subtraction would be negative then the Carry flag is set and the arithmetic result plus 256 is given. Zero results set the Zero flag. Consider the example

ld a, 100		*load A with 100*
ld h, 250		*load H with 250*
sub a, h	*	*subtr content of H from A (sets Cy & resets Z)*
ld c, 10		*load C with 10*
add a, c		*add content of C to A (Cy & Z reset)*

The subtraction sets the carry flag, resets the zero flag, and leaves 106 in A (at *). Then a further 10 is added to A, and, because this does not cause a carry, a borrow, or a zero result, the carry and the zero flags are both reset. At the end of the sequence A would contain 116, and Cy and Z would be reset.

16-bit Additions and Subtractions

The content of BC, DE, or HL, can be added to the content of HL. The content of BC, DE, or HL can be subtracted from the content of HL, but Cy is always included in the subtraction. Instructions for including Cy in the additions are also available, so if Cy happens to be set then an extra 1 is added, but if it is reset then no extra 1 is added. Including the carry flag is used to carry forward the 'carry' or 'borrow' of previous operations into the present one. The mnemonics for the three cases where BC is involved are :

add hl, bc	add the content of BC to HL
adc hl, bc	add BC plus Cy to HL
sbc hl, bc	subtract (BC + Cy) from HL

If you want to make a subtraction from HL without the Carry flag being involved it is necessary to cancel Cy, ie. to make sure it is reset, first. This can be done through a number of instructions, of which *and a* and *or a* have the advantage of leaving the content of A unchanged.

Cy is set if an addition into HL would exceed 65535 , and the arithmetic result minus 65536 is given. If a subtraction from HL would give a negative result then Cy is set and the arithmetic result plus 65536 is given. Zero results set the zero flag.

You can't make direct additions or subtractions of a single register to or from a register-pair, but you could add the content of, say, C to HL , as by:

ld b, 0	*zeroise the high byte of BC*
add hl, bc	*add BC (=C) to HL*

Number Comparisons

Without the ability to compare numbers, computing would hardly be possible. All comparisons are made against the value in A . The mnemonic is cp. For example *cp a, 20* means "subtract 20 from the content of A, and then restore the content of A to its former value". Hence the value in A is left unchanged but the comparison will have had its effect on the flags. If A had contained 20 then the result of the subtraction would have been 0 and the zero flag would have become set. Had the value in A been less than 20 , then the Carry flag would have become set. An absence of these conditions resets the flag concerned; so if A contained any number other than 20 then Z would become reset, and if it contained any number more than 19 then Cy would become reset.

It is possible to compare the value in A with numbers from 0 to 255, with the content of any of the 8-bit registers, or with the content of the memory address pointed to by HL . Obviously no direct comparison with register-pairs is possible. The mnemonic is followed by the subject of the comparison, for example:

cp a, 20   *cp a, c*   *cp a, (hl)*

Bit-wise Comparisons

Three of the logical operations are available for use on the content of A. These are 'AND', 'OR', and 'EXCLUSIVE OR'. The subject of the comparison may be a number, the content of a register, or the content of the memory address pointed to by HL. Suppose the comparison we want is against the number 7 , which in binary is 00000111 and that the content of A happens to be 85 , which in binary is 01010101 .

1. The AND instruction leads to A containing only those bits set that were set in BOTH of the 8-bit groups.

A contains	01010101
7 consists of	00000111
so *and a, 7* leaves	**00000101** in A

AND is useful for 'masking', ie filtering-out particular bits in the accumulator. If you use 'and a,15', for example, the 4 leftmost bits of A will be reset leaving only the 4 rightmost in their original state, but 'and a,240' resets the 4 on the right and preserves the others. Hence, the accumulator could be used to receive two (or more) small numbers from a single memory address, and these then be separated by masking.

2. The **OR** instruction leads to A containing any bit set that was set in EITHER of the 8-bit groups:

A contains	01010101
7 consists of	00000111
so *or a, 7* leaves	**01010111** in A

3. The **EXCLUSIVE OR** instruction leads to A containing any bit set that was set in EITHER ONE, but NOT BOTH of the two 8-bit groups. Hence in the example:

A contains	01010101
7 consists of	00000111
so *xor a, 7* leaves	**01010010** in A

These operations are also useful for their effects on the flags. They always RESET THE CARRY FLAG and the Zero flag is set if the result is zero, but reset otherwise. Obviously a number XORed with itself must always give zero so the instruction *xor a, a* leaves the accumulator empty, sets Z, and resets Cy. Alternatively, both *or a, a* and *and a, a* reset the Carry flag but leave the content of A unaffected. As indicated earlier, they can precede the *adc* or *sbc* operations to cancel Cy.

NOTE ON NOTATION: In operations that *must* involve the A register, it is not usual to refer to A. However, I have written it in so that the structure of the mnemonic is as clear as possible. Thus I have used the form *cp a,20* and *or a,a* etc., whereas the more usual one is *cp 20* and *or a*.

Jump Relative

As BASIC requires the GOTO command, so m/c requires its 'jump' instructions. The first of these is called 'jump relative' because the jump is made a specified number of bytes ahead or behind (ie. relative to) the present address. The mnemonic is jr followed by the jump distance (called the 'displacement'), which is contained in a single byte. Because both forward and backward jumps are required, it is necessary to be able to specify either a positive or a negative number for the displacement, and because a sign bit limits the capacity of a byte to 127, relative jumps can be no larger. As well as the standard instruction, there are four others that order a jump only if certain flag conditions are met:

jr N	*"jump relative"*	*jump N bytes*	
jr c N	*"jump relative carry"*	*if Cy set*	*ditto*
jr nc N	*"jp relative no carry"*	*if Cy not set*	*ditto*
jr z N	*"jump relative zero"*	*if Z set then*	*ditto*
jr nz N	*"jp relative not zero"*	*if Z not set*	*ditto*

A value of N of 128 or more indicates that a backward jump is required (the sign bit has an arithmetic value of 128), the distance of the jump being (256-N). A request for a jump back of 6 bytes on the condition that Z was not set would be written as: *jr nz 250*

The count backwards or forwards is taken from immediately after the address of N, so the first address counted in a jump back is the address of N, and the first one counted in a jump forward is the address after the one containing N. If you know the address jumped from (call it 'f'), and the address jumped to (call it 't'), then the jump distance is

f - t + 1 for backward jumps
t - f - 1 for forward jumps (see diagram on next page)

Addr	Byte
1	24
2	9
3	•
4	•
5	'Y'
6	•
7	•
8	24
9	251
10	•
11	•
12	'X'

A jump forward of 9 bytes to 'X', and a jump back of 5 bytes to 'Y'

The *jr* instructions have the advantage of making the sub-routine 'relocatable', ie. the whole of it could be moved to a different place in memory and the jump instructions would still be accurate because they don't relate to specific addresses. They have the disadvantage of providing only fairly small jumps, though this can be overcome by leap-frogging, ie. arranging that one jump should be to another, thus providing a chain of jumps.

djnz

This is a special version of *jr*. Its full name is 'displacement jump not zero'. It is used exclusively as an economical way of ordering a repetitive loop. The count for the number of repetitions is first put into the B register, the loop procedure is then defined and the instruction *djnz* is added at the end together with the displacement required, which is invariably negative (ie. giving a jump backwards). It is vital of course to keep the *ld b, N* instruction outside the loop or the count will be refreshed at every pass and the program will be stuck in the loop for ever (or until you pull the plug out). That possibility aside, the instruction automatically decrements B and ceases to loop back when the content of B reaches zero. (Page 46 gives an example)

Jump Absolute

The third kind of jump is called 'absolute' because it is made to a specified address. The mnemonic is jp followed by the address in question. As in all such cases, the address is given low byte first. As with jr, there are also four conditional versions, and there is also an unconditional version that allows a jump to the address pointed to by HL. This is useful when you require a jump to an address whose value you don't know when you are writing the program. You arrange for some calculation to put the address into HL and then request the jump to it. Its mnemonic is *jp (hl)*. The six versions are shown on the next page:

jp N N	*"jump"*	*jump to addr given by N N*
jp c N N	*"jump carry"*	*if Cy set ditto*
jp nc N N	*"jump no carry"*	*if Cy not set ditto*
jp z N N	*"jump zero"*	*if Z set ditto*
jp nz N N	*"jump not zero"*	*if Z not set ditto*
jp (hl)	*"jump to (hl)"*	*jump to addr in HL*

Increment and Decrement

The content of an 8-bit register or of the memory address pointed to by HL can be increased or decreased by 1 by the instructions 'inc' and 'dec' respectively. Because the flags are affected according to the result, these instructions are useful in counting operations. If a register is repeatedly decremented and finally reaches zero this sets the Z flag, which will indicate that the count is complete. Additions to and subtractions from registers other than A are not available, so 'inc' and 'dec' are the only ways of changing their contents directly.

The 16-bit register pairs can also be incremented and decremented, but without any effect on the flags. This means that counts of more than 255 can't be made without a bit of subterfuge, but consider:

```
start      ld bc, 10000        Count into BC
start+3    ....                The loop
           ....                procedure.
           dec bc              Decrement the count
            ld a, b              and test
           or a, c               for zero.
            jr nz 'start+3'    Repeat if not zero
           ....                  else continue
```

The count of 10,000 (or any other 16-bit number) is put into BC. After each pass, BC is decremented and the value left in B is put into A. The value left in C is then ORed with it. If either A or C is not zero then the result will not be zero and the program will be told to jump back and go through the loop again. When the count is complete both B and C will contain zero and the jump back will not be made. Loading A from B is necessary because B and C can't be ORed directly.

Call and Ret

The instruction-pair *call* and *ret* are the equivalent of GOSUB and RETURN in BASIC. *Call* is followed by the 2-byte address at which the called sub-routine starts. *Ret* is a 1-byte instruction needing no address. Following a call, the Z80 works through the sub-r until it finds a 'ret' and then returns to the main program where it executes the next instruction. Before starting the 'call' , the processor puts its return address onto the top of the stack, and at the 'ret' it collects the address and returns to it. Obviously it must find the correct address if the return is to be successful, so if the stack has been changed (as by a *push* , for example) then it must be changed back before the sub-r ends. (For more on stack operations see Appendix 6.)

If it doesn't find a 'ret' in the sub-r then the Z80 will go marching on to higher and higher addresses activating whatever it finds there with usually terminal results. It is equally important that there should be no accidental 'ret' in the program not associated with a 'call' ; any such will cause an excursion to a false address and chaos. However, a sub-r may contain any number of *rets* because the first one encountered will be the only one to be activated. The processor never sees any of the programming that follows the 'ret' it responds to.

There are conditional versions of both 'call' and 'ret' , and the condition for the one need not be the same as that for the other. You might have the 'call' conditional on Z being set, and the 'ret' being unconditional, or any other combination.

Sub-routines may be 'nested', ie. a sub-r may be called from within another, and 'recursive', ie. a sub-r may call itself, though in this case the call must be conditional or a closed loop will be formed and the stack will overflow. The mnemonics are :

call N N	*ret*
call c N N	*ret c*
call nc N N	*ret nc*
call z N N	*ret z*
call nz N N	*ret nz*

Block Handling

A pair of instructions that have always impressed me with the beauty of their conception and the convenience of their use are the so called 'block handling instructions'. These allow a block of bytes to be copied to another location in memory. They are:

ldir	ie.	*load, increment, and repeat*
lddr	ie.	*load, decrement, and repeat*

They require all three register-pairs. First you put the address of the DEstination into DE, the address of the source into HL, and the count of Bytes to be moved into BC, then you give the instruction.

In the case of *lddr* the byte pointed to by HL is copied to the address pointed by DE, then all three register pairs are decremented. For *ldir* everything is the same except HL and DE are both incremented. The operation is repeated until the content of BC reaches zero. The original data is left unaffected so you end up with two versions of it unless the new one has partially over-written the old. When the operation is over, HL and DE will both have been adjusted one extra time, ie. they will be pointing to addresses which are just outside the data blocks.

Programming occasionally needs an area of memory to be zeroised or filled with some other invariant byte. For areas of less than 128 bytes a 'djnz' loop is the best solution, but for large ones *ldir* achieves the same effect very neatly. Point HL to the first address and DE to that address +1. Put the required number of bytes into BC, load the HL address with zero (or whatever) and then use *ldir* . The DE address is constantly loaded with what is found in the HL address and in the next iteration HL will point to the previous DE address.

Be careful with *ldir* and *lddr* . If you call one accidentally, or if you put the wrong address into DE, you will have discovered a super way to corrupt your programs.

There are non-repeating versions called *ldd* and *ldi* respectively. A byte is moved and the registers are changed as described above, but the action is not repeated, though you can make them repeat by including them in a loop. This permits other actions to be taken after each byte transfer.

Block Comparisons

There are instructions similar to the above except they involve comparisons instead of copying. In *cpdr* , BC is loaded with the maximum number of comparisons required and HL with the first address. The content of A is then compared with the content of the address

pointed to by HL. If these two are not the same then both BC and HL are decremented, and the procedure repeated until either BC contains zero or a matching comparison is found. In *cpir* the process is the same except HL is incremented every time. The instructions can be used to scan data tables for a particular byte; on return HL points to it and Z is set if a match is found, otherwise Z is reset, HL points to the end of the table and BC contains zero.

The repeating and the non-repeating versions are :

cpir	ie.	*compare, increment and repeat*
cpdr	ie.	*compare, decrement, and repeat*
cpi	ie.	*compare and increment*
cpd	ie.	*compare and decrement*

Push and Pop

Quite frequently there is a need to store the content of a register pair so that the registers can be put to other uses. You can load their contents into two available memory locations but then you will need to make a note of the addresses. Frequently it is more convenient to use the *push* instruction which puts the two bytes onto the top of the stack and decrements SP twice. The bytes can be recovered later by a *pop* which reverses the procedure. In both cases the name of the register pair has to be specified, and a *push* must always be associated with a *pop* and vice versa or the stack will become unbalanced with the usual results.

You can make any number of pushes before popping them if the stack has room enough, but remember that the last pair to be pushed will be the next pair to be popped, ie. they come off the stack in reverse order. You don't have to pop the same registers that you pushed so this gives a convenient way of moving the bytes to a different register-pair. The mnemonics are :

push af	*pop af*
push bc	*pop bc*
push de	*pop de*
push hl	*pop hl*

It is not possible to push or pop a single register so A is always pushed and popped in association with F. (So bear in mind that popping AF may restore an out-of-date set of flags.)

A useful feature of *push* is that the register-pair is left undisturbed. Thus if you push HL three times, you acquire four versions of it; three on the stack and the original.

The Shift Instructions

The contents of any of the 8-bit registers or of the memory address pointed to by HL can be shifted one place to the right or one place to the left. The bit that is pushed out (either the most- or the least-significant bit) is moved into the Carry flag. It is replaced by a zero moving in from the opposite end. These operations are referred to as *srl* and *sla* respectively.

A shift to the right halves the value of the 8 bits concerned (but loses any fraction). A leftward shift would double the value except for the loss of the most significant bit which must somehow be accounted for if a true doubling is to be given. (See the second para. in "Rotations" below.) If you start with bit No 7 reset then this point is already covered because then no set bit is lost (in 8 bits you can double numbers smaller than 128, but not numbers equal to or larger than 128).

There is a second version of the right shift called *sra* . This leaves bit No 7 unchanged but puts the zero into bit No 6. Hence a negative number would not have its sign changed by this operation. The mnemonics and full names of the three shifts are as follows ('R' stands for a permitted location - a register or the addr pointed to by HL) :

sla R	*shift left arithmetical of 'R'*
sra R	*shift right arithmetical of 'R'*
srl R	*shift right logical of 'R'*

The shift instructions play a major role in calculational procedures such as fast multiplication and division, but they are also used whenever bits need to be tested one at a time. The fact that the end bit is moved into Cy at each shift makes it possible to take alternative actions according to whether the bit is set or not. Appendix 1 gives diagrams of these instructions.

The Rotation Instructions

The rotation instructions permit a right- or left-ward movement of the same locations as the shifts and find use in the same applications, but instead of shedding the end bit it is fed back in at the opposite end. There are four such instructions, the first two of which are *rr* and *rl* ; ie. "rotate right" and "rotate left". In the rightward version, Cy is put into bit No 7 and bit No 0 is put into Cy, thus making it a 9-bit rotation in effect. The leftward version is similar except for the direction of movement - Cy finishes in bit No 0 and bit No 7 in Cy.

In a pair of registers we can obtain a true doubling by treating the Low Byte with *sla* followed immediately by *rl* on the High Byte. The first instruction puts bit No 7 into Cy, and the second transfers it from Cy into bit No 8.

The remaining two rotations are *rrc* and *rlc* which mean "rotate right cyclical" and "rotate left cyclical" respectively. They are 8-bit rotations with the displaced bit being reflected in Cy. They allow for sequential bit checking without the loss of bits. Appendix 1 gives diagrams.

'Rotate Digit' & 'DAA'

There are two rotate instructions for use only in BCD calculations. They rotate bits 0 to 3 of A with the bits of the address pointed to by HL four bits at a time. These are :

rld	*rotate left digit*
rrd	*rotate right digit*

'rld' operates through the following sequence; bits 0 to 3 of the HL address are moved to bits 4 to 7 of the HL address, bits 4 to 7 are moved to bits 0 to 3 of A, and bits 0 to 3 of A are moved to bits 0 to 3 of the HL address. 'rrd' operates on the same bits but with a rightward shift in the HL address. BCD stores its numbers in sets of four bits, and these operations allow each set of four bits in the address pointed to by HL to be examined separately in A. 'Decimal Adjust Accumulator' has the mnemonic 'daa' and is used solely in BCD calculations. See Chapter 14.

Exchanges

The contents of HL and DE can be exchanged by *ex hl, de* . This is useful because HL is the only pair that can act as the totaliser in 'add', 'sbc', etc., so in a sequence of arithmetical actions you need to keep preserving its contents whilst freeing it for the next one. It's a pity there is no version involving BC. The instruction *ex sp, (hl)* takes the top two bytes from the stack into HL and replaces them with the two that were in HL. There are other exchange instructions but on the 'PCW' their use is fraught with complication.

Carry Flag Instructions

Reset	There is no instruction to reset Cy but this can be done by *or a*
Compl	The instruction to complement Cy is *ccf*. This sets it if it is reset and vice versa.
Set	To set the flag use *scf*.

Neg, nop and complement

Neg a *nop* are not connected but they go well in a title. 'Neg' means 'negate the contents of the accumulator'. It complements the contents of A and adds one, thus giving the so called 'twos complement' which is equivalent to subtracting from zero. If a subtraction has taken the contents of A below zero, then 'neg' has the same effect as the BASIC command 'ABS'. (See Appendix 3.)

'Nop' doesn't do anything, literally. It stands for 'no operation', and its code is zero. Not the most fruitful command, you might think, but bless the foresight that included it. If the Z80 encounters a sequence of zeroes it happily marches through them without doing anything injurious. Thus a gap between two parts of your program is no problem if it is zeroised. You can also put zeroes in place of bytes that you want to eliminate; this ensures that no addresses will be changed, and that all the 'jr' distances are preserved.

There is an instruction for complementing the contents of the Accumulator : *cpl* . This resets all set bits, and sets all reset bits (thus giving the 'ones complement' of the value in A).

Lots of Bits

There are three operations that can be applied to any bit of the registers, or of the address pointed to by HL. They are :

res N, R   *set N, R*   *bit N, R*

N is the bit No, and R is a register or the memory address. 'Res' means 'reset this bit', and 'set' means 'set this bit'.

res 6,(hl)	*reset bit No 6 of the HL address*
set 3,b	*set bit N 3 of register B*

The 'bit' instruction allows you to test any of the bits to see if it is set or not. The answer is provided by the Zero flag. A zero bit gives Z set. A '1' bit gives Z not set. (In logical parlance the bit and Z are 'in complement'.) Suppose D contains the value 64 which is 01000000 in binary. The following results would be obtained :

bit 7, d	*Z set*	*Z=1*
bit 6, d	*Z reset*	*Z=0*
bit 5, d	*Z set*	*Z=1*
bit 4, d	*Z set*	*Z=1*

etc . . . .

The 'bit' instructions can be used for testing flags that the programmer has devised for himself. You may have decided, for example, to use the 8 bits of a memory address as a block of 8 flags in which sub-routines will record the outcome of their operations. Later routines can then use 'bit' to discover what had occurred in earlier sections of the program, so 'bit' has become a means of communication.

Addressing Modes

Authors more stately than me find use for the following terms:

immediate addressing
direct addressing
indirect addressing
implied addressing
and *relative addressing*

I include them only because they are part of computer mythology. They are not exactly w-language because no simpler alternative phrases exist, it is just that I have not so far ever found a use for them. They suit people who like labels.

Chapter 5

Writing a sub-routine

Without doubt the most convenient way of writing an m/c program is to use an **Assembler**, which is a professionally prepared piece of software (ie. of programming) that may come in a variety of forms.

When using one sort you load the package into the computer before starting to write your own program. When it is in operation, you type in the mnemonics you require in their intended order. With the other sort you type your mnemonics into a separate ASCII text file which later you subject to the assembling action of the Assembler program.

Both sorts have a built-in dictionary that they use to translate the mnemonics into machine code bytes, which can later be placed in memory starting at the address that you selected as the **origin** for your program, and they usually run through your program twice because that is the only way to establish the true addresses for jumps. All professional programmers employ assemblers of one kind or another. If you intend to progress into commercial work then a knowledge of Assembler programming will eventually become essential, though for the hobbyist there are other possibilties.

The main disadvantage of Assembler packages is that they cost money. The very simplest are priced in the region of £50, the most advanced professional versions are several hundred pounds. The more you pay the more you get, but what you get is not necessarily pro rata to the cost, so it is wise to be discriminating before you part with your money. The risk is that if you go for economy then you may soon find that your purchase doesn't cover your requirements (does it handle the whole instruction set, does it give you code that you can move if you need to, and what about linking programs together ?). Alternatively, if you 'go for the best' then you may regret having spent good money on features you don't need and perhaps can't even understand.

Computer folk have not always shown themselves to be brilliant at communicating with actual people, and the worst features of assemblers can be the documentation, which often seems to be based on the assumption that everyone already knows what they do and how to operate them. The PCW Utilities actually include two free assemblers, but they are tricky ones to use and are supplied with no instructions so getting then into operation is not easy, but if you are interested it is worth a try.

To get round some of these problems, before buying one I suggest that you work through the present chapter and get as much practice with programming in the way outlined as you can because this will give you an insight into what m/c is and how it operates and that may help you with your final choice. (My own final choice was to write my own 'Code-Insertion System', which now does everything I want, including printing out the mnemonics with their code bytes and addresses. It gave me many happy hours at the keyboard sorting through the subroutines I needed and it didn't cost me anything.) If you already have an assembler then this chapter may still be helpful in broadening your understanding of m/c in a way that using professional software might not.

Throughout the chapter I have elected to write all bytes in decimal because I can be sure that everyone will understand that, though not everyone knows hex, and also because decimal is easier to input through the keyboard. In case you think that decimal is somehow 'wrong' or 'inappropriate' for computer use, then bear in mind that the computer has no truck with hexadecimal either. Its own language is binary and it is binary that finally whispers through the printed circuits. What we use to produce the whispering is best decided by convenience.

A BASIC program to insert m/c

Load BASIC into the machine and then type in the following short program. When you have checked it over, SAVE it under some short name such as "mc" (short to avoid unnecessary key-jabbing).

```
100   CLEAR, 49999
110   FOR n = 50000 TO 50100: POKE(n),0: NEXT
120   RESTORE
130   FOR n = 0 TO 500: READ k
140   IF k = 99 THEN STOP
150   POKE(50000+n),k: NEXT

200   DATA 0,0,0
490   DATA 201, 99

500   z = 50000: CALL z: STOP

1000  FOR n = 50000 TO 65535
1010  PRINT n; TAB(12); PEEK(n)
1020  a$ = INKEY$: IF a$ = "" THEN GOTO 1020
1030  NEXT
```

What does it do ?

Line 100 Restricts BASIC to address 49,999 and below, thus freeing address 50,000 and above for our m/c use. (Up to 62980 . Above that is reserved for CP/M .) Don't forget the comma.

Line 110 Zeroises the first 101 addresses, thus erasing all traces of previous programs.

Line 120 restores the data pointer to the start of data.

Line 130 allows for the reading of up to 501 data bytes.

Line 140 stops the READ when the last byte '99' is found; the 99 is a marker to indicate that the end of data has been reached.

Line 150 pokes each byte into the next address in sequence starting from 50,000.

Lines 200 to 490 are for DATA lines into which we will put the bytes of our m/c programs ready for insertion into memory.

Line 500 runs the m/c program through the instruction "CALL,z", 'z' having been set to 50,000. This directs BASIC to hand operations over to the m/c program that it will find at address 50,000. After it has run, the m/c program must arrange to make a return to BASIC when its task is complete

Using the program

"Run 1000" allows inspection of the bytes that are in place from 50,000 upwards. Each is shown with its address. Press any key to display additional bytes, and 'STOP' to exit the list.

If you now input "run", you will get the report 'Break in 140. Ok' indicating that the BASIC program has run through, found the '99' byte and then the 'STOP' command.

If you input "Run 1000" followed by an extended key press then a list of addresses starting at 50,000 will be given on the screen, each associated with a zero except for 50,003 which will have a '201' beside it. This indicates that the three zeroes from Line 200 have been put into memory followed by the '201' from Line 490. Press 'STOP' to exit the list.

If you input "run 500" at this stage you will get 'Break in 500. Ok' showing that the m/c program at address 50,000 has been run and the 'STOP' in Line 500 encountered. In fact the 'm/c program' consists of only three 'nop's and the byte for 'ret', which is 201. Hence the machine simply went to the routine and returned from it without doing anything while it was there. However this makes the point that a 'ret' is essential if you are to terminate the m/c program and successfully regain control by a return to BASIC.

In this chapter I have written out the bytes that need to be put into the DATA lines. When you use the insertion program for yourself you will look up the bytes from Appendix 1 and write the DATA lines accordingly.

A mini program

Type the following line and add it to the listing, then SAVE the program under a slightly different name such as "mcl". (This is a way of saving a range of different programs without changing the fundamental one that you can LOAD whenever you need it, though with important programs you will probably want to call them something more distinctive.)

200 data 6,7, 62,10, 33,100,195, 119, 35, 60, 16,251

After saving the modified program enter "run". Then "run 1000" followed by a keypress to reveal up to address 50028 or there abouts. The list of addresses and bytes will now show the above sequence of numbers followed by the '201' from Line 490, with zeroes thereafter. The numbers were inserted into memory by Lines 130 and 150.

If you now enter "STOP", "run 500", followed by "run 1000", you will find additional numbers in memory starting with '10' at 50020 rising to '16' at 50026. These numbers were inserted by the machine code program whose bytes were derived from the following mnemonics :

ld b, 7	*6 7*	*Put the count 7 in B*
ld a, 10	*62 10*	*Put 10 into A*
ld hl 50020	*33 100 195*	*Put 50020 into HL*
ld (hl) a	*119*	*ld the HL addr from A*
inc hl	*35*	*Increment HL*
inc a	*60*	*Increment A*
djnz -5	*16 251*	*Jump back 5 bytes if count not zero*
ret	*201*	*Else return [to BASIC]*

The mini-program illustrates the 'djnz' instruction and 'djnz' is the last but one mnemonic. It starts by putting an arbitrary count of 7 into B (which is always the count register for 'djnz'), though any count up to 255 could have been chosen. The arbitrary number 10 is then loaded into the Accumulator, and HL loaded with the start address (I chose 50020 because it is close to the program bytes and therefore convenient to inspect).

The next five bytes now form a loop that is to be repeated B times. This loop puts the value in A into the address pointed to by HL and then increments both A and HL before looping again. . Hence the sequence of numbers 10, 11, 12, ... up to 16 will be inserted into consecutive addresses starting at 50020.

Now change the end of Line 140 from "..THEN STOP" to "..THEN

GOTO 500", and replace the '10' in Line 200 with '252'. The first change cuts out some key pressing by running the m/c program straight after loading the bytes into memory so you no longer need to enter "run 500". SAVE then enter "run".

"Run 1000" should now give a sequence at 50020 that reads

252, 253, 254, 255, 0, 1, 2

Notice that the increments to A increased it to the maximum value of 255 after which it zeroised and continued the count from there. This is much like a mileometer which, after showing its maximum value of all 9's, starts again at zero. All the registers act like this, and you get the reverse sequence if you decrement them.

To obtain the size of the jump back, the first byte to count is the '251'. Then count back to and including the first byte that you want to loop from. I draw arrows for the jumps on the listings of my programs so the logic of the routine is clear and so the impact of any changes can be seen at a glance.

Had this been a forward jump then the first byte to count would have been the '201' and the count would have been up to but NOT including the byte you want to resume operating from. For backward jumps you subtract the jump size from 256 as indicated in the description of the instruction on page 32.

Testing the flags

EDIT line 200 by erasing the last two numbers, then add line 210 below. The program will then LIST as:

200 DATA 6,7, 62,252, 33,100,195, 119, 35, 60
210 DATA 200, 16,250

When you have SAVEd and RUN this, "run 1000" should reveal that addresses 50025 and 50026 contain '0', not '1' and '2' as they did previously. This is because the mini program is now as indicated on the next page. In this the instruction *ret z* checks the zero flag during each pass of the loop. In the first four passes it finds Z not set so the routine proceeds as before (except that the jump back is now 6 bytes not 5 due to the presence of the '200').

```
ld b, 7        6 7          Put the count 7 in B
ld a, 252      62 252       Put 252 into A
ld hl 50020    33 100 195   Put 50020 into HL
ld (hl) a      119          ld HL addr from A
inc hl         35           Increment HL
inc a          60           Increment A
ret z          200          Return [to BASIC] if Z set
djnz -6        16 250       Else jump back 6 if count not zero
ret            201          Return [to BASIC].
```

The instruction *ret z* checks the Zero flag during each pass of the loop. In the first four passes it finds Z not set so the routine proceeds as before (except that the jump back is now 6 bytes not 5 due to the presence of the '200'). However, during the 5th pass A is zeroised from 255 and this sets the Zero flag. On finding Z set *ret z* orders an immediate return to BASIC so that no further values of A get put into memory.

Now EDIT line 210 to read:

```
210    DATA  214,100, 216, 16,248
```

The mnemonics for the program are now :

```
ld b, 7        6 7          Put count 7 into B
ld a, 252      62 252       Put 252 into A
ld hl 50020    33 100 195   Put 50020 into HL
ld (hl) a      119          ld HL addr from A
inc hl         35           Increment HL
inc a          60           Increment A
sub a 100      214 100      Subtract 100 from A
ret c          216          Return [to BASIC] if Cy set
djnz -8        16 248       Jump back 8 bytes if count not zero
ret            201          Else ret [to BASIC]
```

When the program is now SAVEd and run, inspection shows that the sequence at 50020 and above is *252, 153, 54, 0, 0, 0* etc.

During each loop, 100 is subtracted from the value in A and the Carry flag is checked by 'ret c'. In the first three loops Cy is found not set so the program continues. In the 4th loop the subtraction takes A below zero thus setting Cy so 'ret c' orders an immediate return to BASIC and no further numbers are inserted into memory. Notice that the jump size is now -8 because of the extra bytes.

Multiple Choices

The following development of the mini program illustrates how different actions can be taken in different circumstances. I want to add 100 to the value in A at each pass through the loop, starting with 7 ; if the result is less than 100 or more than 199 then I want that value to appear in memory, but if it is in the range 100 to 199 then I want '0' to appear in memory. The program is longer and overlaps 50020 so the HL address has been moved up to 50029 so that the program doesn't overwrite itself (ie. it doesn't insert inappropriate bytes inside the program) and cause a crash.

```
ld b, 20       6 20         Put count 20 into B
ld a, 7.       62 7         Put 7 into A
ld hl 50029    33 09 195    Put 50029 into HL
inc hl         35           Increent HL
add a, 100     198 100      Add 100 to A
cp a, 100      254 100      Compare A with 100
jr c 9         56 9         If A < 100 jump on
cp a, 200      254 200      Compare A with 200
jr nc 6        48 6         If A ≠ 200 jump on
ld (hl) 77     54 0         Put 0 into HL addr
djnz -15       16 241       Jump -15 bytes if count not zero
jr 3           24 3         Else jump 3 bytes
ld (hl) a      119          ld HL addr from A
djnz -20       16 236       Jump -20 bytes if count not zero
ret            201          Else ret [to BASIC]
```

The DATA lines for the program should be changed to contain the bytes shown in the above listing. When it is run the numbers inserted into 50030 and above are as follows : 0, 207, 51, 0, 251, 95, 0, 39, 0, 239, etc. There are no values between 100 and 200, which was the intention. If it were for actual use the program could be made much more elegant, but this inelegant version is easier to follow (which is usually true).

Strategy of the sub-r

Call the content of A ; (a). The strategy of the routine is that during each loop (a) is compared with 100 and if it is found to be less than 100 then Cy will become set and a jump made to the 'ld (hl), a' instruction. If (a) > 99 then a second comparison is made, this time against 200. If this does NOT set Cy (because (a) >199) then again a jump is made to 'ld (hl), a'. For all other values the program goes to 'ld (hl) 0'. Whichever route is followed, when the count reaches zero then the program ends.

Instructions and bytes

It is as well to bear in mind that it is the bytes in memory that cause an m/c program to have its effects. The processor reacts only to bytes and produces only bytes. The mnemonics are not instructions, though for convenience we speak of them as such. They are no more than what their name suggests; a reminder and a summary of the instructions that their associated bytes bring into action.

Most m/c instructions require 1 or 2 bytes. Some require 3, and a few require 4. None requires more than 4. In all 1-byte instructions the byte is called an **opcode** (short for operation code) because it is a code that leads to the Z80 performing the specified operation. In many 2-byte instructions the two bytes make up the opcode (ie. it takes two bytes to specify the operation concerned), though some consist of an opcode and a defined byte.

A defined byte (abbreviated to DEFB) is one that you specify the value of yourself to suit your own requirements. A defined word (DEFW) is two associated bytes (ie. a High and a Low Byte) that are defined by the programmer, such as an address. Most 3-byte instructions consist of an opcode and a DEFW ; *ld hl N N* for example. All the 4-byte instructions that we will be using have two bytes as opcode and two as a DEFW. (For the sake of completeness: a DEFM is a defined message, and a DEFS is a defined string. Both consist of a string of ASCII codes.)

A slightly cut down set of Z80 mnemonics is listed in Appendix 1 with their decimal opcodes and the number of DEFBs required. These should be used when compiling for the BASIC insertion program. DEFBs are shown as an 'N', and DEFWs by 'N N'.

You can of course use any of the instructions in the set but some that I haven't listed are tricky because of the way in which the CP/M operating system has a prior claim on them. It is not a good idea to use the index registers nor the alternate registers, and the instruction 'halt' is not to be used. CP/M has its own way of sorting out the interrupts. I have restricted programming to the registers and mnemonics described in the text and have been little inconvenienced by this. From now on I suggest you make a point of using Appendix 1 to compile

and run as many test routines as you can. In the course of this, problems will inevitably occur, but it is in finding the causes that you will increase your skill as an m/c programmer, and having a project that you really want to get to grips with is worth any number of 'five-finger-excercises'.

Using an Assembler

If you are intending to buy an Assembler then the following brief description may be of help. They are not all alike so full details are not possible, though the basis of their use is fairly standard.

The Assembler version of the mini-routine might look something like the following (but with the addresses and a byte-count down the left side). First you specify where you want your program to be placed in memory by an 'org' (origin) instruction, and then define the value constants you require, as by *fred equ 99* or *99 = fred* . All future uses of *fred* will then imply this value. The words 'start', 'loop', and 'end' are labels that mark places within the listing, to which you specify your jumps without having to count the jump distances. You can also insert notes down the right side (like REM statements) to explain features of the program.

```
org         equ  C350h
count       equ  7
first       eq   10
addr        equ  C364h

start       ld b      count
            ld a      first
            ld hl     addr
loop        ld (hl) a
            inc hl
            inc a
            djnz      loop
end         ret
```

Chapter 6

Practical Programming

The cardinal and most worthy rule of computing is that programming always starts with an algorithm and the drawing of a flow diagram. An algorithm is a 'logical route' by means of which the program is to achieve its objectives, and the flow diagram shows the sequence of the operations that it will follow. The better class of diagram employs the standard symbols that have been agreed for this purpose; choices are put into diamond-shaped boxes, operations into square ones, etc.

Short routines of the type we have been discussing hardly need such thorough treatment, though it is obviously a good idea to start as you intend to go on, and practice with a desirable technique is always itself desirable, though in all conscience I feel I should say no more about algorithms because I use them very rarely and tend to have more trouble with getting them right than I do with assembling a program in cold blood, but don't be put off by me.

My approach is that if I can conceptualise a sub-routine quite clearly then I get straight on with assembling it, and only if I am unable to grasp its logical niceties with exactitude do I get down to planning it out in this formal way. This leads me into the ultimate computing sin of drawing-up a flow diagram only after I have made a bindles of the programming - and they drum you out of the Worshipful Society of Computer Studies for far smaller crimes than that; though not being a member affords some protection of course.

A generally agreed approach that I do regard with enthusiasm is that of splitting up a large program (and even a not so large one) into well defined tasks and making each of these into sub-routine that is called from a 'central' or 'executive' routine. This is in any case necessary if any of the sub-r's need to be called from more than one location, perhaps even from within each other. The ultimate development of this would be that the executive routine consisted of nothing but a sequence of 'call' commands terminating with 'jr start', though I don't recommend aiming for this unless there is a well thought out justification for it.

If I have any regard for flow diagrams it is for their value for sorting out the executive routine, though I think they need some kind of perking up so that you can more easily follow the 'jumps' and the 'calls', and discriminate between them.

Library Sub-routines

Each program gives rise to its own particular sub-routines, but there are some that can be used more or less unchanged in program after program. Good examples would be the arithmetical procedures such as multiplication and division, which are likely to be needed in a wide variety of applications. There is no point in working them out afresh for each occasion so once this has been done they can be stored for future use. In this way a programmer builds up a library of procedures that suit his needs, and using ones devised by other people is seen as sensible not plagiaristic provided you give credit when it is due.

I started my library on disc, each sub-r being stored separately under its own name, but I soon found this to be cumbersome and I abandoned the idea. I now keep the most useful ones alongside my code-insertion routine so they are all inserted into memory at the same time, it being much easier to erase unwanted ones than to go searching for those that are needed.

I also find it essential to keep a written version of everything in a loose-leaf binder so that the details can be looked up and the need for modifications seen at a glance. However accessible information on the screen may be, there is something a bit more readable about paperwork.

Organising the Memory

My first job when I start a new venture is to allocate regions of memory to the various duties that will need it. These are usually:

Variables	: data, especially the results of calculations
Major strings	: long phrases and pages that will be printed or screened
Minor strings	: short phrases, words, and print instructions
Major routines	: usually those user-selected from a menu
Minor routines	: usually those called by major routines

You may also require storage areas for data of a non-variable kind, and for temporary records that will eventually be filed.

It is better to assess the memory requirements of these various duties fairly generously at the start because if they over-run their allotted areas then you will have to move everything to make room. Without doubt the room required by the strings will be far more than you expected, so allocate them plenty of space. I tend to be mean and try to shoe-horn everything into the least memory so none is wasted, but don't be tempted by this. If it turns out that you were over-generous then you can move everything closer together when you have finished if you want to, and that gives a much nicer feeling than leaving out features that should be included just because you don't want the labour of shifting something so you can fit them in. Moving things always means changing lots of addresses and that can be laborious even if you have been shrewd enough to access everything through jump-tables. (See Appendix 8.)

All working programs put information of some kind into memory through the actions of their sub-routines, and as with everything in computing there are two irreconcilable views on how this should be organised. One view is that data should be placed close to the sub-r that produces it so that the relationship between them can be seen more easily. The other is to keep all data in a single reserved data area regardless of its source. Having tried both I tend to prefer the second on the grounds that sub-routines frequently use each other's output and having the data all in one place makes it easier to keep track of. Though, just to be contrary, I sometimes do it the other way because in that particular application it seems to meet needs better.

The so called variables make up this data. Even if there is no program need for a sub-r to put information into memory (it could pass it on in the registers), there is still an advantage in using memory storage, even if only temporarily, because bytes inserted into memory can be inspected at leisure and the accuracy of the routine producing them assessed, but data kept in the registers is immediately overwritten by later activity so you can't interrupt operations to discover it.

Having apportioned memory, I then rule a sheet of A4 with columns ready to receive the addresses and the names of the variables that I will generate during the programming. I make a point of providing them with a whole page, which is invariably more than enough. (A computer 'page' is any block of 256 addresses that starts with a Low Byte of zero.) I also give each variable the address(es) it needs solely for its own use, ie. I rarely share an address between two variables even if it seems that there could be no clash between them. You never know how the program will develop later, and there will be confusions enough without generating uncertainty about which number it is that you are looking at.

I also try to allocate an easily remembered High Byte to the variables addresses, and put the important ones in first. It is surprising how this aids memory, and that, together with only a single record sheet to inspect, cuts out much laborious thumbing through endless sheets of program details to remind yourself of the address at which something crucial is stored. Attention to seemingly trivial points like these can quintuple the ease and pleasure of writing programs, and ignoring them can have the same effect on the late-at-night-exasperation index.

Programs vary very widely in their need for strings. Calculational routines may need little more than the ability to report the results, but in interactive ones, where there is a lot of correspondence with the user, strings may occupy more memory than any other feature, and you can be sure that ALL your programs will need strings for error-reporting. So certain is this need that your error-handling routine and the strings associated with it are excellent candidates for your library. (At least the user may have to be notified that a disc is full or that he has pressed the wrong key; see Chapter 13.)

The PCW's memory plan

The PCW's operating system is CP/M which requires two sections of memory for its own use. The first is page 0 (0000h to 00FFh, ie. 0 to 255 inclusive) so this is not available to the programmer.

The highest memory address plus 1 that is available is recorded at

0006/7h, and inspection normally yields F606h or (6,246). This address and those higher are used by CP/M and hence are never available. (The presence of an RSX program will be recorded by a change in this stored address.) In addition, memory down to about (128,242) is used during program loading so you cannot load a program that would extend above this, but you can use this region once the program is in. When Mallard BASIC is in place it occupies from 0100h to 7A96h (256 to 31382), not including any BASIC lines that you may have programmed in. The CP/M **stack** grows downwards from F600h [ie from (0,246)]

The memory areas available for program use can therefore be summarised as indicated below:

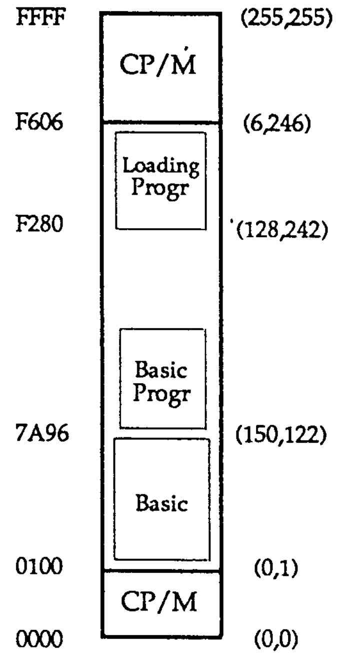
BASIC in place	from	46080	B400h	(0,180)
	to	61440	F000h	(0,240)
BASIC not in place	from	256	0100h	(0,1)
	to	61440	F000h	(0,240)

The lower limit with BASIC in place will naturally depend on the size of the BASIC program. You can have it calculated for you by entering the BASIC command : *print himem - fre("")*

HIMEM is either the address at the top of the TPA (see the next paragraph) or the address in the last CLEAR instruction if one has been used. (The definition of HIMEM given in the manuals is garbled.) *fre(" ")* gives the number of bytes of free space from the top of the BASIC program up to HIMEM. Hence subtracting one from the other gives the address at the top of BASIC. My upper limits are cautious but they still make 15k available when BASIC is in, and nearly 60k available when it is not. (See diagram opposite.)

The operation of CP/M

Page 0 (ie. 0000h to 00FFh) is used by CP/M for the Z80 restart areas it contains and also for storage of its own system variables. The area above this up to F605h is called the TPA, which stands for 'Transient Program Area'. This is the area in which all user programs are placed (including BASIC and the ones you write). The area above the TPA, ie from F606h up to FFFFh, is occupied by the main CP/M systems which are referred to as BIOS and BDOS. These stand for **Basic Input/Output System,** and **Basic Disc Operating System,** (but are not related to the language BASIC.)



The Memory Plan

Using BDOS

Most of our m/c contact with CP/M will be through BDOS, which contains a large number of functions (sub-routines) that allow the PCW to work as it does, and which are available for our use once we've sorted them out. You select the one you want by putting its BDOS function number into the C register, and then you call the address 0005h; ie (5,0). At the same time DE is usually loaded with additional information that BDOS may require such as an address at which to find something. (Using BDOS is described in Chapter 7.)

Program speed

Each of the Z80 instructions requires a specific amount of time for its completion. This time is measured in 'T-states' or 'clock cycles'. The length of a T-state is determined by the speed at which the computer's internal clock is set to run (the 'clock' is an oscillating crystal), which in the case of the PCW is 4 million pulses per second, thus setting the length of a T-state to 0.25 micro-seconds. So short a time may seem too little in which to achieve anything, but as micro-processors measure their internal events in nano-seconds (thousandths of a micro-second), then even a fraction of a micro-second seems like a lazy morning to it. Appendix 2 tabulates the clock cycles for the major m/c instructions.

As with BASIC, alternative m/c programming routes are almost always available and it is usual for inspection of a completed program to show that savings can be made in its run-time. This can be important in routines that involve a large number of re-iterations ; if a single calculation takes a thousandth of a second (a long time in m/c terms), then saving half of this will not be noticeable to the operator, but if the application is to repeat the sequence a million times then the saving would cut the run-time from nearly 17 minutes to only 8.

In addition to these practical considerations, programmers generally take pride in producing the most elegant program. 'Elegance' is not so easy to define, but it is something along the lines of 'style' in design work, though its most noticeable parameters are those of compactness (economy of the use of memory space) and speed. However, the pursuit of speed is best carried out after the program has been shown to run properly. Effectiveness outranks elegance by several battalions, and as I once heard it expressed; "However ashamed you may be of the engine, you can take comfort from the fact that drivers never look under the bonnet."

It is often counter-productive to opt for methods simply because they are fast. *Push de* works nearly twice as fast as *ld (Addr), de* but recording the content of DE for future inspection may be worth any number of saved micro-seconds. And if the sequence onto and off the stack turns out to be inconvenient, then you may spend more time in revamping it than has been saved.

Unless you are programming something very special indeed then you have my personal guarantee that your original un-cleaned-up as-written version will run quite fast enough. The search for speed is a bit like insisting that a hi-fi should have linear responses in the ultra-sonic range as well.

My apocryphal story about it concerns the five minutes I once spent making sure that a sub-r was working at absolutely peak revs, only to notice on completion that it was a procedure that arranged for the program to stop to await a key-press !

Whilst your own m/c sequences will be fast enough in all normal meanings of the word, CP/M is a highly complex set of interacting sub-routines and calls to it may invoke thousands of unseen operations. Naturally these take time. The print instructions are particularly involved and economics made with them will be noticeable. It is much quicker, for example, to move to a print-position in a single bound than to crab across the screen to it one column at a time.

Outputting text through the printer seems fast in typing terms but it takes infinitely longer to print a character than it does to transfer one to the printer buffer, so the processor is kicking its heels during most of the printing operation. If instead of printing a long piece of text all in one go you can feed it in chunks of say 256 bytes at a time, then you may be able to process other batches of work while the physical printing is taking place.

Care with program writing

If you are to use an assembler then its paperwork should give details of where its own routines reside in memory and where you can order your own to be written. It will also store the named variables to suit itself so you won't need to define a separate variables area in most cases.

If you are to use BASIC in the way described earlier then you can isolate your programming area by the *Clear, Addr* command. Naturally this address must be above the highest address that your BASIC program has need of or the two will try to overwrite each other. Although this is such a simple principle, you will suffer a lot of frustration from not attending to it properly. Even if the CLEAR command is in the program it must be RUN to have any effect, and if you add to the BASIC program then you may need to CLEAR to a higher address. You can find out how much room is available above the BASIC program by entering: *print fre (" ")*

Whatever the means of writing it, you must ensure that in operation the m/c program does not insert anything into the restricted areas; those occupied either by CP/M, by the Assembler, or by the BASIC insertion program. It should also not insert bytes into itself unless it is of the so called 'self-modifying' type. If this happens (because a sub-routine has miscalculated an address, say) then these programs will no longer be reliable and at best the system will operate unpredictably. It might even corrupt your discs, so release them when there is any risk.

Friendly Advice
(based on lots of personal experience)

I have to admit to being not a little ashamed of some of my wn reactions to inexplicable errors. It is the easiest thing in the world to curse the machine, its makers, Zilog, the author, Caxton, the cat, and the government for conspiring against you when some detail will just NOT come right. In my case it was always me that had made a booboo, and in your case it will be you. The number of BF errors that you will make will astonish you, and I still haven't found the answer to the pitfall of reading into a listing or into a piece of text the thing that I expect to see there.

If you give the Z80 a good set of input bytes then it will respond unfailingly with a good set of output bytes, but if you make a mistake it has no way of knowing that you intended something else and will always assume that you are as infallible as itself. Checking over your m/c programs before you use them is the only way of revealing errors in good time. After all you are speaking directly to the heart of the computer, and, having bypassed everyone else's programming, there are no friendly error messages available, and no-one else's housekeeping to take care of you. 100% accuracy with your input will therefore be just about enough.

A usual effect of a program error is 'lock-up'; ie. the computer no longer responds to the keyboard and your only option is to pull the plug out and start again. Even if there is no lock-up, if something totally unexpected occurs (the screen may become sprinkled with gibberish, say), then you should restart even if there seems no need because you will have no idea what other mischief has been done. It is better to stop, clear the computer out and reload, rather than soldier on with unreliable material that may have been corrupted in ways that you can't easily detect.

For this reason it is essential to put every m/c program onto disc before trying it, otherwise you could lose several hours of programming effort when the program crashes. Even in cases where you have made only a slight modifications, record the program again before using it. Otherwise you may lose your modifications and not be able to remember what they were.

With larger programs I make a rule to have at least one unused 'pure' copy on disc - one that I know has not been run and therefore can't have been corrupted by hidden errors, but there is obviously a limit to the number of back-ups you can keep, and in this application "the more the merrier" is NOT true. If you keep too many taken at different stages in the development of a program then you'll forget in what ways they differ and be worse off than if you'd kept only one of whose history you are certain. At the time, keeping written records seems tiresome, but if a problem arises a week later you'll be glad you did. The best policy is to devise a system of your own and stick to it. Professionals frequently use a 3-generation "Grandfather, Father, Son" system. 'Son' is the latest version, and a 'grandfather' is discarded when each 'son' is born.

Chapter 7

Screen printing

The character set

The complete set of screen printable characters and their ASCII codes is given in the PCW manual on pages 113 to 118 [547 to 554]. Those with codes larger than 31 can be printed directly by the methods described below. By convention ASCII codes smaller than 32 are reserved for use as control-codes and so are not available for direct printing, but indirect methods are available (see Chapter 9). Also by convention the word "print" means 'display characters on the screen', as opposed to "list" which means 'print onto paper'. When printing or listing, CP/M will ignore a code that it can't interpret.

The BDOS functions

In this chapter we will start to use the BDOS functions, the most interesting of which are summarised in Appendix 4. The first and simplest is BDOS No 0, which has the name "System Reset". Its effect is to clear out any traces of previous operations, and reboot the system (see page 126). It could be brought into action from m/c as follows :

```
ld c 0        14 0        Load C with fnc No
call BDOS     205 5 0     and call it.
     continue . . . .
```

BDOS is always used by calling 0005h after loading C with the required function number. For many functions it is necessary also to load DE, usually with an address. Frequently BDOS reports back with information that you will find in A or in HL or in both, though in the case of function No 0 no such report is made, and no DE address is required.

Function No 12 has the name "Return Version Number". ('Return' means "report back with".) After calling this, H will be found to contain zero with the CP/M version number in L. For version number 2.15, L will contain 2Fh, and for version 3.0 it will contain 30h, etc. Neither of these two functions is of much practical interest, but they illustrate the approach.

Keyboard Input

In contrast, BDOS function No 1, called "Console Input", is very interesting. It makes the program await a key-press (the 'console' is the keyboard) and then puts the ASCII of the pressed key into A. If it is a printable character it is also printed onto the screen at the currently established print position (see pages 70 & 71). Some of the control codes such as TAB are also echoed on the screen, though others are not. To test the function, insert the following byte sequence and run it:

```
ld c 1          14 1          Await a keypress
call BDOS       205 5 0        then put its
ld (50020)a     50 100 195    ASCII into 50020.
ret             201           And finish.
```

This little sub-r awaits a key-press and when one has been made it loads the content of A into 50020. After each RUN, press a key and then use *PRINT PEEK (50020)* to observe the ASCII code of the key you pressed.

In some applications it may suit your purpose to ignore the value in A and use the function merely to halt the program whilst the user reads a message before pressing a key to continue.

Alternatively, by testing the value returned in A, the function can allow the user to pick alternative courses of action, or it can prevent access to a program unless a correct key sequence is typed in. The following sub-routine causes a jump to 'Program 1' if "y" (for 'yes') is pressed or to 'Program 2' if "n" (for 'no') is pressed, and

prevents further action if no key or any other key is pressed.

```
ld c 1          14 1          Function No into C
call BDOS       205 5 0        & bring into action.
cp "y"          254 121       If ASCII is 121 (y)
jp z Prog1      202 P1 P1      then jp to Prog1.
cp "n"          254 110       If ASCII is 110 (n)
jp z Prog2      202 P2 P2      then jp to Prog2.
jr -17          24 239        Else repeat.
        continue ....
```

Using BDOS corrupts (changes) the contents of virtually all the registers so you can't be sure what is left in C after such use and it is necessary to jump back to 'ld c 1', not just to 'call BDOS'. If you need to preserve the contents of a register or of a register-pair whilst BDOS is being used, then 'push' it before calling BDOS and 'pop' it afterwards.

'Numacc'

Function 1 can be used in interpreting numbers typed in at the keyboard. The first requirement is to reject all unacceptable keypresses. The following sub-r does this so we will call it 'Numacc'.

```
Start:
    ld c 1          14 1          Await a
    call BDOS       205 5 0        key press.

    cp 48           254 48        If the ASCII < 48 then
    jr c 6          56 6           jump to repeat.
    cp 58           254 58        If ASCII > 57 then
    jr nc 2         48 2           jump to repeat.
    or a            183           Else reset Cy,
    ret             201            and leave the sub-r.
Repeat:
    ld e 8          30 8          Put 'backspace' in E
    ld c 2          14 2           and print it
    call BDOS       205 5 0        ( see page 64 ).
    jr -24          24 232        Jump back to Start.
```

The sub-r can be extended to make it accept other useful keypresses such as decimal-point, Return, Exit, etc. and to set and reset the flags in a way that will make it clear which of these, if any, has been used.

For example, if it returns Cy set only when Return is pressed, and Z set only when Exit is pressed, then testing the state of the flags will indicate whether the user has happily completed his entry or whether he abandoned it and the input should be disregarded.

Other key-press functions

If you want the pause provided by function No 1 and the ASCII of the pressed key, but you don't want a character to be printed on the screen, then follow function No 1 by a sub-r to print 'backspace' then 'space'. If you merely want to record the pressing of any key without recording the ASCII, then use function No 11 ("Get Console Status"). This puts zero into A if no key has been pressed, and 1 into A if any key has been pressed. If you follow use of the function by 'or a' this will give Z set if A contains zero, so Z set means 'no key pressed'.

More complex situations can be dealt with. Suppose you want no pause in the program if no key has been pressed, but you want to know which key it is if one has been. This can be dealt with as by :

```
ld c 11       14 11        If no
call BDOS     205 5 0       keypress
or a          183           then
jr z PROG     40 N          continue.
ld c 1        14 1         Else call
call BDOS     205 5 0       function No 1.
      continue . . .
```

If there is no keypress then the program continues through the 'jr z', but if there is one then function No 1 will be called and it is very likely that the same key will still be down when this happens. If it isn't then the user will invariably press again. Function No 1 provides the ASCII of the pressed key in A, so you can use it as appropriate.

There is another alternative. Function No 6 ("Direct Console Input/Output") requires E to be loaded prior to use as well as C. If you put FFh (255) into E and then call No 6, A will contain zero if no key has been pressed, or its ASCII if one has, but no character will be echoed to the screen. If instead of FFh you put an ASCII code into E, then that character will be printed whatever key is pressed.

The books print bold italic warnings that you should not mix Direct Console I/O .(such as No 6) and other console I/O functions though it is not clear what they mean by mixing. I have used the above routine and then later used other BDOS functions without any deleterious effect that I was able to detect.

Printing single characters

When you want to print a single character use function No 2. It prints the character whose ASCII code you have loaded in E ; hence to print "?" you would load E with 63 and C with 2 and then call BDOS.

Reading text from the keyboard

The functions considered so far have dealt with only single characters though most occasions require whole phrases from the user, as in completing stock or personnel records, updating files, etc. Function No 1 could be adapted to this, but I turn pale at the prospect of writing a routine for it (including a provision to erase mistakes !).

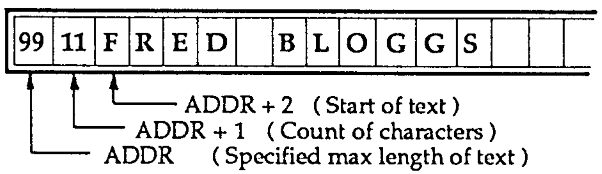
Fortunately CP/M has our best interests at heart and has provided BDOS function No 10 which takes care of just these requirements. It is called "Read Console Buffer". I would have expected it to be called "Write Console Buffer", but let's not haggle over detail.

In computerese a 'buffer' is a small area of memory where bytes can be stored prior to being processed. The printer is provided with a 'printer buffer' into which the processor hurls ASCIIs fast enough to make your head spin and which the printer subsequently plods through at the rate of one every now and then. Buffers are used to balance up the different speeds of operation of different processes.

Once function 10 has been called, the program will await the input of the text, which will be echoed onto the screen at whichever print-position is established at the time. (See pages 70 & 71.) Whilst typing it in, the two 'DEL' keys and the cursor arrows are available for correcting mistakes. When the user has completed his text input he presses 'RETURN'. It is usual then to transfer the text from the buffer to its permanent home. If the user tries to over-fill the buffer (exceed the size defined for it) then a beep will sound and no more characters will be accepted.

To use the function it is necessary to state where you want the buffer to be located in memory and how many bytes you want to allocate to it, (ie. how many characters are permitted to be typed into it) the maximum being 128. The following sub-r illustrates the case of declaring a buffer at 'ADDR' which will have room for 99 characters.

```
ld hl ADDR     33  A  A      Buffer addr into HL
ld (hl) 99     54  99        Put length at  start
ex hl de       235             of buffer, & move ADDR to DE,
ld c 10        14  10          then call the
call BDOS      205  5  0       function
      continue . .
```



A Typical Console Buffer

The total room required by the buffer is 2 more bytes than the number of characters that may be put into it, and the start of the text (its first letter) is at ADDR+2 . The address ADDR is occupied by the authorised length of the buffer (99 in the above case) . In addition, CP/M counts the number of characters that have actually been typed in and enters this value at ADDR+1 . A handy way of transferring the text is therefore

```
ld bc ( ADDR+1) 237 75 A1 A1     Put number of typed
ld b, 0         6  0               chars into BC.
ld hl ADDR+2    33 A2  A2        Point HL to source.
ld de HOME      17  H  H           and DE to home.
ldir            237 176          Transfer.
      continue . . .
```

Note that function No 10 does not add a string-end marker, so you must add your own if one is needed. (See below.)

String printing

BDOS function No 9 (called 'Print String') is available for printing the strings of prepared phrases such as menu-pages, program titles, user instructions and the like. These are made up of the ASCII codes of the relevant characters. First DE is loaded with the address of the start of the string and then the function is called. The string may be of any length but its end must be marked by a **string-end marker** ('delimiter' in w-lang) which by default is the "$" sign whose ASCII code is 36. If your string has not been provided with an end-marker then when it is printed the gibberish that occupies the addresses beyond it will be added to its end until the function comes across a "36" that happens to be lying about in memory, but no other harm will be done.

Conveniently, as well as all the letters, numerals, and punctuation signs, the string may contain a variety of very useful printing instructions called the CP/M **escape-sequences** (for accidental and therefore not very good reasons that don't involve escaping), plus control-codes such as

7:	bel	: the 'beep' sound.
8:	backspace	: move back one character.
9:	tab	: move the print position right to next column whose number is a multiple of 8.
10:	line-feed	: print at this column on the next line down, scrolling up if necessry
13:	carriage return	: move to left margin.

The print control-codes and escape-sequences are described on pages 139 - 141 [581 - 584] of the manual, the latter being listed as 'ESC X'. For m/c use this translates to '27' followed by the ASCII code of 'X'. So 'ESC E', "Clear the viewport [screen] " translates to '27 69'. In the few cases where this is followed by a number shown in italics, the number is retained *per se* in m/c.

The following routine is intended for insertion at address 50,000 to display a string located at 50010 (90,195) so it can be run under the BASIC insertion program described in Chapter 5.

```
ld de 50010    17 90 195     Point to string.
ld c 9         14  9         Select 'Print string'
call BDOS      205  5  0      and call BDOS
ret            201
```

The m/c bytes for this and for the string are given in the following DATA lines.

```
200    DATA  17,90,195, 14,9, 205,5,0, 201
210    DATA  0,0
220    DATA  27,69, 27,89,46,64
230    DATA  42,32,  84,104,105,115, 32
240    DATA  105, 115, 32, 97, 32
250    DATA  115,116,114,105,110,103,32,42
260    DATA  7, 10,10,10,10, 13, 36
```

Write these into the program and run it. '27, 69' clears the screen. '27, 89, L, C' defines the print position according to the values of 'L' and 'C'. L equals the line number plus 32, and C equals the column number plus 32. If both are given the value 32 then printing will start at the top left corner of the screen (experiment with other values larger than 32 to see the effect).

Changing the String-end marker

The PCW screen contains 32 print lines (No 0 to No 31), and 90 print columns (No 0 to No 89). Hence the value of L can be varied from 0+32 to 31+32, ie. from 32 to 63. However, if you try 36 (= line No 4) you will find that your string is ignored. This is because an error in the system programming mistakenly interprets a line value of 36 as the ASCII code of '$', ie. as the usual string-end marker. CP/M therefore prematurely thinks it has found the end of the string and as a result nothing can be printed on line No 4 by this method. This may explain why text on the PCW is so frequently seen scrolling up from the bottom; scrolling and bottom-line printing are immune to the error.

A rather more wholesome solution is to change the string-end marker to one whose ASCII code does not lie in the range 32 to 63. For this I use 255 because it is easy to spot and easy to miss if I haven't included it. It is also the ASCII code of a symbol I am not likely to want to use much in strings (it corresponds to "Equivalent to"). You might like to select your own from the character set given between pages 113 and 118 [547 and 554] of the manual. Zero is sometimes advocated, and it is a good choice, but I prefer to use zero for blanking characters that may be wanted on some occasions but not on others.

The marker is changed by putting the required ASCII code into DE

and calling function No 110. To set it to 255 the sequence is as follows:

```
ld de 255     17  255  0      Put ASCII into DE.
ld c 110      14  110         And call the
call BDOS     205  5  0       function.
     continue . . .
```

If DE is set to FFFFh (255,255) and function No 110 is called, then this is a request to CP/M to report which marker is currently in use but not to make any change to it. The code is given in A.

If you change the marker then all your strings for use with function No 9 must end with '255' (or whatever marker you specified), and when you leave your routine and return to either CP/M or to BASIC then your last instruction must be to change the marker back to '36' because all the strings used by these two systems end with '36'.

Block printing

When function No 9 is used the whole string will be printed. Function No 111 allows 'slicing', ie. the printing of any part of a piece of text. Control is provided through a 4-byte long 'character control block' called the 'CCB' to which DE points when the function is called. The first two bytes of the CCB specify the address of the first character to be printed, and the last two specify how many characters to print (up to 65535 !). Obviously in these circumstances no string-end marker is required. The calling sequence would be to put the required data into the CCB and then :

```
ld de CCB     17  C  C        DE points to CCB.
ld c 111      14  111         Call the
call BDOS     205  5  0       function.
     continue . . .
```

Message Printing

The functions so far described require the address of the text to be known beforehand. For handling a few large-sized strings this is no problem, but many programs require a surprising number of minor strings (possibly several dozen) that vary from only one character (such as 'bel') up to forty or fifty. The problem with these is that if you pack them together to save space then any alteration to an early one means that all the later ones will have their addresses changed, and if you have already written lots of routines that call them at their old addresses you will not be over the moon to have to plod through making revisions.

I have found the best solution to be the one that requires only the order of the messages to be recorded, not their addresses. To print such a message its list No is put into A and the list is then scanned until the correct one is found. The only address required is that of the start of the list, which is obviously also the address of the first message; ie. of message No 0. The printing of a message then requires only 5 bytes :

```
ld a MESNUM    62  N          Messg number into A
call SUBR      205  S  S       & call print routine
    continue . . .
```

The SUBR referred to is the one that locates and prints the message. It lists as :

```
ld hl LIST     33  L          Point to list
or             183            If (A)=0 then
jr  11         40  11           no search is required
push af        245            Store the message No.
ld a (hl)      126            Check each byte
inc hl         35               until a
cp 255         254  255         string-end marker
jr nz -6       32  250          is found.

pop af         241            Then recover the
dec a          61               message No & decrement it
jr nz -11      32  245        If No not zero repeat

ex hl de       235            Else put message address
ld c 9         14  9            into DE and
call BDOS      205  5  0        call 'Print String'.
ret            201
```

The print-position

The screen print-position always stays where the last print operation left it, ie. at the end of the last string printed. In using BDOS

functions No 9 and No 111 and in message printing, it is possible to include in the string an instruction defining where the text is to be displayed so the print position in these cases can be guaranteed.

Naturally this is not possible for single character printing, nor for placing keyboard input, and some pre-composed strings require different print positions at different times. A solution is to have in the variables area a short 'position string' made up as follows (for example) :

```
51217  (17,200)   27    DEFB
51218  (18,200)   89    DEFB
51219  (19,200)   Ln    Required Line Nà+32
51220  (20,200)   Col   Required Colm Nà+32
51221  (21,200)   255   DEFB  (end-marker)
```

If the required figures are put into 51219/20 and this sequence is then 'printed' as if it were a string then the print-position will be transferred to the location specified by it. This would be achieved by :

```
ld hl PRPOSN     33  N  N       Put print posn
ld (PRSTR)hl     34  19  200     into the string.
ld de 51217      17 17 200      Point DE to string.
ld c 9           14 9           Call
call BDOS        205 5 0        'Print string'
    continue . . .
```

More about the Print Position

To keep itself orientated CP/M counts the number of characters that have been printed since the print position was last at the left margin and automatically puts the next one in the next column to the right as it works through the string, and when the print-line is full it moves to the left end of the line below.

This is fine except that no-one thought to explain to it that not all characters are printable (some are control-codes or escape-sequences), and hence it will get its sums wrong and prematurely put your string on the next line down if you do a lot of printing without telling it to mind its own business.

To tell it thus, precede each print-position instruction with a '13' (= carriage return) ; in long strings you may insert several such 13s, all in front of position instructions. Each of them zeroes CP/M's column-count and ensures that printing will be where it is intended. The following section of a long string illustrates the idea:

. . . 66, 101, 114, 116, <u>13, 27, 89, 52, 62</u>, 70, 114, 101, 100 . . .

Note that, on its own, a '13' will also transfer printing to the left margin, but the immediately following print position instruction countermands this effect when it is not desirable.

Finding the Cursor Position

If your program should need to be informed of the current cursor position, use the sub-r listed on page 107.

Chapter 8

Using the Printer

This chapter relates principally to m/c control of the very flexible dot-matrix printer of the '8256' and '8512'. The somewhat limited possibilities of the daisy-wheel printer of the '9512' are explained from page 555 of the manual onwards, though the general principles of what follows applies to both machines.

It is possible to switch the printer on from CP/M (so that it echoes all that goes to the screen) by the key sequence ^S ^P ^Q. (When followed by a letter the symbol "^" indicates the 'control' key which for the PCW is given by 'ALT'. Not to be confused with 3^2 which means 9 !).

This sequence turns off the screen, turns on the printer, and then turns on the screen again, which complexity is needed. Later the same sequence turns the printer off again. The method is not a satisfactory way of printing because in addition to the required text, all else that comes to the screen (such as error reports) gets printed too. Unwanted control codes are sometimes also shown, together with messages that I have never yet established the source nor purpose of, and to cap all, the echoing sometimes stops at times decided not by me.

You can also produce a 'screen dump', ie. have the current screen content printed-out (bit-mapped), by simultaneously pressing 'PTR' + 'EXTRA'. This is useful for recording otherwise non-printable material, though the print size is rather small.

BDOS printing

Happily there is a printer version of the block-print function. It is No 112, and has the *w-language* name of "List block to logical device LST", but fortunately it performs very well in spite of that. The word 'list' is used to refer to printing via the printer, as distinct from 'print', which means 'display characters on the screen'.

As with function No 111, the printing is controlled from a 4-byte CCB to which DE points when the function is called. The first two bytes of it give the address of the first character to be printed, and the other two give the number of characters to print. If the string does not end with a 'line-feed' (10) or a 'carriage-return' (13), then the characters designated will be transferred to the printer buffer but they will not be printed.

One or other of these two control codes is required as a **prompt** to empty the buffer onto paper. The 2-byte character-count decides how many characters will be transferred to the buffer, and the position of the prompt decides how many of these will be printed. Those before the prompt will be printed, those after it won't, at least not immediately.

This makes it possible to join strings from different sources together before they are printed, but it also means that you have to put the prompt where you intend or you will leave debris in the buffer that will become tacked onto your next print. To print all of them, the count in the last two bytes of the CCB should just include the prompt as one of its count of characters.

Text control

The printer won't react to codes it can't interpret, and one of these is 'bel' (7), but the ones opposite are recognised. In fact there is little use for 'carriage return' except as a means of overprinting because 'line-feed' automatically causes the first character of the new line to be printed at the left margin, as does each new use of Function 110. On

the screen 'backspace' causes the next character to obliterate the last; on paper the two are superimposed.

8:	backspace - move one column left.
9:	tab - go to next colm which is multiple of 8
10:	line feed - scroll the paper up one line.
12:	form feed - scroll page out of printer
13:	carriage return - move to the left margin.

The 'line/column' escape-sequence (27, 89, L, C) is ignored by the printer which obviously can print on only the current line. Movement across the paper is effected by using 'TAB' or by inserting spaces into the string.

There is supposed to be a means of halting printing if the bottom of a short page is detected but it doesn't work for me. '27 8' is supposed to set it and '27 9' to unset it. The sequence to give an exact amount of paper-feed does work. This is '27 74 N' where N is in the range 0 to 216. If N = 216 then one inch of paper (25mm) is scrolled up; if N = 108 then half an inch is scrolled, etc.

Underlining

Text can be underlined by incorporating the escape-sequence '27 45 1' into the string. The underline is switched off by '27 45 0'.

Print style

The full range of printer control codes and escape-sequences is given in pages 126 to 135 [561] of the manual, but not all of them work (or perhaps I am incompetent). The most flexible one is 'Select Mode', through which a combination of different effects can be chosen. These are :

bit No	value	effect
5	32	Double width
4	16	Double strike
3	8	Bold
2	4	Condensed
1	2	Elite (=normal)
0	1	Pica (=smallish)
0	0	normal

Thus to start printing in double strike the escape-sequence '27 33 16' would be incorporated into a string. Any futher use of '27 33 X' in which bit No 4 of 'X' was not set would cancel the double strike mode. The bits retain their set/reset condition until you change them, so the following escape-sequences in a string would have the effects:

27 33 32	Start printing double width.
27 33 40	Add Bold to the above.
27 33 2	Cancel the above, print Elite only

Draft / High Quality

High quality printing is given by '27 109 49', and reversion to draft quality by '27 120 48'.

Italics

The italic versions of the numerals, the alphabet, and the punctuation signs are printed if bit No 7 of the ASCII code is set (ie. if 128 is added to the usual ASCII code). This works for all the symbols with codes from 33 (21h) to 126 (7Eh). Not surprisingly the normal 'SPACE' (32) looks much like its italic version, but for some reason neither 127 nor 255 give the "zero without slash" as promised by the manual, nor its italic form. You get an 'i' for the first and a '£' for the second.

Printer graphics

For printing to the screen there is a useful set of graphics with ASCII codes in the range 128 to 159. These allow you to print borders and lines and columns that tidy up the presentation of data quite nicely but sadly they can not be printed directly onto paper and the completely useless symbols tabled on page 135 are offered as an alternative (see the note at the foot of page 134). However, it is possible to print special characters of your own devising through the printer, so you can reproduce the missing ones and others to your own specification. You can even cover a whole page with designs. And if you are up to the task of converting a picture into binary digits then that too can be reproduced on paper.

When producing printer graphics, the 8 dots of each vertical line in the graphic can be represented by an 8-bit number. Zero corresponds to the instruction 'print no dots', and 255 to the instruction 'print all 8 dots'. The number 1 means 'print only the lowest dot', and 128 means 'print only the top dot'. All the other numbers imply other dots or dot combinations.

For draft quality letters six such instructions are enough to print a whole character, and as the dots are separated by a dot-width the characters are 12 dot-widths wide. For the standard range of characters the numbers required to reproduce the dot patterns are stored in the printer's memory, and the variations required to give the various printing styles are taken care of by the firmware (built-in and unchangeable programming).

To print a UDG (user-designed graphic) BDOS function No 112 is used as described above but the printed string should include the escape-sequence '27 75 6 0' immediately followed by the six 8-bit numbers that represent the graphic. You can increase or decrease the width of the UDG by changing the value '6' in the escape sequence, though this number must be matched by the number of data bytes that follow.

The count in the CCB must include the four bytes of the escape sequence plus the number of data bytes, plus the number of any other characters that are to be printed simultaneously. As always the string must end with '10' if it is to be printed immediately, and this counts as a character. The above escape-sequence does not work with BDOS function No 111, ie. it will not print to the screen, and the print control instructions that change the style of normal characters have no effect on UDGs.

To print an 8 x 8 black square, an 8 x 8 hollow square, and a right-pointing triangle, the print string would contain the following sequences respectively:

. . . . 27 75 8 0 255 255 255 255 255 255 255 255 . . .
. . . . 27 75 8 0 255 255 195 195 195 195 255 255 . . .
. . . . 27 75 8 0 255 255 126 126 60 60 24 24 . . .

Double density UDGs

The dots of a UDG can be packed together at double the usual density if the escape-sequence reads '27 76 N 0', followed by twice as many data bytes. This greatly increases the blackness of the print by halving the width of the UDG without affecting its height. It enhances the appearance of the triangle referred to above for use as a pointer. To obtain black squares that are really black and really square or black circles that are really black and circular use 16 data bytes and print them at double density.

Full page graphics

Because a UDG can can be made to any width it can be made wide enough to fill a print line, and several such lines could be used to fill a page, though when printing UDGs there is no 'wrap' onto the next line; any part of the UDG that does not fit between the margins is lost (though if it is followed by normal characters these will be printed on the line below).

Hence a separate print instruction is required for each picture line. To fill the 90 print columns requires 540 data bytes at normal density, or 1080 at double density. These counts are achieveable because the last byte in the escape-sequence (usually zero) is the high byte of the count. The byte sequence for a full line-width UDG (90 columns) will therefore be :

Single density	*27 75 28 2 . . . . .*
Double density	*27 76 56 4 . . . . .*

Library symbols

It is a simple and enjoyable matter to construct print symbols for practically any purpose. If these are kept in their own area of memory, a programming project that is likely to use them can be equipped with sub-r rather like 'Print Message' as described in the last chapter, though in this case the string-end marker would be superceded by a convenient character such as 'SPACE' (32, 20h).

An example program

The following program illustrates the points made in this chapter by printing a string of characters and then several UDGs. The routine is for insertion from address 50,000 onwards.


```
Sub-r:
        ld de 50010     17  90  195     Point to the CCB.
        ld c 112        14  112         Call the
        call BDOS       205 5 0           print function.
        ret             201             Return to BASIC
        DEFB            0               Nop

CCB:
        DEFW            95  195         Address of string.
        DEFW            65  0           String length.
        DEFB            0

String:
        DEFS            65 66 67 9 68 69 70 9
        DEFS            27 33 32  74 75 76 32 27 33 40 77 78 79 32
        DEFS            27 75 10 0  255 255 24 24 24 255 126 60 24 24 32
        DEFS            27 76 20 0  9 9  133 197 7 51 15 15
                        15 255 31 15 15 15 51 7 197 133 9 9
        DEFS            27 33 0 10
```

The compiled assembly language version is given so that the bytes can be transferred into data lines No 200 *et seq* of the BASIC insertion program suggested earlier.

'List Output'

There is a second BDOS function that can be used in conjunction with the printer. This is function No 5 , 'List Output' , which is somewhat similar to No 2 'Console Output'. It transfers the ASCII code that is in the E register into the printer buffer. Obviously printing text by this means would be laborious, but, the function can be used to add extra letters or control codes to a string already in the buffer, from where they will be printed as part of the main string.

If you want to prove that 'List Output' does work, put the printer online and run the sequence on the next page:

```
ld e 83       30 83
ld c 5        14
call BDOS     205 5 0
ld e, 79      30 79
ld c, 5       14 5
call BDOS     205 5 0
ld e, 33      30 33
ld c, 5       14 5
call BDOS     205 5 0
ld e, 10      30 10
ld c, 5       14 5
call BDOS     205 5 0
ret           201
```


Chapter 9

Screen Graphics 1

Special characters

For normal printing to the screen the character set is limited to those listed on pages 113 to 118 [547 to 554] of the manual, but excluding the ones with ASCII codes less than 32. By convention these low numbers are reserved for control-codes and therefore don't print, even though few are actually in use as controls.

The remaining 224 give a wide range of choice that covers most standard requirements, though inevitably many of the characters with foreign accents are of only occasional if any interest. At the same time a user may require a number of non-listed special symbols to suit his own purposes; 'Locoscript', for example, uses custom-made signs to indicate inset-paragraphs and the location of paragraph-ends, to name only two.

One of the advantages of controlling print operations from m/c is that symbols to suit virtually any requirement can be incorporated into the character set and then printed in the normal way by using BDOS function Nos 1, 9 or 111. This is done by altering the pixel pattern associated with a particular ASCII code (the pixel pattern is the set of light-dots that make up the shape of the character when it is displayed on the screen), and to make the alteration it is necessary to gain access to the area of memory where the patterns are stored.

Memory Banks and Blocks

As indicated in chapter 2, the Z80 can address only 65536 (64k) memory addresses, though in fact the PCW is provided with either 256k or 512k of memory depending on the model. So large a storage capacity greatly enhances the machines' ability to manipulate the data required by complex programs, but because the processor can't have access to all of it at once it has to be split into segments.

A convenient subdivision is into **blocks** of 16k, so that four of these constitute the amount of memory that the Z80 can deal with at one time. More than four 16k blocks are installed in the machine but they are 'switched into circuit' only when the processor has need of the information they contain. Each set of four in service together is called a **bank**. Hence each 'bank' consists of 64k.

The blocks are numbered from 0 to 15 for the '8256', and 0 to 31 for the '8512' an d '9512', though the blocks in a bank are not usually in a numerical sequence. Block No 7 is given the name 'common' memory because it is in service in all banks. It provides an area in which co-ordinating instructions can be placed. These remain in force regardless of which bank is in use; if this were not so it would be impossible for different banks to apply themselves to the same task. The contents of the various blocks is as indicated on the next page. The banks are also numbered from 0 upwards, and within each bank the addresses always follow the standard range of 0 to 65535 (0000h to FFFFh).

What is stored in a memory bank is not changed by the action of it being switched into and out of circuit. Its contents are continually refreshed by the normal interrupt sequence that the machine uses to ensure that no information is allowed to leak away from its memory.

The Memory Blocks

Block 0	A BIOS jumpblock.
Block 1	Most of the screen pixel data (the record of which pixels are 'on' and which are 'off').
Block 2	The screen-character shape data, Roller RAM, and some of the screen pixel data.
Block 3	BIOS and BDOS routines.
Blocks 4-6	Most of the TPA.
Block 7	The upper part of the TPA plus the resident part of BDOS & BIO S (at F606h and up; see Chapter 6). All of which is 'common' memory.
Block 8	CCP, disc hash table, data buffersparts of BIOS.
Blocks 9 up	The 'Memory Disc' (see Chapter 11)

(For details of memory switching see Chapter 11 and Appendix 7)

The 'Screen Environment'

There is also a special bank that is not given a number but is called **The Screen Environment.** The blocks that are in service in the various banks, including the Screen Environment, are as follows:

Start-address Hex	R.Biro	Bank 2	Bank 1	Bank 0	Screen Envr
C000	(0,192)	7	7	7	7
8000	(0,128)	-	6	3	2
4000	(0,64)	8	5	1	1
0000	(0,0)	-	4	0	0

Bank 1 we have met before; it contains the TPA, ie. those sections of memory that are occupied by user programs (it is the bank that is made available when the machine is ready for use), but switching the Screen Environment into service is of particular interest to us now because it is the only one that gives access to block 2 in which the data for the shape of the screen characters is kept.

Accessing the Screen Environment

The Screen Environment is switched-in by the 'Screen Run Routine', which itself is accessed through the Extended BIOS Jumpblock in block 0. Because this block is not in service at the same time as the TPA, user programs must access it through another jumpblock in Page 0 from which the address of the 'Jmp-Userf' entry can be derived. This is easier to do than it is to explain.

An example program

Compile the following program, insert it at 50000, and then run it. It changes the the pixel pattern relating to ASCII code 97 - the one that normally displays the pattern for "a". When you have run it, try typing in a line containing the letter "a".

```
START:
        ld bc SUB-R     1 97 195      Point BC to SUB_R
        call userf      205 89 195
        DEFW            233 0         Addr of Scrn Run Rtn
        ret        *    201           Return [to BASIC]
USERF:
        ld hl (0001)    42 1 0        Addr W.boot into HL
        ld de 87        17 87 0       Add 87 to
        add hl de       25             get 'jp.userf addr,
        jp (hl)         233            and then jump to it.
SUB-R:
        ld hl 47880     33 8 187      HL = Addr of 'a'
        ld b 8          6 8           Count of 8 bytes.
        ld (hl) 255     54 255        Change all
        inc hl          35             the bytes
        djnz -5         16 251         to 255,
        ret             201            and return.
```

The listing is in three parts. The first part is a kind of 'executive program', the second sorts out the 'jump userf' address, and the third contains the instruction for making the changes to the pixel pattern.

Part 1 first loads BC with the address of the third part; it is necessary to use BC for this purpose when the Screen Run Routine is being called. It then calls the second part, but within this call sequence is the DEFW (two bytes) which is being used as an 'in-line parameter'. An in-line parameter is data that is in the next one or more bytes that are 'in line with' (ie. immediately following) the instruction in question. Consequently the address of the Screen Run Routine is already known to the 'userf' function as the address it must use when it begins its operations.

In part 2 the address contained in 0001/2h is loaded into HL and 87 is added to give the 'jump-userf' address. A jump is then made to this address, but, because of the unusual structure of the sub-r to be found there, when this sub-r has done its work a 'ret' is made to the place after the 'call' that initiated the sequence (which was 'call USERF'); ie. to the place marked by '*' in the listing. Because in this case a 'ret' is found there, a return to BASIC is then made.

In the third part, the address of the start of the pixel-pattern for 'a' is put into HL and then this address and the seven addresses following it are all loaded with 255.

Note that this program works only because being at address 50,000 and above puts it into common memory, ie. at or above (0,192), 49152, or C000h. Had it been below C000h then its third part would not be accessible while the Screen Run Routine was operating, the pixel pattern changes could not take place, and the most probable result would be a crash.

Pixel patterns

The patterns of all the 256 characters listed in the manual are stored in Block 2 starting at address (0,184 , ie B800h. The section of memory containing them is called 'The Character Matrix RAM'. (RAM just means 'memory'.) Each pattern requires 8 bytes within the Matrix RAM , so the pattern for the character whose ASCII code is 0 takes up the first 8 bytes, the pattern for ASCII No 1 occupies the next 8, and so on up to ASCII No 255 whose pattern is stored from (248,191) to (255,191) ie. from BFF8h to BFFFh.

The Matrix RAM contains 8 bytes per character because each one is displayed on the screen as a set of 8 horizontal lines of dots. The first byte represents the top line of the character, and the eighth byte represents its bottom line. Each dot of light is called a 'pixel'.

The 8 bits of each byte signal the state of the 8 pixels on its line. Each set bit signals an 'on' pixel (a bright dot), and each reset bit signals an 'off' pixel (no bright dot). A byte content of 1 gives an 'on' pixel at the right-hand end of the line, 128 gives an 'on' pixel at the left-hand end. If all the bits are set (byte = 255, FFh) then a continuous line of 'on' pixels is signalled, and if all 8 bytes contain 255 then a bright rectangle is displayed for that ASCII code. It is a rectangle rather than a square because the pixels are packed closer together in the horizontal direction than they are in the vertical direction. The vertical pixel separation is twice the horizontal separation.

After running the above program attempts to print "a" will display a solid rectangle, though this new pattern will be overwritten by the normal "a" pattern whenever CP/M is again loaded into the computer.

Using the Character Matrix RAM

To make up and use a new pixel pattern first select an existing character that is superfluous to your requirements and make a note of its ASCII code. You then find its address within the Character Matrix RAM by multiplying the code by 8 and adding the result to the start address of the Matrix RAM. A simple way of doing this is :

```
ld hl ASCII     33  N  0      ASCII code into HL
add hl hl       41             then
add hl hl       41             multiply
add hl hl       41             by 8.
ld de MATRX     17  0  184    Point DE to Matrix-RAM
add hl de       25             and add so HL points to Char.
    continue . . .
```

Once the start address of the character has been found, then that and the next 7 addresses should be loaded with the bytes that establish its new pattern. The pattern can be most easily worked out by using a block of 16 x 8 5cm squares on a sheet of graph paper onto which the outline of the character is drawn. Pairs of squares one above the other represent each pixel. Any square-pair covered by the outline is counted as a set bit, all others counting as reset. Don't spend too long on the artwork as the result on the screen is often surprisingly like or unlike what you are hoping for so screentests are even more appropriate than in Hollywood. Also bear in mind that if it may need to be underlined or otherwise be matched up with the standard letters then the bottom byte should be left as uncluttered as possible.

Printing the first 32 ASCIIs

If you fancy having available the Greek letters or any of the other first 32 characters given on page 113 [547] of the manual, then transfer the required ones en bloc by means of an 'ldir' operation into a higher place in the Character Matrix RAM , thus obliterating the uninteresting stuff that is already there and taking over the ASCII codes relating to it. Take care that the first addresses POKEd from and into are really the starts of characters or your text will end up looking like something from the original "The Fly" (heads and bodies contributed by different characters !).

The printer uses a different set of character patterns so the changes described above make no difference to paper-printed text. (see Chap 7.)

Chapter 10

Screen Graphics 2

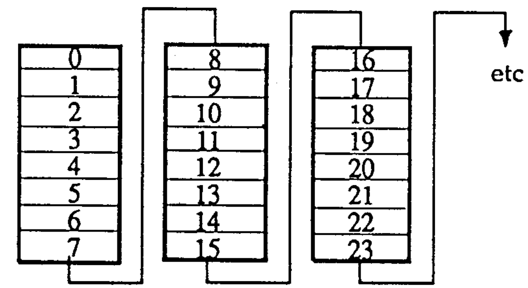
Drawing on the Screen

Blocks 1 & 2 also contain the data that indicates which of the screen pixels are on and which are off, so the screen environment is also the path that gives us direct control of what the screen will display.

The screen is 256 pixels high by 720 wide. As each print character requires a square of 8 x 8 pixels there are 90 print columns and 32 print lines, though the lowest line cannot normally be printed on because it is reserved for system reports. (See 'Status line' below)

The first byte of screen data is (48,89) ie. 5930h in block 1, and the last one is (47,179) ie. B32Fh in block 2. However these addresses are arranged in the way which best suits the printing of 8 x 8 characters so they are not sequential across the screen. Each screen byte represents a horizontal row of pixel-dots and each sequence of 8 of these arranged one below the other gives the data for a print position (a line and column intersection). The next 8 bytes cover the next print position to the right along the same print line. Thus the 90 columns of a whole print line are defined by 720 bytes in a zig-zag pattern, and the next print line down is defined by the next 720 bytes, etc. (See the diagram on the next page.)

There is a further complication in that a particular set of 8 bytes do not apply to a fixed screen position; to aid the process of text scrolling the set always points to the same position within the text wherever it is scrolled to. This procedure is illustrated by the following program which is intended for insertion at address 50000 by the BASIC insertion program. It is very similar to the one used for making modifications to characters.



The sequence of addresses in a print line

```
START:
        ld bc SUB-R     1 102 195
        call USERF      205 94 195
        DEFW            233 0
        ld c 1          14 1          Await keypress
        call BDOS       205 5 0         ( see Chap 7 ) before
        ret             201             returning to BASIC.
USERF:
        ld hl (0001)    42 1 0
        ld de 87        17 87 0
        add hl de       25
        jp (hl)         233
SUB-R:
        ld hl SC-ST     33 48 89      Screen addr to start
        ld bc 720       1 208 2       Count of 720 bytes.
        ld (hl) 255     54 255    *   Each byte to be 255
        inc hl          35            Next addr.
        dec bc          11            Reduce count,
        ld a, b         120             and
        or c            177             check;
        jr nz -8        32 248          repeat if not zero.
        ret             201           Return when finished.
```

When this is loaded and run it will produce a bar of white across the screen, though I can't predict where this bar will be because I don't know the state of your 'Roller-RAM'. If the bar is near the top of the screen then press 'RETURN' a few times until it scrolls out of sight and then RUN 500 to produce another one nearer the bottom. Now change the '255' indicated by '*' to '85', then RUN again.

These changes and the ensuing reports will have scrolled the bar upwards, but in spite of this the new (dimmer) bar will fall on top of the old one and completely blot it out. Note that the two bars appeared at different screen positions though the addresses given for their creation were the same. (Because of the 'await-key' you need a key-press to achieve a return to BASIC.)

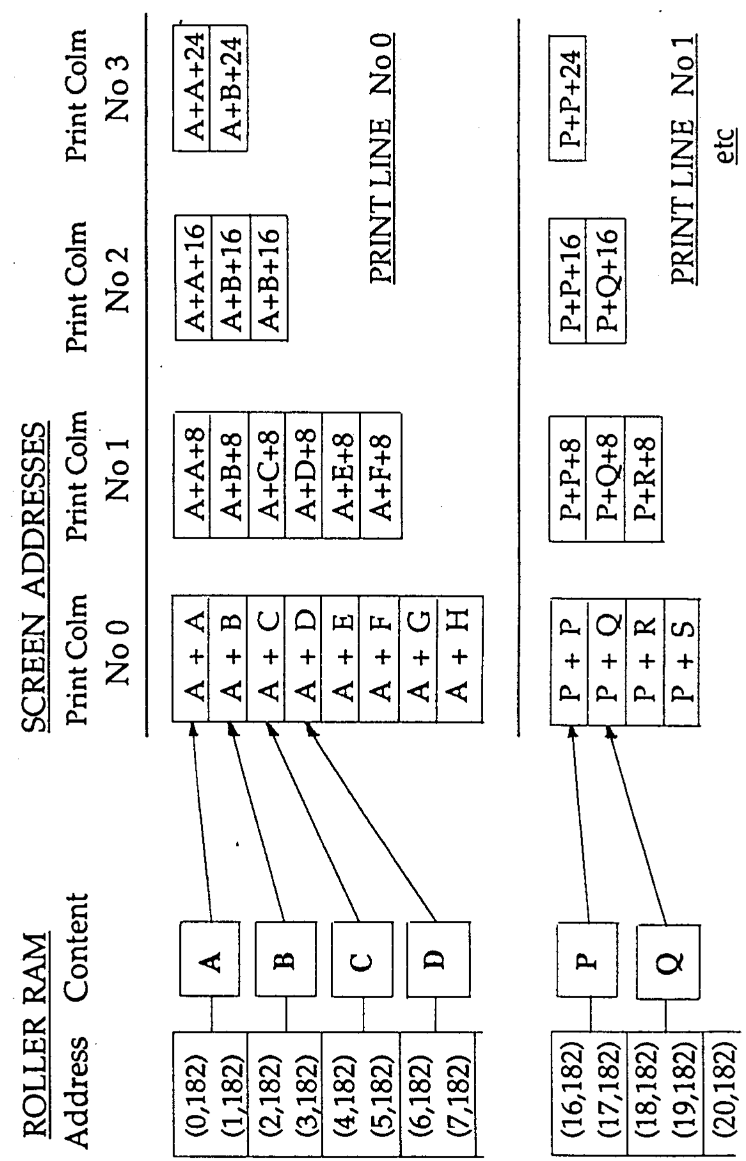
If by this means or otherwise the lowest line (the 'status-line') is written into, it will not normally scroll up as the other lines do but it can be made to do so by *PRINT CHR$ (27)+CHR$ (48)* or the equivalent m/c print instruction ('27 48' in a print string). The effect is reversed by changing the '48' to '49'.

Roller-RAM

'Roller-RAM' is 512 bytes of memory starting at (0,182) ie B600h in block 2. It is rumoured to contain the 256 addresses of the starts of the 8 pixel lines within each of the 32 print lines, and to update these every time they change due to a scroll action.

What it actually contains is the first address for the print-line divided by 2, with the other seven as sequential increments of this value. Hence to derive the address of a pixel-line you have to add its entry to the entry for the print-line (see the diagram on the next page). No doubt this eccentricity serves some secret CP/M purpose, but I can't imagine what.

The real point to Roller-RAM is that, oddly written as it is, it does contain unambiguous data that allows us to pin down the memory addresses of a chosen screen location. In particular, doubling the entry at B600 / B601h, ie. (0/1, 182), gives the address of the start of print-line No 0 (ie. of the byte at the top left corner of the screen), doubling the entry to be found at (16/17, 182) gives the address of the start of print-line No 1, and so on, each useful entry being 16 bytes on from the last. The diagram opposite shows the connection between the contents of Roller-RAM and the addresses of screen-bytes.



The contents of Roller-RAM mapped onto the screen

To demonstrate the efficacy of Roller-RAM's record keeping, change the last part of the earlier program to read as indicated below.

```
SUB-R:
      ld hl (RLLR)   42 0 182    Take 1st entry from Roller-RAM
      add hl, hl     41           and double it.
      ld (hl) 255    54 255      Set all the pixels
      ret            201          and finish.
```

This version will display a short line of pixels at the top left corner of the screen. However, as soon as you leave the m/c routine, BASIC will report "Break at 500. / Ok" and these two text lines will cause the screen contents to be scrolled up thus carrying our pixel-line to oblivion. This explains my inclusion of the 'await a key-press' in the first section of the program. Without it you wouldn't see the pixel-line due to immediate scrolling by BASIC. Unlike the earlier version of the program, however many times it is called this one always puts the line in the same place on the screen.

Screen Co-ordinates

Because printing always starts at the top left of a page it is natural for the Roller-RAM to start from that point of the screen. However most of us have been educated to a co-ordinate system that counts positive as up and to the right so it is natural for us to want the bottom left corner as the 'origin' or zero point of our screen map. That is what I will use in calculations relating to screen positions which will therefore range from [0,0] at bottom left to [719,255] at top right. [0,255] corresponds to top left, and [719,0] to bottom right. To avoid ambiguity, square brackets will indicate that these are positions in a rectangular co-ordinate system and not Red-biro addresses. Pixels are packed together about twice as densely in the horizontal as in the vertical direction, so the value in the 'X' direction should be doubled if a 'square' display is required. This allows two side by side pixels to be illuminated for each point which greatly enhances the brightness - single pixels are not easy to see.

The following explanation is probably not needed, but I will give it anyway. A particular place on the screen is represented by a pair of co-ordinates, the first indicating the 'X' distance rightwards from the left edge and the second indicating the 'Y' distance up from the bottom. Each such place corresponds to a pixel, and a pixel is a bit within a screen address. Hence a place on the screen as far as the computer is

concerned is a bit number within an address. So a program to calculate screen locations needs to return two elements of data; the address and the bit number.

Calculating screen addresses

The procedure for calculating the address of the screen byte for a chosen co-ordinate pair is a matter of common (and probably garden) algebra, but it occupies several stages of deduction and explanation so I have relegated it to Appendix 5, though the calculations can be summarised as :

1. Calculate which print-line we are in.
2. Is a fraction of a print-line involved ?
3. Calculate the effect of the 'X' co-ordinate.
4. Derive addr of print-line from Roller-RAM.
5. Add adjustments to get screen byte address.
6. Find the bit number within the byte.

This gives both the address and the bit number so the latter can be set without disturbing the other bits at that address, hence images can be imposed on a back-ground without corrupting it. For moving images it is a simple matter to record the address and the bit that was last set and then reset it without going through the lengthy search procedure a second time, though such resetting will 'unpick' a background if it is not constantly refreshed. The table on the next page gives the variables of a sub-r to obtain screen addresses.

Insert the following program at 50,000 and put the co-ordinates that interest you into (0/2,200). When the program is run it will provide the relevant screen address in (10,200), and the bit reference in (12,200)

51200	(0,200)	Lo byte of 'X'.	*	PUT CO-ORDS
51201	(1,200)	Hi byte of 'X'	*	IN HERE BEFORE
51202	(2,200)	'Y'.	*	CALLNG ROUTINE
51203	(3,200)	LINE.		
51204	(4,200)	31 - LINE.		
51205	(5,200)	Fraction of line.		
51206	(6,200)	Lo byte of 8 x COLM.		
51207	(7,200)	Hi byte of 8 x COLM.		
51208	(8,200)	Lo byte of Line-address.		
51209	(9,200)	Hi byte of Line-address.		

```
51210  (10,200)  Lo byte of BYTE-ADDRESS.    || SCREEN
51211  (11,200)  Hi byte of BYTE-ADDRESS     || ADDRESS.
51212  (12,200)  SET BIT (1-128 not 0-7)     || & BIT No
51213  (13,200)   Old
51214  (14,200)    address
51215  (15,200)     and bit.
```

When given an appropriate feed (not listed here) it can fill the screen pixel by pixel in about 41 seconds. This may not sound fast, but as there are 184,320 pixels it amounts to one every 0.22 milli-seconds. In cases of moving a relatively small number of pixels against a stationary background, the operation is quite fast, and some increase could be gained by not storing some of the parameters.

<u>Prepare to use Screen Environment</u>

```
START:
        ld bc SUB-R     1 97 195
        call USERF      205 89 195
        DEFW            233 0
        ret             201

USERF
        ld hl (0001)    42 1 0
        ld de 87        17 87 0
        add hl de       25
        jp (hl)         233
```

SUB-R:
<u>Calc 'LINE' & '31-LINE'</u>

```
        ld a (51202)    58 2 200     Put 'Y' into A,
        ld c, a         79            and into C.
        srl a           203 63       Divide
        srl a           203 63        by
        srl a           203 63        eight
        ld (51204)a     50 4 200     Store (31-LINE),
        ld b, a         71           Put into C,
        ld a 31         62 31         and subtract
        sub b           144           from 31.
        ld (51203)a     50 3 200     Store LINE.
```


<u>Calculate part lines</u>

```
        ld a (51204)    58 4 200     31-LINE into A,
        sla a           203 39        and
        sla a           203 39        multiply by
        sla a           203 39        eight.
        add 7           198 7        Add seven,
        sub a, c        145           and subtract 'Y',
        ld (51205),a    50 5 200      then store.
```

<u>8 x Column No</u>

```
        ld hl(51200)    42 0 200     'X' into HL.
        res 0, l        203 133      Obtain the
        res 1, l        203 141       value of
        res 2, l        203 149       '8 x INT(X/8)': col No
        ld (51206)hl    34 6 200      and store.
```

<u>Get Line-Address</u>

```
        ld hl(51203)    42 3 200     'LINE' into
        ld h 0          38 0          HL.
        add hl, hl      41           Multiply
        add hl, hl      41            by
        add hl, hl      41            sixteen.
        add hl, hl      41
        ld de RLLR      17 0 182     R-RAM addr into DE.
        add hl, de      25           HL pts to Line-entry.
        ld e (hl)       94           Transfer
        inc hl          35            it
        ld d (hl)       86            into DE,
        ex hl de        235           then into HL,
        add hl, hl      41            double for Line-addr
        ld (51208)      34 8 200     Store.
```

<u>Get byte-address</u>

```
        ld de(51206)    237 91 6 200 '8 x Colm' into DE
        add hl, de      25            add to Line-addr.
        ld a(51205)     58 5 200     Put
        ld e, a         95            'part-lines'
        ld d, 0         22 0          into DE,
        add hl, de      25            and add to result.
        ld (51210)hl    34 10 200    Store.
```

continued on next page . . . .

Find bit No & set it

```
          ld a(51200)    58 0 200     Low byte of 'X' to A
          and 7          230 7          get 8 x ( X/8 - INT (X/8) )
          ld b, a        71           Then
          ld a, 7        62  7          subtract from
          sub a, b       144            7 to give bit No
          ld b, 1        6  1         Set bit Nà 0 of B
          or a           183          End if A...
          jr z 5         40  5          contains zero.
          sla b          203 32       Else move the set bit
          dec a          61             leftwards using (A)
          jr nz -6       32  251        as a count.
          ld a, b        120          Then store set bit
          ld (51212)a    50 12 200      in 51212.
          or a, (hl)     182          Set this bit
          ld (hl) a      119            of the HL address.
          ret            201          Return to USERF.
END
```

Checking the program

To make sure the above routine was working properly I added the following code at the start of SUB-R:

```
FEED: ld hl(51200)    42 0 200     Increment
      inc hl          35             the value
      ld (51200)hl    34 0 200       of 'X'.
      ld de 171       17 207 2     If it
      or a            183            has not
      sub hl, de      237 82         exceeded 171
      jr c 13         56 13          then jump on.
      ld hl 0         33 0 0       Else reset 'X'
      ld (51200)hl    34 0 200     to zero,
      ld hl 51202     33 2 200       and
      dec (hl)        53             reduce 'Y'
      ld a (hl)       126            by 1.
      or a            183          If 'Y' is now
      ret z           200            zero then  END.
```

together with the following to replace the last 'ret' :


```
RUBOUT:
          ld hl(51213)    42 13 200     Reset
          ld a(51215)     58 15 200       the last
          cpl             47              pixel
          and (hl)        166             that was
          ld (hl) a       119             set before this.
          ld hl(51210)    42 10 200     Record present addr
          ld (51213)hl    34 13 200       as the last address,
          ld a(51212)     58 12 200       and present set-bit
          ld (51215)a     50 15 200       as last set-bit.
          ld b 255        6 255         Slow
          nop             0               the
          nop             0               process
          djnz -4         16 252          down by looping.
          jp SUB-R        195 97 195    Then repeat.
END
```

These additions cause the routine to set each screen bit in turn and then reset the one that was previously set. It gives the effect of a dot of light moving from left to right across the screen and dropping to the next line down at each line-end like the raster dot of a TV screen. If the process is not slowed down by the 'djnz', the dot seems to flicker and move irregularly because of interference from the interrupts and the monitor scan. To initiate it, use this BASIC sequence which starts the dot at 'Y' = 50, and resets the 1st screen-bit to get things going:

```
poke (51200), 0 :   poke (51201), 0 :   poke (51202), 50 :
poke (51213), 0 :   poke (51214), 4 8:   poke (51215), 89 :
print  chr$ ( 27 ) + chr$ ( 69 ) :
z = 50000 :  call z :  stop
```

Screen Clearing

If for some reason the escape-sequence '27 69' cannot be used, you can still clear the screen by including the following code in 'SUB-R'. It makes no use of Roller-RAM but puts zeros into every screen-address regardless of their sequence on the screen. It takes 0.09 seconds.

```
ld hl SCR-ST     33 48 89     First scrn-byte addr
ld (hl) 0        54 0         Zeroise.
ld de HL+1       17 49 89     2nd scrn-byte addr into DE
ld bc 23039      1 255 89     Num of scrn bytes-1.
ldir             237 176      Zero to all bytes
```

continue . . .

A similar approach can be used for partial screen clearing after obtaining the start address from Roller-RAM, but you can't use a single 'ldir' because simple increments to such an address may give values that are beyond the end of screen memory. After each print-line has been cleared you must check that the address pointed to does not exceed (47,179) or B32Fh. If it does then all further clearing must be from the start of screen memory at (48,89) ie. 5930h. Alternatively the start address of each 'to-be-cleared' print-line can be taken from Roller-RAM. (Remember to double the address from Roller-RAM.)

Double setting

The following sub-r sets the pixel to the right of the one set by the main program. If the existing pixel is bit No 0 then it sets bit No 7 of the present address + 8, ie the one lying to the right on the screen. If the present byte is at the right edge of the screen ('X' ≥ 712) and its bit No 0 is set, then the sub-r will not attempt to set a bit still further to the right. With minor modification, the approach can be used for setting the bit on the other side, or for setting several bits to produce thicker lines.

```
ld hl(51210)   42 10 200    This screen-byte addr into HL
ld a (51212)   58 12 200      and set-bit into  A
bit 0 a        203 71       If set-bit is No 0
jr nz 7        32 7           then jump on.

ld b, a        120          Else copy bit to B
srl b          203 56         and move 1 place right
or b           176          Combine this with 1st
or (hl)        182            and with screen-byte
ld (hl) a      119            load all to screen-bt
ret            201          Finish.

push hl        229          Save screen-byte address
ld hl(51200)   42 0 200     Put 'X' into HL,
ld de 712      17 200 2       and
or a           183            subtract 712
sbc hl de      237 82         from it.
pop hl         225          Recover addr.
ret nc         208          Finish if 'X' ≥ 712.
ld de 8        17 8 0       Else
add hl de      25             add 8 to byte addr
set 7 (hl)     203 254        & set its leftmst bit
ret            201          Finish.
```


Line drawing

Because the CP/M screen map is laid out for printing and not for plotting, the most convenient approach to drawing lines is first to develop a suite of programs similar to the one above with the others capable of setting the pixel above, below, to the left, and at each corner of the primary pixel. An executive routine then sorts out in which direction the line is to grow from its start point and how many 'up' or 'down' pixels are required per 'across' pixel, or vice versa. Thus if

$$ABS (X1 - X2) = 4 \times ABS (Y1 - Y2),$$

then the line will consist of segments 4 pixels long in the 'X' direction, each segment touching the last segment corner to corner.

For $ABS (Y1 - Y2) = 6 \times ABS (X1 - X2)$

the segments would be 6 pixels long in the 'Y' direction. It is a good idea to record the details of the last pixel set so that the figure can be easily extended thereafter.

Deleting Pixels

The method of deleting set pixels is shown under 'RUBOUT' on page 95. The set bit is put into A and then complemented. If A is then ANDed with the contents of the screen byte then you can guarantee that this bit will be reset and the others preserved. Note that the alternative of XORing with the original un-complemented bit may not give the desired effect. If the bit in the screen byte has already become reset by some other means, then XORing it with a set bit will set it again.

Vertical Scrolling

The screen contents can rapidly be scrolled up or down by scrolling the contents of Roller-RAM. For upward scrolling DE is loaded with the first address of the RAM and HL with an address greater than this. Call the difference between them 'Diff'. The amount of screen movement will be Diff x 2 pixels, so 'Diff' must be an **even** number or the screen will become hopelessly scrambled. BC is loaded with the number of bytes in the Roller-RAM minus Diff. If the status line is to be included this is given by 512 - Diff, otherwise by 496 - Diff. The scrolling is then achieved by 'ldir'.

For scrolling down DE is loaded with the last address of Roller-RAM which is (255,183) or (239,183), HL with an odd address smaller than this, and BC as above. The scroll action is produced by 'lddr'.

Scrolling-up duplicates the bottom screen pixels, scrolling-down duplicates the top ones, so a feed of new screen data at these places is required if a consistent display is intended. If vertical scrolling takes place in multiples of 8 pixel-lines then printing can follow without problems, but otherwise newly printed characters will be scrambled.

Horizontal Scrolling

There must be some way of using Roller-RAM to scroll horizontally, but being ignorant why it is written the way it is and how it is used except for column No 0, I haven't been able to work one out, though it is easy to scroll the screen data left or right column by column (ie. in 8 pixel jumps). To scroll left DE is loaded with the first screen address (48,89), HL with (56,89), and BC with the number of screen bytes minus 8, ie. with (248,89). The bytes are then moved left by 'ldir'.

To scroll right DE is loaded with the last screen address (47,179), HL with (39,179), and BC with (248,89). The right scroll is produced by 'lddr'.

Scrolling the screen data never produces problems with later printing, but it does produce 'wrap' - the column scrolled off the screen at one side reappears on the other side on the print-line above or below and this has to be overwritten by the incoming new data. It is visually more satisfactory to blank the ejected column prior to scrolling, or to scroll only 89 columns, so that the flicker effect is reduced.

Scrolling horizontally pixel by pixel is achieved by rotating every screen-byte so that the end bit is moved into Cy, but it is complicated by then having to rotate Cy into the byte 8 addresses away. It is only theory so far, but some day I intend to do a pixel-scroll in 8 passes of each print-line, each pass starting one address later ; in the pattern of an 8-threaded screw. Whichever way, each print-line has to be treated as a separate entity.

Chapter 11

The Memory Disc

The PCW models have 64k of memory immediately in contact with the Z80 processor, but a slice of this is occupied by the resident BDOS and BIOS leaving about 60k of TPA for users. The amount of data to which the user can have access is more or less unlimited if disc storage is used, but taking information from discs, working on it, and then redisking the updated version is a distinctly pedestrian procedure ; as anyone who has sat through a monthly accounts package will confirm.

Fortunately for those jobs that require rapid handling of a lot of data there is additional memory providing virtually instantaneous access to a total of 256k in the case of the '8256', or 512k in the case of the '8512' and '9512'. When the memory allocated to CP/M has been subtracted, the availability stands at 128k and 384k resp., and it would be a highly unusual enterprise that felt cramped by figures of this size. This is known as the **Memory Disk** though of course it is not a disc at all - this is the name given to the set of memory blocks, with numbers of 9 and above, which can be treated by CP/M as if they were a disc, at least in the sense that it can write to, or read from, 'files' in this particular memory area.

If you are interested in the technicalities of how the different memory banks are 'switched into circuit', please refer to Appendix 7. For our present purposes suffice it to say that CP/M contains a sub-r called **The Memory Manager** which can be called at FD21h (33,253) on the

'8256' and '8512' or at FD2Dh (45,253) on the '9512'. It is used by loading A with the number of the memory bank required and then calling the function. In what follows I will always use the lower address, '9512' owners should use the higher. Thus the following sequence would switch Bank No 0 into circuit (see page 83 for bank details)

```
ld a, 0          62 0
call MEM_M       205 33 253      [or 205 45 253]
    continue . . .
```

Bank switching

The following sub-r uses bank switching as an alternative to the Screen Run Routine to POKE bytes directly into the screen data, thus setting and resetting pixels within a print line.

```
xor a            175          Zeroise A and switch-
call MEM_M       205 33 253     in Bank No 0.
ld hl, SCRN      33 48 89     Point to screen data.
ld b, 240        6 240        Count 240 bytes.
ld (hl), 85      54 85        Fill byte with '85'.
inc hl           35           Point to next.
djnz -5          16 251       Repeat until count reaches zero
ld a, 1          62 1         Restore
call MEM_M       205 33 253     the TPA.
ld c ,1          14 1         Await a
call BDOS        205 5 0        keypress.
ret              201          And finish.
```

As before, the screen location of the inserted bytes will depend on Roller RAM, but the effects of the technique are limited to only a part of the screen data and it has no access the Character Matrix RAM, nor to Roller RAM, etc. These are all located in block No 2 which is not accessible through a numbered bank but only through the Screen Environment (see page 83).

Switching blocks

As an alternative to bank switching, experiment has yielded an empirical method of addressing, not the banks of the Memory Disc, but its individual blocks. The method is to load A with a value of 35 or more and then call the Memory Manager. The required block is then switched into circuit and is available as if it covered the address range

(0, 64) to (255, 127) (the block must always be addressed in this range)

the number of the switched-in block being given by [(a) - 26].

Do not attempt to use this approach to access CP/M features such as Screen Data, Roller RAM, Character Matrix RAM, etc. To address them use the Screen Run Routine as described in Chapters 9 & 10.

The highest block in use by CP/M is No 8, so you should not write into a block that has be called by an A value of less than 35 An A value of 35 and above will provide contact with the blocks of the Memory Disc, and you can insert data into, or obtain it from, such blocks by addressing them in the range (0,64) to (255,127). This means that the address of the first byte in the block is always (0,64), regardless of the block number, and the address of its last byte is always (255,127).

As each block contains 16k of memory there are 16 blocks in the '8256' (Nos 0 to 15) , and 32 in the '8512' and '9512' (Nos 0 to 31). The highest value placed in A before employing this method should therefore be 41 in the case of the '8256', and 57 in the case of the other machines. To switch-in a chosen block, put the appropriate value into A and then call FD21h, (33,253). When you have finished with the block, load A with 1 and call the address again to re-establish the TPA. The calling sub-routines should be in block 7. As indicated on page 82, block 7 is always in service and your sub-routines that call up new blocks and then switch back to the TPA must be located in it so they are not switched out of circuit. The most likely use of the Memory Disc is as a storage area from which to withdraw data so that your routines in the TPA can work on it, as by:

a) switch-in the new Block
b) 'ldir' requ data into a 'holding area' in block 7
c) switch back to the TPA
d) employ the relevent TPA sub-rs to process data in the holding area

A convenient 'holding area' is the region from (128,242) to (0,246) [F280h to F600h]. As indicated on page 56, this region is available after program loading has been completed but it cannot be a constituent of a program because such a program would be too long to load.

Whatever else has happened, block 7 always starts at (0,192), though you can duplicate it at (0,64) as well if you wish. If the processing sub-routines can all be accommodated in block 7 and are fully self-contained then it will not be necessary to move the data because they can process it where it lies during the time that the new block is in-circuit and then pass on the result of such processing as a few variables. Having the main variables area in block 7 would be convenient in this regard. Whereas you can always move data back and forth between any block and block 7, there is no means of moving it directly between other pairs of blocks, though it can be done indirectly by using block 7 as the intermediary.

It is also possible to locate some sub-routines within the blocks of the Memory Disc and obtain data-processing through them if this is desirable, but this is risky because such routines must unfailingly return control to block 7 regardless of excursions such as errors or unplanned-for results from calculations, and they must be able to operate in the absence of the facilities that may have been switched out of circuit, including the stack. It is therefore better to regard the 'Memory Disc' as just that ; a welcome extra piece of storage.

For your reassurance I have been using the 'Empirical Method' of block switching for three years in the commercial manipulation of accountancy documents. If there had been any glitches in the approach they would have shown themselves with a vengeance by now. Commercial work has the advantage of sharpening the attention wonderfully.

Restoring the TPA

Because it is essential always to return to it, I will risk boring you by restating that the TPA (Bank No 1) is the 'standard' bank that will contain your main operational routines and their variables areas. Switching back to it is achieved by loading A with '1' and then calling FD21h ie (33,253). This instruction will be required immediately after every use of the Memory Disc, and it needs to be located in common memory.

An example program

The following program illustrates a possible application of the block-switching technique. Imagine that an unspecified number of 128-byte

data records have been stored in blocks 9 upwards, and that each has a two-byte serial number in its first two bytes. This routine is given the job of searching through the records to find the one whose number is the same as the number at (0/1,196). If the correct serial number cannot be found, a return is made with Cy set, otherwise Cy is reset and the required document is returned in the 128 bytes from (128,242).

The procedure consists of two sub-rs. The first one, called 'Next Doc', takes the document address detail from (2/4,196) and incremernts it to point to the next document. This has been separated off as an independant sub-r because it is likely that other routines would find a use for it.

The second, called 'Test Doc', is the test routine. It uses 'Next Doc' to locate each document in turn, and tests each to see if it matches the required one. The variables have been allocated as follows:

```
50176   (0,196)   Required
50177   (1,196)     serial number
50178   (2,196)   Last tested document
50179   (3,196)     address.
50180   (4,196)   Last tested doc block No ; ( 35 to 57 ).
```

Next Doc

```
ld hl(50178)     42  2  196      Add 128 to last addr
ld de 128        17  128  0        and test to see if
add hl de        25                result > highest
ld (50178)hl     34  2  196        permitted doc addr
ld de H H        17  129  127      of (128,127).
or a             183
sbc hl de        237 82          If it does not then
ccf              63                finish with
ret nc           208             Cy reset.

ld hl 0 64       33  0  64       Else put (0,64) as
ld (50178)hl     34  2  196      1st addr in next block
ld hl 50180      33  4  196        and increment the
inc (hl)         52                block number.
ld a (hl)        126             But if block number
cp 58            254  58   *      now exceeds max
ccf              63                then finish with Cy
ret              201               set, else reset.
```

'Next Doc' works by adding 128 to the last document address. If the result exceeds (128,127), which is the highest address at which a document could start, then (0,64) is put in as the next doc address because that is the first address of the next block, and the block number is incremented. If the block number now exceeds the maximum for the machine [✱ : 57 for the '512' models, or 41 for the '8256'], then all the documents must have been examined and so Cy is returned set. Otherwise Cy is returned reset. The proper adjustment of Cy is achieved in both parts of the sub-r by complementing it with 'ccf'.

'Test Doc'

Because the first action of the testing sub-r will be to increment the doc address, the program is initialised by putting the [first address - 128] into the variables. Each doc is then tested in turn until

either 1. 'Next Doc' returns Cy set, in which case 'Test Doc' terminates, also with Cy set.
or 2. The two serial numbers correspond, in which case the document is copied, the TPA is restored, and Cy is reset.

As with all example programs, this is by no means the fastest way of dealing with the task, but examples are there to illustrate principles, not to show how clever the author is. In a real life situation, 'Test Doc' would be able to test for correspondence with a much wider range of parameters than just the serial number by testing for compliance with a 'mask' that had been composed through the use of a screen questionaire.

```
'Test Doc'
        ld 128 63         33  128  63     Initialise 'last' doc
        ld (50178) hl     34  2  196      addr = '(0,64) - 128'
        ld a, 35          62  35            and block as
        ld (50180)a       50  4  196        No 35.
Next:
        call NEXT_D       205  N  N       Point to next doc and
        jr nc OK          48  7             jump on if no carry.
        ld a 1            62  1           Otherwise restore
        call MEM_M        205  33  253      the TPA,
        scf               55                set Cy,
        ret               201               and finish
```

continued on next page . . . .


```
OK:     ld a(50180)       58  4  196      Switch-in
        call MEM_M        205  33  253      this block.
        ld hl(50178)      42  2  196      Doc start-addr in HL
        ld a(50176)       58  0  196      LB of No into A
        cp (hl)           190             If requ LB and doc LB
        jr nz Next        32  229           not same try next
        inc hl            35              Point HL & A to
        ld a (50177)      58  1  196      HBs and if these not
        cp (hl)           190               the same then
        jr nz Next        32  222           then try next.

        dec hl            43              Else transfer
        ld de 62080       17  128  242      the document
        ld bc 128         1  128  0         to the
        ldir              237  176          holding area.
        ld a 1            62  1           Restore the
        call MEM_M        205  33  253    TPA.
        or a              183             Reset Cy
        ret               201               and finish.
```

Finding the Cursor Position

Bank 0 contains a sub-routine at (191,0) called 'TE_ASK' ('TE' = "terminal emulation"). A call to it will put the current cursor position into HL, so if your progran needs to know this you can use:

```
        xor a             175             Zero into A
        call MEM_M        205  33  253    Switch to bank 0
        call TE_ASK       205 191  0      Call routine
        ld (addr) hl      34  A  A        Save cursor posn
        ld a 1            62  1           Restore
        call MEMMGR       205  33  253    TPA
        ret               201             Finish
```

Chapter 12

Disc Handling

There are three stages in bringing a disc file into existence ; the file is first Created, something is written in to it, and then it is Closed. Once these three operations have been completed the file is ready for use. To use it you Open it, use it, then Close it again.

'Create'

The BDOS function to create a file is No 22. To use it you first need to load DE with the address of your chosen 'File Control Block'; the 'FCB'. The FCB is a 36-byte area of memory into which you put the description of a file that you wish to manipulate. The data needed in the FCB for creating a file is as follows:

Bytes No	Data
0	Drive Number (A: = 1, B: = 2).
1 to 8	The file name (8 chars).
9 to 11	The file type (3 chars).
12 to 35	All set to zero.

The first byte receives the drive number in which the disc is waiting. 1 = drive A: , 2 = drive B: , etc . . . through to 16 which refers to drive P: , though I doubt if many of us will have one of those. A zero in

byte No 0 specifies the 'default' drive, whichever that may be at the time. (A 'default' value is one that the computer ascribes to something in the absence of a specific instruction from the user.)

On the '8256' and '9512' there is only one disc drive so only '1' (ie. drive A:) really applies, though you can use '0' if you like because A: is automatically the default drive as well, and I am told that a '2' also works because it gets translated to a '1' . On the '8512' you can use '0', '1', or '2', to select between the drives available. However, for any machine if you use a non-recognised drive number you will get a CP/M error message and be returned to the "A>" prompt, thus losing contact with your program (but see Chapter 13).

The next 8 bytes are for the ASCII codes of the filename in UPPER CASE (capital letters). If you don't use upper case then CP/M will not be able to find your file again when it looks through the directory for it. The name may not be more than 8 characters long, and if it is less than 8 characters then the remaining bytes must be filled with 'spaces' the ASCII code of which is 32. If you want CP/M to co-operate with your filenames, don't use the characters listed on page 2 [364] of the manual (CP/M section).

The next 3 bytes receive the ASCII codes of the file-type, again in UPPER CASE. When file names are written out in full, as in the CP/M 'pip' command, for example, the file-name and the file-type are separated by a full-stop (.), but this should *not* be included in the FCB. You can make the file-type anything you wish, but certain letter combinations have special significance and it is better to avoid them except when the significance is intended. The special types include COM, SUB, ENG, BAS, REL, ASM, EMS, SYM and WP. In particular use COM only when you mean it.

The remaining 24 bytes of the FCB are reserved for use by CP/M for storing information during the creation of the file. They should all be zeroised before calling the 'Create' or 'Open' functions and not changed subsequently. (For a quick method of making FCBs see page 123 and Appendix 10.)

Suppose I want to create a file called 'MC.COM' on a disc in drive A:, and that I already have in memory starting at address 50100 (180,195) a string made out with these letters and spaces, which would be;

77 67 32 32 32 32 32 32 67 79 77

and that I wish the FCB to be at address 50176 (0,196). A mini routine for creating such a file could be as listed below.

After the create function has been called, it is possible to check on its success by inspecting A. If A contains 0 to 3 then the create was successful. If it contains 255 then the create was unsuccessful, probably because the Disc Directory was full and no more entries could be written into it. The simple test is to increment A. If this sets the Z flag then A must originally have contained 255 and an error handling procedure should be called. (See Chapter 13.)

```
Prepare FCB:
        ld a, 1          62  1          Insert the
        ld (FCB), a      50 0 196        drive No into byte 0
        ld de FCB+1      17 1 196       Copy the
        ld hl STRG       33 180 195      string
        ld bc, 11         1  11  0       into
        ldir             237 176         bytes 1 to 11.
        ld hl FCB+12     33 12 196
        ld b, 24         6 24           Zeroise
        ld (hl) 0        54 0            the
        inc hl           35              remaining.
        djnz             16 251          bytes.

Create the file:
        ld c FNUM        14  22         Fnc No into C
        ld de FCB        17 0 196       Point DE to FCB,
        call BDOS        205  5  0       and actuate.
              continue . . .
```

You should not attempt to Create a file if one of that name already exists on the disc. If you do you will corrupt the disc and may not then be able to read anything useful from it. If there is any doubt, use the 'Delete' function to erase any such that may exist before creating a new one. Attempting to erase a non-existent file puts 255 into A but causes no problems. See page 114 for a program example.

Quick fix for FCBs

I am indebted to Mr Johs Lind of Denmark for drawing my attention to fnc No 152, called 'Parse File Name', which is a quick and easy way to set up a zeroised FCB. See Appendix 10 for details.

'Open'

BDOS function No 15, 'Open File' is performed exactly as is the Create function, but it is applied to existing files to make them accessible for reading or for having more data written into them. You should not attempt to manipulate a file that has not been recently either Created or Opened, nor change the FCB until the file has been closed again. If the file is found and successfully opened then A is returned containing 0 to 3. If it cannot be found then A will contain 255 which can be tested for by 'inc a' as indicated above.

'Wild Cards'

The use of 'wild-card' characters in file names is described in the CP/M manual on page 8 [370] and later, and a similar procedure can be used from m/c. The asterisk '*' cannot be used, but if a '?' (ASCII 63) is inserted at any place in the file-name or file-type then any character will be regarded as a match for it. The following table shows which functions can be used with wildcards :

YES	NO
15. Open	22. Create
16. Close	23. Rename
19. Delete	

'Write' and 'Read' use the existing FCB which will already contain the wildcards if any.

Memory address

To write bytes from memory into any sort of file, or to read bytes out of one into memory, we have to use yet another set of initials (sorry). 'DMA' stands for 'Direct Memory Access', which harks back to ancient computer times and therefore means nothing nowadays but nobody wants to change it. The term 'Set DMA address' means 'Tell CP/M the address of the piece of memory we are interested in'. In this context a piece is always 128 bytes, and such a piece is called a **record.**

Setting the DMA address is achieved by calling BDOS function No 26 with DE pointing at the address concerned.

When a file is being read from, 128 bytes will be transferred from the disc into the piece of memory that starts at the DMA address. When a file is being written into, 128 bytes will be copied onto the disc from the piece of memory that starts at the DMA address. This applies to all varieties of files.

'Close'

After any sort of file has been accessed it must be 'Closed' before the FCB is used for other purposes and before the computer is turned off. 'Close' is achieved by using BDOS function No 16 with DE pointing to the FCB that was used to create (or open) and process to the file. A successful Close is indicated by A containing 0 to 3. A value of 255 indicates no success, probably because BDOS couldn't find the named file.

The purpose of the Close operation is to update the disc directory with the new details of the file. If you don't get a successful Close then the file contents will not match the disc directory and it will probably be impossible to access it properly in future.

File types and Kinds of files

There is a distinction to be made between 'file type', and 'kind of file'.

The file *type* is indicated by the three letters that follow its name. Files that contain text (which are sometimes given the type letters 'TXT') will contain solely ASCII- and print control codes. Data files (sometimes given the type 'DAT') such as personnel, accountancy, or stores records, will probably hold a mixture of numerical and string data. Whatever may be conventional, you can give these two sorts of file any file-type you like.

COM files form a special group; 'COM' stands for 'command' because the filename can be used as a command to the computer. Because they are to self-run they must contain at least one m/c routine and the strings and data they need for their operations, and they must have the file-type 'COM' or they won't do what is expected of them.

There is more than one way of writing into or reading from files, and it is this that determines what *kind* it is. (See the manual page 55

vol 2 [502] for a description of the different kinds of file.) As far as we are concerned there are two kinds: the **sequential** access and the **random** access.

Whatever kind or type it is, once the file has been created or opened data can be written into it. What is written in will be a sequence of bytes. The bytes can represent anything you like. The file-making process is indifferent to the contents, and all of the 'write' instructions will copy into the file whatever set of bytes you have pointed to.

The first kind we will consider will be sequential files, then random access files. Next we will look at the process of making back-up files, which applies to all types and kinds of file, and at the end of the chapter we will examine the special usefulness of .COM files.

SEQUENTIAL FILES

When writing into sequential files, CP/M takes each batch of bytes and puts them into the file in sequence. Once the sequence has been established it can't be changed. If you want to modify it you

1. copy the data into memory and change it there.
2. delete the original file.
3. make a new one of the same name and put the new data into it.

Write sequential

Once the DMA address has been set it is possible to write the first 128 bytes into file from the DMA address by loading DE with the address of the FCB that was used to create or open the file, and then calling 'Write Sequential'; BDOS function No 21. If the Write operation is successful then on completion A will contain zero.

If A contains a non-zero value then the write was unsuccessful, probably because the disc or the directory is full, or the FCB is invalid. The simplest way of checking this is to use 'or a'. A zero value in A (indicating success) will set the Zero flag.

Read Sequential

Reading from a sequential file is similar to writing to one. The function number is 20. Reading should be attempted only with files that have been Opened, and the FCB used in the Open should be used in the Read. One Read operation copies 128 bytes from the disc into memory starting at the current DMA address.

A successful Read returns zero in A. A non-zero value indicates that it was unsuccessful, probably because an attempt to read beyond the end of the file has been made (A=1), or the FCB is invalid (A=9). Reading into memory a number of 128-byte records can be done in the way described for Write Sequential (see below), or continuing to read until A is found to hold the value 1.

Summary

These procedures may seem long-winded but in fact they flow quite naturally once you have seen the reason for each step. The sequence for writing to a file can be summarised as below (function Nos in brackets), and an exactly similar procedure is required for reading from one:

1. Prepare the FCB with the Drive No, the file name, the file type, and zeroes in the remaining bytes

2. Either Create (22) a new file or Open (15) an existing one by pointing to the FCB with DE and calling the appropriate BDOS function.

3. Point to the section of memory that contains the bytes you want to write to the file by setting the DMA address (26).

4. Write the bytes into the file by pointing DE at the FCB and calling the BDOS function 'Write Sequential' (21).

5. Close the file by pointing DE at the FCB and calling the BDOS function (16).

It may help if you remember that every time you want to tell CP/M which file you are referring to, you point DE to the FCB that describes it. Remember also that BDOS corrupts the registers so it is

necessary to re-load them with their required contents after each use of BDOS.

Making larger files

The above sequence describes the creation of a file containing only one 128-byte record. In fact it is usual to want to write much more than this so it is necessary to repeat the 'Set DMA address' and 'Write Sequential' functions several times.

Suppose we find that we have 2197 bytes that need to be written into a new file. This corresponds to 17 records plus 21 bytes left over. The smallest block that can be copied is 128 so the 21 bytes get a whole 128-byte record to themselves, making 18 records in total. (The 107 bytes beyond the end of the data will be copied onto the disc as well.)

To achieve the copying we go through the initialisation of the FCB and Create the new file. Then we point DMA to the start of our bytes and order the 'Write sequential'. This takes care of the first lot of 128 bytes.

We then add 128 to the first DMA address so that it now points to the second lot of 128 bytes. We then set the new DMA address, and order 'Write Sequential' again (which automatically selects the next part of the disc to write to). This is repeated 16 more times until all 18 records have been pointed to and copied. Then we Close the file.

The tally of 128-byte records and the DMA address cannot be kept in registers because they would be corrupted each time BDOS was called, but the listing below illustrates a routine that might be used for this purpose. It 'pushes' the count and the address onto the stack and reclaims them later by 'pop'. It assumes that the FCB has been prepared as indicated at the start of the chapter, and that the first address to copy data from is 'ADDR'.

```
ld de FCB      17  F  F      Cancel any
ld c DELETE    14  19         file of
call BDOS      205  5  0      the same name.
ld de FCB      17  F  F      Create
ld c CREATE    14  22         a new
call BDOS      205  5  0      file.
```

continued on next page . . . .

```
        inc a          60            If A = 255 then signal
        jp z ERR1      202 E1 E1     'Directory Full'.

        ld de ADDR     17  A  A      First addr of DMA.
        ld b, 18       6  18         Count of records.
Loop:   push bc        197           Store count,
        push de        213             and DMA address.
        ld c, SET      14  26        Set the DMA to
        call BDOS      205  5  0       the addr in DE.
        ld de FCB      17  F  F      Point to FCB
        ld c WRT       14  21          and write a
        call BDOS      205  5  0     128-byte record.
        or a           183           If A ≠ 0 then report
*       jp nz ERR2     194 E2 E2     'Disc Full'.

        pop de         209           Recover last DMA
        ld hl 128      33  128  0      and
        add hl de      25              add 128.
        ex hl de       235           New addr into DE.
        pop bc         193           Recover record count
        djnz Loop      16  228         and loop if not zero.

        ld de FCB      17  F  F      Point to
        ld c CLOSE     14  16          the FCB
        call BDOS      205  5  0       and Close file
        inc a          60            If A= 255 then rep
        jp z ERR3      202 E3 E3     'File not found'.
              continue . . .
```

If an error occurs you will at least wish to signal 'Disc Full', or whatever, so that the operator can deal with the situation, but after the error-handling you don't want to return to the address after the place where the error occurred so CALLing an error routine is not appropriate: you need to *jump* to it, and from there go back to the main 'Menu Routine' (see page 120). Also note that in the above listing there are two 'push' instructions outstanding (ie. two not cancelled by a 'pops') if the jump to 'ERR2' is made (*), and the error-handling routine must account for this. (See Chapter 13 and Appendix 6.)

RANDOM ACCESS FILES

So far we have considered only sequential files in which you couldn't make changes except by deleting a file and replacing it by an up-dated version. However BDOS function Nos 34 and 40 allow us to write directly into a file to replace any of its records, and function 33 allows us to extract any record from it for inspection.

Function No 34 is called 'Write Random'. (In fact these files have nothing whatsoever to do with randomness but I suppose I'll have to swallow my pedantry and stick to established nomenclature.) The name attempts inadequately to imply that you can write a new 128-byte record into the file at whatever point you select (which is obviously therefore not random).

Having Created or Opened the file and set the DMA address, you put the required 16-bit record number (0 to 65536) into bytes 33 and 34 of the FCB, and zero into byte 35. You then call function No 34. If the file already contained a 128-byte record corresponding to the number you chose, then the data in it will be overwritten by the new insertion. If it did not contain one then the function automatically extends the file to provide one, and then fills it. If originally, say, records Nos 0 to 10 existed and you request Write Random into, say, No 16, the file will be extended to include No 16, records Nos 11-15 being full of garbage.

Function No 40 is called 'Write Random with Zero Fill'. It does what fnc No 34 does but first fills the record with zeroes. This is really good if you are faced with 'sparse files' (whatever they are), but is a waste of time otherwise.

Function No 33 is called 'Read Random', and acts in reverse to 'Write Random', ie. it takes a selected record out of the file and puts it into memory starting at the DMA address. As before, you specify the record number in bytes 33 and 34 of the FCB, and set byte 35 to zero.

Using Random Access is particularly convenient if you are dealing with large files that might require an appreciable time to read from and write into because of the amount of data to be shifted. Random access involves only 128 bytes at a time which can be transferred in a second or two, and also greatly assists with the structuring of files because the location of data is not changed by subsequent additions or cancellations.

BACK-UP FILES

'Rename'

BDOS function No 23 allows the names of files to be changed. To achieve this an FCB is set up to contain the description of the old file with zeroes in its first 16 bytes, plus the new name and zeroes in the second 16 bytes.

The drive code No in byte No 16 should be zero ; the drive code No in byte 0 is set to the drive of the disc in question. Hence the contents of the FCB bytes are as indicated below.

Byte Nos	Content
0	Drive code No
1 to 8	Existing file name
9 to 11	Existing file type
12 to 16	All zeroes
17 to 24	New file name
25 to 27	New file type
28 to 35	All zeroes

As usual DE points to the FCB, C is loaded with the fnc No, and BDOS is called at 0005h. A successful Rename is indicated by A containing 0 to 3, an unsuccessful one by A containing 255. The new file name must not be already in use, and this function will not accept 'wild-card' letters.

Back-up files

A useful application of Rename is in making back-up files. In the normal way of things if you take the content of a file into memory so that additions can be made to it, it will be necessary to Delete the old file before Creating the new one under the same name. If a problem arises after the erase has occurred but before the new one has been recorded (a power failure, say, or someone trips over the wires), or if the recording is unsuccessful and you switch off without realising it, then you will have lost all your data.

This can be avoided if instead of erasing the old file it is given a different name. The new file can then be Created under the required name. The change in name is usually slight so that the connection between the two files can be seen at a glance; 'MC.COM' might be become 'MC2.COM', for example. And if you are handling very important files you might choose to introduce another layer of backup; 'MC2.COM' being renamed as 'MC3.COM', before renaming 'MC.COM' as MC2.COM'.

I use one layer of back-up in developing programs. When I press the SAVE key this is programmed to delete the existing backup, rename the present main file as the new backup, and then record the new main file. With care over the detection and reporting of errors this sequence is as safe as I expect to need. To achieve it I have in memory a 'File-name String' that is separate from the FCB and is therefore not altered by the file-handling operations. It is made up as if for Rename as indicated on the previous page. It is used in the the 'SAVE' process as follows

1. To delete the old back-up file, 'ldir' the second half of the string into the FCB, set the Drive No, zeroise the rest of the FCB, and call Delete.
2. To rename the old file as backup, 'ldir' the whole string into the FCB, and call 'Rename'.
3. To create the new file, 'ldir' the first half of the string into the FCB, zeroise the rest of the FCB, and call 'Create'.

Temporary backups are often given the distinctive file-type '.$$$', and some renaming is done by changing the file-type to 'BAK', though I find this less convenient for backing-up a COM file because a 'BAK' file won't self-load until you've changed it to a 'COM' again.

The advantage given by the backups is that although you can still lose one version of the file, it is hardly credible that they will all bite the dust together unless you mutilate the disc, for which mishap few would hold the programming to be culpable.

MAKING AND USING .COM FILES

The beauty of a .COM file is that you have only to type its name and it will load itself into memory and then proceed to run untouched by human hand ; a bit like a genie being summoned from its bottle by a magic word. (Though so far I have had no luck with genies.)

The usual method of using CP/M is to wait for the 'A>' prompt and then enter the name of the .COM file that you want to use. One such name could be 'basic', because Locomotive Software wrote their BASIC into a file to which they gave the name 'BASIC.COM'. When the first part of such a name is typed in, CP/M scans the disc directory for a .COM file that matches it. If one is found it is copied into memory starting at address 0100h ie. (0,1) by the **Console Command Processor** (CCP), which will automatically have been inserted into memory for that purpose.

If you try to load a file that is so long that it would encroach into the upper area needed by CP/M then you will get the report "Cannot load" without any elaboration. (The maximum acceptable file length is just over 60k, ie. approx 480 records.) When loading is complete, the CCP transfers operation to the new program by making a jump to 0100h. Whatever instruction is found at 0100h initiates the new program's sequence of operations.

This gives the basis for writing files by m/c and operating them. Provided the first m/c instruction in a .COM file gives us control then we can do anything we like from there. Hence it is typical for the first section of the file to be devoted to some kind of 'MENU' program which halts operations whilst the user selects from alternatives, or inputs some data. Even if this layout has not been adhered to for some reason, its effect can still be obtained by putting the three bytes of a 'jump to MENU' instruction as the first three bytes of the program, and because I always like to leave options open even when the program's needs seem cut and dried, I like to leave the first ten or so bytes unused (zeroised) so that such an instruction can be inserted later if that is found to be necessary.

The Menu program

A typical Menu program to be found at the beginning of a .COM file would start with a 'Print String' instruction. This would load DE with the address of a menu-page string, put 9 into C, and then call BDOS. A typical page-string would start with 'clear screen' (27 69), then display a title (probably underlined), followed by a list of the options available to the user against each of which would be shown the key that is to be pressed to select it. The subsequent selection of

options would be based on BDOS function No 1 as outlined on page 61. The Menu Program would either start at (0,1) or be preceded by a few zeroes (not less than three), and look something like:

```
        ld de STRG     17  S  S       Print the
        ld c, 9        14  9          Menu
        call BDOS      205 5  0       String
Loop:
        ld c, 1        14  1          And await
        call BDOS      205 5  0        a keypress
        cp K1          254 K1         Compare
         jp z  OPT_1   202 L1 H1       (A) with
        cp K2          254 K2          each of the
        jp z  OPT_2    202 L2 H2       allowed
        cp K3          254 K3         ASCII codes
        jp z  OPT_3    202 L3 H3       and jump
        ...            ...             to the appropriate
        ...            ...             program when
        cp Kn          254 Kn          a match
        jp z  OPT_n    202 Ln Hn       is found.

        ld e BEL       30  7          If no match is
        ld c, 2        14  2           found then sound
        call BDOS      205 5  0        the 'beep' and
        jr Loop        24  L           loop back to repeat
```

In the listing, K1, K2 . . . Kn, are the ASCII codes of keys that are authorised for the user to press to select a menu option. If he doesn't press any key then nothing will happen, and if he presses an unauthorised one then a 'beep' will sound and the program will loop back to await another choice, though there is no need to include the 'beep' feature if it is not wanted. (Most of us don't care for being beeped at, however justified it may seem to the programmer.)

Testing .COM files

Note that a .COM file is always loaded into memory at 0100h (0,1) regardless of the address of its own manufacture, and this has to be taken into account while it is being written ; the addresses of all its internal calls must apply to this low area of memory not to the addresses that apply during its assembly. The snag here is that you cannot try out a program until it is in the final 'COM' version and in position at low memory, and this will complicate the testing process if

you have no testing aids available there (but see Appendix 9).
To overcome this you might write the sub-routines as if they are to operate at the addresses at which they are assembled, but make sure that the Low Bytes of addresses are all the same as they will be when the program is transferred to 0100h. This applies to addresses of the Variables etc., as well as to all 'jump' and 'call' addresses.

Then, immediately prior to making the COM file, and after all testing has been carried out, institute a 'Search and Change' procedure that runs through the program subtracting the necessary fixed amount from the High Bytes of all addresses. The 'Search and Change' is particularly easy to organise if you are assembling with BASIC in place. On one occasion of assembling at high memory because BASIC was in place, I made a point of starting everything from address 51456 (C900h) because this has a red-biro equivalent of (0,201), whereas in the COM file version everything would start from (0,1). Hence all addresses would undergo the simple transformation of having 200 subtracted from their High Byte.

On the subject of testing; you should never pass on from writing a sub-routine, particularly one to calculate numerical results or data addresses, until you have certain evidence that it is doing its job properly. (Yes I know "There is no way in which this one can fail", but don't you believe it.)

The first COM file I ever wrote was called "bel". When you typed its name it loaded itself and sounded two beeps in quick succession. The fact that it worked stunned me into a silence of wonderment, and in the next five minutes I almost wore out the disc with action replays. If it had successfully launched a new Mars Probe I doubt if I would have been any more impressed. I hope all yours are as successful, and that at least some of them achieve a bit more !

MISCELLANEOUS FILE CONSIDERATIONS

A bit more about the DMA address

It is possible to write into or read from a file without having set the DMA address, but then the default location will be 0080h (128,0), so the data area must be the 128 addresses starting from there. When the computer is switched on, the DMA will have this default address, but once you change it it retains the new value until you change it again or you reboot the system (see page 126). Because it is rare to copy only 128 bytes, it is better to think of setting the DMA address as a normal part of file handling.

Assessing Disc Free Space

The amount of free space on a disc in a particular drive can be assessed by using function No 46, though it is necessary first to ensure that that drive is 'logged-in'. The following sub-routine has two alternative starts. Starting at 'Start 1' gives the free space of the disc in drive A:, starting at 'Start 2' gives the same for drive B: If your machine has no drive B:, then you can leave out the 2nd and 3rd operations (ie. omit "24 2 30 1") and always use 'Start 1'.

The first part of the sub-r loads E with the number of the drive we are interested in and stores this by pushing DE, but notice that in this case 0 = A:, and 1 = B:, which is different from the numbering used when constructing an FCB. It then uses function No 13 to reset all the drives, which has the simultaneous effect of resetting the DMA address to (128,0) ie, 0080h.

```
START_1:
        ld e, 0         30 0        Start for drive A:
        jr 2            24 2         and jp to 'push DE'
START_2:
        ld e, 1         30 1        Start for drive B:
        push de         213         Save the value in E.
        ld c 13         14 13       Reset the drives
        call BDOS       205 5 0      and the DMA address.
      * pop de          209         Recover drive No to E
        ld c, 46        14 46        and call
        call BDOS       205 5 0      the function.
        ld hl (DMA)     42 128 0    Collect space in HL
        ret             201          and finish.
```

At * the drive number is put back into E by popping DE, and function No 46 is called. The disc free space is given as a 24-bit number in the first three bytes of the DMA, which we know is now at (128,0). As the result represents the number of unused 128-byte records, it is unlikely that you will have a disc that contains more than 65535 of them (!), so you can ignore the high-byte, and take the result from the low- and middle-bytes. In the listing this is loaded into HL before returning to the main routine

Chapter 13

Error Handling

A computer 'error' can be defined as any unexpected or unwelcome event, and in computing if it is unexpected you can be pretty sure it will be un-welcome.

Errors come in three broad kinds:

1. Those that direct operations to an undesirable place within your program.
2. Those that lead operations outside it by returning to CP/M or to BASIC.
3. Those that prevent, or cause screwed-up, input from the keyboard.

In all three cases you will have lost control of what happens next.

TYPE 1 ERRORS

Typical of a type 1 error would be if a disc were filled during a Write-sequential operation but the sub-routine continued to try to write bytes onto it, thus corrupting it and possibly making all its data inaccessible. Obviously such a situation should not be allowed to occur, and in this case it can be prevented by testing the content of A; the discovery of a '255' should lead the program out of the operation into an 'Error Handling Routine' that would be available to all sub-routines. This should perform at least the following duties:

1. Warn the user that something untoward has happened, and tell him what it is,
2 Await a key-press,
3 Direct operations back to a safe 'restart' location.

It could be designed also to indicate the program address at which the error occurred and this could be helpful to a programmer if there would otherwise be some doubt about it.

To establish the error-handling sequence, the first action of the main program should be to load the content of SP into some chosen address; call this 'ADDR'. This records the stack situation in the pre-start 'all clear' condition. Then, when the error has occurred, just before jumping to the error-handling routine, A is loaded with the number of the error. This tells the routine which error-message to display on the screen. A typical list of error messages might contain:

0 Memory full
1 Disc full
2 Directory full
3 List 'X' full
4 Failed to Erase
5 End of File
6 File not found
7 Code 'X' not found etc...

Naturally the contents of the list will depend on the sort of program being run. The general method of message selection and display is described on page 69, so a simple error handling routine might list as follows:

```
ld hl LIST      33  L  L       Point HL to list of
call PRTM       205  P  P        messages, print the one
ld c, 1         14  1            pointed to by A and
call BDOS       205  5  0        await keypress.
ld sp (ADDR)    237 123 A A    Restore stack pointer
jp MENU         195  M  M      Return to main menu.
```

The reason for reloading SP with the stored value is that this automatically cancels any unfinished business with 'calls' or 'pushes' that may have been short circuited by the jump to the error handling routine. (See Appendix 6.)

TYPE 2 ERRORS

When the m/c routine has been called from BASIC, and an error causes a premature return to it, there is not much that can be done to retrieve the situation. BASIC has its own error handling arrangements (see 'On Error Goto'), but these will be of little use because BASIC will be awaiting a command or running the next bit of program, and be unaware of the m/c error. It is therefore as well to put in a BASIC line that tests to see if the m/c routine has run to completion, rather than just assume that it has. You could, for example, arrange for the last part of the m/c routine to change the value of a variable that has no other use, or to change the value in a reserved memory address, and on return check that this has been done. (See pages 172 and 360 of Vol 2 [509] of the Manual.)

A premature return to BASIC will probably be due to the presence in the m/c routine of an unwanted 'ret', or to an incorrect change in SP causing a valid 'ret' to return the address at the bottom of the stack instead of to the one it was supposed to return to.

The same possibilities exist if the m/c program has been derived from a .COM file (ie. BASIC is absent), except that a premature return will give rise to the 'A>' prompt because operations will have been handed over to CP/M. In this case, in addition to spurious 'rets', any circumstance that gives rise to a 'Warmboot' will provoke the 'A>' prompt. Whatever the cause, once the prompt has appeared you will be unable to get back to your m/c program without switching off the machine and starting again. That guarantees loss of any data that might have been accumulated whilst the program was running.

CP/M Warmboot

When the machine is switched on it performs the so called 'Coldboot'; ie. it loads the necessary system-programming and sets its system variables ready for operations to begin. 'Warmboot' is the name given to a subsequent restart that doesn't involve switching off. 'Warmboot' doesn't reload the CP/M program but it does reset all system variables to the 'coldboot' condition, and then expects you to input a CP/M command. If a type 2 error has occurred this will have lead to a 'warmboot', but your program will still be in memory though you won't be able to get back to it because there is no CP/M though you won't be able to get back to it because there is no CP/M command that provides for this.

All 'boots', fur-lined or otherwise, involve a jump to 0000h, at which is to be found;

jp 3 252.

At (3,252) is to be found;

jp 111 252,

and at (111,252) is to be found a complex set of instructions that do all sorts of abstruse and wonderful things.

It is therefore possible to prevent CP/M warmboots by changing the jump instruction at 0000h. If, once your program is installed and running, it changes the address at 0001/2h, you will be able to redirect all warmboot attempts to your own warmboot procedure, and thus maintain control. This isn't something to be done once the program has been given over for civilian use because the address is used for other purposes also (see page 83), but it is a handy tool for a programmer desparately trying to work out the reason for his program's *kami-kazi* tendencies.

CP/M disc error procedures

Some disc errors may also cause warmboot and loss of control. An example might be that the user inadvertently makes some inappropriate disc request, to which CP/M might respond:

"Drive not ready: Cancel, Ignore, Retry ?"

If a proper disc could be inserted followed by 'Retry', then everything would be fine. If no such disc were available then a return to CP/M would be inevitable because 'Ignore' has no effect and 'Cancel' acts like that. Just to be helpful, CP/M will give you the additional information:

"CP/M Error on A: Disk I/O
BDOS Function = 15 File = FRED.DAT
A>"

though you may feel that that is little consolation.

Fortunately this arrangement can be modified by BDOS function No 45, which is called 'Set BDOS disc error mode'. The function requires that an error-mode be put into E before calling it. The error-modes are;

0 to 253:	Error message displayed followed by warm boot (the normal arrangement; the default setting is 00h).
254:	Error message displayed but no warmboot.
255:	No message and no warmboot.

Error-modes of 254 and 255 can therefore be helpful in maintaining control in case of errors of this type, though naturally you have to put in some alternative procedure of your own.

TYPE 3 ERRORS

Type 3 errors are caused by 'bugs'. (And you know what people who insert 'bugs' are called !) The machine and the system sofware can be assumed to be faultless, so if you get lock-up or something equally uncooperative then it is almost certainly because your program has an error in it.

Normally bugs aught to be revealed by the tests applied to each of the sub-routines before they are linked up to form the program, but it can happen that a bug appears only after the program has been run. A sub-r may be putting a byte into an address that is harmless, and therefore unnoticed, during the tests, but one which has critical significance later when the program is in use. This highlights the need to keep a pure, un-run, copy of programs, particularly if they are complex and not easy to follow through. Once unintended bytes get into a listing they can cause ever increasing corruption of what is supposed to be there until it is quite impossible to trace the original source of the trouble. If you have a good version you can keep copying it to make tests.

Once a bug has entered a program, it can be detected only by 'homing-in' on it, ie. by testing the program up to more and more advanced stop-points. This should show the earliest place in the sequence that the bug operates at, and make discovering it relatively straight forward. If you think you know what is causing the trouble but have not yet made an ordered search, don't persist with your notion too long. I have occasionally dug myself into ever deepening holes by making changes 'that are bound to solve the problem' when what I aught to have been doing was working through a patient and orderly enquiry.

Recording results in memory so they can be inspected later (as indicated in chapters 5 and 6, and elsewhere) can make bug hunting very much easier. If a result isn't coming out right that won't be due to bad luck, it will be because something is wrong with the programming that produces it, or with the programming that feeds data to help in its calculation.

Programs frequently have a need to to store newly produced data onto disc at the end of each keyboard session. In these cases it is **vital** to separate the operating program from the data so that only the latter is involved in the recording. If a bug should get into a program during a period of use, the last thing you want to do is to record it for posterity. Information Theory makes use of a notion very similar to Entropy - "Orderliness never gets any more orderly ; downhill is the only way it knows." We accept the first idea without a murmur, but we need our wits about us to forestall the second.

KEYING ERRORS

There is a particularly exasperating source of all three types of errors ; that of the unintended keypress. The most irritating ones are the function keys (f1 to f8) that occupy dangerous ground between SHIFT, RETURN, DEL etc, and the numeral pad. One of them in particular (I think it is f5) occasionally causes mayhem out of all proportion to its work-a-day worth.

The simple answer is to redefine the keyboard so that keys you have no use for have no effect when pressed. This is achieved by creating a file to use with the CP/M 'setkeys' function. The procedure is described on page 108 [541] *et seq* of the manual. Through 'setkeys' you can instruct any key to produce any 'ASCII code' you wish. The code for 'Don't do anything' is 159.

Your file can then be operated automatically at start-up by referring to it in a line in a 'profile.sub' file.

And if you are not familiar with 'profile.sub', I recommend that you take as long as necessary to swot it up. It allows programs to self-load without any keypressing. I wouldn't be without it.

Chapter 14

Arithmetical routines

Displaying numbers

Almost every program needs to display numbers on the screen or through the printer. In developing a routine for this we will limit our consideration to numbers up to 65535, though the priciples described can be extended to numbers of any size.

When a number is in the computer it will be in 1- or 2-byte form and this must first be converted to decimal digits. This could be accomplished by dividing it first by 10,000 and using the resulting integer as the first digit, then multiplying the fraction by 10, etc., etc.

However my choice is to use repeated subtraction first of 10,000, then of 1000, then of 100, then of 10, thus leaving the units as the remainder. This avoids the errors that would inevitably be introduced by the rounding of the inexact results of divisions. It is also faster for small digits than multiplication, though maybe slower for large ones.

When the decimal digits have been calculated they need to be converted to the ASCII codes of their numerals so that the numerals can be printed. Inspection shows that the ASCII code for "0" is 48, for "1" is 49, for "2" is 50, etc. Hence the ASCII code is obtained by adding 48 to the digit, and this can be done as part of the calculational procedure. The program is in two parts, one as a sub-routine called by the other. I have called the sub-routine "Calcdig" ; separating it off avoids unnecessary repetition of the same code sequences for each digit.

The resulting ASCII codes will be stored in the Variables area, which I will assign to page 200 [ie., it starts at 51200 ; (0,200) ; C800h]. Naturally you will put yours where it is most convenient.

```
51200  (0,200)    27    DEFB
51201  (1,200)    89    DEFB
51202  (2,200)    ln    print-line No
51203  (3,200)    col   print-colm No
51204  (4,200)    -     ASCII of Ten Thousands
51205  (5,200)    -     ASCII of Thousands
51206  (6,200)    -     ASCII of Hundreds
51207  (7,200)    -     ASCII of Tens
51208  (8,200)    -     ASCII of Units
51209  (9,200)    255   DEFB ( delimiter )
51210  (10,200)   Lo    Number to
51211  (11,200)   Hi      be processed
```

In addition to the required ASCII codes, I have also inserted the bytes necessary to produce a 'print-position' string so that the results can be printed at any screen location. (See pages 70 and 71.)

The last two bytes are the Lo- and the Hi-byte of the number whose digits we want. It is not necessary to have this in memory for our present purposes, but you may have other reasons for wanting to store it. The main routine proceeds as follows :

```
Start1:
      ld hl(51210)      42 10 200     Collect number from vars
Start2:
      ld de 10,000      17 16 39      (Start if num already in HL)
      call 'Calcdg'     205 N N       Calc the Ten-thous
      ld (51204), a     50 4 200      Store the digit
      ld de 1000        17 232 3
      call 'Calcdg'     205 N N       Calc the thousands
      ld (51205), a     50 5 200      Store the digit
      ld de 100         17 100 0
      call 'Calcdg'     205 N N       Calc the hundreds
      ld (51206), a     50 6 200      Store the digit
```

continued on next page . . . .


```
      ld de 10          17 10 0
      call 'Calcdg'     205 N N       Calculate the tens
      ld (51207), a     50 7 200      Store the digit
      ld a, l           125           Put units into A
      add a, 48         198 48        Convert to ASCII
      ld (51208), a     50 8 200      Store the digit
      ret               201
```

In each case DE is loaded with the rank of the digit to be calculated (10,000 ; 1000 ; 100 ; or 10) prior to calling 'Calcdig'. 'Calcdig' returns the ASCII value in A which is then stored by the main program in the proper place in memory. At the end, the previous subtractions will have left the units in HL, ie. in L, so this is moved into A and there converted to the ASCII code before being stored. The sub-routine 'Calcdig' is as follows :

```
      xor a             175           Zeroise A & reset Cy
      inc a             60            Increase the count
      sbc hl de         237 82        Subtr digit rank in DE
      jr nc -5          48 251        Repeat if no carry
       add hl de        25            Else restore last subtraction
      dec a             61              and last count increment.
      add a, 48         198 48        Convert to ASCII
      ret               201           and ret to main.
```

The accumulator is to be used to count the number of 'hundreds', 'tens', etc., so it is first zeroised and the Carry flag reset by 'xor a'. A small loop now repeatedly subtracts the value in DE from what is left in HL, and A counts the number of subtractions. If the subtraction takes the result below zero then Cy will become set thus telling us we have gone too far. We therefore add D E back to HL once and take one off the count. The count in A is converted to the appropriate ASCII code and this is taken back to the main routine for storage by it. The ASCII codes. are now all in their proper sequence in memory ready for printing. (See chaps 7 & 8.)

Pseudo-random numbers

The term 'generating random numbers' means something like "outputting a sequence of numbers one at a time in such a way that:

a) the values all fall within specified size limits
b) a large set would contain a roughly equal frequency

of all the allowed members

c) there is no way of predicting a future value".

In practice the inconvenience of meeting all these conditions is too great and the last one is usually waived ; a set of 'pseudo-random' numbers being used instead.

These are not random at all ; on the contrary their sequence is entirely predictable though to a user they seem adequately 'mixed up', and, if there are enough of them, and if operations start at different places in the sequence at different times, then they appear to be random.

A set of pseudo-random numbers in the range 0 to 65535 will be generated if the following sequence of operations is performed each time a new one is required. Any member of the set may be used as the starting point or 'seed', and each new product acts as the seed for the next. The calculation is often performed on the floating-point forms (because the third operation cannot be performed in 16 bits), but 24- or 32-bit arithmetic can achieve the same effect more conveniently (see below).

Add 1
Multiply by 75
Extract MOD 65537 (divide by 65537 and use the remainder)
Subtract 1

Random numbers

There is no way in which a calculational procedure can output a sequence of truly random numbers from the PCW, though randomness, or rather 'unpredictability', can be extracted from human activity and coupled with calculation in such a way that the conditions stated above can be satisfied. In the following two examples the value stored at 'ADDR' has a constant added repeatedly to it, but the additions cease when a key is pressed. As both the current value and the time lag are unknown, the new value cannot be predicted. This is as adequate for all chance- or risk- simulations as die-rolling or coin-tossing would be.

Adding 13 rather than 1 reduces the risk of 'clumping' that is faintly conceivable if extremely short time lags should occur. Adding 13 or 1 gives an excellent spread to the results, but adding most other numbers does not. If the sub-r were to treat a high-byte and a

low-byte simultaneously by adding 13 to one and 1 to the other, then random values in the range 0 to 65535 would be generated. As listed, the sub-r gives values in the range 0 to 255. The cycle time is about 0.5 milli-seconds.

```
ld a,(ADDR)     58  A  A      Take current 'seed'.
add a, 13       198 13        Add 13
ld (ADDR),a     50  A  A        and replace in memory
ld c 11         14 11         Test for a
call BDOS       205 5 0         key-press
or a            183             ( see page 61 ).
jr z -16        40 240        If none then repeat addition
ret             201           Else finish.
```

Die throwing

To simulate die-throws in the range 1 to 6, the following sub-r limits the values that may occur in A and hence in the result. The simultaneous throwing of say three dice is best simulated by three independent calls of the sub-routine. Attempts to obtain three simultaneous die values invariably means that the values for the individual dice cannot independent of each other, though if the value for the first is derived as below and those of the other two from subsequent use of pseudo-random numbers then the dependence need not be noticeable.

```
ld a,(ADDR)     58  A  A      Take current 'seed'
inc a           60              and add 1.
cp 7            254 7         If in range ( not>6 )
jr c, 2         56 2            then jump on,
ld a, 1         62 1          else restore to 1
ld (ADDR),a     50  A  A        and replace in memory
ld c, 11        14 11         Test
call BDOS       205 5 0         for a
or a            183             key-press.
jr z -21        40 235        If none repeat
ret             201             else finish.
```

Multiplication and Division

I obtained the next four sub-routines from magazines and books. They are fast and economical of memory and have shown themselves to be useful in a wide range of programs. They are offered as candidates for a library. In all cases the abbreviation '(a)' means "the content of the A register", and '(hl)' means "the content of the HL register pair", etc.

It is not possible to standardise on which registers shall contain the original numbers because of the unique role that HL plays in additions and subtractions, though you could add extra instructions to achieve standardisation if it seemed desirable. There is no need to have both the 8- and the 16-bit versions in memory at the same time because the latter will perform the same function as the former if the High Bytes are first set to zero, though the calculation time will be a little longer.

8-bit multiplication

This multiplies (h) by (e) and gives the result in HL.

```
ld d, 0         22  0
ld l, 0         46  0        (h) x (e) = (hl)
ld b, 8         6  8
add hl, hl      41
jr nc 1         48  1
add hl de       25
djnz -6         16  250
ret             201
```

8-bit division

This divides (d) by (e) and gives the result in D plus any remainder in A.

```
ld b, 8         6  8
xor a           175
sla d           203 34       (d)/(e) = (d) + (a)
rl a            203 23
cp e            187
jr c 2          56  2
sub a, e        147
inc d           20
djnz -11        16  245
ret             201
```


16 to 32-bit multiplication

This multiplies (bc) by (de) and gives the result in HLDE. If the product is certain not to exceed 65535 then HL can be ignored and the result taken from DE. For larger results, the total is 65536 x (hl) + (de).

```
ld hl 0         33  0  0
ld a, 16        62  16
bit 0 e         203 67       (bc) x (de) = (hlde)
jr z 1          40  1
add hl bc       9
srl h           203 60
rr l            203 29
rr d            203 26
rr e            203 27
dec a           61
jr nz -16       32  240
ret             201
```

16-bit division

This divides (bc) by (de) and gives the result in BC with any remainder in HL.

```
ld hl 0         33  0  0
ld a, 16        62  16
scf             55           (bc)/(de) = (bc) + (hl)
rl c            203 17
rl b            203 16
adc hl de       237 106
sbc hl de       237 82
jr nc 2         48  2
add hl de       25
dec c           13
dec a           61
jr nz -16       32  240
ret             201
```

32-bit Calculations

8- and 16-bit calculations are suitable for most purposes, but occasionally they do not offer adequate precision. Fortunately it is not difficult to provide sub-rs that operate on 24- or 32-bit numbers, though they are noticeably slower when many iterations have to be invoked. My preference is that, rather than have both 24- and 32-bit available together, I provide for only 32-bit because these can do the work of both. 32-bit division is used in, among other things, calculating pseudo-random numbers as described above.

From the discussions in chapter 2, it will be obvious that 32-bit numbers occupy four bytes. In all cases of referring to 'pointing to' such a number by HL, say, I will mean that the number is in memory and HL contains the address of the least significant byte of the number. This will also be the lowest of the four addresses that the number occupies.

32-bit Addition

The following sub-r allows addition of two 32-bit numbers that are currently in memory. Before calling the sub-r, their low bytes are pointed to by HL and by DE respectively. The result is given in DEHL (D contains the most significant byte of the result and L the least significant).

```
ld a (de)      26         Lowest byte into A.
add a (hl)     134   *    Add other lowest byte
ld c, a        79           and store in C
inc hl         35         Point to next
inc de         19           two bytes
ld a (de)      26           and
adc a (hl)     142   *      repeat
ld b, a        71           storing in B.
push bc        197        Save the 2 bytes on stack
inc hl         35         Point to the next
inc de         19           two higher bytes,
ld a (de)      26           and
adc a (hl)     142   *    repeat.
ld c, a        79
inc hl         35         Point to the two
inc de         19           highest bytes
ld a (de)      26           etc.
adc a (hl)     142   *
ld d, a        87         Put the two highest
ld e, c        89           bytes of result in DE.
pop hl         225        And 2 lowest into HL
ret            201
```


32-bit Subtraction

This follows the same pattern as the addition, but the addition operations marked with '*' are changed to subtractions, ie.

> *'add a,(hl) 134'* becomes *'sub a,(hl) 150'*,
> *'adc a, (hl) 142'* becomes *'sbc a,(hl) 158'*.

The number pointed to by HL is subtracted from the one pointed to by DE.

32-bit Multiplications

The following suite of programs allows three kinds of multiplication of the 32-bit number pointed to by HL

START 1. multiplication by the content of A
or START 2. by the 16-bit content of DE
or START 3. by a second 32-bit number pointed to by DE

A 14-byte scratch pad is required the lowest address of which I will call (P,P), and the sub-r for 32-bit addition is needed. If the result would be too large to fit into 32 bits then the calculation is terminated and Cy is returned set. Otherwise the result is returned in HLDE (and in the highest four bytes of the scratch pad), and Cy is returned reset.

```
START1:
        ld de PAD+4     17 P+4 P     Transfer the HL No
        ld bc 4         1 4 0          into the
        ldir            237 176      Pad.
        ld (PAD), a     50 P P       Put (a) into PAD.
        ld b, 8         6 8          Count of 8 bits in B
        jr ZERO         24 15        Go to 'Calculation'.
START2:
        push de         213          Save (de)
        ld de PAD+4     17 P+4 P     Transfer the HL No
        ld bc 4         1 4 0          into the
        ldir            237 176      Pad.
        pop hl          225          Recover (de) into HL
        ld (PAD), hl    34 P P         and put into Pad.
        ld b, 16        6 16         Count 16 bits in B
        jr ZERO         24 20        Go to 'Calculation'.
```

continued on next page . . . .

```
START3:
        push de          213              Save (de).
        ld de PAD+4      17 P+4 P         Transfer the HL number
        ld bc 4          1 4 0              into the
        ldir             237 176            Pad.
        pop hl           225              Recover (de) into HL
        ld de PAD        17 P P             and copy the
        ld bc 4          1 4 0            32-bit DE number
        ldir             237 176            into the Pad.
        ld b, 32         6 32             Count 32 bits in B
ZERO:
        ld hl 0          33 0 0           Zeroise
        ld(PAD+8)hl      34 P+8 P           the rest
        ld(PAD+10)hl     34 P+10 P          of the
        ld(PAD+12)hl     34 P+12 P          Pad.
CALC:
        ld hl PAD+3      33 P+3 P         Point top byte of
        srl (hl)         203 62             the multiplier
        dec hl           43                 and
        rr (hl)          203 30             rotate
        dec hl           43                 each least significant
        rr (hl)          203 30             bit
        dec hl           43                 in turn
        rr (hl)          203 3              into Cy.
        jr nc 19         48 19            If bit reset jump on.

        push bc          197              Else save bit count.
        ld hl PD+10      33 P+10 P        And add multiplicand
        ld de PD+4       17 P+4 P           into
        call 32-Add      205 R R            the result
        ld(PD+10)hl      34 P+10 P
        ld(PD+12)de      237 83 P+12 P
        pop bc           193              Recover bit count but if
        ret c            216                addition overflow, then exit

        ld hl PD+4       33 P+4 P         Rotate the
        sla (hl)         203 38             multiplicand
        inc hl           35                 to the
        rl (hl)          203 22             left
        inc hl           35                 in
        rl (hl)          203 22             five
        inc              35                 bytes
        rl (hl)          203 22             ( 5th is to test
```

continued on next page . . . .


```
        inc hl           35               for overflow )
        rl (hl)          203 22

        ld hl(PAD)       42 P P           Test
        ld de(PAD+2)     237 91 P+2 P       the
        ld a, l          125                multiplier.
        or h             180              If all
        or e             179                bytes
        or d             178                now zero
        jr z END         40 10              then finish.

        ld a(PAD+8)      58 P+8 P         Test 'fifth byte'
        or a             183                of the multiplicand
        jr z 2           40 2             if not 0 (overflow)
        cf               55                 then set Cy
        et               201                and finish
        djnz CALC        16 181           Else repeat if count not 0

END:    ld hl(PD+10)     42 P+10 P        Transfer result
        ld de(PD+12)     237 91 P+12 P      into DEHL
        or a             183              Reset Cy
        ret              201              And finish.
```

32-Bit Divisions

The following three programs allow divisions of the 32-bit number pointed to by HL similar to the multiplications described above:

START 1. Division by the content of A
or START 2. By the 16-bit content of DE
or START 3. By 32-bit number pointed to by DE

The 32-bit subtraction routine is needed, as is an 18-bit scratch-pad whose lowest address is (P,P). If the divisor is the larger of the two (ie. the result would be less than 1) then the division is terminated and Cy returned set. Otherwise Cy is reset and the result is returned in DEHL (and at the top of the scratch-pad).

```
START1:
        ld de PAD+4      17 P+4 P         'HL number'
        ld bc 4          1 4 0              into
        ldir             237 176            Pad.
        ld (PAD) a       50 P P           (a) into Pad.
```

```
        ld hl 0             33 0 0         Zeroise
        ld (PAD+1)hl        34 P+1 P        rest of
        ld (PAD+2)hl        34 P+2 P        divisor
        jr TEST             24 19          Go to Calculation
        push de             213            Save (de).
        ld de PAD+4         17 P+4 P       'HL number'
        ld bc 4             1 4 0           into
        ldir                237 176        Pad.
        pop hl              225            Recover (de) in HL
        ld (PAD)hl          34 P P          and put into Pad.
        ld hl 0             33 0 0         Zeroise
        ld (PAD+2)hl        34 P+2 P        rest of divisor.
        jr TEST             24 18          Go to Calculation.
START3:
        push de             213            Save (de).
        ld de PAD+4         17 P+4 P       'HL number'
        ld bc 4             1 4 0           into
        ldir                237 176         Pad.
        pop hl              225            Recover (de).
        ld de PAD           17 P P         'DE number'
        ld bc 4             1 4 0           into
        ldir                237 176         Pad.
TEST:
        ld hl PAD           33 P P         If divisor is the
        ld de PAD+4         17 P+4 P        smaller of the two
        call 32 SUB         205 S S         then exit
        ret c               216             with Cy set.
ZERO:
        ld a 10             62 10          Zeroise
        ld hl PAD+8         33 P+8 P        rest
        ld (hl) 0           54 0            of
        inc hl              35              Pad
        dec a               61
        jr nz -6            32 250

        ld b, 32            6 32           32 bits until divisor is empty
CALC:
        push bc             197            Save count.
        ld a 13             62 13          13 bytes to rotate.
        ld hl PAD+4         33 P+4 P       Rotate
        sla (hl)            203 38          13 bytes
        inc hl              35              leftwards
        rl (hl)             203 22
```

continued on next page . . . .


```
        dec a               61
        jr nz -6            32 250
        ld hl PAD           33 P P         Subtract divisor
        ld de PAD+8         17 P+8 P        from rotated
        call 32 SUB         205 S S          bytes
        jr c 12             56 12          If divisor smaller then jump on
        ld (PAD+8)hl        34 P+8 P       Else put
        ld (PAD+10)de       237 83 P+10 P   remainder into
        ld hl PAD+14        33 P+14 P       rotated bytes and
        set 0 (hl)          203 198         count 1 into the result

        pop bc              193            Recover the bit count
        djnz CALC           16 216          and repeat if not 0

END:    ld hl(PAD+14)       42 L+14 H      Put result
        ld de(PAD+16)       237 91 L+16 H   into 'dehl',
        or a                183              reset Cy
        ret                 201              and finish.
```

SIN and COS

Calculating the precise values of Sin and Cos for any angle is a complex business involving evaluating series in which the values are in the form of floating-point numbers. Not only is this operationally difficult but programs making use of it are slowed down quite noticeably, and the technique is not suitable in, for example, tactical and strategic games where lots of positions, courses, and distances apart have to be evaluated as quickly as possible.

Fortunately there is a fast alternative if you are willing to accept a modest amount of approximation. In this case the approximation still provides an angular discrimination of a degree (or better if you insist), and results to an accuracy of tighter than 0.5%. The solution is to use a table of pre-calculated values and use the value of the angle as a pointer to the table; and fortunately it isn't necessary to provide Sin and Cos with a table each as their values are the same except for being 90 deg out of phase.

As both Sin and Cos always have values of 1 or less they can't be stored as such in single bytes, but you can use the device of multiplying by 255. This puts 255 into the table when Sin has a value of 1.000, and zero into the table when Sin is zero. The table values are therefore accurate to plus or minus 1 in 255, or about 0.4%, which is conveniently similar to the angular discrimination of plus or minus

1 in 360, or about 0.3%. The fact of having multiplied by 255 is taken into account in the calculations that follow the use of the table. You can increase the angular discrimination to any required level by increasing the length of the table in proportion, but the accuracy of the table content can't be improved without using two bytes per entry, which would allow an accuracy of plus or minus 1 in 65535. A real doubling of accuracy would therefore require a table four times as long.

The table need be only 451 bytes long and gives a result for each whole degree from 0 - 360 deg for both Sin and Cos. Given that the table starts at ADDR, it is filled by using the following BASIC command :

```
defint z :
for n = 0 to 450 :
z = cint ( 255 * sin ( n/57.296 ) ) :
poke ( ADDR + n ), z :
next
```

The routine that accesses the table simultaneously obtains a value for both Sin and Cos, and also provides the sign of each result. It reads the angle from where it is stored in the Variables and puts its results back in there. An angle larger than 360 deg has 360 repeatedly subtracted from it until the result is less than 361. This value is used in the calculation. The next set of addresses from those used earlier are:

51212	(12,200)	Lo Value of the
51213	(13,200)	Hi angle (0 to 360)
51214	(14,200)	Cos x 255
51215	(15,200)	Sin x 255
51216	(16,200)	Sign flags

For two positive results the sign flags are reset (flag value = 0). A negative Cos gives bit No 0 set (flag value = 1), and a negative Sin gives bit No 1 set (flag value = 2). Both flags are set if both Sin and Cos are negative (flag value = 3). The routine is:

```
Initialise :-
    ld bc 0          1 0 0        BC will take the flags
    ld hl(51212)     42 12 200    Put angle into HL
    ld de 360        17 104 1
    or a             183          Reset Cy and
    sbc hl de        237 82         subtract 360
    jr nc -4         48 252         until the result is negative
```



```
    add hl, de       25           then add back 360
    ld (51212)hl     34 12 200    and store result.
    ex hl de         235          Transfer angle to DE.

Evaluate the signs :-
    ld hl 270        33 14 1      If the angle is
    or a             183            > 270 deg
    sbc hl de        237 82         or
    jr c 8           56 8           less
    ld hl 90         33 90 0        than 90 deg
    sbc hl de        237 82         then jump on,
    jr nc 1          48 1           else set the flag
    inc c            12             ( for a negative COS )
    ld hl 180        33 180 0
    or a             183          If it is < 181 then
    sbc hl de        237 82         jump on,
    jr nc 2          48 2           else set the flag
    ld b 2           6 2            ( for a negative SIN )
    ld a, c          121          Put COS flag into A
    add a, b         128            add the SIN flag
    ld (51216)a      50 16 200      and store.

Calculate SIN :-
    ld hl ADDR       33 S S       SIN-table start in HL
    add hl de        25             and add the angle.
    ld a (hl)        126          Extract the SIN byte
    ld (51215)a      50 15 200      and store.

Calculate COS :-
    ld hl ADDR+90    33 C C       COS-table start in HL
    add hl de        25             and add the angle.
    ld a (hl)        126          Extract the COS byte
    ld (51214)a      50 14 200      and store.
    ret              201
```

A common use of Sin and Cos is to assess changes in co-ordinate values, and testing the sign bits will indicate whether the changes should be positive or negative, 0 degrees being taken as 'due North'.

There are a number of ways in which the routine could be modified for better speed or to operate with a shorter table, but this presentation gives the best view of the principle. Different values loaded into HL in "Evaluate the signs" would allow for other orientations, such as 0 deg is 'due East'.

Square Roots

Square roots can be dealt with as for Sin and Cos except that the table holds squares and contains 2-bytes per entry. It is the location of the square that indicates the size of the square root. To obtain integer square roots up to 255 the table should contain $(n + 0.5)^2$ for 'n' in the range 0 to 255, [not the squares of the integers]. You step through the table two bytes at a time until you find the first entry that is larger than the number whose root you wish to know. The count of the steps is the required square root. The table is most easily filled from BASIC.

Page 194 of the June '88 issue of "Personal Computer World" describes a method of finding any power of a number and the second to seventh root of a number by direct computation, but the calculation of the roots is slow. Square roots are of interest in calculating the distance apart of two co-ordinate pairs for use in games etc, though often you can compare the two squares of two distances and avoid square roots entirely.

Binary Coded Decimal

BCD allows precise calculation with large numbers. Whilst calculation in the floating point form may be accurate to one in a million or two, this may not be enough in some applications such as accountancy which needs to take care of the pence even in sums amounting to hundreds of millions of pounds.

For us decimal thinkers the complication of binary originates from the fact that bytes count in 256's. BCD starts with the idea of storing only decimal digits in them so that each one of a sequence of addresses could be treated exactly like the columns in conventional arithmetic. You can then record numbers of any size (and therefore obtain any level of accuracy) just by devoting extra addresses to them. It then refines this concept by taking note of the fact that the numbers up to 9 require only four bits so that two of them can be stored in an 8-bit byte. This halves the amount of memory required to store numbers, but the principle of calculating in tens is adhered to because manipulations are always carried out on half-bytes. And guess what half a byte is called. Any ideas ? That's it : a **nibble**. That's official, honest !

Because it is nibble-based, BCD requires three extra instructions; 'daa', 'rrd' and 'rld', and it also has its own flag called 'half-carry' which is set if additions or subtractions in the four rightmost bits of A give rise to overflow into bit No 4. The instruction 'daa' causes a nibble overflow if appropriate by adding 6 to both nibbles of A and then subtracting it again. Suppose A contains 9 and 1 or more is added to it. In BCD this should give overflow into the nibble on the left though the accumulator won't automatically give this because its four rightmost bits can hold 15 before overflowing. However adding 6 as well pushes overflow into the left nibble thus incrementing its content and setting the half-carry flag. If the left nibble also overflows following 'daa' then this will be reflected in Cy.

A full description of BCD wouldn't be appropriate here, but if you are interested you might like to build up your own set of BCD routines based on the examples of addition and subtraction given below. The convention I have adopted makes it possible to handle numbers up to 127 bytes long (254 digits!) preceded by a sign byte which is zero for 'positive' and 255 for 'negative'. First you need to allocate a memory block for storage of the numbers and it is best to draw this out so that you have clear picture of what each address is for. In multiplication and division extra blocks are needed for the product or quotient, plus a scratch pad for making a note of which stage has been reached. The registers are used as follows :

On entry: (b) = 0. (c) = number of bytes not including the sign byte used for each number, ie. half the maximum number of digits that each number may have. HL and DE point to where each number is stored in memory (they point to the sign byte).

On exit: The result is pointed to by HL and the carry flag is set if there is any overflow. Negative results will be in 'tens complement', ie. ready to give correct results by simple addition.

BCD Addition

```
        add hl bc        9      Make DE point to right-most
        ex hl de         235      byte of 1st num and HL
        add hl bc        9        to right-most byte of 2nd
        ld b, c          65     Put byte count into B
        ld a (de)        26     Add two bytes incl any
        adc a,(hl)   *   142      carry and apply
        daa              39       decimal adjustment.
        ld (de) a        18     Store resulting byte
        dec de           27       and point to the two
        dec hl           43       next bytes to the left.
```

continued on next page ....

```
djnz -8        16 248     Repeat until the count is 0,
ex hl de       235          and point HL to the result.
ret            201
```

BCD Subtraction

The sub-r for BCD subtraction is identical to the one above except that the 6th instruction (∗) should be changed to :

```
sbc a, c       158
```

Usefulness of BCD

You will see from the complexity of such simple operations that BCD is nothing like as fast as conventional Z80 arithmetic. This is the price paid for its ability to handle many digits. Although I have spent many happy hours developing routines to manipulate BCD numbers, I have never used them and don't ever expect to. My accounts programs, which count pennies, are based on 32-bit arithmetic and can deal with values up to about £43 million.

BCD attempts to suggest that it is really OK to manipulate decimal numbers electronically , but, like most pretenses, it involves too many complications to be worth the trouble.

APPENDICES

Arithmetical Instructions

add hl,bc	9	add a,a	135	adc a,a	143
add hl,de	25	add a,b	128	adc a,b	136
add hl,hl	41	add a,c	129	adc a,c	137
add hl,sp	57	add a,d	130	adc a,d	138
		add a,e	131	adc a,e	139
adc hl,bc	237 74	add a,h	132	adc a,h	140
adc hl,de	237 90	add a,l	133	adc a,l	141
adc hl,hl	237 106	add a,(hl)	134	adc a,(hl)	142
adc hl,sp	237 122	add a,N	198 N	adc a,N	206 N
sbc hl,bc	237 66	sub a,a	151	sbc a,a	159
sbc hl,de	237 82	sub a,b	144	sbc a,b	152
sbc hl,hl	237 98	sub a,c	145	sbc a,c	153
sbc hl,sp	237 114	sub a,d	146	sbc a,d	154
		sub a,e	147	sbc a,e	155
		sub a,h	148	sbc a,h	156
		sub a,l	149	sbc a,l	157
		sub a,(hl)	150	sbc a,(hl)	158
		sub a,N	214 N	sbc a,N	222 N

Comparisons

cp a	191	and a	167	or a	183	xor a	175
cp b	184	and b	160	or b	176	xor b	168
cp c	185	and c	161	or c	177	xor c	169
cp d	186	and d	162	or d	178	xor d	170
cp e	187	and e	163	or e	179	xor e	171
cp h	188	and h	164	or h	180	xor h	172
cp l	189	and l	165	or l	181	xor l	173
cp (hl)	190	and (hl)	166	or (hl)	182	xor (hl)	174
cp N	254 N	and N	230 N	or N	246 N	xor N	238 N

Increment and Decrement

inc bc	3	inc a	60	dec a	61
inc de	1	inc b	4	dec b	5
inc hl	35	inc c	12	dec c	13
inc sp	51	inc d	20	dec d	21
		inc e	28	dec e	29
dec bc	11	inc h	36	dec h	37
dec de	27	inc l	44	dec l	45
dec hl	43	inc (hl)	52	dec (hl)	53
dec sp	59				

Call and Ret

call N N	205 N N	ret	201
call c N N	220 N N	ret c	216
call nc N N	212 N N	ret nc	208
call z N N	204 N N	ret z	200
call nz N N	192 N N	ret nz	192

Push and Pop

push af	245	pop af	241
push bc	197	pop bc	193
push de	213	pop de	209
push hl	229	pop hl	225

Jumps

jp N N	195 N N	jr N	24 N
jp c N N	218 N N	jr c N	56 N
jp nc N N	210 N N	jr nc N	48 N
jp z N N	202 N N	jr z N	40 N
jp nz N N	194 N N	jr nz N	32 N
jp (hl)	233	djnz N	16 N

Block operations

ldd	237 168	lddr	237 184
ldi	237 160	ldir	237 176
cpd	237 169	cpdr	237 185
cpi	237 161	cpir	237 177

Miscellaneous instructions

ccf	63	ex hl, de	235
scf	55	ex (sp) hl	253
neg	237 68		
nop	0		
cpl	47		
in a, (P)	219 N	out (P), a	211 N
rld	237 111	rrd	237 103
daa	39		

Bit, Set, & Reset

bit 0,a	203 71	bit 1,a	203 79	bit 2,a	203 87	bit 3,a	203 95
bit 0,b	203 64	bit 1,b	203 72	bit 2,b	203 80	bit 3,b	203 88
bit 0,c	203 65	bit 1,c	203 73	bit 2,c	203 81	bit 3,c	203 89
bit 0,d	203 66	bit 1,d	203 74	bit 2,d	203 82	bit 3,d	203 90
bit 0,e	203 67	bit 1,e	203 75	bit 2,e	203 83	bit 3,e	203 91
bit 0,h	203 68	bit 1,h	203 76	bit 2,h	203 84	bit 3,h	203 92
bit 0,l	203 69	bit 1,l	203 77	bit 2,l	203 85	bit 3,l	203 93
bit 0,(hl)	203 70	bit 1,(hl)	203 78	bit 2,(hl)	203 86	bit 3,(hl)	203 94
bit 4,a	203 103	bit 5,a	203 111	bit 6,a	203 119	bit 7,a	203 127
bit 4,b	203 96	bit 5,b	203 104	bit 6,b	203 112	bit 7,b	203 120
bit 4,c	203 97	bit 5,c	203 105	bit 6,c	203 113	bit 7,c	203 121
bit 4,d	203 98	bit 5,d	203 106	bit 6,d	203 114	bit 7,d	203 122
bit 4,e	203 99	bit 5,e	203 107	bit 6,e	203 115	bit 7,e	203 123
bit 4,h	203 100	bit 5,h	203 108	bit 6,h	203 116	bit 7,h	203 124
bit 4,l	203 101	bit 5,l	203 109	bit 6,l	203 117	bit 7,l	203 125
bit 4,(hl)	203 102	bit 5,(hl)	203 110	bit 6,(hl)	203 118	bit 7,(hl)	203 126

set 0,a	203 199	set 1,a	203 207	set 2,a	203 215	set 3,a	203 223
set 0,b	203 192	set 1,b	203 200	set 2,b	203 208	set 3,b	203 216
set 0,c	203 193	set 1,c	203 201	set 2,c	203 209	set 3,c	203 217
set 0,d	203 194	set 1,d	203 202	set 2,d	203 210	set 3,d	203 218
set 0,e	203 195	set 1,e	203 203	set 2,e	203 211	set 3,e	203 219
set 0,h	203 196	set 1,h	203 204	set 2,h	203 212	set 3,h	203 220
set 0,l	203 197	set 1,l	203 205	set 2,l	203 213	set 3,l	203 221
set 0,(hl)	203 198	set 1,(hl)	203 206	set 2,(hl)	203 214	set 3,(hl)	203 222
set 4,a	203 231	set 5,a	203 239	set 6,a	203 247	set 7,a	203 255
set 4,b	203 224	set 5,b	203 232	set 6,b	203 240	set 7,b	203 248
set 4,c	203 225	set 5,c	203 233	set 6,c	203 241	set 7,c	203 249
set 4,d	203 226	set 5,d	203 234	set 6,d	203 242	set 7,d	203 250
set 4,e	203 227	set 5,e	203 235	set 6,e	203 243	set 7,e	203 251
set 4,h	203 228	set 5,h	203 236	set 6,h	203 244	set 7,h	203 252
set 4,l	203 229	set 5,l	203 237	set 6,l	203 245	set 7,l	203 253
set 4,(hl)	203 230	set 5,(hl)	203 238	set 6,(hl)	203 246	set 7,(hl)	203 254

res 0,a	203 135	res 1,a	203 143	res 2,a	203 151	res 3,a	203 159
res 0,b	203 128	res 1,b	203 136	res 2,b	203 144	res 3,b	203 152
res 0,c	203 129	res 1,c	203 137	res 2,c	203 145	res 3,c	203 153
res 0,d	203 130	res 1,d	203 138	res 2,d	203 146	res 3,d	203 154
res 0,e	203 131	res 1,e	203 139	res 2,e	203 147	res 3,e	203 155
res 0,h	203 132	res 1,h	203 140	res 2,h	203 148	res 3,h	203 156
res 0,l	203 133	res 1,l	203 141	res 2,l	203 149	res 3,l	203 157
res 0,(hl)	203 134	res 1,(hl)	203 142	res 2,(hl)	203 150	res 3,(hl)	203 158
res 4,a	203 167	res 5,a	203 175	res 6,a	203 183	res 7,a	203 191
res 4,b	203 160	res 5,b	203 168	res 6,b	203 176	res 7,b	203 184
res 4,c	203 161	res 5,c	203 169	res 6,c	203 177	res 7,c	203 185
res 4,d	203 162	res 5,d	203 170	res 6,d	203 178	res 7,d	203 186
res 4,e	203 163	res 5,e	203 171	res 6,e	203 179	res 7,e	203 187
res 4,h	203 164	res 5,h	203 172	res 6,h	203 180	res 7,h	203 188
res 4,l	203 165	res 5,l	203 173	res 6,l	203 181	res 7,l	203 189
res 4,(hl)	203 166	res 5,(hl)	203 174	res 6,(hl)	203 182	res 7,(hl)	203 190

OPERATION TIMINGS

The following list indicates the times of the common operations, the numbers being in 'T-states', each of which corresponds to 0.25 micro-seconds.

adc a, N or (hl)	7
adc a, r	4
adc hl, rr	15
add: as above except	
add hl, rr	11
and a, N or (hl)	7
and a, r	4
bit n (hl)	12
bit n r	8
call Addr	17
ccf	4
cp a, N or (hl)	7
cp a, r	4
cpd cpi	16
cpdr cpir	16
cpl	4
daa	4
dec r	4
dec (hl)	11
dec rr	6
djnz	13
ex hl de	4
inc: see 'dec'	
jp Addr	10
jp (hl)	4
jr	12
ld rr (Addr)	20
ld (Addr) rr	20
ld r, N	7
ld r, r	4
ld (rr) a	7
ld (hl) r	7
ld r (hl)	7
ld a (Addr)	13
ld (Addr) a	13
ld hl (Addr)	16
ld (Addr) hl	16
ld rr, N N	10
ldd ld	16
lddr ldir	16
neg	8
nop	4
or: see 'and'	
pop	10
push	11
res b, r	8
res b, (hl)	15
ret	10
rotate registers	8
rotate (hl)	15
rld rrd	18
sbc: see 'adc'	
sub: see 'add'	
scf	4
set: see 'res'	
shift: see 'rotate'	
xor: see 'and'	

END

NEGATIVE NUMBERS

A satisfactory system for handling negative numbers should make it possible to obtain the correct result by additions of either negatives to each other or of negatives to positives without needing to know whether some negatives are involved, and it should be possible to establish the sign of a number by inspection of the sign bit. This is provided by the so called **twos complement** system in which a binary number is converted to its negative value by complementing all its bits and then adding 1, which is the equivalent of subtracting it from zero.

Consider the example of subtracting 5 from 13. First convert the 5 to its twos-complement and then add the result to 13 (and ignore the overflow).

5 in binary is:	00000101	
Complement it,	11111010	
and add 1	11111011	
13 in binary is:	00001101	
Add the negative,	11111011	
to give	00001000	which = 8

8 bits can represent numbers from +127 to -128. 16 bits can represent +32767 to -32768. A positive 8-bit number in A is converted to its negative version by the instruction 'neg', which stands for "negate the accumulator". A negative number would be converted by this to its positive value. Both are numerically the same as subtracting from 256. Thus to obtain the same effect as the BASIC command 'ABS' first test the eighth bit; if it is set then use 'neg', otherwise not. The following procedure negates (bc) and gives the result in HL:

```
or a        183      Cancel Cy.
sbc hl hl   237 98   Zeroise HL.
sbc hl bc   237 66   Subtract (bc) from 0
ret         201
```

And the following transfers an 8-bit number from A into HL whilst preserving its sign bit:

```
ld l, a     111      Number into L.
rl a        23       Sign bit into Cy.
sbc a, a    159      Propagate sign thru A &
ld h, a     103      load it into H.
```

END

BDOS FUNCTIONS

The following table lists the BDOS functions referred to in the text together with the input required in DE or E and the output given in A (or HL in the case of No 12). See text for No 6. The function No is always put into C.

Fn No	Input Name	Output (de) or (e)	(a)	Page
0	System reset	-	-	61
1	Console Input	-	ASCII	62
2	Console Output	ASCII	-	65
5	List Output	ASCII	-	79
6	Direct Console I/O	-	0 = no key/ASCII	64
9	Print String	String addr	-	67
10	Read Consl Buffer	Buff addr	(Txt in Buff)	65
11	Get Console Status	-	0 = no key ; 1= key	63
12	Version Number	-	(Vers Nos in L)	62
13	Reset Disc System	-	(Drives & DMA reset)	123
15	Open file	FCB addr	255 = failure	111
16	Close file	FCB addr	255 = failure	112
19	Delete file	FCB addr	255 = failure.	115
20	Read sequential	FCB addr	0 = success	114
21	Write sequential	FCB addr	0 = success	113
22	Create file	FCB addr	255 = failure	108
23	Rename file	FCB addr	255 = failure	118
26	Set DMA address	DMA addr	-	111
33	Read Random	FCB addr	0 = success	117
34	Write Random	FCB addr	0 = success	117
40	Wrt Randm + Zero Fill	FCB addr	0 = success	117
45	Set Disc Error Mode	mode No	-	129
46	Get Disc Free Space	-	(Fre in DMA)	123
110	Set/Get delimiter	ASCII/FFFFh	(Marker in A)	68
111	Print text block	CCB addr	-	69
112	List text block	CCB addr	-	74
152	Parse Filename	PFCB addr	see text	168

END

SCREEN ADDRESSES

The first requirement is to calculate 'LINE' which is the printline No counting the top line as No 0. In BASIC the equivalent calculation would be

LINE = 31 - INT(Y/8)

though the value INT (Y/8) is also required and is stored in C for later use [call this (c)]. If Y=0 then LINE=31, if Y=255 then LINE=0. The value of LINE allows the start address of the print-line to be obtained from Roller-RAM. The Roller-RAM address for LINE=0 is (0,182), for LINE=1 it is (16,182), etc. Hence the Roller-RAM address is given by

RAM_ADDR = (0,182) + 16 x LINE

The address of the start of the print-line can be extracted from RAM_ADDR.

Consider the case where 'X' = 0. If the value of 'Y' is 7, 15, 23, 31 .. or 255 (ie. 7 + 8n for 'n' between 0 and 31), ie. if the required byte is at the top of a print-line, then the line-address will the same as the screen-address. If the byte is not at the top of the line then the screen-address will be increased accordingly, the increase being given by :

Correction = 7 - (Y - (31-LINE) x 8)
= 7 - Y + (c)

The final correction is for the value of 'X'. Starting at 'X'=0, seven increments in 'X' point in turn to the bits of the leftmost screen byte, but when 'X'=8 bit No 0 of the next-right screen byte is pointed to. Athough this is only one byte to the right on the screen, it is 8 bytes further on in memory. Hence an increment of 8 to 'X' causes an increment of 8 in the address, but smaller increments in 'X' make no difference to it. This correction is the equivalent of :

8 x INT(X/8)

which could be calculated by three right shifts of 'X'
[giving INT (X/8)] followed by three left shifts [multiplying by 8], but the same effect is given more simply by resetting the three right-most bit of 'X'.

END

THE STACK & THE PROGRAM COUNTER

The stack is a small area of memory used for temporary storage of information. It grows down-wards from higher to lower addresses as each new entry is made, and retreats upwards as entries are removed. The start (highest address) is still called the 'bottom of the stack', and the end (lowest address) is called the 'top of the stack'.

The return address for each 'call' is stored at the top of the stack, as are the 'pushed' contents of register-pairs. Following 'push hl' the sequence is:

1. SP is decremented.
2. contents of H are copied into the address pointed to by SP.
3. SP is decremented again.
4. contents of L are copied into the address pointed to by SP.

'Pop hl' follows the reverse procedure but the bytes that HL fed onto the stack stay in place : 'pop' does not remove them it only causes SP to point to the previous entry, though they will be over-written by any future 'push' or 'call'.

The location of the stack can be changed by putting its new location into SP. When choosing a location it is necessary to prevent other operations from over-writing it, and vice versa. The area allocated should be large enough to allow two bytes to be added to it for each case of use, though this is difficult to calculate and it is prudent to be generous. Changing the content of SP can also be used to access earlier entries in the present stack, it being necessary to increment SP twice to point to each earlier entry. This is the equivalent of writing an extra 'pop' into the program but without transferring anything into a pair of registers.

Restoring the Stack

Occasions arise when you need to re-balance the stack, ie. to ignore unwanted data and restore it to an earlier condition, but you don't want to go through a possibly lengthy procedure of individual 'pops', the required number of which may in any case be uncertain. If you have defined your own stack location then re-defining as before will cancel all intervening stack operations and take you back to the stack you had at the start of the program. If you are using the existing

CP/M stack, or if you don't want to go all the way to the start of your own stack, then record the required stack-address in memory and then re-load SP with it at the appropriate moment.

If you can't get a satisfactory return either to BASIC or to CP/M when your m/c program has been run then you can be sure you have an unrequited 'push', 'pop', 'call' or 'ret' somewhere. You can temporarily solve the problem by making your first m/c instruction 'ld (N N),sp' (to record the last address at which BASIC or CP/M was operating), and make the last instruction before the final 'ret' ; 'ld sp,(N N)', though temporary solutions are only temporary.

THE PROGRAM COUNTER

The program counter, PC, is a 16-bit register that keeps track of the address at which the next operation is to be found. When the machine is switched on or reset, PC is loaded with 0000h so operations always begin with the instruction at that location, which is 'jp FC03h'.

Each time the Z80 encounters an opcode it interrogates it to establish the number of bytes in the instruction. For most instructions this number is added to PC, the instruction is executed, and the new address in PC is then jumped to. If the instruction is a 'jp' the address immediately following the opcode is copied into PC and operations proceed from there. For a 'jr' the byte following the opcode is added to PC and operations proceed from there.

For a 'call', the content of PC+3 is put onto the stack, SP is adjusted, and PC is loaded with the call address. The 'ret' puts the top address from the stack back into PC and adjusts SP. This is why it is essential to have balanced every 'push' with a 'pop' (or to have pointed SP to the right entry) between a 'call' and its 'ret'. If you forget to do this before a conditional 'ret' you may get a crash on some occasions but not on others and not be able to see why.

END

SWITCHING MEMORY BANKS

The processor Ports

The Z80 makes contact with the outside world through 'ports', which can pass bytes inwards to the processor, or outwards from the processor to some device connected to it.

There are two ways of operating the ports. In the first the 'address' of the port is loaded into BC and then either the 'in' instruction takes a byte from the external device (a section of the keyboard, say) and puts it into a register ; or alternatively the 'out' instruction feeds the byte that is in the register out into the external device. The mnemonics would be as follows for the register 'R' ;

in R,(c) or *out R,(c)*

However, access to the Memory Disc is gained through the other method of using ports, and we will be concerned only with the 'out' version. In this the required byte is put into A and the port number is specified as part of the instruction code. To output the content of A through any one of the ports the generalised mnemonic and the generalised decimal instruction bytes are ;

out(P), a 211 P *where 'P' is port N.*

'A' is the only register available for use with this instruction

The Memory Manager

The 'Memory Manager' is located at address FD21h (33,253) [FD2Dh (45,253) for the '9512'] in common memory. This is the sub-r that lines up the set of memory blocks that are required to be available to the Z80 at any particular moment. Usually this is Bank 1 (the TPA), but it may be any of the others. The Manager is entered with A containing the Bank No required and this it stores at address FEA0h. It then loads A with each of three values prior to making three 'out' instructions to port Nos F0h, F1h, and F2h (ie. ports 240, 241 and 242). The values put into A and then sent to these ports determine which memory blocks are switched into circuit. The basis of the Memory Manager (for '8256/8512') is as follows:

The Memory Manager:

```
FD21  push hl          229           Save HL
      ld (FEA0h), a    50 10 254     Store A
      dec a            61            If ( A ) = 1
FD26  jr z 30          40 30          jump to FD46.

Banks 0, 2 and N
FD28  inc a            60            Else restore (a),
      ld hl 8381h      33 129 131     load HL,
      jr z 9           40 9          If ( A ) = 0 jp to FD37
      ld l 88h         46 136
      cp 2             254 2         If Bank = 2 then
      jr z 3           40 3           jump to FD37.
      add 86h          198 134       If Bank > 2 then
FD36  ld l a           111           ( L ) = 134 + ( A ).

FD37  ld a 80h         62 128        Set
      out (F0h),a      211 240        the
      ld (0061h)hl     34 97 0        values in A
      ld a l           125            and
      out (F1h),a      211 241        give
      ld a h           124            the 'out'
      out (F2h),a      211 242        instructions.
      pop hl           225           Recover original ( HL )
FD45  ret              201            and finish.

Bank 1 (TPA)
FD46  ld hl 8685h      33 133 134
      ld (0061h) hl    34 97 0       As
      ld a l           125            above.
      out (F1), a      211 241
      ld a h           124
      out (F2), a      211 242
      ld a 84h         62 132
      out (F0), a      211 240
      pop hl           225           Recover original ( HL )
FD57  ret              201            and finish.
```

The 'push' and 'pop' instructions are fairly common features of the sub-routines within CP/M and are included so that data held in HL is preserved for later use if required, but they are not essential to the bank-switching operation. If you follow through the pattern of the sub-r you will see that the values in A used for the 'out' instructions are unequivocal in the cases of calling for Banks 0, 1 and 2. The bytes fed to the ports in order to switch-in these banks are as follows. The block No is equal to the byte minus 128, as shown to the right of the table. Block 7 gets no 'out' instructions.

	F0	F1	F2	blocks
Bank 0	128	129	131	0 1 3 7
Bank 1	132	133	134	4 5 6 7
Bank 2	128	136	131	0 8 3 7

For banks of higher number the value sent to F1 is equal to [134 + (A)] so the sequence continues as;

Bank 3	128	137	131	0 9 3 7
Bank 4	128	138	131	0 10 3 7
Bank 5	128	139	131	0 11 3 7 etc.

Note that only one block is changed for Bank Nos larger than 2.

General rules for block switching

The above switching sequences are those employed by the PCW for its own good reasons, but if you want to swop the blocks about in your own way then the following rules apply. I developed my 'Empirical Technique' before I had fully cottoned on to them.

To refer to a block add 128 to its number, so No 0 becomes '128', No 1 becomes '129', etc. There are four memory ranges in the machine as listed on page 82; give them the following numbers:

hex	red-biro	No
0000 to 3FFF	(0,0) to (255,63)	240
4000 to 7FFF	(0,64) to (255,127)	241
8000 to BFFF	(0,128) to (255,191)	242
C000 to FFFF	(0,192) to (255,255)	243

Forget about the highest range because it should always contain Block 7, but any other range can have any block switched into it by *out (r), a* where 'r' is the range number, and A has been loaded with the block number. Hence to put Block 10 into the bottom range the instructions would be:


```
ld a, '10'        62 138
out ( a ), 240    211 240
```

The PCW tends to keep switching back to the TPA when it has completed an interrupt sequence, so if you have trouble with this (it never surfaced when I was using the 'Empirical Method') use :

```
di                243        Disable interrupts
ld a, BLOCK       62  N2     Select block No
out (a), RANGE    211 R      Switch it in
...      ...                 Your
...      ...                   procedure
ld a, BLOCK       62  N1     Restore
out (a), RANGE    211 R        orig block
ei                251        Enable interrupts
    continue...
```

Don't forget to operate from block 7, and keep your procedure moderately short or the machine will get upset.

Accessing the Memory Disc

There is a BIOS (not a BDOS) function No 27, called 'SELMEM', which accesses the Memory Manager by adding 78 to 'w.boot' to produce the address FC51h ie (81,252), at which is found the instruction 'jp FD21h', ie. 'jump to the Memory Manager'. Before using it A is loaded with the required Bank No. SELMEM is the normal system-entry to the Memory Disc, but it is more convenient for an m/c user to call the Memory Manager direct.

The reason for my development of the Empirical Block-Switching approach was that difficulties naturally arose with the above technique when I attempted to cross a block boundary with a high Bank No in use, as may happen with an 'ldir' operation. I was under the false impression that three new blocks came into force, so each time I crossed the boundary I was overwriting either block 0 or block 7. I am grateful to Johs Lind for clarifying this, though it should have been obvious.

END

PARSE FILE NAME

It is possible to cut out a lot of programming when setting up an FCB by using fnc No 152. First point DE at a 4-byte control block - the "PFCB" (don't blame me!). The first two bytes of the PFCB contain the address of a string that names the file.

This string has four optional parts, but (need I add) at least one option must be used (that's what the instructions say!). The first option is the drive name which can be A:, B:, etc. If you don't specify the drive, the default drive is used. The second option is the file name (up to 8 ASCIIs). The third option is the file-type, which must consist of "." followed by up to 3 ASCIIs. The fourth option may be a password consisting of ";" followed by up to 8 ASCIIs. The whole string must end with one of 16 possible terminators viz: Space (32) Tab (9) Return (13) Null (0) ; = < > . : , [ ] / $ and Verticalbar. This allows you to have several strings end to end in memory and use each one as appropriate.

The third and fourth byte of the PFCB are the address at which you want the FCB to be constructed. When fnc 152 has worked its magic the FCB will be drawn up at that address and zeroised ready for use in 'Open' etc. The password, incidentally, will be inserted in bytes 16 to 23, and its length at byte 26.

If your string had the terminator 0 or 13, zero will be returned in HL. For the other terminators, the terminator address will be returned in HL (which therefore tells you where to find the next string). FFFFh will be returned in HL if you use a duff filespec.

END

BOOKS

An Introduction to Z80 Machine Code

Authors: R. A. & J. W. Penfold
Published: Bernard Berbani Ltd, Shepherds Bush Road, London.

As a dictionary of the mnemonics, this book is extremely good value. It is low priced and gives a description of the full Z80 instruction set, together with the T-states required by each and their effects on the flags. All opcodes are in Hex.

CP/M 80 Programmer's Guide

Authors: B. Morrell & P. White
Published: Macmillan Education Ltd, Basingstoke, Hants, RG21 2XS.

An excellent description of the more commonly used BDOS functions with emphasis on those applying to file-handling. Clear and informative. It briefly describes the use of Assemblers.

The Amstrad CP/M Plus

Authors: D. Powys-Lybbe & A. Clarke
Published: M.M.L. Systems Ltd., 11, Sun Street, London, EC2M 2PS

This is a large, comprehensive, Amstrad-specific book giving a description of the implementation of CP/M on the 'CPC' and 'PCW' models and written by the experts. If you want to know anything about Amstrad CP/M then it will almost certainly be in here. There is a tutorial section that is readable enough and says something about programming with Assemblers, but it is closer to being a text-book than a user-guide, so dipping in for snippets of information is not easy. Most of the book is data tables that are useful in m/c programming but the presentation style is 'professional', so unless you know most of it already and merely want guidance on detail you will be struggling.

END

INDEX