"PCW SUPER Code"

The second best book on programming the PCW.

Loading screens ◊ Machine definition ◊ Interrupt techniques ◊ Pause generation ◊ Stop-watch & interval timing ◊ Draft and NLQ printer fonts ◊ All printer font data, with tips on re-design ◊ Advanced screen printing & screen addressing ◊ Screen manipulations : scrolling, panning, inversions, etc. ◊ Character inversions, rotations, enlargements, etc. ◊ Customised printing ◊ All forms of menus ◊ Block switching and other useful techniques, etc., etc.,

- DOZENS OF PROGRAM EXAMPLES
- Full INDEX and APPENDICES

"The whole book is a gold mine of information. It stands out as an example that programming does not have to be be dull, boring, or confusing. The author talks about machine code as if it is no more complicated than drawing up a shopping list, and his readers will believe him."

Amstrad PCW Magazine
May 1990

£13.95 nett

ISBN 1 871892 01 5

SPA

SPA ASSOCIATES

PCW SUPER CODE

MICHAEL KEYS

P C W

SUPER 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. In addition I am indebted to Iain Stirzaker and Geoffrey Childs who supplied extra material, and were patient. Geoffrey Childs also performed the invaluable service of proof reading the manuscript. 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.

© M. Keys 1990

All rights reserved. No reproduction in any form may be made without written permission.
No paragraph of this book may be made, reproduced, copied, translated, or transmitted without written permission in accordance with the Copyright Act 1956 and its amendments.
The assessment of the suitability for use of the information in this book in particular cases is the responsibility of the user. Neither the author nor the publishers accept responsibility for any consequence of its use.

First published April 1990

ISBN 1 871892 01 5

CONTENTS

1. Aims 7
2. The Set-Up 9
3. Time & Again 24
4. Forgive the Interruption 37
5. Back to Font 50
6. Basics of Screen Printing 69
7. Basics of Screen Addressing 78
8. Screen Manipulations 94
9. Character Manipulations 107
10. Basics of Menu Design 121
11. Key-press Menus 126
12. Cursor Menus 134
13. Loading Screens 142
14. Miscellaneous Output 162
15. Miscellaneous Input 168

Appendices :

1. Equivalent Addresses 182
2. Draft Font Data 184
3. NLQ Font Data 188
4. Block Switching 194
5. Keyboard Memory Map 198
6. Contents of Memory 200
7. Screen addresses 202
8. Im 2 programming 204

Books 210

INDEX 212

Chapter 1

Aims and Intentions

This book is the sequel to my *PCW Machine Code,* also published by Spa.

Its aim is to provide a more detailed description of using machine code to get the very best out of the PCW 8256, 8512, and 9512. In it I emphasise screen control because the screen is the most direct, and perhaps the most directly interesting, contact that is usually made with a computer. But in addition it also deals in detail with other important subjects that readers of my earlier book have expressed an interest in. These include Character Manipuations, Interrupts, Timing, Printer Fonts, and Loading Screens.

To obtain most advantage from the book you will need either to have read my earlier title and to have obtained a little practice with the information contained in it, or in some other way to have become familiar with the Z80 Instruction Set and the methods of using it.

Nothing here is complicated but it would be a waste to have to go over earlier ground a second time so I will assume that you do at least know what machine-code is.

When writing program examples I will give the usual Z80 mnemonics that you can input through your assembler if you have one, but be warned that I do not use a commercial assembler myself so I tend to be a bit lax with the details of syntax: I might occasionally drop the odd comma or miss out a space or two, so don't rely on copying my version verbatim.

I will also give the machine code bytes of the sub-r alongside the mnemonic, and these will be in decimal. The logic of this choice is that programmers familiar with hex will probably be using an assembler and will not be interested in the opcodes nor how they are written. On the other hand programmers who, like myself, input the opcodes directly into memory will find decimal much easier to key in than hex, and correspondence with readers of the earlier book has indicated that a majority of those who mentioned the choice were grateful for the decimal version. So decimal it is.

Even so, to cut down as much as I can on programming frustration for the minority, I have included Appendix 1 that lists the addresses referred to in the text in their decimal, hex, and 'red-biro' equivalents. 'Red-biro', incidentally, is my own idiosyncratic (but terribly sensible) way of writing addresses as two decimal bytes; they are written in brackets with a comma between, low-byte first.

Chapters 6 and 7 summarise the factual aspects of how to access the screen and what the Screen Data is and how to manipulate it. To readers of *PCW Machine Code* this may seem to be partly a duplication, but not everyone has read that title (not yet anyway), and even for those who have it is no bad thing to have the subject matter expanded and collected into a single space from the several chapters through which it was previously dispersed.

When referring to the Amstrad manuals, I will give the page number for the 8256 & 8512 first, followed by the corresponding page of the 9512 manual enclosed in square brackets.

I hope this volume is enjoyed as much as the earlier one seems to have been, and, as was the case with that one, I am always grateful to receive comments and suggestions from readers, even though (and I apologise in advance) I may not always be able to reply on an individual basis.

Chapter 2

The Set-up

THE MEMORY MANAGER

The PCW operates a 'banked' memory system, each bank consisting of 64k (see *PCW Machine Code*). Access to most of the blocks within these banks is provided by a CP/M routine called the "Memory Manager" (see Appendix 4) that starts at address (33,253) for the '8256' and the '8512', but at (45,253) for some '9512' machines (CP/M version 2.1) and at (58,253) for others (CP/M version 2.9).

To use it, the A register is first loaded with the number of the bank required and then a call is made to the relevant address. Thus to switch bank 0 into circuit, the assembler sequence would be:

```
ld a, 0          62  0
call MEM-M       205 33 253
```

In fact, on behalf of speed, I would use 'xor a' instead of 'ld a, 0,' and it just so happens that a 'xor a' instruction precedes the routine, so in the particular case of summoning Bank 0 you can simplify your code by calling the stated addresses minus one, ie. call (32,253), or (44,253), etc.

Switching to Bank 1 is a much used operation because Bank 1 contains the TPA, and every time the system returns from a foray into the nether regions of CP/M that is effectively what it needs to do, and every time you use another bank that is what you need to do also. The operation looks like:

```
ld a, 1          62  1
call MEM-M       205 33 253
continue . . .
```

Routines that you use to access the memory disc should be in Common Memory, as should your stack. The COM file stack and the CP/M stacks are in block 7, but the one used by Mallard Basic and some other applications is not. The common stacks are constructed under the following addresses:

COM file	(0,246)
CP/M (8256/8512)	(74,255)
CP/M (9512)	(24,255)
Mallard	(74,111) approx

To make sure that the current stack is in block 7, use:

```
ld (A,A), sp     237 115 A A    Save this stack address
ld sp, New       49 B7 B7       Make another in Block 7
. . . .                         Your
. . . .                            routine
ld sp, (A,A)     237 123 A A    Recover the original stack
continue . . .
```

When selecting (B7,B7), you can use the safe area of whichever of the listed stacks will not be used for other purposes.

THE BIOS JUMP BLOCK ROUTINES

Bank 0 (see Chapter 7 & Appendix 4) contains a jump table to a set of BIOS routines that can supply useful information concerning the configuration of the machine and other matters. The standard means of access to them is through the CP/M so-called 'USERF' ('user function'), which is used as follows.

Get the address that is stored at (1/2,0), which is (3,252). To this add

87 to produce the 'Jump USERF address', the value of which is (90,252) in the case of the PCWs. Calling or jumping to this will give access to the BIOS routines covered by USERF. In addition, you need to supply the address of the individual function you want in the form of an in-line parameter, and also to supply any other data required by the function in the appropriate registers. The 'in-line' parameter is supplied 'in-line with' (immediately following) the last call preceding the use of 'USERF'. If the address of the required table entry is to be (X,X), the listing could be on the lines of the one shown below.

This is a form I have used before, and it has the advantage of being one that applies in any CP/M environment. I have also used it to develop the modular approach that I will describe later.

```
ld hl, (1,0)     42 1 0       Jump-USERF
ld de, 87        17 87 0        address
add hl, de       25             into HL
ld (A A), hl     34 A A         and store
call B B         205 B B
DEFW             X X          The address of the
  * continue.....               particular function

Address (B,B):
ld bc, N N       1 N N        Load whatever
ld de, N N       17 N N         feed-data
ld a, N          62 N           is needed
ld hl, (A A)     42 A A       And jump to the
jp (hl)          233            Jump-USERF address
```

However, once you know that your machine will always produce (90,252) [but check the value that your machine produces for this address] as the Jump_USERF address, then there is no reason to stick to this particular layout. Indeed some of the following examples use an exactly equivalent but visually different pattern, as in:

```
ld bc N N        1 N N
ld de N N        17 N N
ld a, N          62 N
call USERF       205 90 252
DEFW             X X
  *   continue....
```

In most cases not all of the registers need to be loaded like this with feed data before using USERF; but when I describe the function I will say what data is needed.

The use of the in-line address might seem a bit convoluted, but the address is passed without needing to load it into a register-pair, which may or may not strike you as justification. The in-line technique is possible because the address to be returned to after the 'call' is, as usual, at the top of the stack. This is collected from the stack, the in-line entry is extracted and the true return address (incremented by 2) is put onto the stack to replace it. When the call has been completed, the return is made to the place marked by the asterisks so the rest of the program will run from there.

It is necessary to use the USERF approach for complex functions such as the Screen Run Routine, but to utilise the simpler ones I choose to use the simpler approach of switching-in bank 0 through the Memory Manager, because bank 0 contains block 0, which is where the required jump table resides. All that is needed then is to load the feed-data into the required registers, make a call to the appropriate place in the table, record the returned data, and then switch back to the TPA. The generalised listing is to the pattern:

```
call Bank_0      205 32 353     Switch in bank 0
.......          .......        Load the
.......          .......         data, make
.......          .......         the call, &
.......          .......         record the returns
ld a 1           62 1           Back to the
call MEM_M       205 33 253      TPA
   continue....
```

The jump table starts at (128,0) and the entries in it are each three bytes on from the one before, ie. at (128,0), (131,0), (134,0), ... etc. Some of the more useful ones are as follows.

MACHINE SPECIFICATION

The next two routines make it easier to write portable software because by interrogating them your program can discover which environment it is operating in.

Machine type (227,0)

This could be helpful if you are writing software for unknown machines, though I confess ignorance as to the full details. It returns '1' in A for the '8256' and '8512', and probably some other number for the '9512'.

Machine configuration (230,0)

This supplies enough information to make up for my ignorance concerning the previous entry. The returned data is as follows:

A reg	'0' = one disc drive fitted '255' = two drives
B reg	Number of memory blocks (16 or 32)
C reg	'0' = no serial interface '255' = serial interface fitted
HL pair	Address in block 7 of buffer table

On my machine the buffer table [at (232,255)] contains three entries of five bytes each. These entries show that there are three buffers in use, and give their details as follows:

Byte 0	Bank number
Bytes 1 & 2	Buffer address
Bytes 3 & 4	Buffer size in bytes

Which version ?

It is equally helpful in discriminating between operating environments if your program is given the means of finding out which version of CP/M is in force. Because the versions vary slightly in the locations of their routines their jump tables vary a bit in the addresses they use. For example address (10,252) may contain one of the following bytes, so inspect it to determine the Version No:

Version 1.1	38	Original '8256' version
Version 1.4	38	Main '8256/512' version
Version 2.1	50	Original '9512' version
Version 2.9	48	Latest '9512' version

SCREEN ROUTINES

The following five routines relate to various screen operations or provide information about it.

Screen data (191,0)

This facility is mentioned in Chapter 6 as a means of obtaining the cursor position, but it also reports on the current screen size as well. The full set of normally returned parameters is

B	Top line number	(0)
C	Left column number	(0)
D	Bottom line number	(30 or 31)
E	Right column number	(89)
H	Present cursor line	(0 to 31)
L	Present cursor column	(0 to 89)

Screen reset (194,0)

This clears the screen, moves the cursor to the top left, and initialises the screen settings to their default values (which are listed above).

Status line ask (197,0)

This asks whether the status line is enabled or not. The reply is in A. If A returns zero then the status line is disabled indicating that the line has been incorporated back into the normal screen thus giving it 32 text lines. A value of '255' indicates that the line is enabled, thus reducing the screen text size to 31 lines.

Status line set (200,0)

This enables or disables the status line according to the value in A when the routine is called. Zero disables the line and '255' enables it.

Screen Run Routine (233,0)

This major routine is the primary means of accessing the screen and character data in block 2, which cannot be accessed through a numbered block. The 'modular' approach described in Chapter 7 employs the conventional route via USERF to the Screen Run Routine because the screen- and character- data in block 2 are needed.

Before using the Screen Run Routine, the address of your manipulating routine (which must be in common memory) must be put into BC.

Once blocks 7, 2, 1, and 0 (the Screen Environment) are in circuit, the addresses of interest are:

Screen data	(48,89) to (47,179)
Roller RAM	(0,182) to (255,183)
Char matrix RAM	(0,184) to (255,191)

KEYING ROUTINES

The following routines relate to various aspects of keyboard use.

Key detect (218,0)

This useful routine simulates BDOS fncs 1 or 6, except that more information is provided. The carry flag indicates whether there was a keypress or not (ie. whether there is anything waiting in the keyboard buffer):

Cy set	Key pressed
Cy reset	No key pressed

If a character is available its **key-number** (see the manual on redefining the keyboard, page 109 [544]) is returned in C, but if Cy is reset then the contents of C will be uncertain. Whether any of the shift keys were also pressed will be indicated by the bits of the **B** register as follows:

bit 0	1	not used
bit 1	2	'EXTRA'
bit 2	4	'CAPS LOCK'
bit 3	8	Repeated key
bit 4	16	'NUM LOCK'
bit 5	32	'SHIFT'
bit 6	64	'SHIFT LOCK'
bit 7	128	'ALT'

Bear in mind that the routine searches the keyboard buffer and reports on each character or token that it finds there. So if you press 'SHIFT' and then also 'a' to give upper case "A", it will detect the SHIFT first

and then the shifted 'a', so you will need two calls in succession to obtain this information. The two will return the following data:

1st call	C reg 21	B reg 32	('SHIFT' shifted)
2nd call	C reg 69	B reg 32	("a" shifted)

If you want details of all the keys that have been pressed but you don't know how many have been, you will keep calling the routine until Cy is returned reset. And if you want information on all key-presses whenever they are made, you will do what the machine does, which is to call the routine at interrupts (at every 6th interrupt in fact).

Bit 0 is not used. A key held down is supposed to set bit 3 of B, but it doesn't do that for me. There is no actual 'CAPS LOCK' key, but this function is provided by ALT + ENTER, and that key combination sets bit 2.

See below for an alternative method of key-detection.

Key put (221,0)

This inserts a character or a token into the keyboard buffer from within a program so that the next fetch operation will recover it. (There is room in the buffer for ten, but if overflow occurs any more will be lost.) Hence you can change the keyboard setting by putting in, say, 'SHIFT' plus some harmless key number, and then recovering it by key-detect. (SHIFT LOCK cannot be induced like this.) The feeds are identical to the returns from the previous program.

KEY DETECTION FROM MEMORY

An alternative method of assessing which key, if any, is being pressed is provided by the fact that key-presses are recorded in the bits of the sixteen addresses above and including (240,191) in block 3. To use this data it is only necessary to switch-in block 3 and test the appropriate bit. When I first came across this method, I used the following piece of code to discover which bit of which address relates to which key.

The routine starts at the lowest address and tests each bit of each address until it finds a set bit. If it doesn't find one (no key pressed)

it starts again. When it finds a set bit it exits to store the address containing it and also stores the value left in B minus one, which is the bit number.

```
     ld a, 1           62 1           Await
     call BDOS         205 5 0          a key-press
Loop:
     di                243
     ld a, Block       62 131         Get block 3
     out (242), a      211 242          into Range 2
     ld hl Lowest      33 240 191     Point to lowest address
     ld a, 16          62 16          16 addresses
Loop2:
     ld b, 8           6 8            8 bits per address
     rlc (hl)          203 6          Test the bit
     jr c Report       56 13          Jump out if set
     djnz Rotate       16 250         Else try next bit
     inc hl            35             Next address
     dec a             61             Reduce count of addresses
     jr nz Loop2       24 244         And test this address if nz
     la a, Block       62 134         If zero, restore
     out (242),a       211 242          the block
     ei                251              and the interrupts
     jr Loop           24 227           and start again
Report:
     ld a, Block       62 134         Restore
     out (242), a      211 242          block 6
     ei                251              and restore interrupts

     ld (Addr), hl     34 A A         Store the address
     ld a, b           120            Bit No + 1
     dec a             61             Bit No
     ld (Addr2), a     50 A2 A2       Store bit No
     ret               201            Finish
```

Tables of which bit relates to which key are given in Appendix 5, though you can calculate them for yourself. If the the key number is 'n', then:

Key numbers	Offset	Bit number
0 to 71	INT n/8	n Mod 8
72	9	7
73 to 80	10	n - 73

The offset is counted from the first table address, which is (240,191).

This method of key-detection is superior to other methods in some respects. Because the combination of set bits when several keys are pressed is additive, it enables you to detect any combination of keys, and because the bits are reset as a key is released, key-release as well as key-press can be detected.

OTHER KEY FUNCTIONS

The remaining key routines accessible through the jump table are likely to be employed only once during the initialisation of the program. This could be attended by using the Amstrad utilities in a 'profile.sub' file, but I personally find the programming approach more convenient, and it it is certainly more flexible. You will need to access the last two through the USERF function because they use ASCII codes etc. (which need access to block 2).

Repeat speed (224,0)

If a key is held down there is normally a delay of 0.6 seconds before a second character is generated. (This gives clumsy typists time to lift our fingers.) Once repeating has been established identical characters are generated every 0.04 seconds, ie. at a rate of 25 per second. These default settings can be changed by calling (224,0) with the new values, which are in 1/50ths of a second, in HL.

H reg Start delay
L reg Repeat delay

Set key (215,0)

This function allows you to assign a particular ASCII code or expansion token (see below) to a given key-press, and you can state which shift state, if any, you are referring to. The feed data is:

B reg ASCII or expansion token
C reg Key number (manual page 109 [544])
D reg 1 = normal
2 = shift (caps)

4 = ALT
8 = shifted ALT
16 = EXTRA

A value of '3' in the D reg, for example, would specify both 'shifted' and 'normal'. The following code resets the upper case 'Z' (shifted "z") to print "a" instead of "Z" (by way of example only; I'd be surprised if you come across an urgent need for this particular conversion).

```
ld b, 97         6  97          ASCII 'a' into B
ld c, 71         14  71         Key num 71 (='z')
ld d, 2          22  2          Specify 'shifted'
call SCR_RN      205 90 252     Call USERF
DEFW             215  0         Address of 'Set Key'
   continue....
```

Any other conversion can be set along the same lines, but if you specify a key number greater than 79, which is the highest available, then no action will be taken.

Set Expand ('212,0)

This sets the string to which an expansion token (see below) is to be expanded. If the string won't fit into the table because the latter is full, the change is not made. Your only course then is to take over or shorten the entries of other tokens, but I have never tried that. On completion Cy is set if the expansion was accepted, so test it to see.

The feed data is:

B reg The expansion token (128 to 159)
C reg The string length (0 to 31)
HL pair Address of the expansion string

USING EXPANSION TOKENS

Expansion tokens are the 'ASCII codes' in the range 128 to 159. They are not really ASCIIs, just the reference numbers to the entries in the table that stores the expansion strings, and obviously not all of them are used. The number of the first entry is 128, that of the second is

129, etc. If you (or the makers) allocate a particular string to token number 128 (say) then the first entry in the table will be that string. If you have also used the routine at (215,0) to set a key to token 128, then when the key is pressed the string will be printed. So instead of getting a mere "z" out of key number 71, you can if you wish get

"HI FOLKS, HOW ARE YOU TODAY?"

though I won't insist on it.

There are quite a few cases in which expansion tokens are very useful. Apart from putting cheery messages onto the screen with a minimum of effort, when used in conjunction with a high-level language such as BASIC, or with CP/M, the host system will try to execute it. So if you set the function keys to "GOTO 100", "GOTO 200", "FRED", or whatever, you will be able to choose a single keystroke to access the required bit of program or run the file called "fred.com", and this facility can be integrated into a menu.

For example, the menu choices might look like:

" *Press the Initial letter of your selection :*
Bert
Fred
Harry
Sheila "

If the normal and the shifted versions of the keys "b", "f", "h", and "s" have been set to tokens that expand to the full name of each of these individuals, then their names (and any other details about them that you include) will appear at the current print position when their key is pressed.

The token table

When allocating tokens to strings bear in mind that most of them are allocated already, so in re-allocating them you will be over-writing the original entries. When a token entry has been over-written and allocated to a new key, the old key may still be allocated to it, so two keys will produce the new response.

The token table is 150 bytes long and starts at (118,40) in block 0 in

my machine. In case yours may be different, to locate it look for the byte sequence:

1 3 1 26 1 26 . . . .

There are 32 entries in it, and they may vary in length from 1 byte to 32 bytes. The first byte of an entry defines the length of the text of the entry, and that may vary from 0 to 31. The text consists of ASCII codes. Hence the first entry, which is " 1 3 ", specifies that this entry is 1 byte long, and that the text byte is '3'. The 'empty' tokens such as '159' (see below) have no text, and therefore have entries consisting of a zero.

The table contents

When you get your machine, tokens 128 to 154 are allocated to the 'control keys' "Ctrl C", "Ctrl Z", etc. (see manual page 112 [538]). Tokens 155 to 159 are unallocated at cold-boot. In fact the 'empty string' of token 159 is used as the "don't do anything" entry; keys allocated to token 159 have no effect when pressed, and allocating them so is a good way of uncluttering the keyboard so that the operator is not constantly making keying errors.

Expansion tokens can't be nested, ie. you can't put one within an expansion string and expect to see it expanded; you'll get the screen graphic corresponding to the ASCII number concerned. And you can't re-allocate such keys as 'SHIFT', 'ALT' etc. If the key is numbered in the keyboard diagram (see the Manual page 109 [544]) then it can be re-allocated, if it isn't numbered then it can't be.

The size considerations on the previous page indicate that only 118 bytes of the table's 150 are available for text (32 bytes are used to specify the lengths of the entries), and entries that would make the table exceed its permitted length, or which are themselves longer than 31 bytes, are not accepted.

Example Program

The routine will typically be used in conjuction with the previous one, as the following example illustrates. It sets 'f8' to print the 10-letter string at address (S,S).

```
ld b, 128       6 128         1st token
ld c, 10        14 10         Ten letters
ld hl N N       33 S S        Point to the string
call USERF      205 90 252
DEFW            212 0         Address of 'Expand'

ld b, 128       6 128         Same token
ld c, 77        14 77         Point to the f7/f8 key
ld d, 2         22 2          Shifted (=f8 only)
call USERF      205 90 252
DEFW            215 0         Address of 'Set'
    continue.....
```

DISC FACILITIES

The earlier routines in the bank 0 jump table relate to disc drive set up and handling. They are not likely to be of interest, but the following bare bones indicate what is available.

Disc drive initialise (128,0)

This resets all the disc parameters to the default values and turns the motor off. The default times for the drive motor are:

Motor on timeout	1 sec
Motor off timeout	5 secs

Because mechanical systems suffer more wear from stopping and starting than they do from running, the motor is set to run on for five seconds after the last read or write request to ensure that this isn't just a pause for housework. The routine also resets five other parameters

Disc drive set (131,0)

This sets the above parameters to the new values in a 26-byte data block called the XDPD (Extended Disc Parameter Block).

Read sector (134,0)

Write sector (137,0)

Check sector (140,0)

Format a track (143,0)

Login (146,0)

Select format (149,0)

Get status (152,0)

Read ID (155,0)

Init parameter block (158,0)

ditto *Extended block* (161,0)

Motor on (164,0)

Motor off (167,0)

All but the last two disc drive routines should not be used without detailed information on the data required for satisfactory operation, though I have it vaguely in mind to publish something covering that one day.

The action of the last two in the list are, incidentally, given by the very much simpler bits of code:

```
ld a, N          62 N
out (248), a     211 248
```

If 'N' is given the value 9 the motor is switched on. The value 10 switches it off.

Chapter 3

Time and again

Time of day

CP/M operates a clock that an m/c programmer can use to display the time of day, though it is not much good for interval-timings of the 'stop watch' variety. In fact the clock is something of a calendar as well because it also records a count of days, though you would have to leave the machine permanently switched on to make use of its long term recordings because the data is zeroised at each coldboot. The clock works by counting the number of interrupts, which occur at 1/300ths of a second.

To obtain a readout BDOS fnc No 105 is called with DE pointing to a 4-byte 'TOD' block, which will receive the required information ('TOD' might mean 'time of day'?). Bytes 0 & 1 get the day number, byte 2 gets the hour number, and byte 3 gets the minutes number. Additionally the number of seconds is put into the A register. But bear in mind that all these numbers are in BCD, except for the 'days' count which is a 16-bit number. With the TOD block located at the address 'TODAD', the data availibility is as in the following diagram.

Bytes 0 & 1: Days	16 - bit number
Byte 2: Hours	TENS | UNITS
Byte 3 : Minutes	TENS | UNITS
A reg : Seconds	TENS | UNITS

The time data in the TOD block

In fact you don't need to call fnc 105 to get the data because it is also stored at five addresses in high memory. The 'hours' are put into (246,251), 'minutes' into (247,251), and 'seconds' into (248,251). Addresses (244/5,251) store the count of days, but setting the time to 23hrs 59mins and waiting for the next increment indicates that the 'days' are stored as a simple 16-bit number, and not in BCD as the other data is.

(244,251)/(245,251) : Days	16 - bit number
(246,251) : Hours	TENS | UNITS
(247,251) : Minutes	TENS | UNITS
(248,251) : Seconds	TENS | UNITS

The time data in memory

Displaying the time

The time data can be displayed on the screen at your chosen location either continuously (with a constant update for each new second) or, if you prefer it, briefly at chosen moments so that the user is made aware how time is passing (as might be appropriate in games, tests, etc., where elapsed time is significant).

The first requirement is a 'time-string' for printing by fnc 9 into which the relevant ASCII codes will be inserted. Lets say this is to be at address ADDR, which in red-biro formulates to (A0,H), with (A1,H), (A2,H), etc., being the addresses following in sequence. The time-string will be composed of the bytes listed below, which assume that you want only the 'hours', 'minutes' and 'seconds', but not the 'days', though the 'days' can be added easily enough.

The first five bytes determine the screen position at which the time-string is to appear. If they are both given the value '32' then it will be printed at the top left. If they are given the values '63' and '77' respectively, it will appear at the bottom right. The '13' at byte 0 resets CP/M's column count to zero to ensure that printing is where it is intended in spite of CP/M's mutinous tendency.

Address	byte	purpose
(A0,H)	13	'Carriage return'
(A1,H)	27	Escape sequence
(A2,H)	89	'print at'.
(A3,H)	L	Line No (top line=32)
(A4,H)	C	Colm No (left colm=32)
(A5,H)	32	Space
(A6,H)	32	Space
(A7,H)	DEFB	ASCIIs of the two
(A8,H)	DEFB	'hours' digits.
(A9,H)	46	"." (separator).
(A10,H)	DEFB	ASCIIs of the two
(A11,H)	DEFB	'minutes' digits.
(A12,H)	46	"." (separator).
(A13,H)	DEFB	ASCIIs of the two
(A14,H)	DEFB	'seconds' digits.
(A15,H)	32	Space
(A16,H)	32	Space
(A15,H)	36	String-end marker.

The four '32' bytes (spaces) ensure that the time digits are visually distinct from any text that may have encroached into their area. You can make the figures stand out in reverse video by replacing the first two 32s by '27 112' and the second two by '27 113', though in reverse video my preference would be to have an extra space before and after the digits as well. The DEFBs are the ASCII codes of the digits that will be inserted into the string by our time interpreter sub-r.

The two '46' bytes are the ASCII of "." which will act as visual separators between the three elements of the time, though you could use others (32 for 'space', 47 for "/", or 58 for ":", for example). The '36' is the standard string-end marker (delimiter), though if you have been using BDOS fnc 110 you will presumably need a different value in here.

Calculating the time

To display the time you can either extract the data from your own TOD block, or from the PCW's storage in high memory, whichever you prefer. For no particularly good reason I have chosen to use a TOD block.

```
Start
      ld de, TODAD    17  N  N      Update
      ld c, 105       14  105        the
      call BDOS       205  5  0     TOD block

Seconds   .....       (see below)
```

This gets the time data (not including the 'seconds') into the block. The next step is to preserve the seconds value that will have been put into the A register because A is about to be used for other purposes:

```
HLDIG
      ld b, a         71            Save the 'seconds' into B
      and 15          230  15       Discard the 4 high bits
      add 48          198  48         and convert ASCII
      ld l, a         111           Put the 'units' into L
      ld a, b         120           Recover original A
   *  srl a           203  63       Then discard the
      srl a           203  63         4 low
      srl a           203  63         bits to get
   *  srl a           203  63         'tens'
      add 48          198  48       Convert to ASCII
      ld h, a         103           Put the 'tens' into H
      ret             201           Return to main
```

I have actually made this into a sub-routine and have called it 'HLDIG'. It returns the required two ASCII digits in HL so they are ready for insertion into the string. As a sub-r it makes the same code sequence available for obtaining each part of the display without this

needing to be written out three times. Hence to put the 'seconds' figure into the string, continue the main sequence with:

```
Seconds
     call HLDIG       205  N   N      Convert the value in A
     ld (A13), hl      34  A13 H       and put result into string
            continue with 'minutes' . . .
```

The same approach is used to obtain the 'hours' and 'minutes' except that they have to be fetched from the TOD block first. When these too have been dealt with the string is printed, so the final three steps of the main routine are:

```
Minutes
     ld a, TODAD+3  58  N   N      BCD into A
     call HLDIG     205  N   N      Convert it to
     ld (A10), hl    34  A10 H       'minutes' & put into string

Hours
     ld a, TODAD+2  58  N   N      BCD into A
     call HLDIG     205  N   N      Convert it to
     ld (A7), hl     34  A7  H       'hours' & put into string

Print
     ld de, A0       17  A0  H      Print
     ld c, 9         14  9           the
     call BDOS      205  5   0       string
     ret            201             And finish
```

Notice that because the time display procedure will be called repeatedly to provide each update, it too is a sub-routine and therefore ends with a 'ret'.

Program technique

Within HLDIG, I have chosen to use 'srl a' (see ✻ .... ✻) because it makes the point of the sub-routine more obvious. However,

```
     rra          31
     rra          31
     rra          31
     rra          31
     and 15       230  15
```


would have been preferable on grounds of both brevity and speed. The alternative is quicker and involves less code, and any spurious bits that it may rotate into the top end are eliminated by the 'and 15'. There will be other cases like this in future examples, but I will stick to my practice of writing obvious code even if it is not clever code.

Time updating procedure

The above sequence and its sub-routine will fill the time string with the current time and display it on the screen.

Because the CP/M clock doesn't register time intervals shorter than a second, the display will obviously not change more frequently, but, if you are making a continuous display of the time and you require a fair degree of accuracy and a regular 'tick' from your clock, then you need to call the procedure much more often than that because you cannot guarantee that any call will exactly coincide with the change to the next second. If you call it ten times a second, then the length of your 'seconds' will be a second plus or minus a tenth of a second, etc.

If the time display is to be spasmodic (ie. used as a kind of prompt), then you will display it whenever it is needed and then pause either for a specific interval or until the user presses a key, at which point you will cancel the display by printing a cancel-string such as the "Blank line" [escape sequence (27 75)] or a set of 'spaces'.

Setting the time

The CP/M clock is set to zero when the computer is turned on, but you can reset it to any time you like by inserting the necessary data into the TOD block and calling BDOS fnc 104. (Fundamentalist CP/M users think that day zero is 1st Jan 1978, but there's no reason why you should.) Alternatively you can insert your information into the memory addresses referred to earlier because CP/M uses them to store the data it works from. If you want to insert a value of '1' (one minute, say), then you would put '1' into the relevant byte, but for a value of '10', you would put in '16', because the four left bits count in 16s. Similarly the value of '11 minutes' would be given by a byte value of 17 (1 x 16 + 1). '23 hours' would be given by '35' (2 x 16 + 3), and '59' minutes by '89' (5 x 16 + 9).

Using fnc 104 makes it possible to reset to a precise time-signal. To

get an exact correspondence with Greenwich Mean, or whatever, put the data into the block and then run:

```
ld c, 1          14  1        Await
call BDOS        205  5  0     a key
ld de, TODAD     17  T  T
ld c, 104        14  104      Set the
call BDOS        205  5  0     time
      continue . . .
```

You press any key when the time signal corresponds to the time you have put into the block.

PAUSES

There are several ways of measuring time intervals. The simplest is based on the fact that m/c operations take a known time for their execution, so this can be used to introduce fixed 'pauses' into a program. Others methods can be devised with just about as much complexity (and accuracy) as you like, but let's start with the simplest.

Because programmers are usually hell-bent on making their code run as fast as possible, it may seem unlikely that pauses could ever be desirable, but, whilst the housekeeping and calculation side of programs should be fast to be convenient, their interactive aspects need to be paced to the mood of the user. If you are zapping aliens you probably won't want regular intermissions for soothing music, but a banker tending his millions, or a genius planning his career, might feel better reassured if the display reacted to his forefinger with gentlemanly dignity, rather than with confusing, perhaps even insulting, haste. I'm personally not much enamoured of computers that constantly prove that they think far faster than I can.

For these 'gentlemanly pauses', the duration need have no precision about it so a simple loop will be good enough. The following one pauses for the number of seconds indicated by BC and is fairly accurate (when tested against interrupts the error was about 1 in 300, say 0.3%). On entry, B requests the number of whole seconds and C requests the fractions. If you want one second you put '1' in B and zero in C and then call the sub-r. The content of C indicates the number of 1/256ths of a second, so half a second would be given by '128' in C, a quarter by '64' in C, etc.


```
ld a, 232     62  230      The
ld hl (0)     42  0  0      looping
ld hl (0)     42  0  0      procedure.
dec a         61
jr nz -9      32  247
dec bc        11           Decrement the
ld a, b       120            count
or c          177            of 1/256ths
jr nz -16     32  240      Repeat if not zero
ret           201          Else finish
```

The first five instructions load HL from an arbitrary address 460 times, which takes about 1/256th of a second. BC is then decremented to count the loops and the sub-r ends when BC contains zero. Because BC is decremented before being tested, if you start with zero in BC this will be interpreted as a request for 65536 loops, ie. for 4 mins 16 secs, which is the longest pause the sub-r can provide.

Pause timing

If you devise your own loop and you want to know just how long your pause lasts, include it in a loop that repeats it say a thousand times and measure the time or the count of interrupts for the thousand. If you are seeking real precision you'll need to calculate the time taken by the looping procedure that you devised for the test (which is usually very small in comparison) and deduct it.

Inerrupts occur every 1/300 of a second regardless of the duration of the interrupt. However if you have timed a pause length with a short interrupt in operation (no ISRs, say), and you subsequently add to the length of the interrupt by adding an ISR, then your pause timing will no longer be accurate.

Uses of pauses

Some computer programs need drama; adventure games for example, and pauses find use in them to increase the sense of tension and uncertainty. If, faster than a midge's blink, you get the answer to all your appraisals of (guesses about) the current dangerous situation (like, should I kill the dragon now or smile at the princess first?), then the game soon deteriorates into a rather meaningless sequence of

keypresses. If on the other hand the computer waits briefly before responding, then an illusion that your response needs thinking about can be created, and this lifts the human interest. Your choice seems to count for more.

Other programs simply need to be slowed down to induce a reassuring atmosphere. The desirability for pauses between menu pages is referred to in Chapter 10, and there are other situations where pauses induce a more friendly 'human scale' feeling. Many people who don't, and probably don't want to, think at electronic speed will be reassured by packages that present information and choices in a leisurely way.

'Halt' pauses

Although the m/c instruction 'halt' is proscribed in *The Amstrad CP/M Plus* , I have received more than one assurance that it can be used without calamities occurring. The instruction causes operations to cease until the next interrupt occurs, so it creates a natural 'pause' that would be 1/300th of a second long in loops such as the following, which would generate a pause of 1/10th of a second:

```
ld b, 30        6  30        Count of 30 interrupts
halt            118
djnz -3         16  253      Loop 30 times
```

INTERVAL MEASUREMENT

The approach used in producing pauses can also be used to produce a stop-watch timer. Let's start by devising a program that sounds a beep and then times how long the user takes before pressing a key. The time is to be displayed on the screen to plus or minus a thousandth of a second, and we won't bother with times longer than 9.999 secs (longer than that could be hardly be called a 'response').

Once again we need a time-string, and a start-string also. The latter simply clears the screen, sets the print position, and then sounds the beep to begin. It consists of the bytes

13 27 69 27 89 44 72 7 36

The time-string is to hold four digits and a decimal point. It consists of:

```
(A0,H)    DEFB    Secs
(A1,H)    46      Decimal point
(A2,H)    DEFB    Tenths
(A3,H)    DEFB    Hundredths
(A4,H)    DEFB    Thousandths
(A5,H)    27      Print
(A6,H)    89        in
(A7,H)    60        lower
(A8,H)    72        screen
(A9,H)    36      String end.
```

The escape sequence before the end-marker resets the print position to the bottom of the screen so that the pressed key does not tack on a letter at the end of the time digits.

As we intend to measure in thousandths of a second we need a 'clock' that 'ticks' at that speed, and it turns out that the following segment of code fits the bill quite nicely:

```
ld e, 92        30  92
dec e           29
jr nz -3        32  253
```

All it does is to load E with the value '92' and then decrement this number until zero is obtained. As a timing element it has the advantages of being short and simple as well as being easy to adjust; if the clock runs too slow then the loaded value can be reduced to less than 92, and if it runs too fast the value can be increased. I obtained the '92' by including the element in a loop that repeated it for a timed 30 seconds and adjusting the value until the loop count came to as near to 30,000 as I could manage. Hence the elapsed time of the element plus one pass of the loop was as close to 1/1000th of a second as my patience would allow (within 0.5%). Because the loop counts itself and ceases to loop when a key is pressed, it has all the characteristics we need for our counter of thousandths of a second. It is listed below.

The count of loops is to be recorded in HL, so HL is first zeroised. Each time the timing element is run, the count in HL is incremented and saved by 'push hl'. BDOS fnc 11 is then used to test for a key-press, and the count recovered by 'pop hl'. If no key-press is detected then the sub-r continues, but if one is found it terminates with the count in HL.

```
TIMER
    ld hl, 0        33  0  0      Zeroise the count
    ld e, 92        30  92        The
    dec e           29              timing
    jr nz -3        32  253         element
    inc hl          35            Increment the count
    push hl         229             and save it
    ld c, 11        14  11        Test for a
    call BDOS       205  5  0       key-press
    pop hl          225           Recover the count
    or a            183           If no key
    jr z -16        40  240         then repeat
    ret             201           Else return to main
```

We now need to insert the ASCII version of the count into the time-string, which is arranged by:

```
FILL
    ld de 1000      17  232  3    Insert
    call CALCDIG    205  N  N       the
    ld (A0), a      50  A0  H       seconds
    ld de 100       17  100  0    Insert
    call CALCDIG    205  N  N     1st decimal
    ld (A2), a      50  A2  H       place
    ld de 10        17  10  0     Insert
    call CALCDIG    205  N  N       2nd
    ld (A3), a      50  A3  H
    ld a, l         125           Insert
    add 48          198  48         3rd
    ld (A4), a      50  A4  H
    ret             201           Ret to main
```

This uses the sub-r CALCDIG to work out the 'thousands' in HL, which is the number of whole seconds, then the 'hundreds', which is the 'tenths of a second', etc. CALCDIG operates by repeatedly subtracting 1000 (or whatever) from what is left in HL and counting the subtractions. It then adds '48' to the count to obtain the ASCII code of the relevant digit and returns this in A.

```
CALCDIG
    xor a           175           Zeroise the count
    inc a           60            Increment the count
    sbc hl, de      237  82       Subtract the DE value
```



```
    jr nc -5        48  251       Repeat if no carry
    add hl, de      25            Else restore the last subtraction
    dec a           61              and decrement the count
    add a, 48       198  48       Convert to ASCII
    ret             201           Return to FILL
```

The complete timer

Having produced the pieces, the whole operation can be accomplished by combining them as indicated on the following page. As written, this leaves some cleaning up to do, which I leave in your capable hands. For one thing the display is bald; it gives no instructions and it doesn't say what the displayed number is. Nor does it ask you if you want another 'go'

It would also be as well to reject the result if the first digit has an ASCII code of more than 57 (because the content of HL exceeded 9999, equivalent to 10 secs or more), otherwise the displayed number will have a strange first digit.

```
    ld de START     17  S  S      Print
    ld c 9          14  9           the
    call BDOS       205  5  0       'start' string
    call TIMER      205  T  T     Set the timer going
    call FILL       205  F  F     Fill the time-string
    ld de TIME      17  A0  H     Print
    ld c 9          14  9           the
    call BDOS       205  5  0       'time-string'
    ret             201           Finish
```

Very accurate time of day

There is no objection in principle to combining the CP/M clock with a timer based on the above so that fractions of a second could be displayed continuously, but

a) the difficulties of truly synchronising the two systems would make the union inconveniently difficult
b) the timer operates in 'run-time' thus preventing any other program being operated whilst it was in use.

The only way round these is to operate the 'precise clock' entirely from interrupt counting, thus making it possible to display tenths and hundredths of a second by counting in sets of three and sets of thirty interrupts. It isn't possible, nor very desirable (unless you run a spurious TV game show), to display the time continuously at a finer accuracy than this. Interrupt counting is described in the next chapter.

Chapter 4

Forgive the interruption

I am indebted to Iain Stirzaker for clarifying some of the finer points of interrupts, particularly those relating to IM 2.

Without exception computers need interrupts. Interrupts introduce a kind of 'time-sharing' in which the machine takes a little bit of time for its own purposes and gives you the rest. If you subsequently time-share your fraction with someone else, then that's your business.

The 'own purposes' include all the activities that the machine must take care of at regular intervals (its internal housekeeping), plus user services such as checking to see if a key has been pressed recently and taking the appropriate action if one has been. To a programmer the details of all this are not of much importance compared with the fact that as regular as clockwork (if the clockwork is very precise) the machine leaves whatever user program is being run and goes off by itself for a microsecond or two. This happens every 1/300th of a second to an accuracy that you needn't check up on.

TYPES OF INTERRUPT

The interrupt signals are in fact generated by the ULA (Uncommitted Logic Array), and there are two types of them: **Maskable** and **Non-maskable.** As the name implies, the Z80 cannot ignore the non-maskable interrupts, but the instruction *di* (disable interrupts) causes it to pay no heed to the maskable type, though these can later be put back into force by the instruction *ei* (enable interrupts).

If you want to know whether the maskable type interrupts are enabled or disabled, use the sequence:

```
ld a, r           237  95
jp pe, Prog1      234  N1  N1
jp po, Prog2      226  N2  N2
  continue . . .
```

As an alternative, the first line could read

```
ld a, i           237  87
```

In one case the contents of the Refresh Register are loaded into A, and in the second case the contents of the Interrupt Register are. Both cause the Interrupt Flag to be copied into the Parity Flag and this is later interrogated by the 'jump' instructions. The code therefore causes a jump to the address (N1,N1) if the interrupts are enabled, or a jump to address (N2,N2) if they are not.

Although the PCW also makes use of the non-maskable interrupts, they are not hugely interesting and we will restrict our attentions to the maskable type.

MODES OF INTERRUPTION

The Z80 has three maskable interrupt modes. These are numbered 0, 1, and 2, and selection between them is by the instructions *im 0*, *im 1*, or *im 2* respectively. When the PCW is switched on, the Z80 automatically selects Mode 0 but almost immediately it is switched into Mode 1, and after that Mode 1 is the PCW's standard mode. Mode 0 is almost not interesting at all unless you are a hardware fanatic.

INTERRUPT MODE 1

In Mode 1, when an interrupt is signalled to it by the ULA, the Z80 completes the instruction that it is in the process of executing, preserves the contents of the Program Counter on the stack, and disables the interrupts. It then makes a jump to address (56,0) where it finds the instruction to make another jump, this time to address (161,253), which is the start of the standard interrupt sequence that has been programmed into the machine. (It would be wise to check this address on your machine; the range of PCWs has annoying variations in things like addresses). Such sequences, whether built-in or written by interrupt programmers such as yourself, go under the generic title of **Interrupt Service Routines** (ISRs).

In fact most of the standard ISR resides in block 0, so the part of it in block 7 simply saves some registers, switches-in block 0, and then makes a jump to the address found at (167,254) [but found at (119,254) in the case of the '9512']. For my machine this found address is (64,30); check yours.

Using Mode 1

Happily, you can write an additional ISR of your own and persuade the PCW to run it just before it goes off to its own housekeeping. You do this by inserting your ISR in block 0 and changing the address at (167,254) [(119,254)] so that it points to your ISR, not to the usual one, but naturally your ISR must conclude with a jump to the built-in ISR so that is always run as well.

Happily also, there are some unused areas in block 0 into which your code can be placed so that it is accessible to the interrupt procedure, and you can put more code into block 7.

No	Start	End	bytes	comment
1.	* (64,0)	(95,0)	32	Zeroised at startup
2.	(240,37)	(223,38)	239	Uncertain
3.	(232,38)	(0,48)	large	Uncertain

Note that address (64,0) (*) is used by the '9512', so Area 1 for that machine starts at (65,0). Between areas 2 & 3 are some bytes

that cause mayhem if you disturb them, and the safety of these two areas is not guaranteed in all circumstances. When programs are loaded from disc the lower part of Area 3 is used, though its upper part above say (0,42) is believed to be relatively free from interference.

On the subject of safe areas, the CP/M copyright message lies between (74,246) and (124,246) in block 7, so you can put anything you like into this very high area knowing that it will not be overwritten.

The 3 Essential Steps

Running a standard PCW-type of ISR (ie. one that is intended for Mode 1 operation) involves three separate stages:

1. Put the bytes of the ISR into block 0
2. Change the interrupt start address
3. Use the ISR.

Step 1

To insert a program into block 0 you have to 'switch-in' block 0. As indicated earlier there are several ways of doing this but for no especially good reason I will select Bank 0 (which contains block 0) using the Memory Manager. (Alternatively you could invoke the Screen Run Routine which also invokes block 0, or employ the Empirical Method with 'Bank 26'). My listing looks like:

```
ld hl, ( )      42 167 254    Get interrupt start address
ld ( ), hl      34 A A          and store
ld a, 0         62 0          Select Bank 0
call MEM_M      205 33 253
ld de N N       17 64 0       Destination
ld hl N N       33 P P        Source
ld bc N N       1 B 0         Number of bytes
ldir            237 176       Transfer them to Block 0
ld a, 1         62 1          Back to
call MEM_M      205 33 253      the TPA
    continue.....
```

This program and the bytes of the proposed ISR must be in common memory. The latter must be waiting at (P,P) in the above case, and the total number of them is 'B'.

Remember that the ISR must terminate with a jump to the usual interrupt start address, so before you do anything else you must extract it and save it in common memory; at the address (N,N) say. Then the return to the normal interrupt address can be achieved by:

```
ld hl, ( )      42 N N
jp (hl)         233
```

which is the way that your PCW-type ISR should terminate.

Step 2

Changing the stored interrupt start address is simple enough but it is necessary to disable the interrupts while it is being done. The operation should look like:

```
ld hl 64        33 64 0       New start address
di              243           Interrupts off
ld ( ), hl      34 167 254    Insert it
ei              251           Interrupts on
    continue....
```

For clarity I have used the simple expedient here of poking the new address into (167,254), which is what my system needs. On the '9512' you would poke it into (119,254).

If your ISR is needed whilst only a part of your program is running, ie. if it becomes surplus to requirements at some stage, then you may as well switch it off to speed the rest of the program up. It is in any case necessary to switch it off if you intend to replace it by an alternative. Switching off is the reverse of switching it on:

```
ld hl I_START   33 64 30
di              243
ld ( ), hl      34 167 254
ei              251
    continue....
```

To install a replacement ISR you would run the above code and then go back to Step 1 to copy the new bytes into block 0 and follow the procedure as before.

Step 3

The method of using the ISR will naturally depend on what it consists of. Usually the ISR simply updates some data and makes it available for inspection. The following example forms the basis of a precise clock by counting interrupts.

Precise timing

A basic ISR for counting interrupts might consist of:

```
push hl        229          Save HL
ld hl, ( )     42  A  A     Get existing count
inc hl         35             and increment it
ld ( ), hl     34  A  A     Save the new value
pop hl         225          Restore HL
jp I_START     195  64  30  Go to the normal Interrupts
```

HL is pushed at the start and popped at the end so that its content is preserved. It is **essential** to save in this way the contents of at least HL and AF if they will have their values changed by the ISR. Their original values are required by BIOS when the ISR has completed its task. It may be necessary to preserve the contents of DE and BC as well but I have never checked on this, I have always pushed them at the start and popped them at the end whenever they are to be used in the ISR.

And, to be on the safe side, it is as well to assume that any 'calls' made during the ISR will corrupt all the registers, so 'push' and 'pop' them all if any 'calls' are made.

In the last line I have used the start address that applies to my machine. You must check on the value for the machine in question.

All this ISR does is to increment the value stored in the word at (A,A). You could use this to test the accuracy of your pause loops, or otherwise to time intervals of short duration by a sub-routine such as the one at the top of the next page. On completion, HL will contain the count of the interrupts that occurred during the interval, but because they happen 300 times a second the count will rise to a maximum of 65535 in 218.45 seconds, ie. in 3 mins 38 secs, so the interval must be shorter than this if the count is to mean anything.


```
ld hl 0        33  0  0     Zeroise the
ld ( ), hl     34  A  A       count
.....          .....        The timed
.....          .....          interval
ld hl, ( )     42  A  A     The final count
    continue....
```

A more comprehensive ISR for interval timing would increment a full set of addresses in block 7, each of which stored the count of hundredths, tenths, seconds, etc., to whatever range you wish to go. The following example records up to 9.99 seconds, though you could extend to counting minutes, hours, and days as well if you felt inclined. This 50-odd byte long routine could be put into area 2, though if you wanted it in common memory it could be pointed to by a three-byte entry at (64,0) in block 0, ie:

```
jp ISR         195  Z  Z
```

The timer routine at (Z,Z) could consists of:

```
ISR
    push hl        229          Save HL and AF
    push af        245            because used later
    ld hl N N      33  A  A     The 1st address
    inc (hl)       52           Increment it
    ld a, (hl)     126            and check it
    cp 3           254  3       If 2 or less
    jr c N         56  38         then finish else
    ld (hl), 0     54  0          reset to zero
Hundredths
    inc hl         35           And increment
    inc (hl)       52             the 2nd address
    ld a, (hl)     126          As above
    cp 10          254  10        except
    jr c N         56  29         count in tens not threes
    ld (hl), 0     54  0
Tenths
    inc hl         35
    inc (hl)       52
    ld a, (hl)     126          Ditto 3rd addr
    cp 10          254  10
    jr c N         56  20
    ld (hl), 0     54  0
```

continued on the next page . . .

```
Seconds
     inc hl          35
     inc (hl)        52
     ld a, (hl)      126            Ditto 4th address
     cp 10           254  10
     jr c N          56  11
     ld (hl), 0      54  0
Tens
     inc hl          35
     inc (hl)        52
     ld a, (hl)      126            Ditto 5th address
     cp 10           254  10
     jr c N          56  2
     ld (hl), 0      54  0

     pop af          241            Restore AF
     pop hl          225             and HL
     jp I_START      195  64  30    Go to normal Interrupts
```

The procedure is to increment the address (A,A) at every interrupt, but when the count reaches '3' (A,A) is restored to zero and the next address is incremented. Hence the second address is incremented every 1/100th of a second. When the content of this address reaches '10' it is reset to zero and the next one is incremented. Thus the third one is incremented every tenth of a second, etc.

(Whilst on the subject of timing, remember that all ISRs make the interrupts last longer. This naturally impacts on any program that is based on interrupt procedures of the usual length. So if you had previously carefully calibrated a loop-type pause under the original regime, you will now find that it takes longer to run, about 0.5% longer in fact.

And if you want to be really nit-picking the situation is complicated further because for two out of three interrupts the delay is very short, but during those interrupts when the contents of several addresses have to be changed (when the 'seconds' change all five addresses need attention) the delay is proportionately longer. A judicious rehash could produce the same count without the irregularity but I take it that, with such concern over precision, you will have no trouble in getting it right. Delegation, this is called.

To make the question of timings clearer:

1. Interrupts occur at 1/300ths of a second regardless of what you do.
2. If you make the interrupt last longer by adding complications, there is less time to be allocated to your normal program, hence it can do less in a given period, so your loops won't loop as often in a given time.

INTERRUPT MODE 2

This is the most powerful of the three modes and it is therefore disconcerting to find it proscribed in *The Amstrad CP/M Plus*, page 140, though Iain Stirzaker has carefully worked through the means by which it can be used. Whereas a Mode 1 interrupt always jumps to (56,0) to start its excursion, in Mode 2 the programmer decides for himself where this jump shall be to and he informs the Z80 of his requirements by specifying the **interrupt vector**. This 'decide for yourself' approach is obviously more flexible and it permits operations that are not possible otherwise.

The interrupt vector

To initiate Mode 2 operations, it is necessary to employ the Interrupt Register, which we will call 'I'. The interrupt vector contains the address at which the ISR starts, and 'I' contains the high byte of the interrupt vector. Normally the low byte of the interrupt vector is 255 (see below for abnormally). Suppose that we put X into 'I' and then we initiate Mode 2. The interrupt vector has been declared to be (255,X) so the Z80 looks at the address (255,X) and extracts from it the start of the ISR. Note that (255,X) is not the start of the ISR. The start of the ISR is **contained within** (255,X). [The low byte of the ISR-start is at (255,X), and the high byte of the ISR-start is at (0,X+1).]

Abnormally

It is theoretically possible to connect as many as 128 interrupting peripherals to the PCW data bus. If any such are connected, then any one of them may supply the low byte of the interrupt vector so the latter will probably not now be 255. This arrangement means that you can have a different ISR for each peripheral, though you may not want so many. If you want to guarantee that the same ISR is run regardless of circumstances, then fill page Y with 'X' bytes, add one more 'X' at the start of the next page, put Y into 'I', and start your ISR at (X,X).

Choosing the Interrupt Vector

In the simple case where the low byte is 255, there are obviously only 256 values that can be ascribed to the interrupt vector, and many of these are not available because the two addresses concerned are occupied by CP/M or by your own TPA routines. The choice is limited still further by the fact that the vector should be within common memory (ie. X should be larger than 191). If all such addresses are occupied, you can still free a pair within your own program by preceding them by the instruction *jr* 2. The surrounding program will then ignore them.

Switching Mode 2

The method of setting the vector and then switching to IM 2 (which activates it) consists of:

```
di          243        Disable the interrupts
ld a, N     62  N      Put the high byte
ld i, a     237  71      into the I register
im 2        237  94    Switch to IM 2
ei          251        Restore the interrupts
    continue . . .
```

It is obviously necessary to have put the address of the ISR into the interrupt vector, and to have written and installed the ISR before switching to IM 2.

To get back to Mode 1 from Mode 2 use the following piece of code


```
di          243        Disable the interrupts
im 1        237  86    Back to IM 1
ei          251        Restore the interrupts
    continue . . .
```

This will re-establish the standard PCW mode of operation.

A program example

Appendix 8 gives a program example of how to set up and operate within Mode 2, and it illustrates the power of Mode 2 by artificially creating a closed programming loop and then escaping from it at a key press. Normally you would need to switch the machine off to get out of a loop like this.

The loop consists of constantly jumping back to a 'print string' operation (BDOS fnc 9), the printed string being

> *"You are now in an endless loop.*
> *To escape hold down the ALT key until you regain control."*

At every interrupt the ISR checks whether the ALT key is being pressed. If it isn't then no action is taken and the looping continues. If it is then the ISR puts a safe 'restart' address on the top of the stack to replace the normal one so that operations continue from the safe address when the interrupt is complete. The safe restart leads to the printing of the boast:

> *"We have regained control - thanks to IM 2"*

just to confirm who your deliverer is.

IM 2 and BDOS

Because BDOS fiddles about with the interrupts itself, it is important not to test for the keypress and escape on account of it if the machine happened to be involved in a BDOS call at the moment that this particular interrupt occurred. The example program therefore sets up a BDOS-trap by creating a flag at address (158,2). It sets the flag if BDOS is in use and resets it otherwise, and no key tests are made if

the flag is set. There are other occasions when one ought not to make escape attempts on account of interrupt interference, for example using IM 2 with USERF functions would require an additional trap.

Exiting from an ISR

All returns from ISRs should be made by means of:

```
...        ...        The ISR code
...        ...
rst 56     255        Do the normal housekeeping
ei         251        Restore the interrupts
reti       237 77     Exit from the ISR
```

The first of these instructions makes a jump to (56,0) just as a standard PCW interrupt does, and this ensures that the usual housekeeping duties will be attended to. The last instruction is the form of 'ret' that is used to terminate an ISR, though it is only minutely different from a normal 'ret'.

Halt, again

The example program uses the instruction 'halt' in the BDOS-trap. This halts all operations until the next interrupt occurs, which provides time for the keypress to be noticed in case the loop should involve a lot of BDOS use with very little inter-linking code. If you leave it out you may have to wait a long time for your finger to have the desired effect.

IM 1 VERSUS IM 2

Being more or less free to pick your own location for your ISR, and being able to decide for yourself where to return to in adverse circumstances makes IM 2 distinctly 'more powerful' than IM 1, and programmers, like everyone else, are generally keen to have all the power they can get, though this very freedom can cause dilemmas.

When you pick your code location you have to be sure that nobody is going to overwrite it. That will be no problem when you are the only programmer, so IM 2 is then to be preferred. However, if the user may chose to load an RSX in tandem with your programming, then that will try to hide itself at the top of the TPA, just where you chose to try to hide your code. Block 0, in contrast, is safely out of the way of being overwritten so IM 1 might be preferable.

WHAT ISRs CAN'T DO

ISRs can't do long things gracefully, so keep them short. Just to see what it was like I have made my programs run at less than 1/50th of the usual speed by inserting a ISR that contained a very long loop. I'm sure even slower speeds are possible though the machine simply gives up trying somewhere fairly close to this level. Exactly where I can't say.

In the case quoted the screen display appeared and disappeared with all the whizz of glaciers on level ground and my curiosity faltered when it was taking a second or so to print each character. But it is interesting to see in slow motion exactly what does go on when your programs run. I detected two unecessary repetitions in the display of my home-grown assembler; these are noticeable only when the cursor is crawling at snail's pace. And it was a bit of a surprise to see the screen still scrolling at the speed of light whilst BDOS printing could only trudge.

Chapter 5

Back to Font

This chapter is about the CP/M 'fonts' for use with the dot-matrix printers of the '8256' and the '8512'. If you require a nice new font for the daisy wheel printer of the '9512' then you will need to buy a new daisy wheel from the range available.

If you require a nice new font for the dot-matrix machines then my best advice is to buy one of the many excellent packages (and some not so excellent) that have been made available by professional software houses. The trade press advertises them extensively.

If I still haven't put you off then you may be feeling that what follows is exactly the sort of data you would willingly several times have given your eye-teeth for. If so then your eye-teeth may still be in danger because the nitty-gritty of font design is a bit complicated. However, that said, it is entirely possible to compose and install a completely new set of characters to your own requirements and print with them satisfactorily. In this regard, unless your modifications are to be slight, it might be easier to start from scratch with your own new

set than to try to modify the characters already in place. Either way the facts are are as follows.

Printer operation

PCW Machine Code gives a more or less complete description of how the dot-matrix printers work and how to use them. I won't go over the same ground again, but the basic facts are that printed characters (and in this chapter if I write "printed" I will mean "paper-printed", ie. 'listed' if you prefer the buzz) are made up of vertical 'bars' of dots, which I will call 'bars'. Each bar is described by an 8-bit byte. A byte of value '1' means that the bar will have only its top dot 'set', and a byte value of '128' means that only the bottom one will be. (This is 'upside down' compared with the arrangement for printer graphics.)

The lowest dot of the bar is reserved for underlining so letters usually use only the upper 7 dots.

There are only 128 printable characters because ASCIIs with the top bit set (ones larger than 127) signal that the character is to be printed in italics, so an ASCII of '193' (65 + 128) would be a call to print "*A*" (italic capital 'A'). ASCIIs lower than 32 can be printed only if you make a special request for them, but the ones provided are almost useless so don't bother (see pages 134 and 135 of the manual), though these, the ones of ASCII less than 32, are obvious candidates for you to experiment on; perhaps you might even manage to bring them into the world that most of us inhabit.

Font data

If that were all that there is to be said about fonts then paper printing would be as easy as screen printing and we could all rest easily in our beds, but computer printers are expected to output an extraordinarily wide range of text types, and it would be prohibitively expensive of memory to have separate Character Matrix-Rams for every type, particularly as there are 128 symbols and they require roughly six bytes each. To get round this difficulty, font data is compressed into a few relatively short tables, and of these we will first consider the draft quality.

DRAFT QUALITY

There are three data tables relating to the draft quality characters. These are in use whenever a version of draft printing is taking place, so any modifications you make to them will affect all draft characters whether, bold, wide, condensed, normal, or whatever.

The tables reside in block 8 starting (in my machine) at (92,103). They probably start there in yours too, but I suggest that you check to make sure that the sequence of bytes at that address is

87, 1, 93, 1, 102, 1, 108, 1 . . .

in your case as well. If it isn't then you should do a search until you find such a set and base everything that follows on your modified address.

The three tables are, in sequence

(92,103)	1. Offset table
(94,104)	2. Label translation table
(179,104)	3. Bar specification table

Offset table

The first table is 258 bytes long and contains pairs of bytes. Each pair is an offset that is used to locate the font data for a particular ASCII code. The offset for ASCII code No 0 is the first entry, that for ASCII code No 1 is the second entry, etc. As the table starts with the byte sequence given above, the offset for ASCII No 0 is (87,1). The offset for ASCII No 1 is (93,1), etc.

Number of bytes per character

The number of bytes required to describe a character varies from none in the case of "space", to 9 in the case of "'e' with a grave accent". There may be some that use more than nine, but I haven't noticed any yet. The number of bytes that relate to a character is indicated by the difference between its own offset and that of the one following.

The first four values in the offset table are:

	(87,1)	Offset for	Char 0
	(93,1)	ditto	Char 1
	(102,1)	ditto	Char 2
and	(108,1)	ditto	Char 3

Subtracting the first from the second gives a result of '6', indicating that the first character requires six bytes. Subtracting the second from the third gives a value of '9' so the second one requires nine bytes. Similar treatment shows that the third also needs six bytes, etc.

Although there are only 128 printable characters, there appear to be 129 entries in the offset table, but the last one does not refer to a character. It is provided only so that the size of the last character can be deduced.

Bar specification table

If you add the offset extracted from Table 1 to the start address of the first table, ie. to (92,103), you will obtain the address within Table 3 at which you will find the data that describes the character concerned. Thus, if we add (87,1) to (92,103) we get (179,104), and (179,104) is the address at which the data for ASCII No 0 starts. This symbol (which is "a" with a grave accent; see page 135 of the manual) is described by the 6 bytes at that address and the next five. (Six bytes are required in this case but not in all cases.) To get the bytes for ASCII No 1, you would add (93,1) to (92,103) and use the bytes from that address and those following, etc.

Label translation table

Unfortunately the bytes obtained as above are not the bytes that describe the set/reset condition of the dots within a bar, though they do say which bar design is being referred to. I will therefore call them 'bar labels'. The dot patterns can be obtained only by reference to the second table, in which the entries are the bytes that are associated with each bar label. This second table starts at (94,104). Thus, if you obtained bar labels of 0, 1, 2, 3, 4, and 5 from the third table (as described above), then you would look up the first six bytes in the translation table, which are in fact

0, 64, 1, 4, 65, 32

The dot patterns in this (hyperthetical) case would therefore be:

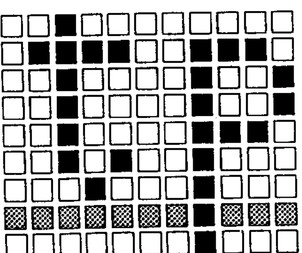
no dots
bottom dot
top dot
third dot,
top and bottom dots,
sixth dot.

In the case of draft quality this complication saves no memory space because you could eliminate one whole table by not doing the translation. However, the saving is significant in the case of NLQ letters, and it makes sense to use the same methodology for both.

The characters supplied with the machine do not use every possible dot combination, they use only 85 such combinations so the translation table contains only 85 entries. The contents of the offset table and of the label translation table are listed in Appendix 2.

Offset-table modifications

The high byte of each entry in the offset table would normally be much less than 15, so the top four bits of it would be unused. To shoe-horn yet more data into the available space the high byte sometimes has its top bit set (128 is added to the byte), and sometimes bit Nos 4, 5, and 6 are also set.



The printer-head pins used for standard letters, descenders, and underlining

If the top bit is set this indicates that the character has a 'descender', ie. it has a donward pointing leg like "g", "p", "q", etc., that pro-

trudes below the print line. Normally characters use only the upper 7 pins of the printer head, and they can therefore be specified by using only bit Nos 0 to 6. Bit No 7 is used in underlining, and also in a few characters. The ninth and lowest pin in the print head is reserved for 'descenders' like those in the letters indicated (see the letter 'p' in the diagram opposite), and is made use of by signalling that the whole of the character should be printed one pin lower than usual. This signal is provided by bit 7 of the high byte of the offset. Because the character is printed lower than usual, descenders are able to protrude below the underline and all the bars of such characters can still be specified in 8 bits, though naturally they can't use the top pin as well.

To obtain the significance of the other high bits, mask them out and treat them as a binary number in their own right. Hence bit No 4 on its own is equivalent to '1', bit Nos 4 and 5 together are equivalent to '3', bit No 6 is equivalent to '4', etc. This derived value is the number of blank bars that are needed before the character proper starts.

An example of the latter is given by full-stop ".", which consists of all blank bars except for the one with the dot in. Its offset is (100,66). With the relevant masking, the high byte yields a corrected value of 2, and the value from bits Nos 4, 5, and 6 is '4' (because bit No 6 is set). Thus 4 blank bars are required before the dot is printed. The technique is obviously effective because this character is described and properly positioned in the print line by only one byte of the data table.

Similarly, the 'comma' has an offset of (96,178), which indicates that the character has a descender and that 3 blank bars are required before the body is printed.

When using a pair of offsets to deduce the number of bytes in a character, it is obviously necessary to strip out these modifications from the high byte before doing the subtraction.

Bar specification table mods

Some further data-packing techniques are used to increase the information content of the entries in the third table.

If the top bit of an entry is set, this indicates that a **blank bar** is required before this bar.

If, ignoring the top bit, the entry has a value of 122 to 126 then this is a signal that the next bar should be **repeated**. The number of repetitions is given by subtracting 121 from the repetition byte, so the maximum number of repetitions is 5, which would be signalled by a value of 126 (or 254). There is not much point in using 122 because this would signal one repetition which could as easily be achieved by two identical labels. Repetition bytes may also indicate that a leading blank bar is required before each repetition, but this is the only other function that they can perform.

Locating character information

In order to gain easy access to the data that describes characters, it is necessary to develop some simple routines that will output the information we require. Because the data is all held in block 8, we must have our routines in common memory so they don't get zapped when it is being accesssed.

The first requirement is to obtain the necessary offset and the number of bar labels involved. This is provided by the following routine. Before using it, the relevant ASCII code must have been inserted into the address (A,A) in common memory. The next four addresses above this are also required for reporting back.

```
FIND
    ld a 2           62 2          Switch in block
    call MEM_M       205 33 253      No 8
    ld hl (A,A)      42 A A        Recover the
    ld h, 0          38 0            ASCII code into HL
    add hl, hl       41            Double it
    ld de TABLE      17 92 103       and add it
    add hl, de       25              to Table start

    ld e (hl)        94            Low Byte of offset into E
    inc hl           35
    ld a (hl)        126           Hi Byte into A
    and 15           230 15        Reject the 4 top bits
    ld d, a          87              and put into D
    ld (A1,A) de     237 83 A+1 A  Store for report
    push de          213             and save
    ld a (hl)        126           Hi Byte again into A
    and 240          230 240       Extract the top 4
    ld (A2,A),a      50 A+3 A        bits and store for report
```



```
    inc hl           35            Point to next offset
    ld e (hl)        94            Low Byte into E
    inc hl           35            Hi Byte
    ld a (hl)        126             into A
    and 15           230 15        Reject top 4 bits
    ld d, a          87              and put into D

    pop hl           225           Recover 1st offset
    ex hl de         235           Swop the two
    or a             183             and subtract 1st
    sbc hl, de       237 82          from 2nd
    ld a, l          125           Result into A
    ld (A4,A) a      50 A+4 A        and report it

    ld a, 1          62 1          Switch back to
    call MEM_M       205 33 253      the TPA
    ret              201             and finish
```

There are two bytes per entry in the offset table so the ASCII code is extracted from (A,A) and doubled to give the number of bytes we have to jump over from the start of the table. This is added to the table start address so that HL now points to the offset for our character.

The offset is extracted into DE having filtered out the mods to the high byte, and the offset is stored at (A+1,A). The high byte is then taken into A again and its top four bits (the mods) are stored for future reference in (A+3,A).

We then step on to extract the next offset into DE, and recover the first one by 'pop'. Subtracting the first one from the second one gives the number of bytes (bar labels) relating to our character, so this is stored at (A+4,A). The information now in memory therefore consists of:

(A,A)	The ASCII code
(A+1,A)	LB of offset
(A+2,A)	HB ditto
(A+3,A)	Mods to HB
(A+4,A)	Count of bytes in character

Accessing character information

Having located the labels relating to a character, it is a simple matter to extract them for inspection. The following routine is intended for

use after the one above and it employs the data reported by it. It transfers the bar labels into addresses (A+5,A) and above, where they can be inspected.

```
SHOW
    ld a, 2          62  2          Establish
    call MEM_M       205 33 253       block 8
    ld hl (A1,A)     42  A+1  A     Get the offset
    ld de TABLE      17  92  103      and add the table
    add hl, de       25               start address.
    ld de A5 A       17  A+5  A     Point to storage
      *                *
    ld bc (A4,A)     237 75 A+4 A   Get the count of
    ld b, 0          6  0             bytes into BC
    ldir             237 176        Transfer
    ld a, 1          62  1          Back to
    call MEM_M       205 33 253       the TPA
    ret              201            Finish
```

The bytes supplied by this routine are the ones prior to interpretation, ie. they are the bar labels. You have to strip off any modifications and then use the interpretation table to get the pattern of on/off dots that make up the character. The content of the Interpretation Table is given in Appendix 2. For a complete interpretation you also need to combine these mods with the ones implied in the offset, if any, which are stored at (A+3,A).

Changing character bar-labels

If you want to modify existing characters and see what effect your modifications have (which is how I analysed the font data), re-write the routine above but insert

```
    ex hl, de        235            Swop source & destination
```

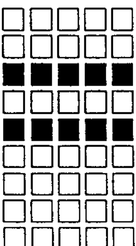
at the place marked by the asterisks (*). Run FIND and SHOW to extract a list of the bar labels in the character. Then change one or more of these in the way you think is relevant, and then run the modified program (the one with 'swop' in it). This will insert the modified labels back into table. If the character is then subsequently printed, the effects of the modifications will be visible.

EXAMPLES OF DRAFT QUALITY CHARACTERS

The four symbols "=" (61), ">" (62), "," (44), and "E" (69) illustrate most of the techniques that are used to pack symbol information into minimum space. The examples are clarified by the diagrams that accompany them.

'Equals'

This has the 62nd entry in the offset table, and its offset is (179,2), which has an unmodified high byte. The next offset is (181,2) so the character has only two bytes describing it. Adding (179,2) to (92,103) gives (15,106), at which are to be found the two labels '125' and '10'. The first of these is in the range 122 to 126, so it indicates that the '10' must be repeated (125-121) additional times, ie. 4 additional times.



The 'Equals' sign

A bar label of '10' gives a byte value of 20, and 20 is made up by setting only bit Nos 2 and 4. Hence the 'equals sign' consists of two parallel lines of dots and because there are five repetitions there are five dots in each line.

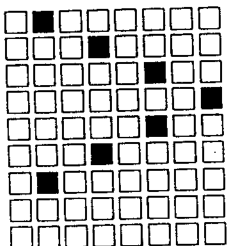
'Larger than'

This has the 63rd entry in the offset table, and its offset is (181,18), which has bit No 4 of the high byte set. This says that the character starts with one blank bar (bit No 4 means '1'). The next offset is (185,18) so there are four bytes in the character. These four bar labels turn out to be

4, 134, 138, and *135*

The last three have bit No 7 set so a blank bar is required in front of each. The stripped labels and their interpretations are then

stripped labels:	*4*	*6*	*10*	*7*
bytes:	*65*	*34*	*20*	*8*

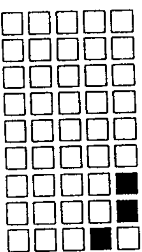


'Larger than'

If you map these out onto a piece of graph paper (as above) you will find that they do indeed look like a right-pointing arrow head. And if you change all the bar labels to the stripped versions you will find that the new symbol is compressed into a shorter width due to the removal of the blank bars.

'Comma'

The offset for 'comma' is (96,178), so the top bit and bit Nos 4 and 5 of the high byte are all set. The setting of the top bit signals that this character is to be printed one pin lower than usual, and that makes the tail of the comma stick down below the print line. Bits 4 and 5 combine to a signal value of '3', so three blank bars are required before the body of the comma starts.



'Comma'

Subtracting the stripped offset from the next stripped offset shows that the character contains only 2 bar labels, and these are found to be '17' and '30', which on translation turn out to represent bytes of 128 and 96. '128' is the bottom pin (normally pin 8, but in this case pin 9), and '96' is the next two pins above (64 + 32). The shape of the comma is obtained from these three dots.

Capital 'E'

The offset for "E" is (220,2) so there are no mods to the high byte. Its four bar labels and their translations are

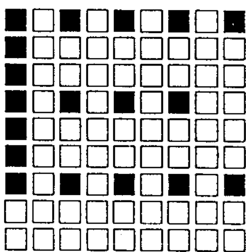
labels:	*8*	*251*	*13*	*132*
stripped labels:	*8*	*123*	*13*	*4*
bytes:	*127*	*r*	*73*	*65*

The first label translates to a byte of '127', which signals seven 'on' dots to create the vertical part of the letter.

The second label is '123' with its top bit set. This requests 2 additional repeats of the next byte, its top bit requesting blank bars between them.

The third label translates to a byte of '73', which maps to the top, the middle, and the bottom dots of the usual seven in the height of a letter. The original and the two repeats of this byte form the main part of the prongs of the 'E'.

The last label translates to a byte of '65', which consists of the top and bottom dots only (64 + 1), thus extending the top and bottom prongs by one dot. The top bit of this label is set indicating that a blank bar is required in front of this byte also.



Capital "E"

DESIGNING NEW SCRIPTS

Because everything has been done to squeeze the current font into the minimum space, you will meet problems if you try to obtain graceful modifications of it. There simply isn't the room for a lot of changes unless you are willing to move a great deal of data and ensure that every consequence of all the moves has been catered for. It is perhaps better to scrap most of what exists and concentrate your attention on producing good re-designs for the fewest number of characters that will cover your requirements.

Whilst it is obviously part of the game to change the data, there is nothing to be gained from changing the ASCII codes that relate to the common characters, so '65' should continue to mean "A", and '97' to mean "a", etc. This is in any case essential if you want to use proportional spacing occasionally. The widths of proportionally spaced characters are derived by the printer from a look-up table, and this is listed in the conventional ASCII sequence. (See page 68.)

The re-designed offset table

When the printer goes in search of the pattern of dots for ASCII code that it is to print next, it must find the offset for that character at the expected place in the offset table, so the structure of the table can't be changed. The offset for "A" must always be the 66th entry, etc. However, although the offset for "A" must always be in this place, its value can be anything within reason that we choose.

Lets suppose we are aiming to produce a set of 96 new characters in the range of ASCII codes 32 to 127. If we could guarantee that no other ASCII would ever be sent to the printer (and that is not easy to guarantee), then we could disregard their offsets, we could even zeroise them all. However, if in these circumstances a rogue ASCII did somehow get into the system, it would be better to have a predictable response to it than an unpredictable one, and I am suggesting that all 'rogues' should print a recognised 'error symbol', such as a black blob, say. This would be provided by the following two specfiers:

bar labels:	*125*	*15*
bytes:	*r*	*28*

The first requests 4 repetitions without any blank bars, and the second defines three dots one above the other.

(On the subject of barring rogue ASCIIs, it would help if you re-specified the keyboard so that it would output only the acceptable ones. This is described on page 108 [541] *et seq* of the manual and also in Chapter 2.)

The offset table would therefore need to point to these two bar labels for all ASCIIs outside our chosen range, but because the blobs are two labels long the offsets must all differ by 2. The start of the new offset table would therefore be:

87, 1, 89, 1, 91, 1, 93, 1, . . . etc

This incrementing by 2 continues up to the entry for ASCII No 31 (the last rogue), so the table will actually start with 32 offsets that differ by 2.

The re-designed label translation table

The label translation table supplied with the machine contains all the dot combinations that the authors discovered were necessary to construct their characters. They may also cover your needs too, in which case you may as well leave them as they are. However, they include only 85 of the 256 possible combinations of dots, so you may find that you can do a swop for some of the ones you don't want, and perhaps add a few more at the end.

The end of the label translation table is defined only by the start of the bar specification table that follows it, and this is defined by the size of the first offset. Because they didn't want any more varieties of bytes, the authors set the first offset to (87,1), ie. to the address just beyond the 85th label translation. If you want more translations (you can have as many as you like up to 120; remember the repetition bytes), accommodate them by sizing your first offset accordingly.

The re-designed bar specification table

The bar specification table starts wherever the label translation table ends, and extends up to a maximum address of (255,107). The current version uses up to only (104,107) (the rest of it is packed with 255s), so there is quite a bit of room in the system for exuberant, not to say flamboyant, calligraphy, provided you start from scratch.

In the case cited, the first 32 entries in the specification table would be repetitions of '125 15', which would be the specifiers for the error blob.

THE NLQ FONT

The operational philosophy of the NLQ font is quite like the one for the Draft font, except that the printer makes two passes over the text. It also prints at the slow head speed (which is half the draft head speed) and this doubles the packing of the dots in the horizontal direction. For the second pass it moves the paper up by half a dot width and then prints a different set of dots. The effect is to double the dot packing in the vertical direction also. The doubling of the packing in both directions means that the text looks much blacker, much more even, and has a smoother profile, but because there are two passes at half speed, and because printing takes place in one direction only to give optimum alignment on the two passes, the printing rate is about a sixth of what is achieved in draft mode.

Inspection of the bar specification table shows that many NLQ labels exceed 128 and are therefore requesting blank bars. This is because the packing of the dots is already high due to the low head speed (the dots frequently overlap), so if the blank bars are eliminated the characters become narrower but there is no perceptible increase in density or definition.

As with the draft font there are three tables in block 8 that contain the NLQ data, and they operate as the draft ones do. The start addresses are:

(104,94)	1. Offset Table
(106,95)	2. Label Translation
(94,96)	3. Bar Specification

The contents of the offset table and of the label translation table are listed in Appendix 3.

Bytes per character

Because twice as many dots may be printed in a character width, many NLQ characters require more than 20 labels for their complete

specification. The bar specification table is therefore much longer and extends up to the start of the draft font tables at (92,103). As before the number of bytes required is indicated by the difference between adjacent offsets.

Label translation

The NLQ label translation table contains two bytes per character. The second one specifies the dot pattern for the first pass and *vice versa.* There are 121 labels in use, so the table contains 242 bytes.

Whereas the coded translations are in most cases the same as were described for draft, the NLQ label '122' signals that the two bytes following it are not labels but binary dot patterns that are to be used as they stand and so are not to be translated. Hence the sequence

'122 1 128'

would mean "print the top dot on the 2nd pass and the bottom dot on the first pass". '122' is used in "comma", "full-stop", and "apostrophe" (and in others) where it miraculously converts an otherwise unintelligible dot pattern in the familiar shapes we so love and admire.

EXAMPLES OF NLQ CHARACTERS

I will illustrate the NLQ characters by describing the same examples as before, though because of the double pass and the way dots are packed together, the final shape of characters is not as obvious as it is for their draft version, and it is obscured further in the case of the letters by the fact that this is a 'serif' font in which the horizontals and verticals all end in spikey bits.

'Equals'

The 62nd offset is (128,5) and the next one is (132,5), so the character is four labels long. When the offset is added to (104,94) you get (232,99), at which address are to be found the labels on the next page:

Labels:	*127*	*11*	*139*	*139*
Stripped labels:	*r*	*11*	*11*	*11*

For these the translation table gives the following pairs of bytes:

First pass:	-	*20*	*20*	*20*
Second pass:	-	*0*	*0*	*0*

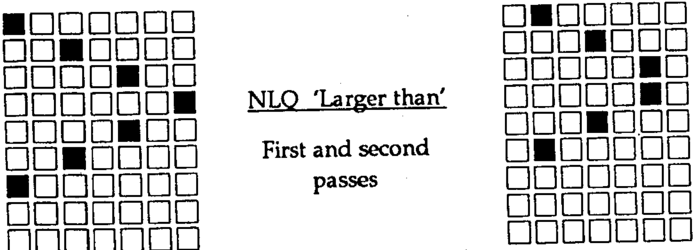
The first byte requests 6 repetitions of the following one, and the last two bytes add two more. All the Bytes are '20' as was the case for the draft version. Nothing is printed on the second pass.

'Larger than'

The 63rd offset is (132,37) and the following one is (139,5) so the character uses 7 labels. Bit 5 of the high byte is set so the character starts with 2 blank bars. The labels and their translations are:

Labels:	*1*	*142*	*135*	*145*	*139*	*177*	*133*
Stripped labels:	*1*	*14*	*7*	*17*	*11*	*49*	*5*
First pass:	*65*	*0*	*34*	*0*	*20*	*0*	*8*
Second pass:	*0*	*33*	*0*	*18*	*0*	*12*	*0*

If you draw out these two sets of bytes you will find that they are both right pointing arrow head shapes. The first one is bigger and pointed, the second is smaller and blunt. They superimpose to make a densely packed arrow head.



NLQ 'Larger than'

First and second passes

The dots of the second pass are printed half a dot lower than those of the first pass.

'Comma'

This character has an offset of (192,84) and contains 9 labels in three triplets. Each triplet consists of a '122' and its two binary bytes, so the specification is:

122 160 64 122 0 160 122 96 64

If you plot these out it is possible to persuade yourself that they look like a comma.

Capital 'E'

The offset is (233,5) and there are 11 labels which translate as:

Labels:	*1*	*129*	*146*	*252*	*13*	*250*	*12*	*93*	*129*	*14*	*7*
Stripped lbls:	*1*	*1*	*18*	*124*	*13*	*122*	*12*	*93*	*1*	*14*	*7*
First pass:	*65*	*65*	*63*	*r*	*73*	*B*		*93*	*65*	*0*	*34*
Second pass:	*0*	*0*	*127*	*r*	*0*	*B*	*12*		*0*	*13*	*0*

When plotted out these two sets of dots conform pretty much to what you would expect.

MODIFYING NLQ

If you intend to modify the NLQ font then obviously you have a serious project on your hands. Apart from anything else there is no gap between the NLQ tables and the ones used in Draft, so the former cannot be expanded without affecting the latter. This is why I keep referring to starting from scratch.

However, it has occurred to me that the space available in and after the draft tables, particularly if you eliminate the first 32 characters, might well make it possible to design a draft font to an NNLQ specification (Nearly Near Letter Quality) by having denser dot packing at least in the horizontal direction. This might be achieved by cutting out all the blank bars and repeating most of the others. If you get it right, you would then have a neat font that printed at draft speed, but I'm not promising anything.

MODIFYING LOCOSCRIPT

The font data for the printed Locoscript 1 letters is kept in the file called MATRIX.STD , and that for Locoscript 2 is in MATRIX.PRI . If you take your version into memory you will be able to modify it in broadly the way described for the CP/M NLQ font, though there are some differences. For one thing the 'binary label' 122 becomes 123 in Locoscript. It is an inconvenient fact that there are many versions of Locoscript 2, and these vary in ways that make a general statement impossible, though no doubt you can deal with your own version if you experiment.

PROPORTIONAL SPACING

Proportional spacing is toggled by '27 112 x', where 'x' may have the values 0 or 1. Its look-up table occupies the 64 bytes preceding the NLQ tables so it starts at (40,94) with the byte sequence:

152 119 68 160 139 ....

Each byte of it contains the width data for two characters, each in four bits, so the individual width values may vary from 0 to 15.

The operation of the table is illustrated by the 17th entry, which is 160. This separates into the values 10 and 0, which relate to SPACE and "!" respectively. "!" has the narrowest width allocation of zero, as have ";" and ":", which combine to make the 30th table entry zero. The 19th entry is 138, which indicates a width of 8 for "$" and 10 for "%" (8 x 16 + 10 = 138).

If you modify the table contents you will be able to observe the effect on the relevant letters when they are printed in proportional spacing mode.

Chapter 6

Basics of Screen Printing

BDOS functions

The BDOS functions of CP/M are called by putting the function number into the C register and then making a call to (5,0). In some cases the DE register-pair needs to be loaded with an address before the call is made, and in some cases BDOS reports back with information in A or in HL or in both.

All printing of text on the screen is achieved by using one of the following BDOS functions:

Fnc 1	Halts operations to await the next keypress, which is recorded in A and if printable is echoed on the screen.
Fnc 2	Prints the single ASCII in E.
Fnc 9	Prints the string pointed to by DE.
Fnc 10	Halts operations to await keyboard input of a text string, which is printed on the screen and also stored in the declared buffer.
Fnc 111	Prints the text described by a 4-byte CCB.

The Print Position

Until it is modified, the screen print position stays at the end of the last piece of text that was printed, but new text can contain as many position instructions as you like, thus allowing any part of it to be placed in a desired screen location. If necessary the 'text' can consist of nothing but a print-position instruction. This would shift the print location ready for a subsequent action such as key-board input for funcs 1 or 10, for example (which obviously can't themselves declare the required location), or to allow printing of a string at different locations at different times.

The most comprehensive control of print position is given by including the escape sequence

27 89 L+32 C+32 . . .

in the material to be printed. 'L' is the line number (for the top line L=0), and 'C' is the column number (for the leftmost column C=0). This sequence automatically moves the print position to the place specified by L & C, but be warned that there is a system error (a bug in CP/M) that interferes with printing on line 4. (See 'String Printing'). It is necessary occasionally to include the control code 'Carriage Return' (=13) into text items because without it CP/M will get its sums wrong and print on the next line down even if you haven't told it to. On its own CR moves printing to the left margin, but if you follow it immediately by a print position instruction this effect will be overridden.

The print position is affected by the inclusion in the text of the control codes:

BACKSPACE	8	
TAB	9	
LINE FEED	10	(scroll if on bottom line)
CR	13	
SPACE	32	

or of an escape sequence consisting of:

CURS UP	27 65	(no scroll)
CURS DOWN	27 6	(ditto)
CURS RIGHT	27 67	(ditto)
CURS LEFT	27 68	(ditto)
CURS HOME	27 72	(go to top left)
CURS UP	27 73	(scroll down if on top line)

Keyboard Input

Functions 1 & 10 take in data from the user, the others print only information that is already in memory. Function 1 makes it possible for the user to respond to program enquiries such as menus or other simple choice situations. If called repeatedly it can be used for accepting more complex inputs (such as multi-digit numbers, or the input of ID codes), but its usefulness for text input is quickly eroded by the complication of providing for the correction of errors. However, because a return from it is made at each key-stroke, illegal keystrokes (non-numerals say, or invalid code characters) can be detected and dealt with as they arise.

Function 10 has the opposite characteristics. It can accept up to 128 characters at a single input, the user can employ the 'DEL' keys and the right/left arrows to correct errors on the hoof, and the cursor keys can be used to move back and forth within the previously input text. However, because the function has no discrimination, and no return is made until the input is complete, illegal input will not be detected until the end of what might be a lengthy keying session, which can in some circumstances lead to muttering about "bad design" and "not user friendly".

To use Fnc 10 it is necessary to declare the location of the buffer that will receive the input (its address is put into DE before calling BDOS), and to specify the maximum number of characters it may contain. You put this maximum, which may not exceed 128, into the first buffer address. CP/M will count the actual number typed in and enter this value at the second buffer address, and the text will start at the third buffer address. If the user attempts to input more than the specified maximum, a beep will sound and no more will be accepted. The user indicates that his input is complete by pressing 'RETURN', but the '13' for this is not inserted into the buffer nor added onto the character count.

Single characters

Fnc 2 finds only occasional use for printing such characters as "?", meaning "I don't understand", or "!", meaning "that looks wrong to me" (though to chess buffs it might actually mean "that looks good to me"!), but in all cases except when there is a serious shortage of printing space it is preferable to print the full phrase.

String printing

A 'string' is a sequence of ASCII codes. For use with Fnc 9 the string must be terminated by an end-marker (delimiter), which conventionally is "$", ie. 36. However an error in the system programming makes it preferable to change the end marker to a value that lies outside the range 32 to 64. Unless this change is made (see the next heading) it is not possible to print directly onto the line whose number is given by [delimiter minus 32], which is normally line 4.

To print a string its address is put into DE before calling the function. The string may contain any of the control codes and escape sequences listed in the manuals on pages 139 to 141 [581 to 584].

Changing the End-Marker

The new value of the end-marker is notified to CP/M by using Fnc 110. You put the value into DE (which is the same as saying that zero is put into D and the marker value is put into E) before the function is called. If instead you put (255,255) into DE, then the function will report back with the value of the current end-marker in A, without making any change to it. My preference for the marker value is 255, so I load DE with (255,0) before calling the function, though zero is also a good choice. If you change it, don't forget that you must change the it back to 36 before returning to CP/M because it uses 36.

I am told that Basic uses its own printing routine, one that owes nothing to CP/M, and its own terminator is zero, which need not be reset before a return to Basic.

Block Printing

To use Fnc 111, DE is first given the address of a 4-byte Character Control Block (CCB), the first two bytes of which contain the address of the start of the text, and the second two contain the count of the number of characters to be printed. This allows text with no end-marker to be printed, and you can of course print different sections from within a block at different times just by changing the address and the count.

Text Control

Several of the CP/M escape sequences provide useful control of the text being printed. They are used by including them in a string. When it is printed they come into effect from whatever location they occupy.

Deletes

There are several escape sequences for deleting text from the screen, none of which change the cursor position:

27 69	Clear the whole screen
27 74	Clear to the bottom of the screen from and including the cursor position
27 75	Clear to the end of this line
27 77	Delete all this line
27 78	Delete the character under the cursor
27 100	Clear from the top of the screen down to and including the cursor position.

'Clear screen' is used every time a new text page, such as a menu, is to be displayed. If you don't clear the screen then the new text will be indecipherably superimposed upon the old.

'Delete character' is not one I have ever used, but the others find application in deleting messages such as prompts or minor error warnings, and in cases where the users wishes to start his input again from scratch.

Emphasis

Page headings etc, can be underlined by preceding them with 27 114, but remember to terminate the heading with 27 117 or all subsequent text will be underlined as well.

'Reverse video' is valuable in menu pages and in drawing attention to staus reports. It is given by 27 112, but remember to terminate it by 27 113.

Cursor on/off

There are occasions when the cursor blob interferes with a screen display. To avoid this it can be disabled by 27 102, and switched back

on again by 27 101. As an alternative to turning it off you can sometimes move out of the way, to the bottom of the screen for example.

Cursor position

Particularly when data is being input, it may be necessary to return to a previous screen location although its Line and Column values are not be known when the program is being written. Such screen locations can be recorded by 27 106, and then re-established by 27 107.

Summary

The following piece of code gives an example of the use of most of these various screen printing techniques. I have assumed that it is OK for you to insert it at address (0,200) upwards, which is above the reach of a short BASIC program, should there be one in residence. When it finishes it makes a 'ret'. If you are working from BASIC this will be appropriate, otherwise it may not be, in which case replace the 'ret' (201) with 'rst 56' (255), which hands over to CP/M, or 'jp MENU' if you are working from a menu.

```
Section 1: ( 0,200 )
     ld de 255        17 255 0       Change
     ld c 110         14 110          the
     call BDOS        205 5 0         end-marker
     ld de 255 255    17 255 255     Then
     ld c 110         14 110          check
     call BDOS        205 5 0         it and
     ld (N N), a      50 254 199      store it at (255,199)
Section 2
     ld de STR1       17 100 200     Print the
     ld c 9           14 9            first
     call BDOS        205 5 0         string
Section 3
     ld de BUFFER     17 180 200     Establish
     ld a 20          62 20           a buffer
     ld (de), a       18              at ( 70,200 ) for
     ld c 10          14 10           20 characters.
     call BDOS        205 5 0
Section 4
     ld de STR2       17 140 200     Point to the 2nd
     lc c 9           14 9            string and
     call BDOS        205 5 0         print it
```



```
Section 5
     ld a, (N N)      58 181 200     Get number of letters
     ld (N N), a      50 172 200      & put into the CCB.
     ld de CCB        17 170 200     Point to CCB
     ld c 111         14 111          and print
     call BDOS        205 5 0         text again
Section 6
     ld de STR3       17 160 200     Point to 3rd
     lc c 9           14 9            string and
     call BDOS        205 5 0         print it
Section 7
     ld e '?'         30 63          Print a
     ld c 2           14 2            single
     call BDOS        205 5 0         "?".
Section 8
     ld de STR3       17 160 200     Point to 3rd
     lc c 9           14 9            string and
     call BDOS        205 5 0         print it again
Section 9
     ld de '$'        17 36 0        Revert
     ld c 111         14 111          to normal
     call BDOS        205 5 0         end marker.
Section 10
     ld c 1           14 1           Await a
     call BDOS        205 5 0         keypress

     ret              201            Then finish.
```

The string at address (100,200) is as follows:

```
27 69                   Clear screen
13 27 89 40 48          Position
73 110 112 117 116 32   "Input
115 111 109 101 32        some
116 101 120 116 58        text: "
13 27 89 40 70          Position
27 114                  Underline on
255                     End marker
```

and the one at (140,200) as follows:

```
27 117                  Underline off
13 10 10 10             CR + 3 lines down
9 9 27 112              2 tabs infrom left margin, plus inverse on
255
```

and finally the one at (160,200):

27 113 *Inverse off*
13 10 10 10 9 9 255 *As above*

The content of the CCB located at (170,200) should be:

182 200 0 0

Explanation of example program

The first section changes the string end-marker to '255'. It then requests what the new marker is and stores the returned figure at (254,199) where you can inspect it at leisure.

The second section prints the string at (100,200) using BDOS fnc No 9. This string clears the screen, sets the print position to line 8 column 16, and then provides the words "Input some text:", followed by a new position instruction to go to line 8 column 38. It then turns on the underline mode so that any following text will be underlined.

The third section establishes a 'Console Buffer' at (180,200) and puts the value '20' into the first byte of it. The user must now input up to 20 characters of text and press 'ENTER' (or 'RETURN'). Because the underline mode had been established earlier, the text will appear underlined.

The fourth section prints the string at (140,200) using fnc 9 again. This string turns off the underline mode, moves to the left margin (13=CR) and down three lines (10=line feed), followed by right by two TABS (16 columns). It then turns on the reverse video mode, so that text appearing next will be in black on a white background.

In the fifth section we require to print the text currently in the console buffer. However this text is of unknown length and has no end-marker, so only fnc 111 is appropriate to print it. The start of the text is known to be at (182,200) so I made a point of inserting this address into the CCB at (170,200). The function also needs to know the number of characters to print so this is obtained from (181,200) into A and then transferred into the CCB at (172,200).

The sixth section prints the string at (160,200), which turns off the

inverse video and makes the same position shift of three lines down and 16 columns across.

The seventh section now prints a "?".

The eighth section uses the last string again to move the cursor position clear of the printed material.

The ninth section reestablishes the original end-marker (assumed to be '36').

The tenth section pauses until a key is pressed before exiting the program.

After the program has been run, you can verify

a) the value of the selected end-marker by inspecting (254,199).
b) the permitted length (20) of the console buffer, and the number of characters typed in (second byte of the buffer), and the text itself. The buffer is at (180,200).
c) the fact that the CCB now also includes the number of chars to print [at (172,200)].

Finding the cursor

Bank 0 contains a sub-routine called 'TE_ASK' ('TE' stands for 'terminal emulation') that reports the position of the cursor. Your program can therefore check to see where a text input has got to and react accordingly. TE_ASK is at (191,0) and it returns the cursor position in HL. H gives you the line number and L the column, but with one of those inconsistencies that could unhinge a sensitive constitution, these numbers don't have the usual '32' added (CP/M was obviously written by a team dispersed in several buildings). 'TE_ASK' thinks the top line is line No 0, and the left column is column No 0. This facility is described on page 14

Chapter 7

Basics of Screen Addressing

The Memory

The Z80 processor, which is the operational heart of the PCWs, can have access to only 64k of memory at a time even though the machines are provided with more than this. The lowest memory address is zero and the highest is 65535 when expressed in normal decimal (also called denary) notation. In hexadecimal notation the address range is 0000h to FFFFh, and in my own 'Red-biro' notation it is (0,0) to (255,255).

The normal memory map

When the computer is switched on, the map of its 64k of memory is as indicated in Appendix 6.

CP/M occupies two memory areas within this, the lower of which starts at 0 and extends up to 255 (which memory range is called 'page 0'). The upper area extends down from (255,255) to (6,246). The memory between these two areas is called the 'TPA', which

stands for 'Transient Program Area'. All your programs, and features such as BASIC if it is in use, occupy the TPA.

Additionally, when a program is loaded into the machine, the loading is handled by a sub-routine that is inserted just below the upper CP/M area, and which extends down to (128,242), but when loading is complete this region is again available for other use.

The standard COM file stack extends downwards below (0,246), so the first pair of bytes onto the stack will occupy (255 & 254,245). See page 10 for other stack addresses.

Memory Blocks

The whole of the memory in the PCW, whether it be 256k or 512k, is sub-divided up into **blocks** of 16k. In the '8256' there are 16 such blocks (numbered from 0 to 15), and in the '8512' and '9512' there are 32 (numbered from 0 to 31). Because the Z80 can address only 64k at once, only 4 of the blocks are 'in circuit' (in contact with the Z80) at any time. The others are 'out of circuit', but still preserve whatever memory content has been put into them so that it is ready for use when called upon.

Memory Banks

A group of 4 'in-circuit' blocks is called a **bank**. A bank always consists of 64k, and the banks are also numbered from zero upwards. The standard bank, ie. the one described above that contains the TPA, is numbered as 'Bank 1'.

Every bank contains Block 7, regardless of the other blocks in it, so Block 7 is called 'common memory'. Block 7 is always addressed in the range (0,192) to (255,255), and you will see from the diagram in Appendix 6 that the upper part of CP/M, the loading program, and the upper part of the TPA are therefore all in Block 7. Block 7 acts as an area of co-ordinating memory through which the actions of all the banks can be applied to whatever task is in hand.

As well as occupying two regions in Bank 1, CP/M reserves five other blocks entirely for its own use. Within these it stores the data and the sub-routines that it needs for controlling the computer, and

for providing the various services that it makes available. The content of the blocks is summarised in Appendix 6.

The Screen Environment

Whilst most of the banks are referred to by their numbers, one has no number but is given the name 'The Screen Environment'. This bank is of particular interest to us because it is the only one through which we can gain comfortable access to the screen data and the data from which the screen characters are generated. It also contains the 'Roller-RAM' without which screen locations could not be addressed. The blocks in service in the various banks are listed in Appendix 4.

Notice that the only bank that gives simultaneous access to the whole of the screen data is the Screen Environment because that is the only one that features Block 2.

The screen map

Everything on the computer screen is formed from a pattern of pixels, a pixel being a single dot of light. There are 184,320 pixel positions arranged in 256 horizontal lines of 720 pixels per line.

Each print position (the screen space occupied by a printable character) consists of 8 lines of 8 pixels, the eight lines being one above the other. The 8 pixels in a pixel-line are represented by the 8 bits of a byte. In it bit No 0 refers to the rightmost pixel, and bit No 7 to the leftmost one. Each set bit signals an 'on' pixel (a bright dot), and each reset bit signals an 'off' pixel (no bright dot). Hence a byte value of 1 would light only the rightmost dot, 128 would light the left-most one, and 255 would light all of them to give a line of light one print character wide. Each other value of the byte would imply a particular combination of lit and unlit pixels. For example, a byte value 85 (01010101) would imply 'off' pixels alternating with 'on' pixels to give a line that appears to be dimly lit.

If all eight of the bytes representing a print position contained 255, ie. if all 64 of the pixels were switched on, then the position would be filled with a rectangle of light. It would be a rectangle not a square because the vertical separation of the pixel-lines is twice as great as the horizontal separation of the pixels within a line.

The record of which pixels are on and which are off is to be found in the Screen Data, which starts in Block 1 at (48,89), and extends into Block 2 up to (47,179). The circuitry that controls the VDU screen constantly scans this data and sets and resets its pixels according to what it finds there. Every time it finds a set bit it turns on the corresponding pixel, and every time it finds a reset bit, it makes sure that the corresponding pixel is switched off.

Hence, any changes made to the memory bytes in this area will immediately change the screen display. For example, if you wanted to light up the entire screen you would fill the Screen Data area with 255 bytes, and if you wanted to clear the screen you would fill the Screen Data with zeroes.

However, the PCWs were conceived as text handling machines so the Screen Data is organised in the way that best suits the printing of characters. The eight pixel-lines of a character position are represented by eight bytes in sequence in the screen data area, and those of the character position to the right are represented the next eight bytes, etc. (See the diagram in Appendix 7).

Hence if you want to draw a straight line across the screen, you can not achieve it by filling 90 bytes in sequence; you would have to fill 90 bytes spaced at 8-byte intervals.

The situation is complicated still further by the fact that a given Screen Data byte does not relate to a particular place on the screen. To aid the scrolling of text, the byte relates only to a fixed place within the text. Suppose the byte at our chosen address is found to correspond with the top of a letter 'Z' within the word 'Zebra'. Wherever the text is scrolled to, as long as 'Zebra' stays in view, our byte will always point to the 'Z', though if the print line containing 'Zebra' is scrolled off the screen, the byte will be re-allocated to the top of a new letter; one that occupies same column that was occupied by the 'Z'.

Roller-RAM

CP/M uses a 512-byte data table called Roller-RAM to ascertain which bytes in the Screen Data apply to which physical locations on the screen. The table starts at (0,182), in Block 2, and can therefore be accessed at the same time as the Screen Data by using the Screen Environment.

What Roller-Ram contains is 256 addresses (hence 512 bytes) which point to the 256 pixel-lines down the left edge of the screen, though for reasons unclear it stores them in a slighly mangled way. The diagram in Appendix 7 shows how the addresses in the table map onto the screen, but let me work through it blow by blow.

1st entry in R-RAM

The first address stored in Roller-RAM (ie. the one indicated by the first two bytes of the table) is half of the address of the top left pixel-line of the screen.

Suppose we examine Roller-RAM and find that its first two bytes are 152 and 44. These two numbers are the low and high bytes of half a red-biro address, so we have to double their value to get the actual address. Twice (152,44) is (48,89), hence the address Roller-RAM is telling us about is (48,89).

[In case the result of the doubling is not obvious; doubling (152,0) gives (304,0) which is actually (48,1), and doubling (0,4) gives 0,88). Adding these gives (48,89). In practice you will not need to do the doubling youself because there are lots of m/c instructions that will do it for you.]

So what Roller-Ram is telling us is that, in this case, whatever we do to the content of address (48,89) will show up in the top left corner of the screen. Hence if we put '255' into (48,89) then a line of light will appear in the top left corner.

The address we obtain from Roller-RAM is certain to be an address within the Screen Data area; in the above case it happens that it was the first address of the Screen Data [(48,89)]. Frequently the first address stored in Roller-RAM will not be the first address of the Screen Data, but it will always be the address of the byte at the top left corner.

Rest of Roller-RAM

The next seven addresses in Roller-RAM are simple increments of the first address. Hence if you add the second entry to the first entry you will get the address of the second pixel-line down from the top left

corner. If you add the third to the first you will get the address of the third pixel line down, etc. (See the large diagram in Appendix 7.) The 8th entry in Roller-RAM, like the first, is also half a screen address, but in this case it is half the address of the top left pixel-line of the second print line. (Which is numbered as print-line No 1.)

The sequence of further entries continues in this pattern; every 8th entry being half a pixel-line address.

Summary of Roller-RAM

The first entry in Roller-Ram (its first two bytes) has to be doubled to discover the address of the top left pixel-line.

To find the address of the first pixel-line in print line 1, double the 9th entry (which is given by the 17th & 18th bytes).

To find the address of the first pixel-line in print line 2, double the 17th entry (which is given by the 33rd & 34th bytes), etc . . .

In general terms, the address of the *first pixel-line of any print line* can be derived from two stages of calculation as follows:

1. Required place in Roller-RAM = (48,89) + 16 x LINE
2. Required screen address = 2 x (the address stored here)

The Status Line

The lowest screen print line is referred to as the 'Status Line' because it may be separated from the body of the screen and made available for displaying status and error messages, etc. When this separation is in effect the status line will not scroll with the rest of the screen. The act of separating it is referred to 'Enabling' the status line, and this is achieved by printing the escape sequence "27 49". It can be reabsorbed back into the body of the screen by 'Disabling' it, which is given by printing "27 48". (See also page 14 in Chapter 2.)

THE SCREEN RUN ROUTINE

Whilst the Memory Manager can access the numbered banks (see pages 9 and 194), the Screen Environment has no number and to access it the 'Screen Run Routine' has to be used instead.

The Screen Run Routine was referred to in Chapter 2, but because it finds use in so many screen-display applications I have evolved the following 'modular' approach, which makes the process of writing such applications a little simpler. It also standardises on the use of registers and variables addresses so that applications can use each other's output. Because it involves bank switching it must be located in common memory, as must the routines that use it. It is an obvious candidate for a library and in mine it takes up the first part of block 7, but you will obviously put anywhere it else in block 7 that is convenient.

The Modular pattern

When using the modular pattern, the first 20 bytes of block 7 are given over to storage of variables. Not all of them are used so there is scope for expansion. The variable names are:

Address	Name	Notes
(0/1,192)	-	Temporary data
(2/3,192)	-	ditto
(4,192)	ASCII	
(5,192)	-	
(6,192)	Line	0 to 31
(7,192)	Colm	0 to 89
(8,192)	Length	1 to 90
(9,192)	Line count	1 to 31
(10,192)	-	spare
(11,192)	-	spare
(12,192)	-	Storage
to . .		of
(19,192)	-	ASCIIs

There are three sections to the program that gives access to the screen material (See *PCW Machine Code*),

1. An executive section
2. Calculation the 'Jump Userf Addr'
3. The sequence of operations to be performed

Sections 1 and 2 are identical in all applications so the same piece of code is needed by them all. The code, which I have called 'HANDL' (Handling Routine), starts at (20,192) and lists as follows:

HANDL (20,192)

```
call USERFN      205  26  192     Section 1
DEFW             233  0
ret          *   201

ld hl, (1)       42  1  0         Section 2
ld de, 87        17  87  0
add hl, de       25
jp (hl)          233
```

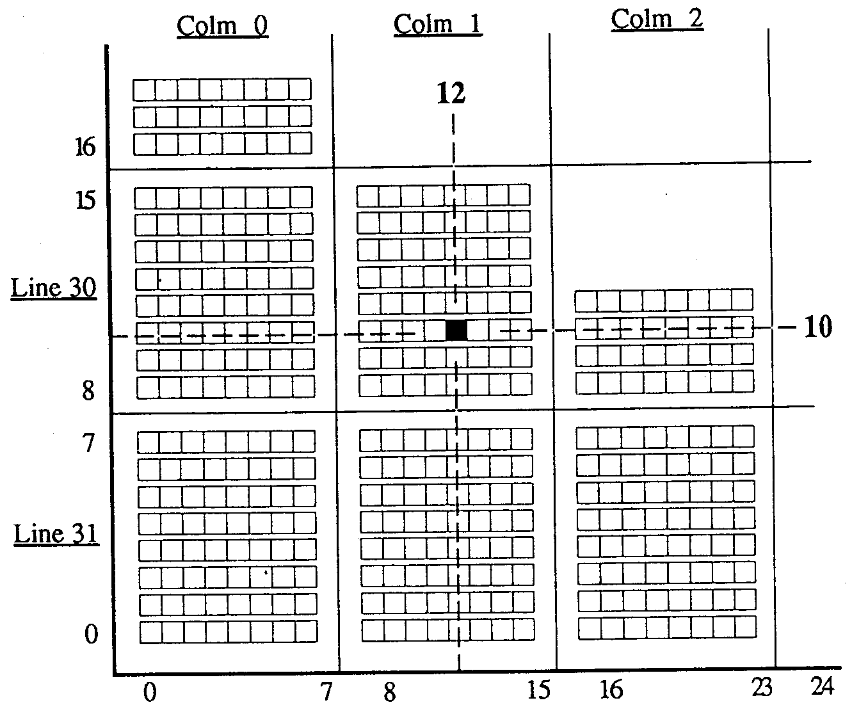
IMPORTANT NOTE

A program making use of HANDL must point BC to the address of the application and the application must terminate with a 'ret'. HANDL also terminates with a 'ret' (*) so it will return to the place after whatever 'call' preceded its use.

Two examples of this are two sub-rs that find use in many screen programs so they too should be residents of block 7. They are FADDR ('Find Address'), which finds a screen address of the top byte of a print position, and BLEN ('Bar Length'), which calculates the number of bytes in a given length of a print line. (See page 91.)

When using FADDR, the line and column numbers must have been put into (6/7,192). If successful, the address is returned in HL with Cy reset. If an impossible line or column number is specified then Cy is returned set indicating that the HL value should be ignored. Because it is important not to poke data bytes wrongly into areas used by the operating system, as a further safety measure the sub-r also tests the calculated address. If it lies outside the screen data area then Cy is returned set, however I inserted this extra level of checking for a special occasion but it is irrelevant in most cases. If you want to speed up this often-used routine then remove the code between /*....*/, but always test Cy on return.

```
FADDR (35,192)
     ld a, (6,192)     58  6  192     Line No
     cp 32             254  32        Reject if
     ccf               63               larger
     ret c             216              than 31

     inc a             60
     ld hl, ROLLER     33  240  181   Roller-Ram start minus 16
     ld de, 16         17  16   0     Step size
     add hl, de        25             Step to
     dec a             61               entry for
     jr nz -4          32  252          this line

     ld e, (hl)        94             Extract  Roller
     inc hl            35               entry
     ld d, (hl)        86               into  DE
     ex hl, de  de,hl  235            Then into  HL
     add hl, hl        41             Double it

     ld a, (7 192)     58  7  192     Colm No
     or a              183            Finish
     jr z, 11          40  11           if zero
     cp 90             254  90        Reject
     ccf               63               if more
     ret c             216              than 89

     ld de, 8          17  8  0       Adjust
     add hl, de        25               address
     dec a             61               for
     jr nz -4          32  252          colm No
  /*...
     ex hl, de         235            Address into  DE
     ld hl SCREEN      33  48  89     Lowest screen address
     or a              183            Reject
     sbc hl, de        237  82          result
     ccf               63               if
     ret c             216              too low
     ld hl SCREEN      33  47  179    Highest screen address
     sbc hl, de        237  82          reject if
     ret c             216              too high

     ex hl, de         235    ....*/  Result back into  HL
     ret               201              and finish
```

The Line and Column positions of the co-ordinates (12,10)

Universal screen addresses

FADDR finds the address of the top of a print location, so there is not much left to do to find the address and bit number of any screen pixel. The screen is 256 pixels high by 720 wide. Taking each as a co-ordinate point, the screen co-ordinates range from [0,0] at bottom left to [719,255] at top right. The print position (line, colm) that contains the point 'P' whose screen co-ordinates are (x,y) is given by

$$\text{line} = 31 - \text{INT}(y/8)$$
$$\text{colm} = \text{INT}(x/8)$$

The number of pixel lines that 'P' lies physically below the address of (line, colm) is given by

pxlns = (y/8 - INT (y/8)) x 8

The bit number of the pixel within the derived address is given by

bitno = 7 - (x/8 - INT (x/8)) x 8

The diagram on the previous page shows all this for screen co-ordinates (12,10), which lie within Line 30, Column 1.

Calculation of the address and bit number of a co-ordinate pair can therefore be calculated as in BITADR if the (x,y) values are supplied as 'y' in A and 'x' in HL. The listing of BITADR and the full procedure for using it are as follows. If you 'call' address (C,C), HANDL will make a return to the location after your call because it terminates with a 'ret'. (See the 'Alternative patterns' heading below for other methods of use):

```
Address ( C,C ):
    ld (2 192), hl     34  2  192     Store 'x'
    ld (4 192), a      50  4  192     Store 'y'

    ld bc, BB          1  B  B        Point to BITADR
    jp HANDL           195  20  192     and then use HANDL

BITADR  Address ( B,B )
    ld a, (4 192)      58  2  192     Collect 'y'
    srl a              203  63        Divide it
    srl a              203  63          by 8
    srl a              203  63
    ld b, a            71             Then into B
    ld a, 31           62  31
    sub b              144            Sub from 31
    ld (6 192), a      50  6  192     Store 'Line'
    ld hl (2 192)      42  2  192     Collect 'x'
    srl h              203  60
    rr l               203  29        Divide it
    srl h              203  60          by 8
    rr l               203  29
    srl a              203  63
    rr l               203  29
    ld a, l            125            Colm into A
```



```
    ld (7 192) a       50  7  192       and store

    call FADDR         205  35  192     Line/Colm address
    ld a (4 192)       58  4  192       'y' into A
    and 7              230  7           'pxlns'
    ld e, a            95
    ld d, 0            22  0
    add hl de          25               Add to Line/Colm address
    ld (0 192) hl      34  0  192         and store
    ld a (2 192)       58  2  192       'x'/256
    and 7              230  7
    ld b, a            71
    ld a, 7            62  7
    sub b              144
    ld (5 192), a      50  5  192       Store
    ld c, 1            14  1
    or a               183              If 'bitno' = 0
    jr z 5             40  5              finish
    sla c              203  33          Else shift the
    dec a              61                 bit in C
    jr nz -5           32  251            by 'bitno' times
    ld a, c            121              Result into A
    ret         *      201              Finish
```

When the program has been run, the address containing the co-ordinates is returned in (0,192) and in HL, and the particular bit number within this address is returned in (5,192). Additionally, A is prepared so that the set bit within it is the same bit that the co-ordinates refer to. Thus if 'P' corresponds to bit No 4 of address (A,A), then (A,A) will be in (0,192) and in HL, (5,192) will contain 4, and only bit No 4 of A will be set.

The accumulator can therefore be used directly to influence the screen byte. If then (with the Screen Environment in place; see the note on the next page) you do

```
    or (hl)        182
    ld (hl), a     119
```

you will set the bit at the chosen co-ordinates without disturbing any other bits. (If it was already set it will stay so.) You would complement it (set it if reset and vice versa) if the first instruction were changed to 'xor (hl)', and you would be sure of resetting it with the sequence

```
    ld b, a           71
    or (hl)           182
    xor b             168
    ld (hl) a         119
```

Proof of the accuracy of BITADR can be derived by doing the following with the Screen Environment in place :

```
PROOF
    push af           245
    ld (hl), 255      54  255
    xor (hl)          174
    ld (hl), a        119
    ld c, 1           14  1
    call BDOS         205  5  0
    pop af            241
      continue...
```

This will produce a bar of light at the appropriate place on the screen, but missing from it will be the pixel that corresponds to the chosen co-ordinates.

Note on use

In case it is not clear: BITADDR returns the address and the bit number you want, but the address relates to Blocks 0, 1, & 2 (ie. from the Screen Environment). When you return from BITADR, the TPA will have been re-established so blocks 4, 5, & 6 will now be in place again (Bank 1). Hence poking something into the returned address without re-establishing the Screen Environment will not affect the screen, it will affect only Bank 1. You can use 'PROOF' either with HANDL in the way described for all other screen applications, or you can temporarily tack it on to the end of BITADR at the place marked by the asterisk (*), just before the final 'ret'.

Alternative use patterns

The modular approach really is modular so it is flexible enough to be used in a variety of ways. If the pattern is not clear then stick to one version until you see the point of it. After that you will be free to pick and choose as you see fit.

An alternative pattern that could be used would be to include part of what is listed above into your main program, and then make a 'call' to 'HANDL', as follows:

```
    program listing . . .
    ld (2 192), hl      34  2  192      Store 'x'
    ld (4 192), a       50  4  192      Store 'y'
    ld bc, BB           1  B  B         Point to BITADR
    call HANDL          205  20  192      and use HANDL
        *
    continue . . .
```

In this case the storing of 'x' and 'y' and the loading of BC are part of the body of the main program, so a 'call' is made to 'HANDL' instead of a 'jump'. HANDL then makes a return to the place marked by the asterisk.

Byte counting

The second and much shorter standard sub-r 'BLEN' finds use in such operations as making cursor bars or otherwise manipulating specific lengths of print lines. On entry the required number of columns to be manipulated must be at (8,192). The result is returned in HL with Cy reset. Any bar length including zero is accepted, so it is possible for the bar to extend over several print lines. It is the responsibility of the programmer to ensure that when in use such a bar does not encroach beyond the end of screen data. (See 'PCHK' below.)

```
BLEN   ( 94,192 )
    ld hl, (8,192)     42  8  192      Number of columns
    ld h, 0            38  0             into HL
    add hl, hl         41              Multiply
    add hl, hl         41                by
    add hl, hl         41                8
    ret                201             Finish
```

One way of checking that the end of a bar is not beyond the end of the screen is to use 'PCHK' (Position Check). There are 2880 print locations on the screen (32 times 90). PCHK tests to see if the number of the line/colm intersection in (6/7,192) plus the bar length in (8,192) exceeds 2880. If it does Cy is returned set.

```
PCHK
    ld hl (6,192)   42  6  192
    ld h, 0         38  0
    add hl, hl      41
 *  push hl         225             Store twice HL
    add hl, hl      41
    add hl, hl      41
    push hl         225             Store 8 times
    add hl, hl      41
    push hl         225             Store 16 times
    add hl, hl      41
    add hl, hl      41              64 times
    pop de          209
    add hl, de      25              Add 16 times
    pop de          209
    add hl, de      25              Add 8 times
    pop de          209
    add hl, de      25              Add 2 times  (=90 times)
    ld a, (7,192)   58  7  192
    ld c, a      7  9               Colm No into  C
    ld a, (8,179)   58  8  192      Bar length into  A
    add c           129             Add the two
    ld d, 0         22  0
    ld e, a         95              Into DE
    add hl, de      25      ....*   Total into  HL

**. ld de, 2881     17  65  11      Max allowable + 1
    or a            183             Subtract from
    sbc hl, de      237  82           actual and
    ccf             63                adjust Cy
    ret             201     ....**
```

PCHK first multiplies the line number by 90 (90=64+16+8+2) to count the number of print positions in full lines and then adds the number of extra columns. 2881 is then subtracted from this total of print positions, which ought to set the carry flag. Cy is therefore complemented so that it is returned reset if the count was acceptable, and set if it was not acceptable.

Program technique

The first part of PCHK between the single aterisks could be shortened to:


```
    push hl         229             Put the HL  value
    pop de          209               into  DE  also
    ld b, 89        6   89
    add hl, de      25              Add it a further
    djnz -3         16  253           89  times
```

but this is very much slower than the version given.

The last part between the pairs of asterisks follows my principle of being visually obvious, though I think I would prefer:

```
    ld de, -2881    17  191  244    Put  65536-2881  into DE
    add hl, de      25              Add the negative
    ret             201
```

Character addresses

Although not strictly a screen consideration, a common requirement of handling screen operations is finding the relevant addresses of required characters within the Character Matrix RAM. This service is performed by the following sub-routine located at (150,193) in my machine. On entry the required ASCII is in A. Its address is returned in HL.

```
FASC    Address  ( 150,193 )
    ld l, a         111             ASCII into
    ld h, 0         38  0             HL
    add hl, hl      41              Multiply
    add hl, hl      41                by 8
    add hl, hl      41
    ld de, RAM      17  0  184      Matrix RAM address
    add hl, de      25              Character address
    ret             201             Finish
```

Note on USERF

I developed the above modular approach to make it simpler to use screen routines that need to share each other's data, but the modular approach is but one way of stating how the Screen Run Routine operates. However, USERF can be presented in other ways, and I employ some of them elsewhere in the book. This point was covered in Chapter 2, 'The Set-Up'.

Chapter 8

Screen manipulations

The basic idea of the modular approach to screen data handling was developed in Chapter 7, and an application of it is described under REVBAR (on page 133), which reverses a length of print line to make a cursor. The technique of screen manipulations can be further extended to a wide variety of applications, several of which are described below.

SCREEN BLOCKS

Whereas REVBAR reverses a continuous series of bytes to make a bar, 'MABLK' ('Make a Block') can perform a variety of manipulations on an area of screen. The area can be any size but it must be square or rectangular. You specify its top left corner by line and column, its width in columns, and its height in lines. Irregular areas can be treated by several calls of the technique.

You can subject the area to any operation that can be specified for the A register, such as 'xor a' (blank), and 'cpl' (reverse), as well as 'ld a, N', 'or N', 'and N', 'xor N', etc. 'Ld a, 85' gives a dimly lit block, 'ld a, 255' gives a brightly lit one. The only restriction is that

these operations must be describable in one or two bytes of opcode, and before you use the sub-r HL must be loaded with them. If there is only one byte then the other must be specified as zero.

On entry the data supplied to the sub-r is as follows:

E reg or (6,192)	Line number of top	(0 to 31)
D reg or (7,192)	Left edge column	(0 to 89)
C reg or (8,192)	Width in colms	(1 to 30)
B reg or (9,192)	Height in lines	(1 to 32)
HL register-pair	Two opcodes	

Once again it is the responsibility of the programmer to see that sub-r doesn't do silly things through receiving silly data, though some safeguards are built in. As with REVBAR, there are two starts which are used according to the location of the data. If you are feeding fresh data in DE and BC, then use Start 1. If the data is already in the variables area, then use Start 2.

```
MABLK
Start 1
      ld (6,192) de      237  83  6 192    Put data
      ld (8,192) bc      237  67  7 192       into variables
Start 2
      ld (PROG) hl       34  X+19  192     Opcodes into program
      ld bc PROG         1   X   X         Point  BC  to program
      jp HANDL           195  20  192         as required by HANDL

Program:   Address  (X,X)
      call BLEN          205  94  192      Bytes into  HL
      ld (0,192) hl      34   0   192      Store
Line
      ld bc, (0,192)     237  75  0  192   Bytes into  BC
      call FADDR         205   35  192     Screen address into  HL
      ret c              216
      ex hl, de          235                  into  DE
Byte
      ld hl 47 179       33  47  179       Last screen address
      ld a (de)          26                Get byte
      DEFW        *      N  N              Opcodes
      ld (de), a         18                Replace byte
      inc de             19                Next
      or a               183               If now beyond
      sbc hl, de         237  82              end of screen data
```

```
ret c            216            then finish

dec bc           11             Decrement count
ld a, b          120              of bytes
or c             177              and loop
jr nz Byte       32  239          if not zero

ld hl, 6 192     33  6  192     Next line
dec (hl)         53               down
ld hl, LINES     33  9  192     Else point to Line
dec (hl)         53                 count & decrement
jr nz Line       32  220        Next line if not zero
ret              201            Else finish
```

You will see that MBLK is very similar to REVBAR except that it treats each line in turn and loops to the next one if the line count has not reached zero. The two opcodes are inserted at the point marked by the asterisk (*), which is 19 bytes on from (X,X).

CUSTOMISED PRINTING

The process of adding edging and other embellishments to screen displays is normally slow because of the time taken by CP/M to work out lots of screen positions. Because we are programming for ourselves (and have our wits about us) and not for Joe Public (who hasn't) we can cut out nearly all the error checking that CP/M has to do and save some time.

The fastest way of printing is to have the whole screen display ready prepared in the Memory Disc and rapidly poke it into place. This is the approach used in making loading screens (see Chapter 13) and it is OK for a small number of important displays, but as each one occupies 23,040 bytes (90k) they are something of a luxury item.

However, the following four customised methods are much more economical of memory and are each tailored to a particular type of printing requirement. The first is a universal approach that produces very attractive effects for screen filling and information display. The second is a replacement for conventional printing where a sequence of ASCIIs is required at consecutive screen locations. The third and the fourth are ideal for placing invariant bytes (such as borders and other decorations) onto a single line or into a single column.

They are quite fast when used in their specialised roles, and operate by poking the ASCII bytes directly into the screen data area in the Screen Environment. In all of them the tables must be in Common Memory unless you have made special arrangements to access them with the Screen Environment in place. All of them will screen print any ASCII from 0 to 255.

Printing from a table

This approach uses small data tables that are economical on memory, and it is universally applicable to any printing situation. In two seconds it produces a whole screen from only 34k of data table compared with 90k for bit-mapping, and generally this time would be much reduced because it would be unusual to need the whole 34k; the table needn't store data for screen locations that will be blank or left unchanged.

An attractive feature is that the table can store the data in any order you like. Hence you can arrange for the display to appear in intriguing patterns; from the centre outwards, maybe, or in waves, or with a random speckling. You can even use it to produce 'animated patterns'. In some applications you may decide to slow it down a bit for these effects to be at their best. A pause (see Chapter 3) before the final 'jr nz Loop' would do the trick.

In the data table each data item is stored as three bytes. The first two are the position, Line (0 to 31) and Column (0 to 89), and the third is the ASCII to be printed. Printing is given by PRTABL, which is listed below. On entry the initialisation data is supplied as follows:

```
HL pair or ( 0,192 )    Address of data table
DE pair or ( 2,192 )    Number of chars to print
```

```
PRTABL
Start 1
    ld (0,192), hl    34  0  192        Data into
    ld (2,192), de    237  83  2  192     variables area
Start 2
    ld bc, PROG       1  X  X           Load BC as
    jump HANDL        195  20  192        required by HANDL
```

```
Program    Address (X,X)
Loop:
    ld hl, (0,192)     42  0  192     Table address
    ld a, (hl)         126            Line
    ld (6 192), a      50  6  192       store
    inc hl             35
    ld a, (hl)         126            Colm
    ld (7 192), a      50  7  192       store
    inc hl             35
    ld a, (hl)         126            ASCII
    inc hl             35
    ld (0 192), hl     34  0  192     Next entry address

    call FASC          205 150 193    Get character address
    push hl            229            Store

    call FADDR         205 35 192     Screen address in HL
    pop de             209            Character address in DE
    ex hl, de          235            Swop
    ld bc, 8           1  8  0          8 bytes
    ldir               237 176        Transfer

    ld hl, (2 192)     42  2  192     Decrement count
    dec hl             43               of
    ld (2 192), hl     34  2  192       characters
    ld a, h            124            Check
    or l               181              remainder
    jr nz Loop         32  213          & loop if not zero
    ret                201            Else finish
```

Sequence printing

PRTABL is a universal method of printing any kind of data allowing for random access to the screen for attractive effect. However, most printing is more sequential than that so less positional information needs to be tabulated. PRSEQU requires only a sequence of ASCIIs, the screen start position, the number of characters, and the address where the sequence of ASCIIs is stored. It is a replacement for normal CP/M printing with either fnc No 9 or No 111. Because PRSEQ can be used to print a 'slice' out of a long sequence, it is useful in, say, printing abbreviated names of months by calculation from a long list; if some calculation relating to month No 3 puts the appropriate address in to HL, then the sub-r would print 'MAR' out of

'JANFEBMARAPR...' On entry the data supplied is:

```
        E reg     Print line            (0 to 31)
        D reg     Print column          (0 to 89)
        A reg     Number of chars       (1 to 255)
        HL pair   Address of ASCII sequence

PRSEQU
    ld (6 192), de     237 83 6 192   Data into
    ld (0 192), hl     34  0  192       variables
    ld (2 192), a      50  2  192

    ld bc PROG         1  X  X        Load BC as
    jump HANDL         195 20 192       required by HANDL

Program:   Address (X,X)
Loop:
    ld hl, (0,192)     42  0  192     Table address
    ld a, (hl)         126            Character into A
    inc hl             35             Point to next
    ld (0,192), hl     34  0  192       and store

    ld a, (hl)         126            ASCII
    call FASC          205 150 193    Get character address
    push hl            229            Store

    call FADDR         205 35 192     Screen address in HL
    pop de             209            Char address in DE
    ex hl, de          235            Swop
    ld bc, 8           1  8  0          8 bytes
    ldir               237 176        Transfer

    ld hl, 7 192       33  7  192     Increment
    inc (hl)           52               Colm No
    ld hl, 2 192       33  2  192     Decrement count
    dec (hl)           53               of characters
    jr nz Loop         32  224          & loop if not zero
    ret                201            Else finish
```

Repetitive line printing

For repetitive printing of identical characters (for which it is excessive to have to specify every Line/Colm and ASCII value) the situation

ation can more easily be handled by calculation. In this case PRLIN fills the bill. It is quite fast for printing the horizontal bits of borders, and by inserting the right data you can print single characters. On entry to it the data supplied is:

E reg	Print line	(0 to 31)
D reg	First column	(0 to 89)
L reg	ASCII code to print	(0 to 255)
H reg	Number of chars to print	(1 to 255)

```
PRLIN
    ld (0,192) hl      34  0  192        Data into
    ld (6,192) de      237 83  6 192       variables

    ld bc, PROG        1  X  X           Load BC as
    jump HANDL         195  20  192        required by HANDL

Program:  Address  ( X,X )
    ld a (0 192)       58  0  192        ASCII
    call FASC          205 150 193       Get character address
    ld (2 192) hl      34  2  192        Store
Loop:
    call FADDR         205 35 192        Screen address in HL
    ld de (2, 192)     237 91 2 192      Character address in DE
    ex hl de           235               Swop
    ld bc 8            1  8  0             8 bytes
    ldir               237 176           Transfer

    ld hl 7 192   *    33  7  192        Increment colm
    inc (hl)           52                  No
    ld hl 1 192        33  1  192        Decr character
    dec (hl)           53                  count
    jr nz Loop         32  233             & loop not zero
    ret                201               Else finish
```

Repetitive column printing

PRCOL is designed for printing a single character in a vertical column so it does the sides of borders very nicely - much faster than CP/M attempting the same effect. It is identical to PRLIN except that the address in the instruction marked by the asterisk (*) is changed so that the line is incremented not the column.

ld hl 7 192 is changed to *ld hl 6 192*

SCREEN SHIFTING

As well as printing to the screen, there is frequently a need to move parts of the display about, to make room for new information, for example, or simply for pleasing effect.

Scrolling up the whole screen can be effected by using 'Line Feed' (10) on the bottom text line, and scrolling it down can be effected by the escape sequence 27 73 whilst the cursor is on the top line.

Scrolling only a part of the screen is visually more impressive, however. When the scrolled section is roughly central and surrounded by stationary text or graphics, a sophisticated 'Window' effect can be produced. This is further enhanced if new text is fed into the window by Custom Printing, or by some other means, and you can provide the window with a border if you want to. Obviously when a scroll up is made the top line of the area is lost (overwritten by the line below), and the lowest line is duplicated on the line above it. To keep a consistent display, fresh text (or blanks) must be fed onto the bottom line as soon as the scroll is complete. Similar but inverted considerations apply to scrolling down.

Horizontal screen movements are also possible (as shown by Locoscript when you are least expecting it). Similar considerations apply to them though they are, if anything, rather more impressive than scrolls if handled properly. The sequence; 'complete sideways withdrawal of text, pause, new text appears' looks good.

Partial scrolling

The sub-r 'SCRUP' scrolls up any part of the screen when the following data is supplied:

E reg or (6,192)	Top line (lost)	(0 to 30)
D reg or (7,192)	Leftmost Colm	(0 to 89)
L reg or (8,192)	Num of columns	(1 to 90)
H reg or (9,192)	Num of lines	(1 to 32)

```
SCRUP
Start 1:
    ld (6 192), de     237  83  6  192
    ld (8 192), hl     34  8  192
```

```
Start 2:
      ld bc PROG          1   X   X
      jp HANDL            195  20  192

Program:    Address ( X,X )
      call BLEN           205  94  192      Number of bytes
      ret c               216                  to transfer
      ld (0 192) hl       34   0   192      Store
Loop:
      call FADDR          205  35  192      Address of
      ret c               216                  acceptor line
      ld (2 192) hl       34   2   192      Store
      ld hl TOP           33   6   192
      inc (hl)      *     52                Donor line is
      call FADDR          205  35  192         one line below
      ret c               216

      ld bc (0 192)       237  75  0  192   Byte count
      ld de (2 192)       237  91  2  192   Acceptor address
      ldir                237  176          Transfer

      ld hl COUNT         33   9   192      Reduce
      dec (hl)            53                   count
      jr nz Loop          32   225          Loop if not zero
      ret                 201               Else finish
```

It obtains the byte count from BLEN, and transfers that many bytes into each line from the line below. At each loop it increments the top line number and decrements the count until it reaches zero.

The program for partial downward scrolls 'SCRDN' is identical except that the 'inc (hl)' marked by the asterisk (*) is replaced by 'dec (hl)', and the L reg and (6,192) contain the *bottom line* number (the one to be overwritten).

Panning right & left

'Panning' is the horizontal equivalent of scrolling. The following listings pan a selected width of a selected number of lines either right or left by one character. It is not a technique I have ever used in public, but it looks pretty snazzy, particularly if only one line is panned, and there must be uses for it other than in Locoscript. If you want to

hypnotise someone with your message, you can make a sort of screen ticker-tape with it - pan the message off the screen sideways whilst feeding new characters in from the other edge. As with scrolling, you always need to feed in a new character or characters at the receptor end of the panned area. The entry data is

E reg	Top line of panned section
D reg	1st panned column (leftmost)
L reg	Number of columns to be panned
H reg	Number of lines to be panned

```
PANLEFT
      ld (6 192), de      237  83  6  192
      ld (8 192), hl      34   8   192

      ld bc, PROG         1    X   X
      jp HANDL            195  20  192

Program:    Address ( X,X )
Loop:
      call BLEN           205  94  192      Bytes
      ret c               216
      ld b, h             68                   into
      ld c, l             77                   BC
      call FADDR          205  35  192      Start of line
      ret c               216                  into HL
      push hl             229                  and save
      ld de 8             17   8   0           8 bytes to next
      add hl de           25                   char to right
      pop de              209               Orig
      ldir                237  176          Move all left

      ld hl 6 192         33   6   192      Next
      inc (hl)            52                   line down
      ld hl 9 192         33   9   192      Reduce count
      dec (hl)            53                   of lines
      jr nz Loop          32   228          Loop if not zero
      ret                 201               Else finish
```

A similar sub-r provides a one-character pan to the right, but in this case note that DE refers to the column at the right end (which is again the first one to be panned), not the one at the left end, which is usual. The calculation of the addresses before 'lddr' could be smartened up a bit, but this version is easier to follow.

```
     E reg    Top line of panned section
     D reg    1st panned column (rightmost)
     L reg    Number of columns to be panned
     H reg    Number of lines to be panned

PANRT
   ld (6 192), de    237 81 6 192
   ld (8 192), hl    34 8 192

   ld bc, PROG       1 X X
   jp HANDL          195 20 192

Program:  Address ( X,X )
Loop:
   call BLEN         205 94 192    Bytes
   ret c             216
   ld b, h           68              into
   ld c, l           77              BC
   call FADDR        205 35 192    End-of-Line
   ret c             216             address into HL
   ld de, 7          17 7 0        Move to
   add hl, de        25              bottom byte
   push hl           229           Save address
   ld de, 8          17 8 0        Subtract 8 bytes to
   or a              183             point to next
   sbc hl, de        237 82          char to left
   pop de            209           Original into DE
   lddr              237 184       Transfer

   ld hl, 6 192      33 6 192      Next line
   inc (hl)          52              down
   ld hl, 9 192      33 9 192      Decrement count
   dec (hl)          53              of lines
   jr nz Loop        32 222        Loop not zero
   ret               201           Else finish
```

For a pan left, the start point is the first byte in the first column to be panned. All the bytes to the right of this are transferred 8 addresses to the left by 'ldir'. For a pan right, the start point is at the bottom byte of the char in the rightmost column, so 7 is added to the address of this column. All the bytes to the left of it are then moved 8 addresses to the right by 'lddr'.

TOP TO BOTTOM SCREEN INVERSION

Whereas the ideas described so far have involved the movement of the data within the screen data area, the display can also be manipulated by changing the contents of Roller-Ram (though you need to restore it to its original condition if you want life to proceed normally thereafter). In this connection I am indebted to Geoffrey Childs for pointing out the significance of Port 245, which can be used to set up a new R-R. The idea is indicated in the following program which inverts the screen contents so that the top line becomes the bottom line, etc., and every line is itself turned upside down. As with all such programs it needs to be in Common Memory.

```
SCINV
   ld bc, X X        1 X X         Point to sub-r
   call Screen-R     205 90 252
   DEFW              233 0
   ld a, 94          62 94         Switch to
   out (245), a      211 245         new Roller
   ld c, 1           14 1          Await a key
   call BDOS         205 5 0
   ld a, 91          62 91         Switch back to
   out (245), a      211 245         original Roller
   ret               201

Sub-r  Address ( X,X )
   ld de, R-R        17 0 182      Original Roller
   ld hl, New-end    33 254 198    End of new Roller
   ld b, 0           6 0           Count 256 operations
Loop:
   ld a, (de)        26            Transfer
   ld (hl), a        119             a
   inc de            19              pair
   inc hl            35              of
   ld a, (de)        26              bytes
   ld (hl), a        119
   inc de            19
   dec hl            43            Point
   dec hl            43              to next
   dec hl            43              address in new Roller
   djnz Loop         16 244        Repeat if count not zero
   ret               201           Else finish
```

(Program after G T Childs)

Bearing in mind that Roller-Ram contains pairs of bytes each pair being an address, the program transfers the contents of the normal Roller to a new one, but it starts at the far end of the latter so that the data appears in it in reverse order. (Note that to obtain a count of 256, which is the number of addresses to be copied, B is loaded with zero for the 'djnz'. B is not tested until after it has been decremented so zero is not found until 256 decrements have occurred.)

When the 'out (245), N' instruction is invoked, this tells CP/M that Roller-Ram is to be found at address (0, 2xN). Our new Roller starts at (0,188) so the value of N in this case is '94'. Because the Roller addresses are fed in from the 'far end' of our new Roller, they are fed in at (254,189), and this location is progressively reduced (by an 'inc hl' and three 'dec hl's) after each transfer.

After the inversion, the program awaits a keypress and then switches back to the original Roller by setting the value of N to 91, which corresponds to the usual Roller address of (0,182).

The new Roller address is actually within the Character Matrix Ram, but it corrupts only those characters of ASCII code larger than 127. Because Common Memory is contiguous with the Character Matrix Ram, I would have expected to be able to locate the new Roller in Block 7, but this doesn't work for some reason.

Chapter 9

Character manipulations

This chapter deals with making modifications to the screen printable characters listed on page 114 to 118 [547 to 554] of the manual, as well as to any custom-made ones you may have generated.

Rotations and inversions

The following manipulations are useful where you need mirror images, or inverted or rotated forms, of exiting characters but you don't want to redesign them by trial and error. Such modified characters find application in vertical printing (for graphs, etc.) and in games where images need to keep their appearance inspite of changing direction. Unfortunately PCW characters suffer deformation if rotated through 90 degrees because a print position is twice as high as it is wide, but they are still quite recognisable and even have a special appeal of their own.

In all cases the information required is

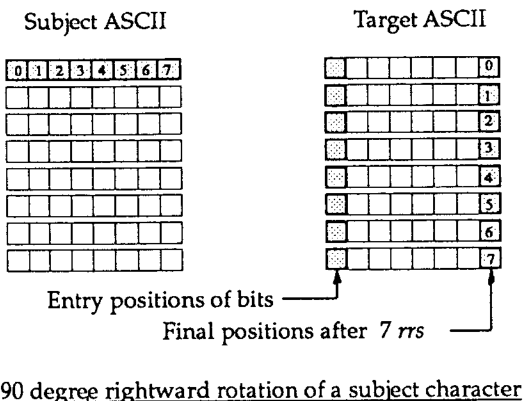
L reg	Subject ASCII
H reg	Target ASCII

Thus if on entry HL contained (65,19) then the modified version of "A" would be written into ASCII No 191 ("TM"), and subsequent printing of ASCII 191 would print the new version of "A".

Rotate right through 90 degrees

```
RRCHAR
    ld (0,192), hl      34  0  192

    ld bc, PROG         1  X  X
    jp HANDL            195  20  192

Program:  Address  ( X,X )
    ld a, (0 192)       58  0  192       Subject ASCII into A
    call FASC           205  150  193    Get its address
    push hl             229                and save

    ld a, (1 192)       58  1  192       Target ASCII into A
    call FASC           205  150  193    Get its address in HL
    pop de              209                and subject address in DE

    ld c, 8             14  8            8 loops
Loop2:
    push hl             229
    ld a, (de)          26               Subject byte
    ld b, 8             6  8             8 bits
Loop1:
    sla a        *      203  39          Bit into
    rr (hl)      *      203  30            ( HL )
    inc hl              35               Next target
    djnz Loop1          16  249

    pop hl              225              Original target
    inc de              19               Next subject
    dec c               13               Decrement count
    jr nz Loop2         32  240          Repeat if not zero
    ret                 201              Else finish
```


The sub-r first works out the addresses within Character Matrix RAM of the source and destination characters. DE is given the first value and HL the second. Because this is to be a right rotation, the top byte of the source character will appear as bit 7 of all the destination bytes (see diagram). However, it is actually put into their bit 0s and at each insertion the byte concerned is rotated right. By the time the sub-r ends these bits will have been shifted as far as bit 7.



90 degree rightward rotation of a subject character

Rotate left

The sub-r for giving a leftward rotation is identical except that the two instructions marked with asterisks are reversed:

```
sla a     to    srl a     203  63
rr (hl)   to    rl (hl)   203  22
```

Character inversions

Characters can be inverted so that the top becomes the bottom and also the left becomes the right by two consecutive rotations as above, the first being into a dummy ASCII.

Mirror-image inversion (which is different) can be performed about either of the two principle axes. Inversion about the horizontal axis swops the top and the bottom but preserves left and right. Inversion about the vertical axis preserves top and bottom but swops left and right.

Top to bottom inversion

```
                L reg    Subject ASCII
                H reg    Target ASCII

TBINV
    ld (0,192), hl    34  0  192

    ld bc, PROG       1  X  X
    jp HANDL          195  20  192

Program:  Address ( X,X )
    ld a, (0 192)     58  0  192       Subject ASCII
    call FASC         205  150  193
    push hl           229              Save its address

    ld a, (1 192)     58  1  192       Target ASCII
    call FASC         205  150  193
    ld de, 7          17  7  0         Add 7 to point
    add hl, de        25                 to its last byte
    pop de            209              Subject address into DE

    ld b 8            6  8             8 bytes
Loop:
    ld a, (de)        26               Transfer
    ld (hl), a        119                bytes
    inc de            19                 to the inversion
    dec hl            43                 location
    djnz Loop         16  250
    ret               201              Finish
```

Side to side inversion

```
                L reg    Subject ASCII
                H reg    Target ASCII

SSINV
    ld (0,192), hl    34  0  192

    ld bc, PROG       1  X  X
    jp HANDL          195  20  192
```



```
Program    Address (X,X)
    ld a, (0 192)     58  0  192       Subject ASCII
    call FASC         205  150  193    Get its address
    push hl           229                and save

    ld a, (1 192)     58  1  192       Target ASCII
    call FASC         205  150  193    Get its address into HL
    pop de            209              Subject address into DE

    ld c, 8           14  8            8 loops
Loop2:
    ld a, (de)        26               Subject byte
    ld b, 8           6  8             8 bits
Loop1:
    srl a             203  63          Transfer bit
    rr (hl)           203  22            into ( HL )
    djnz Loop1        16  250

    inc hl            35               Next target
    inc de            19               Next subject
    dec c             13               Decrement count
    jr nz Loop2       32  242          Repeat if not zero
    ret               201              Else finish
```

Diagonal rotations

You can also rotate/invert about the diagonal axes by choosing a combination of the directions for 'shift a' and 'rotate (hl)', but there is little point in it as far as letters and numerals are concerned because the visual impact appears different in different cases. Rotating about the top-left/bottom-right axis gives the impression of rotating letters that have a vertical axis of symmetry (such as 'A', 'W' and 'Y') through 90 degrees to the left, but those that have a horizontal axis of symmetry (such as 'B', 'C', 'D', etc.) seem to be rotated to the right. Rotating about the other diagonal has the opposite effect.

Making large characters

The following routine makes characters of twice the usual size (four times the area). As before, you specify the ASCII of the character that is to be enlarged and which ASCII code is to receive it, though in this

case the three following RAM locations are also used to store the shape, a quarter in each. The storage sequence is

Top left quarter	in	First Ram location
Bottom left quarter	in	Second
Top right quarter	in	Third
Bottom right quarter	in	Fourth

The feed data is

L reg	Subject ASCII
H reg	Target ASCII (1st Ram location)

```
LARGE
    ld (0,192), hl      34  0  192

    ld bc PROG          1   X  X
    jp HANDL            195 20 192

Program:  Address ( X,X )
    ld a, (0 192)       58  0  192    Subject ASCII
    call FASC           205 150 193   Get its address
    push hl             229             and save
    push hl             229             it twice

    ld a (1 192)        58  1  192    Target ASCII
    call FASC           205 150 193     address into HL

Left side
    pop de              209           Subject address
    ld c 8              14  8         8 bytes
Loop1
    ld b, 4             6   4         4 left bits
    ld a, (de)          26            Subject byte
Loop2
    sla a               203 39        Move the bit into Cy
    rl (hl)             203 22        Then into (HL) and
    sla (hl)            203 38          move it again
    djnz Loop2          16  248       Do this 4 times

    ld a, (hl)          126           Get the HL byte
    srl a               203 63          and move it right
    or (hl)             182           Combine with (HL)
    ld (hl) a           119             & put it back into (HL)
```

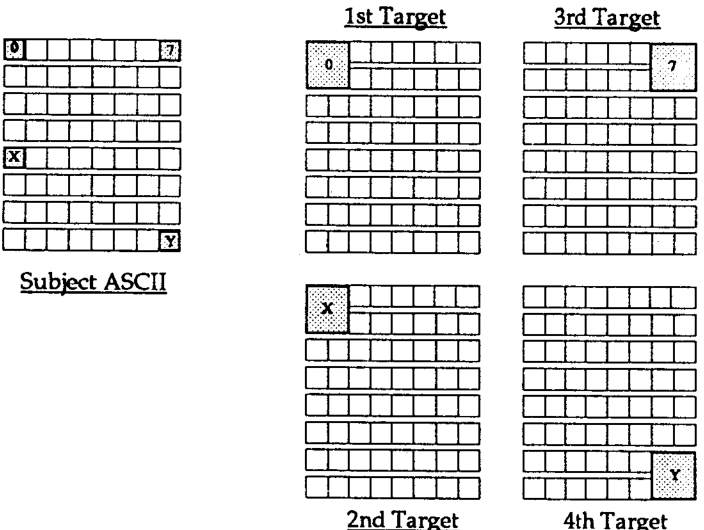


```
    inc hl              35            Duplicate at the
    ld (hl) a           119             next address below

    inc hl              35            Next subject &
    inc de              19              target bytes
    dec c               13            Decrement the byte count
    jr nz Loop1         32  233       Loop if not zero

Right side
    pop de              209           Recover subject
    ld c, 8             14  8
Loop3:
    ld b, 4             6   4
    ld a, (de)          26
Loop4:
    srl a               203 63        All as
    rr (hl)             203 30          above
    srl (hl)            203 62          except
    djnz Loop4          16  248         reversed movement

    ld a, (hl)          126
    sla a               203 39
    or (hl)             182
    ld (hl), a          119
    inc hl              35
    ld (hl), a          119
    inc hl              35
    inc de              19
    dec c               13
    jr nz Loop3         32  233
    ret                 201           Else finish
```

The program enlarges the left side of the subject into the first two target ASCIIs, and then does the right side into the other two. The top left bit of the subject must be quadruplicated in the top four left corner bits of the target, and so on throughout the subject and target.

Consider the left side. The four left bits of the subject byte are mapped into the target byte, but at twice the spacing; bit 0 goes to bit 0, bit 1 goes to bit 2, bit 2 to bit 4, and bit 3 goes to bit 6. After these have been copied, the target byte is copied into A, and A is shifted to the right so that all its set bits are to the right of the originals. A is then ORed with the original, duplicating all the set bits, and is loaded back

into the target byte. Each of the original set bits now has another set bit to the right of it. This completed byte is then duplicated into the byte below. Each subject bit is therefore mapped to four bits in the target.



Enlargement of a Subject ASCII on to four target ASCIIs

Making characters

If you have designed a set of new characters for a particular application (see *PCW Machine Code*), each time the program is loaded it needs to re-establish their shape. This can most easily be done by having a standard routine in Common Memory and calling it from the initialisation sequence, which can be located anywhere, as can the data table that contains eight bytes for each character. The characters are inserted into the CP/M Character Matrix RAM at consecutive locations starting at a chosen ASCII code, and can be printed by any screen printing technique. The sub-r is called with the following supply of data:

HL pair	Addr of data table
E reg	Number of chars to make
D reg	1st ASCII code


```
MCHAR
     ld (0 192), hl     34  0  192
     ld (2 192), de     237 83  2 192
Loop
     ld hl, (0 192)     42  0  192      Table address into
     ld d, h            84              HL and copy
     ld e, l            93               to DE
     ld bc, 8           1  8  0         Point to
     add hl, bc         9                next entry
     ld (0 192), hl     34  0  192       and store
     ex hl, de          235             Back into HL
     ld de, 12 192      17  12  192     Transfer character
     ld bc, 8           1  8  0          bytes to
     ldir               237  176         variables

     ld bc, PROG        1  X  X
     jp HANDL           195  20  195

     ld hl, 2 192       33  2  192      Decrement count
     dec (hl)           53               of characters
     jr nz Loop         32  223          and loop if not zero
     ret                201             Else finish

Program:  Address ( X,X )
     ld a, (3 192)      58  3  192      ASCII code
     call FASC          205  150  193   Its address
     ex hl, de          235              into DE
     ld hl, 12  192     33  112  192    Character bytes
     ld bc, 8           1  8  0         Transfer
     ldir               237  176         8 bytes
     ret                201             Return
```

Designing new characters

When it comes to actually composing the new characters, the following design tool is quite handy. It gives a giant version of the character and a normal sized one, so you can watch the shape develop on both of them.

It uses ASCII codes 253 and 254. The first one takes the new design and additionally the new bytes are also stored at (245-252,227) from where you can transfer them to any ASCII you wish. ASCII 254 is used to make a fully lighted recatangle (all 255s). To modify the shape

being designed move the cursor over the byte you want to change by pressing the up and down arrows and then press RETURN. The bar cursor will be replaced by the typing cursor. Type in one, two, or three digits to state the byte value you require for that location and press RETURN when you have done it. When you have have entered the desired value for all eight bytes (you can change them as often as you like), press EXIT.

The addresses I used to develop the listing are as given below, but you can change all of them except the ones for the 'modular screen routines'.

Feed Data Required

The program requires a set of 255 bytes to make the 'Block' character, a space to accomodate the new character data, and a 'Position String', as follows:

```
( 235 - 242,227 )    All 255s
( 245 - 252,227 )    Bytes of the new character
( 120,226 )          Position String made up of:
                     '27 89 X X 36'
```

The main variables addresses for screen data storage are given below as offsets from (0,192):

```
0    Menu top line reference    (21)
1
2    This line number           (1 to 8)
3
4    ASCII
5
6    This line number           (21 to 28)
7    Cursor column number       (29)
8    Cursor width               (19)
9    Cursor bottom line         (28)

10 to 17   ASCIIs
```

The main screen string used to display the developing character, etc. starts at address (130,226), and is made up as follows:

```
27 69 27 89  36  62  27  114
"DESIGN A CHARACTER"  27  117
```



```
27 89 40 56    134   138 ...(8)... 138   140
27 89 41 56    133   32 ... (8)... 32    133
27 89 42 56    133   32 ... (8)... 32    133
27 89 43 56    133   32 ... (8)... 32    133
27 89 44 56    133   32 ... (8)... 32    133
27 89 45 56    133   32 ... (8)... 32    133
27 89 46 56    133   32 ... (8)... 32    133
27 89 47 56    133   32 ... (8)... 32    133
27 89 48 56    133   32 ... (8)... 32    133
27 89 49 56    131   138 ...(8)... 138   137

27 89 44 80    150  154  156
27 89 45 80    149  253  149
27 89 46 80    147  154  153

27 89 63 66
36
```

The main body of the string (excluding the border round it) is made up of eight lines of eight '32' ASCIIs (that take the place of "32...(8)...32" in each of the above lines). The first '32' is at address (177,226) and the first on the subsequent lines are all 14 bytes on from the one before. As the design proceeds the '32's (SPACES) will be replaced by ASCII 254s (full rectangles) for each set bit of the designed character, thus creating a giant image of it.

There is also a requirement for a lower screen string at address (69,227). This is used in conjunction with the cursor to enable input of the new bytes values of the designed character. It is made up from the following positions and text:

```
27 89 53 64    "Top line:"
27 89 54 64    "2nd line:"
27 89 55 64    "3rd line:"
27 89 56 64    "4th line:"
27 89 57 64    "5th line:"
27 89 58 64    "6th line:"
27 89 59 64    "Bot line:"

27 89 63 64
36
```

The following listing of the program is in the form to be 'jumped' to, and hence it jumps back to your MENU (whatever that is) on com-

pletion. If you want to 'call' it, replace the two lines marked by asterisk (*) with

```
    ret c             216  0  0

CHDES:   Address ( 0,228 )

    ld a, 253         62  253          Transfer any
    ld hl, Bytes      33  245 227        existing bytes to
    call MCHAR        205 5  193         ASCII 253
    ld a, 254         62  254          Make the
    ld hl, Bytes      33  235  227       block
    call MCHAR        205  5  193        character

    ld de, String     17  130  227     Print the main
    ld c, 9           14  9              string
    call BDOS         205  5  0
    ld de, String     17  69  227      Print the lower
    ld c, 9           14  9              string
    call BDOS         205  5  0
    ld de, L/C        17  21  29       Establish the
    ld hl, W/B        33  19  28         bar cursor
    call CSCAN        205  2  194      Finish if
    jp c MENU    *    218  X  X          'EXIT' pressed

    call ADDRS        205  110  228    Else get address
Loop1:
    call REVBAR.2     205  121  192    Cancel cursor
    ld hl, (6 192)    42  6  192       Add 32 to Ln &
    ld de, 32 46      17  32  46         Col, and 14
    add hl, de        25                 extra to Col
    ld ( ), hl        34  122  226     Into position string
    ld de, String     17  120  226     And print
    ld c, 9           14  9              the position
    call BDOS         205  5  0

    ld a, 3           62  3            Specify a maximum of
    ld ( ), a         50  Y  Y           3 digits
    call INTDIG       205  I  I        Accept the digits (see Chap 15)
    ld a ,( )         58  Z  Z         Collect result
    ld hl, ( )        42  224  227     Address for this
    ld (hl), a        119                byte

    ld b, 8           6  8             Eight bits
    ld hl ( )         42  220  227     Address of large display line
```



```
Loop2:
    ld c, 32          14  32           Test each bit
    sla a             203  39            of A, & load
    jr nc 2           48  2              address with ASCII 254
    ld c, 254         14  254            for set bit or
    ld (hl), c        113                ASCII 32 for reset.
    inc hl            35               Next
    djnz Loop2        16  244

    ld a, 253         62  253          Put the bytes
    ld hl, Bytes      33  245  227       into ASCII
    call MAKE         205  5  193        253

    ld de, String     17  132  226     Reprint the main string
    ld c, 9           14  9              without the
    call BDOS         205  5  0          'clear screen'

    call CURSOR2      205  13  194     New cursor with
    jp c MENU    *    218  X  X          old specifications

    call ADDRS        205  110  228
    jr Loop1          24  176          Repeat
```

The routine makes use of the following sub-routine 'ADDRS' for deducing the relevant address within the large display and the address at which this new byte should be stored. It calculates from the line number at which the bar cursor is now placed.

```
ADDRS      Address   ( 110,228 )
    ld a, ( )         58  6  192       Cursor line No
    sub 20            214  20            minus 20
    ld ( ), a         50  2  192       Store
Large display:
    ld hl, N N        33  163  227     1st address minus 14
    ld de, 14         17  14  0        Add 14
    add hl, de        25                 for
    dec a             61                 each
    jr nz -4          32  252            line number
    ld ( ), hl        34  220  227     Store result
```

concluded on next page . . .

Byte storage:

```
    ld a, ( )       58  6  192
    ld hl, N N      33  244  227    1st address minus 1
    inc hl          35
    dec a           61
    jr nz -4        32  252
    ld ( ) hl       34  224  227    Store

    ret             201
```

It also needs the integer input routine 'INTDIG', which is described in Chapter 15.

Chapter 10

Basics of Menu design

Menu Style

Menus are usually the primary means of contact that users have with the program, and it can be helpful to the program's acceptabilty to design them well. Generally, users are not interested in the technical niceties of the package, and won't be impressed by your expertise. Computers already have a name for doing wonderful things so the fact that your program performs competently will be taken for granted.

On the other hand, menus that are untidy and difficult to understand, or which use unfamiliar words, or don't explain what is expected, will convince the user that the program is not to be trusted, even if it is technically flawless.

There is no prohibition against menus that look quite different from each other when they are to perform quite different functions, but if a sequence of similar menus is needed it is a good idea to pay attention to two requirements.

The first is to establish a certain amount of uniformity; the menus should at least bear a family resemblance to each other. It is good practice, for example, to use the same column number for the left side

of the text. Otherwise when one menu is called from another the text seems to jerk sideways. Similarly, unless the menus are of radically different sizes, they should start on the same print line so that the text of each is on the eye-line of the previous one.

The second requirement is that the user should have clear notification of which menu it is that he is looking at; and this should extend be- yond the simple matter of the title. For example the main menu may be decorated with 4 asterisks on each side of the title, but its sub- menus could be given fewer than that. Alternatively you might show the main title in reverse video, but the titles of lesser menus in normal video, all other features being the same. Subliminal clues like this will help to orientate the user and make him more comfortable with the program, though he may not know why.

The most significant thing you can do for your menus is to make them polite because there is nothing in this world more infuriating than a smart-arsed machine. So don't beep unecessarily. Beeps are an excel- lent way to indicate that something unusual needs attention, but if I press a wrong key I don't expect a mere servant to get uppity with me (Locoscript, I think, is intolerable, and the standard pitch of the PCW beep is as aggravating as it could be).

Also make a practice of keeping the user informed. If the program needs to go off by itself to do something secret (access a disk maybe), don't leave him looking at an untidy mishmash of what was happen- ing a minute ago; blank the screen and display a simple explanation of why there is a pause.

Marketing men will have no difficulty with taking these notions on board, though we eggheads tend to the mistaken belief that packaging is always trivial, though the most wonderful program in the world is no good if no-one will use it.

User Instructions

'Help' screens are a good idea in some cases: when the user can't remember what to do next, he summons such a screen to prompt him. However, a comprehensive set of them takes an awful lot of program- ming effort, and I prefer to supply a full set of written notes that con- tain plenty of cross-references through which the user can locate the answer to his particular query. Written instructions are almost always needed anyway because screen text is not a satisfactory substitute for

the paper equivalent. Apart from being more readable, paper-work is more portable, it can be scanned at leisure, and two pages can be studied simultaneously.

Choices

Menus are devices for offering choice. If a program is to offer only a single menu at the beginning and will then get on with its task without further help, you might as well put all the possible choices onto a single screen because once he has made his selection the user will not be pestered again. In these cases the text lines can be quite compli- cated because they will be read only once. But make sure he knows what an allowable response consists of: is he to press only "y" and then wait, does he also have to press 'ENTER' as well, or does he have to type "yes" in full, and what is he to do if neither 'yes' nor 'no' seems appropriate ?

In cases where the program will constantly be asking for information it is preferable to make the menus short (not offering too many choices, and having short text lines) so the user can read them easily and re- spond quickly.

However, making menus brief could mean that there will be more of them, and too many short ones in a seemingly endless chain are as irritating as a few very complicated ones. By thinking of the user as a valued customer who you want to keep on the right side of, you can usually devise some means of turning aside his wrath. Pleasantries and a bit of decoration might help, but when all else fails at least be polite.

Frills

A simple style can stay simple or you can embellish it with borders if you wish. Additionally sections can be boxed off by using the 'graph- ics' indicated with their ASCII codes on page 116 [551] of the manual, though so many are required that I tend to adopt the simplest presentation I can get away with.

However, if you find these goodies really appealing, you can write a string for a top line (with two downward pointing corners) and one for a bottom line (with two upward pointing corners), and print these when required, together with the interconnecting vertical edging of course. The verticals are the real problem. Each pair of 'left edge'

and 'right edge' ought to be included as the first and last character of a text line, but this means that each text line will be longer and that every screen line will need to be printed, so printing is slower. If you try to print them repetitively on their own by using a print instruction in a loop, you will find that they take ages to appear because CP/M has to work out the two print positions at each pass of the loop.

The problems of speed are all solved by the Customised Printing techniques described in Chapter 8, but you still need to put the graphics data into a data table, which is laborious. Professional software houses pay a lot of attention to presentation and if your work is for sale then OK, but if it is for you and your mates I would check to see if they care before going too fancy.

Types of Menu

There are essentially three ways of making menu choices and the menu will need to be designed to suit the method being used. The methods are:

1. Selection by key-press
2. Selection by cursor
3. Selection by text cancellation

1. Keypress

In the first type, the text describing each option is accompanied by a letter or a number indicating which key(s) to press to activate that option. In some cases it may be necessary to press the key and then to confirm it by pressing 'RETURN' as well; this second variant is called **buffered key selection**. When more than one key has to be pressed (as in a multi-digit number input, or typing a whole word), buffered keying can hardly be avoided.

2. Cursor

In the second type the text of the options are listed in the usual way but an option is chosen by moving a cursor next to it and by then pressing 'RETURN', or some other key indicating acceptance. A more impressive variant is for the 'cursor' to be a bar of inverse video that illuminates the whole of each option-text as it is moved between the choices.

Whatever kind of cursor is used, it is usual to move it (either sideways, or, more usually, up and down) by using the four direction arrow keys in the numeral pad, though any of the alphanumeric keys can be made to perform this function as well. A still more professional touch is created if the cursor automatically jumps back to the start of the list when it has transitted all the choices, and this can be a distinct aid to users when the list is long.

3. Cancellation

This less common approach offers each option singly in turn. When the 'reject' key is pressed (I like to use the space-bar for this) the old option is erased and a replacement is displayed in place of it. If the 'accept' key is pressed then the program acts on the option being displayed.

Although not often used, this method may be the only practical solution if each option is very complicated and needs a lot of text for its description. It also finds use in such fields as teaching, in personality or intelligence assessment, and in adventure games, where there may be reasons for not disclosing the full range of choices. It is also useful when most of the screen is occupied by some kind of document or a diagram leaving only the status line (the lowest line) on which to display the options one at a time. Whatever variant is used, it is important to provide for the situation in which the user has rejected all the offered options. If you don't he and the monitor may be left staring blankly at each other, though the monitor will be the one with more patience.

The next two chapters deal in more detail with key-press and cursor menus.

ASCII 13

Note that both 'RETURN' and 'ENTER' have an ASCII code of '13', so when I refer to either of them I am implying either one.

Chapter 11

Key-press Menus

NON-BUFFERED SELECTION

Non-buffered keying means that the program responds as soon as an authorised key is pressed. For single key inputs it is much to be preferred to buffered keying. Suppose that keys 'J', 'K', and 'x' are authorised responses, but no others are to be accepted. A listing to await a keypress and then accept only these is illustrated on the next page.

The sequence awaits a keypress by calling BDOS function No 1. If none is received then nothing will happen, but if a key is pressed the function puts the relevant ASCII code into the A register and the program is allowed to continue. The received ASCII is then compared with the ASCII of "J", and if the two are the same a jump is made to 'program 1' by the 'jpz PROG_1' instruction. If the two are not the same then there is no jump and the next comparison is made, this time with the ASCII of 'K'. If this comparison matches, a jump will be made to 'program 2', etc. This sequence of comparison and 'jump-if-zero' can be extended as long as you like, though the example shows only three such tests because there are only three allowed keypresses.


```
Loop:
     ld c 1          14  1          Await a
     call BDOS       205 5 0          keypress

     cp 'J'          254 74         Compare to ASCII of 'J'
     jpz PROG_1      202 P1 P1        & jump if they match.
     cp 'K'          254 75         Compare to ASCII of 'K'
     jpz PROG_2      202 P2 P2        & jump if they match.
     cp 'x'          254 120        Compr to ASCII of 'x'
     jpz PROG_3      202 P3 P3        & jump if they match.

     jr loop (-22)   24  234        No match so go for another key
```

When the sub-r has tested the ASCII in A against all the allowed values and no match has been found it meets the 'jr loop' instruction. This forces a return to the beginning of the sub-r (to the place marked by the label 'Loop') so fnc 1 is called again and another keypress is awaited.

Note that function No 1 will automatically echo the pressed key onto the screen if it is printable. The character will appear at the current print position whatever that happens to be, and this is reason enough for moving the cursor to the bottom of the screen as suggested below; this gives the echoed characters their own little patch of screen so they don't get tacked onto the menu text. The echoing has the advantage of telling the user what his keypress actually was (as opposed to what he thought it was), but if you prefer to avoid it then print 'backspace' and 'space' (8 and 32) just before the 'jr loop' instruction.

Our natural agressive programming instinct is to insert a 'beep' on detection of a wrong key (ie. immediately before the 'jr loop'), its implication being that anyone who can't tell one key from another is a prat who needs to be pulled up short. I press wrong keys all the time and I'm not a prot (stet), and neither, I imagine, are you. If you feel that comment is unavoidable, think of something kinder than a beep (a gentle warble maybe?).

Text for single-key Menus

A key-press menu is almost always written as a single string that will alone occupy the screen, and it is almost always displayed by using BDOS function No 9 ("Print string").

The following menu-page string illustrates the main features required in a style I quite like. Although written onto separate lines for easier analysis, it would occupy a continuous area of memory starting at some suitable address that we will call 'MENU'.

```
27 69                         Clear the screen.
27 89 40 64                   Start at Line 8, Colm 32.
27 114                        Start underline.
36 36 32 77 65 73 78          "** MAIN MENU **"
32 77 69 78 85 32 36 36
27 117                        End underline.

27 89 43 60                   Print at Line 11, Colm 28.
27 112                        Start inverse video.
32 97 32                      " a "
27 113                        End inverse video.
32 32                         2 spaces.
70 105 114 115 116 32         "First choice"
99 104 111 105 99 101

27 89 45 60                   Print at Line 13, Colm 28.
27 112                        Start inverse video.
32 98 32                      " b "
27 113                        End inverse video.
32 32                         2 spaces.
83 101 99 111 110 100         "Second choice"
99 104 111 105 99 101

27 89 58 60                   Move cursor to low screen.
36                            String-end marker.
```

The screen print positions can of course be varied to suit your requirements, but remember that the 'print at' escape sequence requires 32 to be added to both the line and the column numbers. The following listing would be suitable for displaying the menu.

```
ld de, MENU     17 M M        Address into DE,
ld c, 9         14 9            and call the
call BDOS       205 5 0         print function.
    continue . . .
```

If you compile and display it, you will see that it is actually part of a menu showing only two options, but more can be added by repeating the same general pattern as often as required.

Choice of keys

In my example, to the left of each option there is a block of inverse video showing the key to be pressed to activate it. This menu uses letter-keys. Number-keys are an alternative, but they allow at most 10 options whereas letters allow up to 52 because upper and lower case have different ASCII codes, though you could hardly display all 52 on screen at the same time. The real attraction of letters is that you can use the initials of the option and these are more easily remembered; you might use 's' for 'save' or 'supplier', 'c' for 'calculate' or 'customer' or 'credit note', etc. Equally you could use 'S' for 'save' and distinquish it from 's' for 'supplier', or whatever.

Upper or Lower ?

On the subject of upper versus lower case, you may wish to prevent distiction between the two (so that an input of either "n" or "N" stands for 'no', for example). This can be arranged by following the call of fnc 1 with an instruction to set bit No 5 of A to ensure that A then contains the ASCII of a lower case letter. (The ASCII of a lower case letter is always 32 more than the ASCII of its upper case.) Then, in the testing that comes later, you would need to test for only lower case not for both lower and upper. Alternatively, you could reset bit No 5 to ensure that A then always contains an upper case letter. Either way, the subsequent comparisons would all be in the case you selected, and the program sequence (for lower case) would start:

```
Loop:
    ld a, 1         62 1
    call BDOS       205 5 0
    set 5 a         203 239        Convert to lower case
        continue . . .
```

The cursor position

To make the inverse video blocks distinguishable, it is necessary to leave a blank line between options, and unless the normal screen cursor is disposed of it messes up the appearance of the last line. Moving it down to the bottom of the screen is an easy solution, and this is desirable anyway (see above); but alternatively you can use the "Disable cursor blob" escape code, which is 27 102, but then you must remember to enable it (27 101) when the menu has been used.

If you make a point of using 'disable' anywhere in a program, it is a good idea to attach 'enable' at the beginning of all other types of string; if the blob is already enabled then enabling it a second time will do no harm, and this way it doesn't get forgotten (once you have become used to a cursor blob, trying to input text without one can be quite unnerving!).

BUFFERED SELECTION

With buffered selection the user makes one or more keystrokes that will be recorded but otherwise ignored until he presses an 'OK' key. The OK is usually indicated by 'RETURN'. This allows the user to check his input before committing himself to it, but it is principally a way of inputting whole words or other character combinations that may be authorised by the programming. You can, of course use buffered selection for single key inputs, but I can't see any point in it unless, for some reason, it is vital not to hit the wrong key in which case the method gives the user extra time to check his input before pressing the 'accept' key. It may also be the case that you have used buffered keying elswhere in the program (for some multiple character input, say) in which case you may use it also for single character input, just to be consistent.

Buffered keying arranges to store the input somewhere, and when it gets a '13' it examines the stored material to see if it is listed amongst its authorised sequences. If all the authorised sequences are known to be of the same length, it also usually rejects inputs that are too long or too short without bothering to scrutinise them.

Function 10

This arrangement is easily provided by Fnct No 10 ("Read Console Buffer") which

a) stores the input in the buffer that has been declared for it.
b) refuses to accept more characters than the buffer is authorised to accept (thus preventing a 'too long' input).
c) counts the characters that have been typed in (thus making it possible to check for a 'too short' entry).

Suppose our program requires the user to key in a 5-character code that it will compare with a list of authorised codes. This might be the case if the user were being asked to type in his ID (identity) code before being allowed to access sensitive information, or if an accountant were asking for the display of records of the particular client who is identified by the code.

Setting up the Buffer

First the console buffer is established by inserting '5' (the allowed maximum number of characters in this case) in its first address, loading DE with the address, and then calling the function. The function will await the input and also simultaneously insert into the buffer and echo onto the screen. It will also sound a beep if more than 5 characters are typed and refuse to accept more. The user signifies that his input is complete by pressing 'RETURN'.

If, as I suggested, '5' is stated as the maximum buffer size, then there will be no opportunity to input too many characters, and this can be a disadvantage in security situations where you should avoid giving any clues as to what is authorised practice. Instead declare the buffer size to be something higher that the true number, 10 say, so he has the opportunity to type in too many and be disqualified.

Character count

Following the establishment of the buffer, the user will type in his text, and terminate with 'RETURN'. We then test the count in the second buffer address to see if 5 characters have actually been typed in (CP/M will not include the 'RETURN' key in the count).

A program example

Suppose that the screen is to request a code input by displaying the query:

"Which code ?"

Assuming that the rest of the screen has been prepared as necessary, this prompt will be available as a string such as

```
27 89 40 60             Print posn for query.
87 104 105 99 104 32    Text.
99 111 100 101 63
27 75                   Erase rest of line.
27 89 40 76             Print posn for reply.
36                      End-marker.
```

Suppose also that our buffer is to start at the address 'BUFF'. The process of printing the query string, establishing the buffer, and checking the count could be achieved by the following listing.

```
Loop:
    ld de, QUERY      17  Q  Q     Print
    ld c, PR_STR      14  9          the
    call BDOS         205 5  0       query.

    ld hl, BUFF       33  B  B     Address in HL, and
    ld (hl), 5        54  5          insert max num of chars.
    ex hl, de         235          Address into DE,
    ld c, C-BUFF      14  10         and call
    call BDOS         205 5  0       the fnc.

    ld a, (BUFF+1)    58  B+1 B    Count into A, and
    cp 5              254 5          compare with 5.
    jr nz Loop        32  232      If not 5 repeat.
     continue . . .
```

This procedure points DE to the query-string and prints it. The buffer is declared and filled. The count of characters is compared with 5. If the count is not 5, the program jumps back to print the string again, but if the count is 5 the program proceeds. This explains the presence in the string of the escape sequence '27 75'; if a repeat occurs this deletes the previously printed input characters, and the print position instruction that follows it re-establishes the screen location for the revised reply.

Testing the ID code

The program will have in memory a list of the authorised 5-letter codes, though each entry will in fact be more than 5-bytes long if it also contains a reference number (such as a folio number) or other data that the program will find it useful to extract when a correct entry has been located. In this case assume that the reference number is the

single byte that follows the code, so that each list entry is 6 bytes long. Suppose also that the list starts at the address 'LIST', and that an entry of all zeroes indicates that the end of the list has been reached. This pattern would be obtained if the whole list were zeroised before any entries were put into it. Call the address of the start of the console buffer text 'TXT' (so TXT = BUFF+2).

The intention is that the sub-r will return with the carry flag set and zero in A if the code is not found, but it will return with Cy reset and the relevant code number in A if it is found. A possible listing to follow on from the earlier one that prepared the console buffer might be:

```
FOLIO
    ld hl, List-6     33  L-6 L    6 addrs before list start
    Ld de, Text       17  T   T    Start of text
Next
    ldd bc, 6         1   6   0    Next list entry
    add hl, bc        9

    ld a, (hl)        126          If it is
    or a              183            zero then
    scf               55             set Cy and exit
    ret z             200            because list has been exhausted

    push de           213          Save the
    push hl           229            the two addresses
    ld a, 5           62  5        Count of 5
    call COMPARE      205 C   C    Comp text & list entry (page 163)
    pop hl            225
    pop de            209
    jr c Next         56  237      Not same so go for next

    ld de,5           17  5   0    Point to
    add hl, de        25             Folio number
    ld a, (hl)        126          Extract it into A
    or a              183          Reset Cy
    ret               201          And finish
```

Chapter 12

Cursor Menus

There are two main approaches to creating a cursor; you can either display a 'pointer' beside the text, or you can convert each section of text in turn into reverse video. Both approaches can be achieved by a print operation or by the more flexible method of manipulating the screen data.

Printed cursors

The simplest of all pointers is a custom made cursor rectangle generated by subjecting a screen print-position to inverse video. To do this we first need a cursor-string made up as follows:

```
        27 102      Disable the cursor blob
        27  89 A X  Print at
        27 113      Inverse video off
        32          Space
/*... 27  89 B X    Print at
        27 112      Inverse video on
        32          Space (the Moving Cursor)
        27 113      Inverse video off          ...*/
        27  89 C X  Print at
        27 113      Inverse video off
        32          Space
        36          End-marker
```


As an alternative to a simple rectangle you can use other symbols such as those given by ASCII codes 62 or 252 (assuming the cursor is on the left of the text). Alternatively you can design a pointer of your own as described in *PCW Machine Code* Chapter 9, and in Chapter 9 in this book. Whichever you use, it should replace the middle '32' in the string.

The value of 'X' is the number of the column (+32) in which the cursor is to appear. You choose that to suit yourself, but note that there are three places in which the same value of 'X' must be inserted.

When printed, the string first disables the usual flashing cursor so that it doesn't interfere with the display (but remember to turn it on again later). For the moment ignore the rest of the string except for the section between the two markers /*...*/ , which contains the 'SPACE' that will act as the new moving cursor and the escape sequences that position it and convert it to inverse video and then switch back to normal. The line on which it appears will be determined by the value of 'B'. If you want to start it on the 9th line down, you will give 'B' a value of 40, etc.

We now wait for an up or down arrow to be pressed. If a down arrow is detected we add one to 'B' and print the string again, thus printing the cursor on the line below. If we detect an up arrow, we subtract one from 'B' and print again, this time making it appear on the line above.

Unfortunately, if we do nothing else we will then have two cursors one above the other, but the first and last parts of the string are designed to get rid of the one we don't want. If we make sure that 'A' is always one less than 'B', and that 'C' is always one more than 'B', then the unwanted cursor will always be overprinted by a blank when the new one appears on the line above or below. Thus on detection of a down arrow 'A', 'B', and 'C' should all be incremented, and on detection of an up arrow they should be decremented.

If you want the cursor to jump two lines at a time then 'A' should be two less than 'B', and 'C' should be two more than 'B', and the values should all be increased or decreased by two at each movement.

You should also test the value in 'B' so that up arrows are ignored if the cursor is too high up the screen, and so that down arrows are ignored if it is too low.

Bearing in mind that the arrow keys have ASCIIs of '31' for up and

'30' for down, the following listing would control the cursor if the string is located at Addr and the cursor is to move one line at a time:

```
CURSC
     ld de, Addr    17  N  N       Print
     ld c, 9        14   9          the
     call BDOS      205  5  0       string
Await
     ld c, 6        14   6         Await
     ld e, 255      30  255         a keypress
     call BDOS      205  5  0       (no echo)

     cp 13          254  13        If 'RETURN' or
     ret z          200              'ENTER' pressed then finish

     ld hl Addr+20  33  N  N
     ld c, (hl)     78             'C' into C

     cp 30          254  30        Down arrow ?
     jr nz 3        32   3           if not then jump on
     inc c          12        *    Else increment 'C'
     jr 5           24   5          and jump on

     cp 31          254  31        Up arrow ?
     jr nz Await    32  231          if not then get next key
     dec c          13        *    Else decrement 'C'

     ld (hl),c      113            New 'C' into string
     dec c          13             New 'B' into C
     ld hl, Addr+11 33  N  N         and into
     ld (hl), c     113              string
     dec c          13             New 'A' into C
     ld hl, Addr+4  33  N  N         and into
     ld (hl),c      113              string

     jr Start -47   24  209        Go to start
```

The sub-r starts by printing the string and then awaits a key-press with fnc No 6, which is preferable to fnc No 1 in this case because it records the pressed key in A without echoing it on the screen. To use it in this way, E is first loaded with 255.

The value in A is then compared with '13'. If it is 13 then the sub-r terminates because this indicates that the user has pressed 'RETURN' to show acceptance of a particular menu item. The identity of the item

accepted can be found by testing the value of 'B' (the line number of the cursor) which is at Addr+11 in the cursor string.

Then the value of 'C' is stored in the C register ready for incrementing or decrementing.

The value of the pressed key is now compared with '30' (down). If it isn't 30 then a jump is made over the next three bytes, but if it is 30 then the value in C is incremented and a jump made to the insertion into the string of the revised value. If it wasn't 30 then it is tested against '31' (up). Failure of this test gives a jump back to await another key-press. If it was 31 then the value in C is decremented. These tests have therefore eliminated incorrect key-presses and adjusted the value in C to accord with a correct one.

The adjusted value in C (which is the new 'C') is put back into the string. We know that 'B' is one less than 'C', and that 'A' is one less than 'B', so the the value in C is decremented to obtain their values and these are also put back into the string.

A jump is then made to the start of the program to reprint the string and await the next key-press.

If you wish to insert extra code to control the range of movement of the cursor it should be put in at the places marked by the asterisks "*". In doing this, bear in mind that the sub-r needs the value in A (the code of the pressed key) so this will have to be 'pushed' and later 'popped' (or briefly moved into another register) if you use the A register for testing the line value of the cursor.

Inverse video by printing

To move a bar of inverse video up and down through the text lines of a menu, the text needs to be provided with the proper escape-sequences for each line. The following example gives the idea:

```
13 27 89 42 62  27 113    "First line of text"   27 113
13 27 89 43 62  27 112    "Second line . . ."    27 113
13 27 89 44 62  27 113    "Third line . . . ."   27 113
. . . etc
```

As shown, the second line will be printed in inverse and the others in normal video. To move the inverse it is necessary to change the '112'

to '113' so that the second line will then print in normal mode, whilst setting the inverse in the line above or below, depending on which arrow key is pressed. To achieve this it is necessary to keep a record of the addresses of the bytes that may need to be changed, or this can be done by simple calculation if the line lengths are all the same. In either case it is essential to limit the 'range of travel' or the inverse instruction will be inserted outside the string and the code on each side of it will become corrupted. After each change the string has to be reprinted. Identification of which text option the user chose is most easily achieved by an independent count (in a spare address) that is incremented by down arrows and decremented by up arrows.

Cursors by screen data manipulation

It is much more convenient to produce reversed video cursors by manipulating the screen data, and the technique can be used to make either simple pointers or inversed text lines. As well as being easier to use, the technique operates more quickly because there is no requirement to keep reprinting the menu text. It uses the screen routines developed at the end of Chapter 8, plus the one listed below, called 'REVBAR', which actually makes the reversed video bar. REVBAR must be in common memory. My address for it is (114,192), which is what I will use in all references to it.

On entry to it, the following information should be in the specified registers or in the screen variables area:

E reg or (6,192)	Line number	(0 to 31)
D reg or (7,192)	Start column number	(0 to 89)
A reg or (8,192)	Number of columns	(1 to 90)

Data outside the limits is rejected with Cy set, otherwise Cy is reset. There are two start addresses. The first is used if the data is in the registers, or the second if it is already in the variables area.

```
REVBAR
Start1:     Address  (114,192)
    ld (6,192), de    237 83 6 192    Put data into variables
    ld (7,192,) a     58  8  192        if not already in.
Start2:     Address  (121,192)
    ld bc, PROG       1   X  X        Point BC to program
    jp HANDL          195 20 192        as required by HANDL
```

continued on next page . . .


```
Program:    Address  ( X,X ) = (127,192)
    call FADDR      205 35 192    Screen address to HL
    ret c           216           Finish if error
    ex hl, de       235           Then into DE
    call BLEN       205 94 192    Bytes to HL
    ld b, h         68            Then into
    ld c, l         77             BC
Loop:
    ld hl, 47 179   33  47 179    Last screen address
    ld a, (de)      26            Extract a byte
    cpl             47              reverse it
    ld (de), a      18              and replace
    inc de          19            Point to next
    or a            183           If beyond
    sbc hl de       237 82          end of screen
    ret c           216             then finish
    dec bc          11            Decrement count
    ld a, b         120             of bytes
    or c            177             and loop if
    jr nz Loop      32  240         not zero
    ret             201           Else finish
```

Note that REVBAR reverses in either direction so a bar already reversed is restored to normal by it. When using it to produce menu cursors, call it, then change the line number in the variables before calling it again. This un-reverses the previous line and reverses the new one.

Cursor scanning

In this case it is easy to keep the cursor within bounds, and to send it to the other end of its travel if it attempts to exceed its range. Hence in the following example if the user has scanned all the options downwards and presses the down arrow again, the cursor is flipped to the top of the menu by replacing the value in (6,192) with the upper line number. The same technique is used to flip it downwards from the top. My version is at (2 ,194), so I will use that address when referring to it. The data required in the feed is:

E reg	Top line of menu	(0 to 30)
D reg	Left Colm of cursor	(0 to 88)
L reg	Cursor width	(1 to 90)
H reg	Bottom line of menu	(1 to 31)

```
CSCAN     Address   (2,194)
    ld (0 192), de   237 83 0 192
    ld (6 192), de   237 83 6 192
    ld (8 192), hl   34 8 192
Loop:
    call REVBAR.2    205 121 192
Await:
    ld c, 1          14 1            Await a
    call BDOS        205 5 0           key
    cp RETURN        254 13          Finish if it is
    ret z            200               'RETURN'
Exit:
    cp EXIT          254 27          If 'EXIT'
    jr nz 2          32 2
    scf              55                set Cy
    ret              201               and finish
Down:
    cp DOWN          254 30          If not a down arrow
    jr nz Up         32 26             jump to Up
    call REVBAR.2    205 121 192     Else cancel the cursor
    ld hl, 9 192     33 9 192        Point to bottom line
    ld a, (6 192)    58 6 192        This line number
    cp (hl)          190             If not the same
    jr nz 8          32 8              jump on
    ld a ,(0 192)    58 0 192        Else put the top line
    ld (6 192), a    50 6 192          number into this line
    jr Loop          24 215            number and repeat
    inc a            60              Not same so
    ld (6 192) a     50 6 192          move down
    jr Loop          24 209          And repeat
Up
    cp UP            254 31          If not an up arrow
    jr nz Await      32 208            get another key
    call REVBAR.2    205 121 192     Cancel the cursor
    ld hl, 0 192     33 0 192        Point to top line
    ld a, (6 192)    58 6 192        This line number
    cp (hl)          190             If not the same
    jr nz 8          32 8              then jump on
    ld a, (9 192)    58 9 192        Else put bottom line
    ld (6 192), a    50 6 192          number into this line
    jr Loop          24 185            number also and repeat
    dec a            61              Not the same so
    ld (6 192), a    50 6 192          move up
    jr Loop          24 248          And repeat             END
```


The program starts by making a cursor bar on the top line of the menu. This line number is stored at both (0,192) and (6,192). As the cursor is moved, the value at (6,192) is changed but the one at (0,192) is kept as a reference. The bottom line (the lowest allowed position) is stored at (9,192). It then awaits key-press. If this is '13' then an exit is made with Cy reset. If it is '27' (EXIT) then a return is made with Cy set. Testing Cy indicates whether the user has completed his use of the cursor or merely selected a cursor position. If the key is '30' the procedure for moving the cursor down is followed. If it is '31', the up procedure is used. No other key has any effect.

Other uses of cursors

Making cursors by screen data manipulation has even more interesting applications. Menus with more than one column of choices can easily be accommodated (as is done in the main disc-manager for Locoscript documents), even if the columns have different text widths or start in different screen positions. It is also possible to inverse words within a normal text item (as opposed to in a list) to draw the users attention to them for a 'yes/no' reply.

Because printed cursors obliterate anything underneath them they can't be superimposed on existing text, but screen data cursors can be put anywhere without affecting existing material.

Chapter 13

Loading Screens

What you see is whatever you like

This chapter describes various methods of transferring pictures or patterns to the screen and then animating them. A common use of these techniques is in making **loading screens**; diverting displays to be shown during the time that large chunks of data or long programs are being sucked into the machine from disc. No conceivable arrangement of dots of light can be ruled out as a candidate for performing this service, but whatever they look like they are all aimed at keeping the comsumer re-assured and impressed whilst something unavoidably boring is happening. Not all of them succeed (though I'm sure yours will) and a point to bear in mind on the design side is that the same display will (presumably) always be shown, and whilst it may be diverting on the first occasion, after a dozen showings the novelty will have worn off so there is something to be said for a gentle, low key, approach that doesn't grate on the nerves after the *n*th display.

In addition to the methods described below, text can be brought to the screen in visually interesting ways by some of the customised printing techniques described in Chapter 8. As with all programs involving block switching, the ones in this chapter need to be in common memory.

Screen data

The record of which screen pixels are 'on' and which are 'off' is kept in blocks 1 and 2 between addresses (48,89) and (47,179). Any changes made to the bits stored between these addresses will instantly appear as a change to the screen display. Try this for example, it clears the screen by turning it white:

```
WCLS    Address ( W,W )
      ld a, 8            62  8          Screen
      out (a), 248       211 248          off
      ld bc, SUBR1       1  S1  S1      Point to sub-r
      call USERF         205 90 252     Call
      DEFW               233 0            Screen Run Routine
      ld a, 7            62  7          Screen
      out (a), 248       211 248          on
      ret                201            And finish

Address ( S1,S1 )
      ld hl Scrn         33  48  89     Start of screen data
      ld (hl), 255   *   54  255        Prime with a 255
      ld de Scrn+1       17  49  89     Next byte
      ld bc Count        1   255 89     Screen bytes minus 1
      ldir               237 176        Run through
      ret                201              and finish
```

The routine points BC to the Sub-r at (S1,S1) and then invokes the Screen Run Routine. The sub-r fills each screen byte with a 255, thus turning it white. The starting and finishing 'out' instructions are a tip that I obtained from Geoffrey Childs' book *Streamlined Basic*; the effect of them is to switch the screen off (A = 8) before the clearance starts, and then back on again when it is complete (A = 7). Without these the clearance almost always looks a bit ragged because it is obliged to take place in two parts, the size of the parts depending on the state of Roller-Ram. With them in place it flicks off and on again in the new state almost instantaneously.

You can get variations by changing what (HL) is loaded with at (*). If with '0' then the screen will be cleared to black, '85' will produce a 'grey' clearance, etc. The numbers 0, 85, and 255 all represent symmetrical arrangements of pixels and so the display they produce is 'homogeneous'. Most other numbers produce various sorts of vertical banding because the set pixels of one line corresponded to the set pixels of the one above and below, but you can produce pleasant 'hatch-

ing' by filling a pixel line with one number and the one below by a compensating number. Among the good pairs are 102 and 153, 136 and 34, and others. A good trio to use in this way is 36, 73, and 146. Rotating a number before filling the next pixel line can produce attractive diagonal bandings, but use the cyclical 8-bit rotations such as 'rlc a'. Using two such rotations between pixel lines produces bands inclined at 45 degrees to the edges of the screen.

Does CP/M have a loading screen?

I'm not sure if the bars of black on white that appear when you insert your CP/M disc actually rate as a 'loading screen', but, whilst they may not be the last word in graphic art, they do have the merit of encouraging you to think that an anticipated event is about to occur, and in my book that is at least half the point of using them.

Just to prove that it can be done without a lot of hassle, the following routine mimics the CP/M bars exactly.

```
SBARS
      ld bc, SUBR2     1  S2  S2      Point to next Sub-r
      call USERF       205  90  252   Call the
      DEFW             233  0         Screen Run Routine

Address  ( S2,S2 )
      ld hl, Roller    33  0  182     Start of Roller-RAM
Loop:
      ld e, (hl)       94             Half 1st screen
      inc hl           35               addr
      ld d, (hl)       86               into DE
      ex hl, de        235            Swop the two
      add hl, hl       41             Get actual screen address
      ld a, (hl)       126              and test the screen
      or a             183              byte. If it is not zero
      jr nz Bar        32  14           jump on to zeroise it
      inc hl           35             Else
      inc hl           35               move
      inc hl           35               4 pixel lines
      inc hl           35               down the screen.
      ld a, (hl)       126            And
      or a             183              repeat
      jr nz Bar        32  6            the test
```

continued on next page . . .


```
      ld hl, 15        33  15  0      Byte was zero in both cases so
      add hl, de       25               point to Roller addr of next
      jr Loop          24  233          pixel line and repeat.
Bar:
      ld b, 90         6  90          90 bytes per line
Loop2:
      ld (hl), 0       54  0          Zeroise a pair
      inc hl           35               one above
      ld (hl), 0       54  0            the other
      ld de,.7         17  7  0       Then move to each
      add hl, de       25               adjacent print position
      djnz Loop2       16  245          until line complete.
      ret              201            Finish
```

Before using 'SBARS', the screen must be reversed as by using 'WCLS' described above, or by some other method. Each time 'SBARS' is called it adds another black line to the screen, so you can then use it as CP/M does - during the loading of a program - by calling it each time the DMA address is increased prior to looping back for the next record (see *PCW Machine Code*, page 114, and page 147+ below). The program puts 2 horizontal bars into each print line so the screen can hold only 64 bars. If your file is more than 64 records long then you will need to invoke 'SBARS' every two or three records or you will overwrite the Roller-Ram. Invoking it for every record is in any case a bit fast.

If you just want to see the effect you can repeatedly call 'SBARS' from a simple await-key loop, but exit the loop before the screen fills up.

PRE-COMPOSED SCREENS

Much more visually interesting effects can be obtained from storing one or more alternative 'screens' in memory, and quickly inserting one in place of the one we wish to dispense with. Because of the amount of data involved it is clear that the only practical storage place for them is in the 'Memory Disc', ie. in blocks Nos 9 and above. So the screen 'insertion' would consist of transferring the data from a pair of storage memory blocks into blocks 1 and 2.

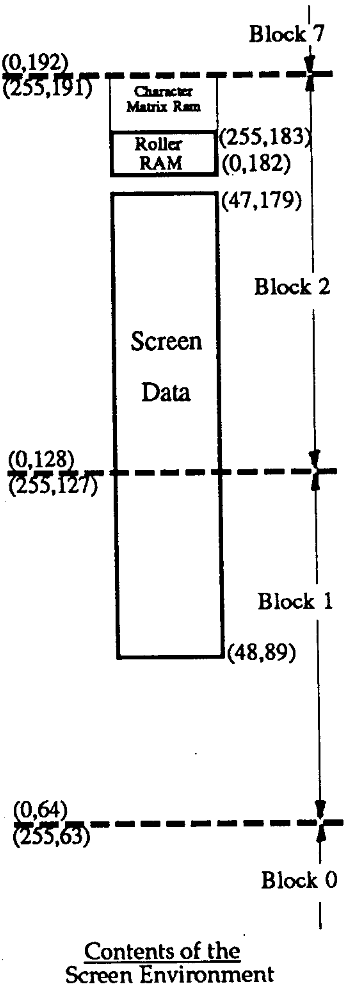
Two blocks are needed because there are 90 pages of screen data and 2 of Roller-Ram, plus some intervening bytes, (see opposite) making a total of nearly 95 pages in all (24,272 bytes in fact), whereas a single block can hold only 64 pages.

The quickest and simplest approach to transferring data from one block to another is to use the block switching technique described in Appendix 4 (which is reproduced from *PCW Machine Code*). It is important with this technique not to get the blocks tangled up, though keeping them straight is not difficult.

Storing screen data in memory

First construct your screen. If it is to be text then it can be assembled in any of the usual ways. If it is to be a picture or a geometric pattern, then you will presumably use a piece of graphics software, with which the use of a light pen makes the task comparatively easy

Once the screen has been prepared, it can be stored by a routine such as the one following. The storage area chosen in this case is blocks 9 & 10, though any pair of blocks would do. (See "Choosing blocks" below). First block 10 is connected to port 240 [address range (0,64) to (255,127)] and the latter's data is transferred to the former. This particular pattern of attaching the blocks to ports is not significant. As long as the two are in circuit together it wouldn't matter where they were, though naturally you mustn't disturb block 7 in the top address range.



Contents of the Screen Environment

It is necessary to disable the interrupts to ensure trouble-free block switching, and the time interval between disabling and enabling them again should be kept as short as is reasonably possible. It is also necesary to put block 0 back in place so that it is immediately available when it is wanted. It is not necessary to do anything additional to restore port 241 because the restoration of the TPA handles this by putting the usual block 5 into it.


```
STORE
    di                    243          Disable interrupts
    ld a, '10'            62  138      Connect block 10
    out (240), a          211  240
    ld a, '2'             62  130      Connect block 2
    out (241), a          211  241
    ld hl, '2'      *     33  0  56    Copy the block
    ld de, '10'     *     17  0  0
    ld bc, Count          1  0  64
    ldir                  237  176
    ld a,'9'              62  137      Connect block 9
    out (240), a          211  240
    ld a, '1'             62  129      Connect block 1
    out (241), a          211  241
    ld hl, '1'      *     33  0  89    Copy the block
    ld de, '9'      *     17  0  0
    ld bc, Count          1  208  38
    ldir                  237  176
    ld a, '0'             62  128      Restore block 0
    out (240), a          211  240
    ei                    251          Restore interrupts
    ld a, 1               62  1        Restore TPA
    call MEM_M            205  33  253
    ret                   201          Finish
```

Re-establishing the screen

Fetching the stored screen for display is the reverse procedure. All the '33s' become '17s', and vice versa (see the asterisks *). Note that it is not a good idea to copy the whole of block 1 because when it was copied back into memory it would restore an out-of-date disc directory in its lower addresses.

Choosing blocks

If the 'memory disc' has been used, then its directory and the data in it will start at block 9 and use progressively higher blocks to suit its needs, so there is reason enough not to use the lower blocks for other

purposes unless you are sure this point can be ignored.

If you select a high block number (higher than 16) on the assumption that your program is to be run in a '512' machine (they have 32 blocks) but later it is actually run in an '8256' (they have only 16), then the machine will automatically adjust the block number by subtracting 16 to make it viable. This may be very convenient in some cases, but it would produce a profound hiccup if your chosen block number were between 16 and 24. The subtraction of 16 in these cases would produce numbers between 0 and 8, and hence the operating system would be overwritten.

Storing screen data on disc

Access to the screen data can be obtained only through block switching or by using the Screen Run Routine, and both of these prevent file handling because you can't make system calls with the interrupts disabled nor with the Screen Envirionment in place. In consequence the data must be transferred first to Common Memory and be filed from there.

Because there is no urgency about putting the screen data onto disc, the simple approach is to transfer it in chunks into block 7 and then disc it from there, though it is important to take care with the boundaries of the chunks and to ensure that it comes back into memory in an identical sequence. To help in this I like to pack it out to an exact number of 128-byte records, and transfer it to block 7 in two equal pieces.

As already stated, the quantity of data including Roller-Ram is (208,94) bytes, so make this up to 96 pages and transfer in two lots of 48 pages. If you transfer each in turn into address (0,196) in block 7, then they will probably not overwrite any programming you may have in residence at the start of the block, and they will fit comfortably under the CP/M and stack area that starts at about (0,246). So a typical arrangement for copying onto disc might be:

```
Create File:
    ld de, FCB      17  N  N     Delete any file of
    ld c, Delete    14  19         the same name'
    call BDOS       205 5  0
    ld de ,FCB      17  N  N     Create a new file
    ld c, Create    14  22
    call BDOS       205 5  0
```



```
    inc a           60           Test
    jp z Error      202  E  E     for error
    continue . . .
```

This makes sure you are not trying to make a second file of that name, and creates a new one ready for the copying. Then follows:

```
    ld b, 2         6   2        Two copying
Loop:
    push bc         197            operations

    ld bc, SUBR9    1   S9  S9   Point to sub-r and
    call USERF      205 90  252    establish blocks 1 & 2 by
    DEFW            233 0          Screen Run Routine
    ld a, 1         62  1        Restore TPA
    call MEM_M      205 33  253
    jp File         195 F   F    Goto 2nd part of prog

Address  ( S9,S9 )
    ld hl, (A A)    42  A  A     Get the start address
    push hl         229          Save it
    ld bc, Count    1   0  48    And add Count
    add hl, bc      9              ready for
    pop hl          225            next copy
    ld de, Common   17  0  196   Copy
    ldir            237 176        to block 7
    ret             201          Return to Screen Run Routine
```

These operations establish the Screen Environment and copy the first lot of bytes from the address stored at (A,A) into block 7, and the address at (A,A) is incremented to the address for the second transfer. The TPA is re-established and then follows the filing operation.

```
Address (F,F)
    ld de Screen    17  0  196   Start of screen data
    ld b, 190       6   96       Count of records
Loop1:
    push bc         197          Store count
    push de         213            and address
    ld c SetDMA     14  26       Set DMA address
    call BDOS       205 5  0
    ld de FCB       17  N  N     Write
    ld c, Sequ      14  21         128 bytes
```

continued on next page . . .

```
call BDOS       205  5  0      to file
or a            183            Test
jp nz Error     194  E  E       for error
pop de          209            Recover DMA
ld hl 128       33  128  0      address and
add hl, de      25              add 128
ex hl, de       235
pop bc          193            Recover count of
djnz Loop1      16  228          records & repeat if not zero

pop bc          193            Recover count of
djnz Loop       16  X            operations & repeat if nz.
```

This returns to 'Loop' for copying the second batch, or closes the file by:

```
ld de, FCB      17  N  N       Close
ld c, Close     14  16
call BDOS       205  5  0
inc a           60
jp z Error      202  E  E      Jump if Error

ret             201            Else finish
```

These operations cover the 'straight' copying of screen data onto disc, so the bytes are stored in the order in which they occur in memory, and naturally they would be recovered by a similar process in the same sequence. A quite different approach to storage is described under 'Picture Building' below.

Artificial screens

As well as saving screens that have been generated on the VDU (which is buzz for 'screen'), you may sometimes want to create one numerically, or in some other artificial way, and in cases where there is no existing Roller-Ram it will be necessary to install an artificial one. To have the desired effect it must occupy the memory location that is (209,2) bytes above the top of the screen data because the screen usually ends at (47,179) and the Roller-Ram usually starts at (0,182). Additionally, the contents of the Ram must follow the rules:

1. Entries are in batches of eight 16-bit numbers, and within a batch each entry is 1 more than the one before.

2. The first member of each batch is half the address of the start of the print line to which it refers.
3. The first entry of one of the batches must be half the lowest address of screen data.
4. The first member of each batch is 360 (=720/2) larger than the first member of the batch preceding.

In relation to rule 2, I have found that the entry referred to cannot always be obtained by dividing the address by 2. The entry when multiplied by 2 must equal the address, but if I am right then the multiplication must have lead to overflow. Hence in the case of the entry for the lowest byte of normal screen memory, the address is (48,89), but the entry must be (152,172), not (152,44). [Doubling either of these gives (48,89) if you ignore the overflow in the first case.] I should point out that there are those who say that this is rubbish. They claim that a simple halving will do. Try it and see.

A simple sub-r to fill out a Roller-Ram for a standard screen could be:

```
RRAM
    ld hl, Roller       33  0  182     Start of Roller-Ram
    ld de, 152 172      17  152  172   Half lowest screen address
    ld b, 32            6  32          32 lines
Loop:
    ld c, 8             14  8          8 bytes per line
Loop2:
    ld (hl), e          115            Address
    inc hl              35              into
    ld (hl), d          114             table
    inc hl              35
    inc de              19             Next address
    dec c               13             Decrement count of bytes
    jr nz Loop2         32  248        Next byte
    push hl             229
    ld hl, 352          33  96  1      360-8
    add hl, de          25             Content for next line
    ex hl, de           235            Into DE
    pop hl              225
    djnz Loop           16  239        Goto next line
    ret                 201            Or finish
```

Port 245

For an alternative method of creating a Roller-Ram, see page 105.

Resetting Roller-Ram

All the quoted examples and the ones that follow in this chapter take it for granted that Roller-Ram will be in an unpredictable condition at the time that a screen is recorded. To cover all eventualities it is as well to make this assumption, but you can significantly reduce the amount of time that your programs spend on calculating addresses if you are able to reset the Roller-Ram to a known state before the screen is composed and recorded. This could be achieved by:

```
SRES
    ld a, 8         62  8        Screen off
    out (a), 248    211 248
    call RRAM       205 R  R     Reset Roller-Ram
    ld e, 27        30  27       Cursor
    ld c, 2         14  2         to
    call BDOS       205 5  0      top
    ld e, 72        30  72        left
    ld c, 2         14  2
    call BDOS       205 5  0
    ld a, 7         62  7        Screen on
    out (a), 248    211 248
    continue . . .
```

You can now guarantee that the address of the top left byte is (48,89), and that all the others follow in the predictable pattern (see Appendix 7).

ANIMATION

There are many ways of achieving movement or change in a screen display and I will lump these all together under the rather grand title of 'animation'.

Picture building

The simplest type of change is the one given by the 'CP/M loading screen' described earlier. This arranges for some kind of incremental addition to the display as time passes. Instead of adding black lines you could arrange to add new parts of the final picture so that it builds

up slowly from either the top or the bottom of the screen, or from both simultaneously. The additions can be either print lines, or pixel lines. The latter give a smoother effect but take 8 times as many operations, though that is no penalty if you are looking for a snazzy result.

A pixel line across the screen consists of 90 bytes, and a file record holds 128, so the data for a pixel line can comfortably be accommodated in a record, and records of program material and screen data could be inter-leaved alternately in the file. Your file loading program would then be constructed so that it directed alternate file records to the screen and to memory storage. As there are 256 pixel lines, you will have to adjust the inter-leaving rate if your file contains more or less than 256 program records, which it usually will.

If there are not enough program records, then handle the surplus pixel records as a batch at the end, but introduce pauses so that they appear on the screen at the same rate as earlier. If there are too many program records, then feed a few in at the start before the picture building begins. This will in any case be necessary because some programming has to be loaded before anything else can be achieved.

The following bare bones of a loading program indicate the loading procedure. It assumes that the first and all odd numbered records are program material, and that the second and all even numbered ones are screen data. The count put into BC is the number of program records (not the number of records in the file).

```
    initialise with 'Open', etc . . .
    ld de, Addr        17  A  A     1st addr for prog records
    ld bc, Count       1   C  C     Count of program records
Loop:
    push bc            197          Store count
    push de            213            and address
    ld c, SetDMA       14  26       Set the address for
    call BDOS          205 5  0       this program record
    ld de, FCB         17  F  F     Point to FCB
    ld c, READ         14  20       Read this
    call BDOS          205 5  0       program record
    or a               183          If an error then
    jp nz Error        194 E  E       goto error procedure

    ld de, PixelStore  17  P  P     Set fixed address into
    ld c, SetDMA       14  26         which pixel line
```

continued on next page . . .

```
call BDOS        205  5  0      is to be read
ld c, READ       14  20         Read the
call BDOS        205  5  0        pixel line record
or a             183            If an error then
jp nz Error      194  E  E        goto error procedure
call LINE        205  L  L      Call the routine to deal
                                  with the pixel information
pop de           209            Increment the
ld hl, 128       33  128  0       program DMA by 128
add hl, de       25
ex hl, de        235
pop bc           193            Recover the record
dec bc           11               count and decrement it
ld a, b          120
or c             177
jr nz Loop       16  205        Repeat if count not zero

finish with Close, etc . . .
```

The program is all as usual except that in addition to reading a program record at each pass, it reads a pixel line record as well, and then calls 'LINE' to deal with this. In all cases the pixel record data is put into the same 128 bytes in common memory so that LINE always knows where to find it. LINE, which has a lot in common with SBARS, lists as follows. This version assumes that the picture is being imposed upon a black screen. If you intend to start from a white screen, then change the zero at (*) to '255'.

The procedure for saving a screen to disc would be identical except that the information would flow in the opposite direction.

```
LINE
    ld bc SUBR7     1  S7  S7      Point to Sub-r
    call USERF      205  90  252     and call the
    DEFW            233  0           Screen Run Routine
    ld a, 1         62  1          Restore the
    call MEM_M      205  33  253     TPA
    ret             201            Return to Loading program

Address ( S7,S7 )
    ld hl Roller    33  0  182     As in SBARS

Loop: ld e, (hl)    94
```



```
    inc hl          35
    ld d, (hl)      86
    ex hl, de       235
    add hl, hl      41
    ld b, 8         6  8
Loop2:
    ld a, (hl)      126
    cp 0       *    254  0
    jr z Fill       40  9
    inc hl          35
    djnz Loop2      16  248
    ld hl 15        33  15  0
    add hl, de      25
    jr Loop         24  235
Fill:
    ld b, 90        6  90          90 bytes per line
    ld de Pixels    17  P  P       Point to pixel store
Loop3:
    push bc         197            Save byte count
    ld a, (de)      18             Copy a byte
    ld (hl), a      119              to screen address
    inc de          19             Next stored byte
    ld bc 8         1  8  0        Add 8 to
    add hl, bc      9                screen address
    pop bc          193            Recover count
    djnz Loop3      16  245          and repeat if not zero
    ret             201            Else return to the loading prog
```

The sub-routine always starts at the address of the top pixel line of the screen and examines each first byte down the left edge to see if this is the pixel line to fill. If the byte is zero and this is a black screen then it is the one to fill, so you must arrange for every new pixel line to start with a byte that is not zero. If the screen is to start from white, then each first new byte must not be 255.

This fractionally restricts the make-up of the left edge of the picture, but as in both cases there are 255 alternative bytes to choose from, not even the most temperamental among us could claim that his artistic intengrity was being encroached upon. And if you really must start from a black screen and have a black left side to the picture, then make each first new byte a '128', and each last new byte a '1'. This would produce a barely noticeable thread of light symmetrically down both edges of the screen.

Random picture building

Instead of feeding all the new pixel lines in at the top of the screen, it is possible to introduce them randomly, or in some complex pattern that tantalisingly reveals parts of the picture out of normal sequence.

To achieve this, each 90 bytes of pixel line data should be preceded by a line number when they are stored in their record. As there are 256 pixel lines, the single-byte numbers 0 to 255 will cover them all. The sub-r 'LINE' must then be modified to read this byte and position the pixel data accordingly. This would be achieved by:

```
LINE2
    ld bc SUBR7      1   S8  S8
    call USERF       205  90  252
    DEFW             233  0
    ld a, 1          62   1
    call MEM_M       205  33  253
    ret              201

Address  ( S8,S8 )
    ld a, (P,P)      58  P  P      Line number into  A
    and 248          230  248      8 x INT (A/8)
    ld l, a          111           Result
    ld h, 0          38  0           into  HL
    add hl, hl       41              and double it
    ld de Roller     17  0  186    Start of Roller-Ram
    add hl, de       25            Point to print line entry
    ld e, (hl)       94            Extract half of address
    inc hl           35               of start of this
    ld d, (hl)       86               line
    ex hl, de        235
    add hl, hl       41            Double it
    ld a, (P,P)      58  P  P      Extra
    and 7            230  7          pixel lines
    ld e, a          95              into
    ld d, 0          22  0           DE  and
    add hl, de       25              add to screen address

Fill:
    ld b, 90         6  90         90  bytes per line
    ld de Pixels+1   17  P+1  P    Point to pixel store
```



```
Loop2:
    push bc          197           Save count
    ld a, (de)       18            Copy a byte
    ld (hl), a       119             to screen address
    inc de           19            Next stored byte
    ld bc 8          1  8  0       Add 8 to
    add hl, bc       9               screen address
    pop bc           193           Recover count
    djnz Loop2       16  245         and repeat if not zero
    ret              201           Else return to the loading prog
```

The first task of the sub-r is to locate the relevant entry in Roller-Ram. There are 8 pixel lines per print line and 16 bytes of Roller-Ram per print line so the pixel line number is put into A and the result

8 x INT (A/8)

is obtained. This could be derived by dividing A by 8 and then multiplying the integer result by 8, but it is more simply obtained by

and a, 248

which masks out the lowest three bits. This is then doubled to count the bytes from the start of Roller-Ram (16 bytes for every 8 pixel lines). When the entry for the start of the print line has been extracted and doubled, the remainder produced by dividing A by 8 (obtained from 'and a, 7') is added to it. HL now holds the address of the pixel line.

Speckling

There are several ways to make your picture appear apparently ramdomly at locations throughout the screen rather than progressively from the top edge, but inevitably they all involve more programming. Picture building by 'speckling' (the random appearance of its pixels all over the screen), is a favourite one of these.

A lot of labour can be saved by making the speckling appear to be random whilst actually making it quite orderly. Hence instead of lighting up ten pixels on ten different lines in succession, light up ten at widely spaced positions on the same pixel line at the same time,

and then move to ten on another line. Further effort will be saved by introducing nearly all of the speckling in the centre of the screen first, whilst moving gradually outwards to the edges.

A simple way of achieving the effect is to have two, three, or more versions of each pixel line. The first of these is only a sparse skeleton of the true line, with subsequent versions as more and more complete copies of it. The most sparsely filled ones are screened first and then, when all of them have been screened, their more advanced versions are gradually superimposed upon them. You can cut down on the number of such lines by restricting the area covered by speckling to only the centre of the screen, and fill the rest of it in a more economical way.

It is also possible to introduce bytes (or print-locations of 8 bytes) rather than pixels in a speckled manner. This would be done by the data table method described under 'Table Printing' in Chapter 8 (see page 97). The table would contain 3-byte entries that defined the byte and its required screen position. As there are 23,040 bytes throughout the screen your table would be longer than 64k if you tried to describe the whole screen in this way, and whilst that would not be impossible it would at least be formidable. It might seem possible even to table individual bits in this way but there are 8 times as many of them and locating them, though possible, would be a lengthy procedure.

Movement

Scrolling and panning of the screen are described in Chapter 8. They are the most economical method of achieving movement. The selected technique can be called, as with picture building, after each new file record has been loaded.

More complex forms of movement can be achieved by having several screens in memory and displaying them in sequence like the 'cells' of film animation, and it is possible to combine scrolling and panning with multiple displays because only the screen data is manipulated - the originals cells continue to be available in memory for re-screening.

AN EXAMPLE PROGRAM

The following example fills the screen with vertical lines and then 'animates' the middle part of the screen. After the lines appear, press any key. When the movement stops press any key to return to your calling program.

```
di                  243
ld a, '2'           62  130        Block 2
out (a), 241        211  241         into 2nd address range
ld hl Start         33  0  64      Start of the bank
ld de Start+1       17  1  64      Next address
ld bc Count         1  0  52       Byte count  (no Roller-Ram)
ld (hl), Byte       54  128        Light one pixel
ldir                237  176       Distribute
ld a, '1'           62  129        Block 1
out (a), 241        211  241
ld hl Start         33  48  89
ld de Start+1       17  49  89
ld bc Count         1  208  38
ld (hl), Byte       54  128
ldir                237  176
ei                  251

ld a, 1             62  1          Restore to TPA
call MEM_M          205  33  253
ld c, 1             14  1          Await key
call BDOS           205  5  0
ld hl 1 1           33  1  1       Initialise count and byte
ld ( ), hl          34  A  A       Storage address
ld bc SUBR9         1  S9  S9      Point to sub-r
call USERF          205  90  252
DEFW                233  0

ld c, 1             14  1          Await key
call BDOS           205  5  0
ret                 201            Return to calling program
```

The sub-routine, which uses the byte value in (A,A) and the flag in (A+1,A) and produces the movement, is as follows:

```
Address  ( S9, S9 )
    ld b, 159          6  159          Number of repetitions

Loop:
    push bc            197             Store
    ld b, 8            6   8           Num of print lines to animate
Loop2:
    push bc            197             Store
    ld hl, Roller      33  208  182    Address of 14th line in Roller
    push hl            229             Store
    ld e, (hl)         94              Half line address
    inc hl             35                into
    ld d, (hl)         86                DE
    ex hl, de          235             Then into HL
    add hl, hl         41                and double it
    ld b, 8            6   8           8 bytes per print line
Loop3:
    push bc            197             Store
    push hl            229             Store line address
    ld b, 90           6   90          Bytes accross screen
    ld a, (A,A)        58  A   A       Get the screen byte
    rlc a              7               Rotate it
    ld (A,A), a        50  A   A       Replace byte in memory
    ld de, 8           17  8   0
Loop4:
    ld (hl), a         119             And poke to screen
    add hl, de         25              Next screen address to right
    djnz Loop4         16  252         And repeat
    pop hl             225             Point to
    inc hl             35                next pixel line down
    pop bc             193             And repeat
    djnz Loop3         16  231           for whole print line
    pop hl             225             Recover address in Roller
    ld de 16           17  16  0         and point to
    add hl de          25                next print line entry
    pop bc             193
    djnz Loop2         16  211         And repeat

    ld hl Byte         33  A   A       Obtain the screen
    ld a, (hl)         126               byte for processing
    rlc a       *      7               Jazz
    rlc a       *      7                 it up
    ld a, b            71              Into B also
    or a               183             If the byte
```



```
    jr z 4             40  4             is not zero
    cp 255             254 255           and not 255
    jr nz 8            32  8             then jump on
    ld a, (A+1,A)      58  A+1  A      Else get the flag into A
    xor 1              238 1             toggle it with '1'
    ld (A+1, A), a     50  A+1  A        and replace it
    ld a (A+1,A)       58  A+1  A      Make sure it is in A
    or a               183             If it is zero
    jr z 5             40  5             jump on
    scf                55              Else set Cy
    rl b               203 16            and add it to B (extra bit)
    jr 2               24  2
    srl b              203 56          If flag is '1', just rotate B
    ld (hl), b         112             Put byte back in store

    pop bc             193             Get byte of repetitions
    djnz Loop          16  171           and repeat if not zero
    ret                201             Else finish
```

The screen byte is stored at (A,A). The sub-r extracts it and rotates it left for each screen filling. When each screen filling is complete, it sets the carry flag and shifts this into the byte to make a thicker line. When it has thickened all the way to 255, it is 'thinned' by shifting a bit out of it.

The flag at (A+1,A) can have a value of '0' or '1'. One value sinals 'rotate left', the other signals 'rotate right'. If the screen byte is found to contain either '0' or '255' the flag is changed to the opposite value by 'xor 1', and the rotation direction reverses.

The two rotations at (*) complicate the movement by sometimes re-inforcing it and at others opposing it. This gives a 'beating' effect.

Chapter 14

Miscellaneous Output

There are a number of small sub-rs that are useful in practically all programs. Frequently a status report needs to be made, a question needs to be asked, or data requested. Inputing strings into programs is also a chore if you are not equipped for it. These miscellaneous but vital functions are covered in this chapter and the next.

Message printing

This technique was referred to in *PCW Machine Code*, but it is worth repeating. Instead of noting the addresses of a whole lot of message strings, it is more convenient to record only their sequence in a list and number them from zero up. If you subsequently change any of them, the alterations to their addresses will not matter because you provide a 'find-and-print' routine that reproduces them when required wherever they are. You put the message number into A and call the routine. The following example can actually handle three different lists that start at (N1,N1) (N2,N2) and (N3,N3) respectively; the appropriate one is selected by the calling address. This is handy when you have three (or more) distinctly different kinds of string.


```
MESPR      Address  ( M,M )

Start 1:
      ld hl N1 N1      33  N1  N1      Start of 1st list
      jr 8             24  8
Start 2:
      ld hl N2 N2      33  N2  N2      Start of 2nd list
      jr 3             24  3
Start 3:
      ld hl N3 N3      33  N3  N3      Start of 3rd list

      or a             183             If (A) = 0  no then
      jr z Print       40  11             search needed
Next:
      push af          245             Save  A
      ld a (hl)        126             Test each
      inc hl           35                 byte for
      cp ENDMKR        254  36            end-marker
      jr nz -6         32  250         Repeat if not one
      pop af           241             Else recover
      dec a            61                 A  and decrement
      jr nz Next       32  245         Loop if not zero
Print:
      ex hl de         235             Else put address to  DE
      ld c 9           14  9              and print
      call BDOS        205  5  0          the string
      ret              201             And finish
```

The list is scanned byte by byte and the message number in A is decremented each time a string end-marker is found. When it reaches zero, HL will contain the address of the message. This is tranferred to DE and the message is printed. (When I first started with m/c, I could not believe that a routine containing so many operations could possibly scan a long list and print a message from the end of it without an embarrassing delay, but I tried it anyway. In fact it does it so fast you can't tell it from normal string printing. That's m/c for you.)

Comparing sequences

There is often a need to test to see if two byte sequences, particularly two strings, are the same. A typical application would be in comparing a password or an identity phrase with the contents of a console buffer that has recently been filled from the keyboard.

To use the following sub-r for this, first load HL and DE with the two start addresses and A with the number of comparisons to be made (the length of string to be compared). If the two sequences do not match then Cy is returned set and HL and DE will be pointing to the first two non-matching bytes. If a complete match is found Cy is returned reset. In both cases A will contain the number of matching pairs detected before return.

```
COMPARE
     ld b, a         71          Count into B
     ld c, a         79            and into C
Loop:
     ld a, (de)      26          Compare pairs of bytes
     cp (hl)         190           in turn
     jr nz 6         32  67      If not the same then jump
     inc hl          35          Else point
     inc de          19            to the next two
     djnz Loop       16  248       and repeat
     ld a, c         121         If count now
     or a            183           zero reset Cy
     ret             201           and finish
Not:
     ld a, c         121         Original count
     sub b           144           minus remainder
     scf             55          Set Cy and
     ret             201           finish
```

Alternatively, and more economically, this routine could have been based on 'cpir', but then it would not have provided the count to the non-matching pair, though this may not matter in most cases.

Listing numbers

As part of my programming library I have collected a set of routines that examine the bytes of my programs and paper-print a list of them and the mnemonics to which they translate, along with jump distances and the Reference Numbers of programs to which jumps or calls are being made. This process of interpreting bytes into the programs that gave rise to them is called **dis-assembly**. My Dis-assembler is a very convenient tool because it not only provides me with a permanent record of the programs, but it also shows up any faults because if the coding is wrong then non-existent or unrelated mnemonics will be printed out to indicate that it is.

There are times when the dis-assembler routines (which in any case are too long to re-produce here) are not available to me, and at others they are not appropriate because the bytes I want to examine are a mixture of data and program that can't be interpreted in the usual way. In these cases I use a much simpler routine that just prints (lists) the numbers onto paper where they can be analysed at leisure. The printed page format is ten 3-digit numbers per print line with three spaces between each. At double line spacing this puts 300 numbers onto a sheet of A4, but you can make that 600 if you print at single spacing.

The program requires a CCB at addresses (c0 to c3,N), the last of which should be zeroed. The third byte should contain '71' for single line spacing, or '72' for double line spacing.

```
      ( c0,N )      String address Lo Byte
      ( c1,N )      ditto Hi Byte
      ( c2,N )      71 or 72
      ( c3,N )      Zero
```

The program also needs a 7-byte variables area consisting of:

```
      ( v0,N )      Count of numbers per line
      ( v1,N )      Lines to print
   ( v2/v3,N )      String address
   ( v4/v5,N )      Address of this number
      ( v6,N )      Duplicate count of nums
```

Before using the routine addresses v0 to v5 should be primed with the required data. Space is required for a 75-byte printer string, the address of which should be put into the first two bytes of the CCB.

The short main routine lists as follows:

```
MAIN
     ld hl v1        33  v1  N     Increment the
     inc (hl)        52              line count
Next:
     ld a, (v0)      58  v0  N     Duplicate number count
     ld (v6), a      50  v6  N       in (v6,N)
     ld hl v1        33  v1  N     After each line
     dec (hl)        53              decrement the line count
     ret z           200           End if zero
```

continued on the next page . . .

```
      call Init        205  I  I      Initialise the print string
Loop:
      call Load        205  L  L      Load a number into string
      ld hl v6         33   v6  N     Decrement the number count
      dec (hl)         53
      jr nz Loop       32   247       And repeat if not zero

      ld de CCB        17   c0  N     Point to CCB
      ld c, 112        14   112         and print
      call BDOS        205  5  0         the string

      jr  Next         24   223       And repeat
```

The main routine prepares the ground for the work done by the sub-routines which are as follows:

```
INIT   Address   ( I,I )
      ld hl, (c0)      42   c0  N     Point to start of string
      ld b, 70         6    70        70 bytes
Loop2:
      ld (hl), 32      54   32        Fill the string
      inc hl           35               with spaces
      djnz Loop2       16   251
      ld (hl), 10      54   10        Terminate
      inc hl           35               with
      ld (hl), 10      54   10          two 'Newlines'
      ret              201
```

'INIT' prepares the string by filling it with spaces and adding the print prompts at the end. 'LOAD' fills the print string as follows:

```
LOAD   Address   ( L,L )
      ld hl, (v4)      42   v4  N     Next source address
      ld a, (hl)       126            Extract the byte
      inc hl           35             Point to next
      ld (v4), hl      34   v4  N       and save new address

      ld (G,G), a      50   X  X      Insert byte for
      xor a            175              interpretation and
      ld (G+1, G), a   50   X+1 X       zeroise high byte
      call Get         205  G  G      Get digits

      ld hl, (v2)      42   v2  N     Address in string
```



```
      push hl          229            Put address
      pop de           209              into DE also
      ld bc 6          1  6  0        Then
      add hl, bc       9                add 6
      ld (v4), hl      34  v2  N        and store new address

      ld hl Result     33  R  R       Copy
      ld bc 3          1  3  0          numerals
      ldir             237  176         into string
      ret              201            And finish
```

'LOAD' takes the next byte address into HL, and extracts the byte into A. HL is then incremented to point to the next byte. The byte is put into memory as the low byte ready to be processed by 'Get Digits', and the high byte is zeroised. The next string address is extracted into both HL and DE, 6 is added to the HL version and this is returned to storage ready for the next insertion. The numerals produced by 'Get Digits' are then transferred into the string.

The sub-r 'Get Digits' is one of universal interest and was described in *PCW Machine Code.* It converts any 8- or 16-bit number into the corresponding numerals ready for display.

Chapter 15

Miscellaneous Input

Is that OK?

This is the most frequent and the most necessary question to be asked of the program user. The string is stored as a message (see page 162) together with the necessary escape sequences for position etc. Lets say its number is 'K'. The string first clears the bottom two lines of the screen, and is made up as:

13 27 89 62 32	Next to bottom line
27 74	Clear bottom two lines
27 89 63 62	Print on bottom line
"Is that OK?"	Text
32 32 32 32	Spaces before reply
36	End-marker

The string is displayed and a key-press is awaited. Only 'yes/no' replies are acceptable. To make keying for non-typists easier the following equivalent keys may be used:

YES	NO
'y' or 'Y' ENTER RETURN	'n' or 'N' SPACE

The assessment of the pressed key is performed as follows:

```
OK
      ld a, K          62  K            Print the
      call MESPR       205  M  M          query
Await
      ld c, 1          14  1            Await
      call BDOS        205  5  0          a key
      res 5, a         203  175         Convert to Capitals
      cp 'N'           254  78          If key is
      jr z  No         40  14           'N', 'n'
      or a             183              Test for 'Space' ( =0 )
      jr z  No         40  10           jump on
Yes
      cp  RETURN       254  13          If key is
      jr z             40  4              ENTER,
      cp 'Y'           254  89            RETURN
      jr nz Await      32  233            'Y' or 'y'
      or a             183              Then reset  Cy
      ret              201              And finish
No
      scf              55               If 'No' then
      ret              201                set  Cy  & finish
```

If any of the 'yes' keys is pressed a return is made with Cy reset. If a 'no' key is pressed Cy is set. Testing Cy on return indicates the choice. Note that resetting bit 5 of A will reduce 32 to zero, so 'Space' is tested for as a zero.

Number input

There can't be many programs that operate without a need to accept numbers from the keyboard. The main routine 'INTDIG' must be able to

a) ignore non-number input
b) take account of digit positions
c) restrict the number of digits to a permitted maximum.

and it uses the sub-r 'NUMACC' to achieve this. NUMACC awaits each key-press in turn and rejects non-numerical input except for 'RETURN' and 'EXIT', but you can adapt it to accept decimal points as well (see the note on deimals at the end of this section). Its truth-table is as follows:

Key	Code	Cy	Z
ENTER/RETURN	13	SET	reset
EXIT	27	reset	SET
Number	48-57	reset	reset

Hence if a return is made with Z set, then 'EXIT' has been pressed and the input should be ignored. A return with Cy set indicates that the input of digits is complete and their value should be calculated. When a number key is pressed its ASCII will be returned in A for processing and both Cy and Z will be reset. Any routine that uses NUMACC will therefore test Cy and Z to assess the value of the input.

```
NUMACC   Address ( A,A )
Start
    ld c, 1         14  1          Await a
    call BDOS       205  5  0        key
    cp ENTER        254  13        If not ENTER/RET
    jr nz 3         32  3            then jump on
    inc a           60             Else reset Z
    scf             55               set Cy
    ret             201              and finish

    cp EXIT         254  27        If not EXIT
    jr nz 2         32  2            then jump on
    xor a           175            Else set Z , reset
    ret             201              Cy & finish

    cp '0'          254  48        If smaller than
    jr c Cancel     56  6            48 then goto Cancel
    cp ':'          254  58        If larger than
    jr nc Cancel    48  2            57 then goto Cancel
    or a            183            Otherwise reset Cy
    ret             201              & Z and finish
Cancel
    ld a, C         62  C          Print the 'cancel'
    call MESPR      205  M  M        message
    jr Start        24  221        And go for next key
```


The 'cancel string' is stored in one of the message lists and consists of:

8 32 8 36

It moves the print position left by one column, prints a SPACE to obliterate the last character, and then moves left again ready for the re-placement.

INTDIG needs a minimum of 11 bytes of variables but we will give it 16 to get a singing-and-dancing version. These are

V/V+1	16-bit result
V+2	Max number of digits
V+3	Digits remaining
V+4/V+5	Addr of next digit
V+6	27
V+7	89
V+8	DEFB Line+32
V+9	DEFB Colm+32
V+10	DEFB Ten-thousands
V+11	DEFB Thousands
V+12	DEFB Hundreds
V+13	DEFB Tens
V+14	DEFB Units
V+15	36 End-marker

Before using INTDIG you should establish the required print position on the screen so that the digits appear where you want them to. The print position specified at (V+8)/(V+9) will apply only when the stored string is re-printed. Naturally it won't affect the screen location of the digits as they are keyed in.

Put the maximum allowed number of digits (up to 5) into V+2. It is necessary to have a facility for limiting the number of digits in order to provide a 'restart' technique, and because more than 5 digits will give unreliable results. In fact results from five-digit numbers larger than 65535 will also be anomalous. Note that mistakes cannot be erased with the DEL keys. When one is made (ie. if a wrong digit is input) then keep typing until the maximum is exceeded because this will provide a restart. Alternatively press 'EXIT' to abort.

The numerical value of the digits is returned in V+0 (low byte) and V+1 (high byte). If you later need to print the number somewhere else, put the screen position into V+8 and V+9 and print from V+6.

```
INTDIG    Address  (I,I)
    ld a, S             62   S                Save the cursor
    call MESPR          205  M    M             position
Start
    ld a, E             62   E                Erase any previous
    call MESPR          205  M    M             input
    ld hl 0             33   0    0           Zeroise the result
    ld (V), hl          34   V    X             and also zeroise
    ld (V+11), hl       34   V+11 X             from units to
    ld (V+13), hl       34   V+13 X             thousands
    ld a, (V+2)         58   V+2  X           Copy the maximum also into
    ld (V+3), a         50   V+3  X             V+3  as a count
    ld hl, V+9          33   V+9  X           Store the 1st digit
    ld (V+4), hl        34   V+4  X             address minus 1
    inc hl              35                    Store an artificial
    ld (hl) '0'         54   48                 'zero' result
Digits
    call NUMACC         205  A    A
    ret z               200                   Finish if 'EXIT' pressed
    jr c Shift          56   23               Jump on if 'ENTER' pressed
    ld hl (V+4)         42   V+4  X           Last address used
    inc hl              35                    Point to next
    ld (V+4)            34   V+4  X             and store
    ld (hl) a           119                   Store ASCII
    ld hl V+3           33   V+3  X           Decrement the
    dec (hl)            53                      count and goto
    jr nz Digits        32   236                next if not zero
Finish
    ld c, 1             14   1                Await
    call BDOS           205  5    0             an extra key
    cp 13               254  13               If this is not ENTER or RET
    jr nz Start         32   195                start from scratch
Shift
    ld a, (V+14)        58   V+14 X           Units into A
    or a                183                   If not a zero go
    jr nz Calc          32   18                 to Calculate
    ld bc, 4            1    4    0           Else shift 4 bytes
    ld de, V+14         17   V+14 X           Move all
    ld hl, V+13         33   V+13 X             digits to a
    lddr                237  184                higher address
    ld a, 48            62   48               And insert a '0'
    ld (V+10) a         50   V+10 X             in Ten-thousands
    jr Shift            24   232              Try again
```

continued on next page ....


```
Calculate
    ld hl, 0            33   0    0           HL will hold the result
Ten-thousands
    ld a, (V+10)        58   V+10 X           Ten-thou ASCII in A
    ld de, 10000        17   16   39
    call EVAL           205  E    E           Evaluate ten-thous
Thousands
    ld a,, (V+11)       58   V+11 X           Thous ASCII etc
    ld de 1000          17   232  3
    call EVAL           205  E    E
Hundreds
    ld a, (V+12)        58   V+12 X           Hundreds ASCII etc
    ld de, 100          17   100  0
    call EVAL           205  E    E
Tens
    ld a, (V+13)        58   V+13 X           Tens ASCII etc
    ld de, 10           17   10   0
    call EVAL           205  E    E
Units
    ld a, (V+14)        58   V+14 X           Units ASCII into A
    sub 48              214  48                 convert to number
    ld e, a             95                      and
    ld d, 0             38   0                  add
    add hl, de          25                      to HL
Store & finish
    ld (V) , hl         34   V    X           Store the result
    push hl             229                   Save
    ld a R              62   R                Print
    call MESPR          205  M    M             'restore' cursor
    pop hl              225                   Recover HL
    xor a               175                   Reset Cy
    inc a               60                      and Z
    ret                 201                   Finish
```

The contribution of each digit is calculated and added into HL by the sub-r 'EVAL', which lists as follows. On entry DE contains the digit value 10,000 1000 or whatever), and A contains the ASCII of the digit.

```
EVAL    Address  (E,E)
    sub 48              214  48               ASCII to decimal in A
    or a                183                   If zero
    ret z               200                     then finish
    add hl de           25                    Repeatedly add digit value
    dec a               61                      into HL until
    jr nz               32   252                until A is zero
    ret                 201                   Then finish
```

INTDIG first saves the cursor position in case it is needed again, and then the sequence of operations is:

Start
Any previous input is erased in case a restart is being made. The erase string is a message made up of

27 107	Restore cursor position and save again
27 106	
27 75	Erase to end of line
36	End marker

The result is zeroised, as are the digits of the units, the tens, the hundreds and the thousands. The maximum number of digits is copied into V+3 to act as a decrementing count, and the address minus one of the storage place of the first digit is put into V+4. An artificial zero is then put into the ten-thousands in case no further digits will be typed in, ie. in case 'RETURN' is the first key pressed.

Digits
The digits are taken in from NUMACC and stored in the address contained in V+4, which is incremented each time. The count in V+3 is also decremented. If NUMACC returns Z set then the program terminates also with Z set. If it returns Cy set then no more numbers are required and a jump to 'Shift' is made.

Finish
If the count in V+3 reaches zero then an extra key-press is required. If this is anything except 'RETURN' or 'ENTER' (which signal that the user is happy with his input) then a restart is made. If it is either one then the program continues in 'Shift'.

Shift
Although anything up to 5 digits may be permitted, less than the maximum may actually have been typed in. It is therefore necessary to shift them up in memory until V+14 (the units) contains a byte other than zero. At each shift a '0' numeral (ASCII 48) is put into V+10 (ten-thousands). When the necessary amount of shifting has been done, they will be in their proper places for printing and for calculating from.

Calculate
The total of the contributions of each of the digits is to be added into HL by EVAL, which is called with DE containing the digit rank

(ten, a hundred, etc), and with A containing the ASCII of the digit. 48 is subtracted from the ASCII to convert it into a number and the rank is added into the total (A) times. For the units, 48 is subtracted from A, and this is added to total.

Store
The result is stored at V (but is also available in HL) and Cy and Z are reset to indicate a successful completion. The saved cursor position is discarded by the message 'Restore' (= 27 107 36).

If you want to accept decimals as legal input, then make the digit store one byte larger to accommodate the decimal point, and don't count in units, count in the lowest rank that will be used. So if you need to deal down to hundredths, for example, then count in hundredths. Then the result '255' would actually mean '2.55'. NUMACC will need to recognise "." as legal and it should be inserted into the digit store. The program also needs to recognise that inputs of either "555.00", "555.0", "555." or "555" all mean 555.00 and this can be achieved by changing the shifting procedure. Instead of shifting until the units contain something other than zero, shift until the decimal point is in its correct place and put '48' in place of any zeroes after (in higher memory than) the decimal point. Before the shifting starts, the program must check through the digits. If it doesn't find a "." amongst them, then it must insert one itself as the last digit and then shift as before.

Inputing strings

Assemblers usually make it possible to insert strings into memory even if you don't know the ASCII codes; it is enough to enter 'String Mode' and press the alphanumeric keys relating. For anyone who hasn't got this facility a great deal of time and trouble will be saved by using the following home-grown equivalent.

It is available whenever I am compiling and I (note) **jump to** it by pressing capital 'X'. When summoned it asks you for the address at which the string is to be inserted, which you enter as two decimal bytes, each followed by 'ENTER', low byte first. After confirming that the address is correct (SPACE='no', ENTER='yes'), you type in the string.

For letters and numerals etc., you simply press the keys. For numerical input (the numbers used escape-sequences, for example,) you

signal "this is going to be a number" by pressing the boxed '+' (to the left of the space bar) to start the number, and 'ENTER' when you have completed it. When the string is complete you press 'ENTER'.

The routine can be placed anywhere in memory, at the address 'Addr' say. It needs five bytes of variables storage starting at (V,N) into which go the following data.

```
V/V+1    Start address of string
V+2      Max allowed bytes
V+3      Flag   1=digit, 0=ASCII
V+4      Count of bytes
```

Note that you need to put the required number in at V+2 before calling the routine, but none of the other variables need priming. I normally keep the max input at 50, a nice round number. The reason for having a maximum is that the program will insert the bytes into memory as you type them, so if you forget and go berserk you may overwrite something.

The page string at (P,P) is as follows:

```
27 69  13 27 89 35 32 27 114
"WRITE A STRING"
27 117 10 10 13
"Start addr bytes:"
27 75 9 9 36
```

The input of the second byte is placed by a position string at (B,B):

```
13  27 89 38 68 36
```

The inputed text is placed by a position string at (T,T):

```
13 10 10 32 32
"String:"
9 9 36
```

We also need a string at (L,L) to move to the next line down to print any numbers typed in:

```
13 10 36
```

Because there is more than one occasion of checking to see if the maximum number of bytes has been exceeded, this operation is given to a

sub-routine at (M,M):

```
MAX
      ld hl, V+2        33  V+2  N     If max number
      ld a, (V+4)       58  V+4  N       not reached
      sub (hl)          150              then
      ret nz            192              proceed

      pop hl            225              Else cancel this
      ld e, 7           30  7              'call' and
      ld c, 2           14  2              beep
      call BDOS         205  5  0
      ld de, TEXT       17  T  T         Start text
      ld c, 9           14  9              again
      call BDOS         205  5  0
      ret               201              And return to main
```

The main program lists as follows:

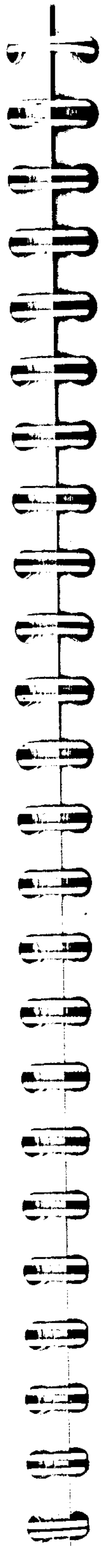
```
MAIN  Address (A,A)
      call MAIN         205  A+6  A    Call this prog
      jp MENU           195  N  N        and return
Program
      ld hl, 0          33  0  0       Zero flag and
      ld (V+3), hl      34  V+3  N       byte count
      ld a, 3           63  3          Set to 3
      ld (Digits), a    50  D  D         digits max
      ld de, PAGE       17  P  P       Page string
      ld c, 9           14  9            print
      call BDOS         205  5  0        it

Accept address
      call INTDIG       205  I  I      Interpret 3 digits
      ld a, (RESULT)    58  R  R       Take low byte
      ld (V) a          50  V  X         and store
      ld de ,BB         17  B  B       Set posn for
      ld c, 9           14  9            2nd
      call BDOS         205  5  0        byte
      ld e, ","         30  44         Print ","
      ld c, 2           14  2
      call BDOS         205  5  0
```

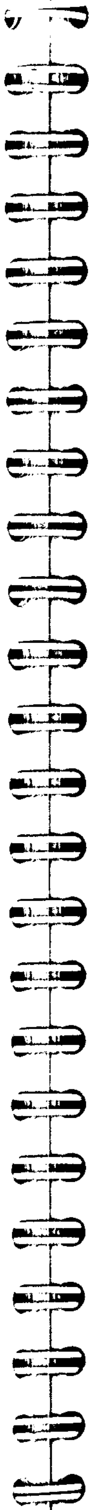
continued on next page . . . .

```
     call INTDIG       205  I  I       Interpret 3 digits
     ld a, (RESULT)    58   R  R       Take high byte
     ld (V+1), a       50   V+1  X     and store

     call OK?          205  K  K       If not OK
      jr c PROG        56   200          repeat

WRITE STRING
     ld de, TT         17   T  T       Position
     ld c, 9           14   9            string
     call BDOS         205  5  0         for text
Next
     ld c, 1           14   1          Await a
     call BDOS         205  5  0         key
     cp ENTER          254  13         If 'ENTER'
     ret z             200               finish
     ld hl, V+4        33   V+4  X     Increment count
     inc (hl)          52                of characters
     cp '+'            254  22         If boxed +
     jr z NUMS         40   18           goto nums
Aciis
     ld hl, (V)        42   V  X       Else put ASCII
     ld (hl), a        119               into memory
     inc hl            35                and point to
     ld (V), hl        34   V  X         next address
     ld (hl), 0        54   0          Zeroise it
     call MAX          205  M  M       If too many, end
     jr Next           24   225        Else next char
Numbers
     ld hl, V+3        33   V+3  X     Test flag, if last
     bit 0 (hl)        203  70           was also a number
     jr nz 8           32   8            then jump on
     ld de LL          17   L  L       Else print on
     ld c, 9           14   9            next line
     call BDOS         205  5  0         down
     ld hl, V+3        33   V+3  X     Set the flag
     ld (hl), 1        54   1            to 'nums'
     call INTDIG       205  I  I       Accept value
     ld hl, (V)        42   V  X       Get mem addr
     ld a, (RESULT)    58   R  R         and insert
     ld (hl), a        119               this value
     inc hl            35              Point to
     ld (V), hl        34   V  X         next mem
```

concluded on next page . . . .


```
     ld de, LL         17   L  L       Print on
     ld c, 9           14   9            on next
     call BDOS         205  5  0         line down
     call MAX          205  M  M       Finish if too many
     jr Next           24   178        Else next
```





APPENDICES

EQUIVALENT ADDRESSES

The following table gives the equivalent in hex and decimal of the 'red-biro' addresses used in the book. They are in ascending order.

(x,y)	Hex	Decimal
(0,0)	0000	0
(1,0)	0001	1
(5,0)	0005	5
(64,0)	0040	64
(128,0)	0080	128
(131,0)	0083	131
(191,0)	00BF	191
(194,0)	00C2	194
(197,0)	00C5	197
(200,0)	00C8	200
(218,0)	00DA	218
(221,0)	00DD	221
(224,0)	00E0	224
(227,0)	00E3	227
(230,0)	00E6	230
(233,0)	00E9	233
(0,1)	0100	256
(158,2)	0293	670
(64,30)	1E40	7744
(240,37)	25F0	9712
(232,38)	26E8	9960
(118,40)	2876	10358
(152,44)	2C98	11416
(48,49)	3130	12592
(0,64)	4000	16384
(48,89)	5930	22784
(104,94)	5E68	24168
(106,95)	5F6A	24426
(94,96)	6068	24670
(92,103)	675C	26460
(94,104)	685E	26718
(179,104)	68B3	26803
(0,128)	8000	32768
(47,179)	B329	45871
(0,182)	B600	46592
(255,183)	B7FF	47103
(240,191)	BFF0	49136
(0,192)	C000	49152
(6,192)	C006	49158
(10,192)	C00A	49162
(20,192)	C014	49172
(35,192)	C023	49187
(94,192)	C05E	49246
(114,192)	C072	49266
(127,192)	C07F	49279
(120,192)	C078	49272
(150,193)	C196	49558
(2,194)	C202	49666

(x,y)	Hex	Decimal
(0,200)	C800	51200
(100,200)	C864	51300
(140,200)	C88C	51340
(160,200)	C8A0	51360
(170,200)	C8AA	51370
(172,200)	C8AC	51372
(180,200)	C8B4	51380
(120,226)	E278	57976
(130,226)	E282	57986
(132,226)	E284	57988
(177,226)	E2B1	58033
(69,227)	E345	58181
(235,227)	E3EB	58347
(0,228)	E400	58368
(110,228)	E46E	58478
(128,242)	F280	62080
(0,246)	F600	62976
(6,246)	F606	62982
(74,246)	F64A	63050
(244,251)	FBF4	64500
(248,251)	FBF8	64504
(3,252)	FD03	64515
(90,252)	FD5A	64602
(33,253)	FD21	64801
(45,253)	FD2D	64813
(111,254)	FE6F	65135
(119,254)	FE77	65143
(167,254)	FEA7	65191
(255,255)	FFFF	65535

PRINTER DRAFT FONT DATA

1. LABEL INTERPRETATION

The following list shows the bytes to which the labels translate. They are given in the order in which they appear in memory. (The Translation Table contains only the bytes on the right not the label numbers, which are shown here for ease of reference.)

Label	Byte	Label	Byte	Label	Byte
0	0	30	96	60	168
1	64	31	24	61	160
2	1	32	126	62	156
3	4	33	82	63	154
4	65	34	80	64	136
5	32	35	31	65	121
6	34	36	6	66	112
7	8	37	122	67	102
8	127	38	81	68	100
9	68	39	70	69	95
10	20	40	62	70	89
11	2	41	41	71	86
12	16	42	3	72	83
13	73	43	254	73	76
14	84	44	192	74	74
15	28	45	125	75	67
16	9	46	97	76	61
17	128	47	88	77	52
18	56	48	69	78	50
19	120	49	63	79	38
20	66	50	48	80	36
21	40	51	33	81	30
22	5	52	22	82	25
23	85	53	21	83	15
24	124	54	17	84	10
25	72	55	12		
26	54	56	7		
27	42	57	250		
28	60	58	204		
29	18	59	188		

2. BYTE LOCATION

The following list shows the bytes that appear in the Translation Table, but this time in ascending order of their own magnitude. The label with each byte shows where the byte is to be found. If you want to know if a byte appears in the table look for it in this list.

Byte	Label	Byte	Label	Byte	Label
0	0	41	41	89	70
1	2	42	27	95	69
2	11	48	50	96	30
3	42	50	78	97	46
4	3	52	77	100	68
5	22	54	26	102	67
6	36	56	18	112	66
7	56	60	28	120	19
8	7	61	76	121	65
9	16	62	40	122	37
10	84	63	49	124	24
12	55	64	1	125	45
15	83	65	4	126	32
16	12	66	20	127	8
17	54	67	75	128	17
18	29	68	9	136	64
20	10	69	48	154	63
21	53	70	39	156	62
22	52	72	25	160	61
24	31	73	25	168	60
25	82	74	74	188	59
28	15	76	73	192	44
30	81	80	34	204	58
31	35	81	38	250	57
32	5	82	33	254	43
33	51	83	72		
34	6	84	14		
36	80	85	23		
38	79	86	71		
40	21	88	47		

3. OFFSETS

The following are the (data packed) offsets from (92,103) at which the set of labels for each character is to be found. The number of labels in a character is given by subtracting one (stripped) offset from the next.

Specials:

ASC	Offset	ASC	Offset	ASC	Offset	ASC	Offset
0	(87,1)	8	(130,17)	16	(186,1)	24	(242,1)
1	(93,1)	9	(135,1)	17	(190,1)	25	(250,1)
2	(102,1)	10	(143,1)	18	(198,1)	26	(2,2)
3	(108,1)	11	(151,1)	19	(204,1)	27	(10,2)
4	(114,33)	12	(157,1)	20	(213,1)	28	(18,2)
5	(117,33)	13	(162,1)	21	(221,1)	29	(25,2)
6	(121,1)	14	(171,1)	22	(230,33)	30	(32,2)
7	(129,65)	15	(179,129)	23	(233,1)	31	(41,2)

Standard set:

ASC	Char	Offset
32	Space	(46,2)
33	!	(46,66)
34	”	(47,34)
35	#	(50,2)
36	$	(55,2)
37	%	(62,2)
38	&	(71,2)
39	’	(79,66)
40	(	(80,34)
41	)	(83,50)
42	*	(86,2)
43	+	(91,2)
44	,	(96,178)
45	-	(98,2)
46	.	(100,66)
47	/	(101,18)
48	0	(108,2)
49	1	(114,34)
50	2	(116,2)
51	3	(121,2)
52	4	(126,2)
53	5	(133,2)
54	6	(142,2)
55	7	(151,2)
56	8	(159,2)
57	9	(163,2)
58	:	(172,2)
59	;	(173,178)
60	<	(175,18)
61	=	(179,2)
62	>	(181,18)
63	?	(185,18)
64	@	(190,2)
65	A	(197,2)
66	B	(206,2)
67	C	(210,2)
68	D	(215,2)
69	E	(220,2)
70	F	(224,2)
71	G	(228,2)
72	H	(234,2)
73	I	(238,34)
74	J	(241,2)
75	K	(246,2)
76	L	(251,2)
77	M	(254,2)
78	N	(5,3)
79	O	(12,3)

ASC	Char	Offset
80	P	(16,3)
81	Q	(20,3)
82	R	(27,3)
83	S	(33,3)
84	T	(37,3)
85	U	(42,3)
86	V	(48,3)
87	W	(57,3)
88	X	(66,19)
89	Y	(73,3)
90	Z	(82,3)
91	[	(87,35)
92	\	(90,19)
93	]	(97,35)
94	Up	(100,35)
95	_	(103,3)
96	acc	(105,51)
97	a	(107,3)
98	b	(112,3)
99	c	(116,19)
100	d	(119,3)
101	e	(123,3)
102	f	(132,19)
103	g	(137,131)
104	h	(146,3)
105	i	(150,35)
106	j	(153,19)
107	k	(158,3)
108	l	(162,35)
109	m	(165,3)
110	n	(170,3)
111	o	(175,3)
112	p	(179,131)
113	q	(183,131)
114	r	(187,19)
115	s	(192,3)
116	t	(196,19)
117	u	(202,3)
118	v	(207,3)
119	w	(216,3)
120	x	(225,3)
121	y	(230,131)
122	z	(239,3)
123	{	(244,35)
124	bar	(248,67)
125	}	(249,35)
126	~	(253,3)
127	0	(3,4)
Table end		(12,4)

END

PRINTER NLQ FONT DATA

This lists the NLQ label translations in the order in which they appear in memory. (The Translation Table contains only the pairs of bytes on the right not the label numbers, which are shown here for ease of reference.)

1. LABEL INTERPRETATION

Lbl.	pass: 2nd	pass: 1st	Lbl.	pass: 2nd	pass: 1st	Lbl.	pass: 2nd	pass: 1st	Lbl.	pass: 2nd	pass: 1st
0	0	0	30	2	2	60	1	2	90	3	2
1	0	65	31	16	64	61	0	3	91	2	1
2	0	64	32	8	32	62	60	124	92	28	29
3	0	68	33	2	0	63	128	34	93	16	36
4	0	1	34	16	32	64	152	2	94	16	33
5	0	8	35	24	16	65	2	64	95	0	42
6	0	4	36	4	1	66	33	8	96	9	32
7	0	34	37	0	5	67	32	8	97	35	6
8	8	0	38	128	0	68	0	33	98	36	2
9	1	0	39	1	64	69	16	1	99	32	5
10	0	4	40	12	8	70	60	120	100	34	2
11	0	20	41	0	9	71	128	2	101	33	2
12	0	84	42	63	62	72	60	60	102	18	8
13	0	73	43	56	56	73	33	65	103	8	17
14	33	0	44	0	85	74	0	80	104	22	1
15	32	0	45	4	80	75	24	56	105	5	16
16	0	32	46	48	32	76	0	70	106	18	2
17	18	0	47	0	72	77	4	65	107	4	16
18	63	127	48	8	16	78	0	66	108	16	4
19	0	2	49	12	0	79	1	65	109	1	17
20	4	64	50	16	65	80	64	0	110	1	12
21	0	16	51	0	81	81	40	16	111	4	9
22	36	0	52	32	32	82	36	16	112	4	8
23	16	0	53	36	1	83	41	0	113	9	2
24	0	40	54	32	4	84	37	0	114	10	1
25	0	128	55	32	1	85	24	0	115	8	1
26	0	69	56	0	17	86	20	0	116	2	6
27	32	64	57	8	8	87	4	12	117	2	5
28	12	28	58	8	4	88	5	2	118	1	4
29	16	16	59	4	4	89	2	4	119	5	0
									121	1	3
									122	1	1

2. BYTE LOCATION

The following list shows the bytes that appear in the Translation Table, but this time in ascending order of their own magnitude. The label with each shows where the byte is to be found. If you want to know if a byte appears in the table look for it in this list.

Byte	1st pass label No	2nd pass label No
0	0 8 9 14 15 17 22 23 33 38 49 80 83 84 85 86 119	0 1 2 3 4 5 6 7 10 11 12 13 16 19 21 24 25 26 30 37 41 44 47 51 56 61 68 74 76 78 95
1	4 36 53 55 69 91 104 114 115 121	9 39 60 79 109 110 118 120 121
2	19 30 60 71 88 90 98 100 101 106 113	33 64 65 89 91 116 117
3	61 120	90
4	6 10 54 58 59 89 108 118	20 36 45 59 77 87 107 111 112
5	37 99 117	88 105 119
6	97 116	
8	5 40 57 66 67 102 112	8 32 48 57 58 103 115
9	41 111	96 113
10		114
12	87 110	28 40 49
16	21 29 35 48 81 82 105 107	23 29 31 34 50 69 93 94 108
17	56 103 109	
18		17 102 106
20	11	86
22		104
24		75 35 85
28	28	92

Byte	1st pass label No	2nd pass label No
29		92
32	16 32 34 46 52 96	15 27 52 54 55 67 99
33	68 94	14 66 73 101
34	7 63	100
35		97
36	93	22 53 82 98
37		84
40	24	81
41		83
42	95	
48		46
56	43 75	43
60	72	62 70 72
62	42	
63		18 42
64	2 20 27 31 39 65	80
65	1 50 73 77 79	
66	78	
68	3	
69	26	
70	76	
72	47	
73	13	

Byte	1st pass label No	2nd pass label No
80	45 74	
81	51	
84	12	
85	44	
120	70	
124	62	
127	18	
128	25	38 63 71
152		64

3. OFFSETS

The following are the (data packed) offsets from (104,94) at which the set of labels for each character is to be found. The number of labels in a character is given by subtracting one (stripped) offset from the next.

Specials:

ASCII	Offset	ASCII	Offset	ASCII	Offset	ASCII	Offset
0	(246,1)	8	(92,2)	16	(253,2)	24	(162,3)
1	(9,2)	9	(117,2)	17	(15,3)	25	(176,19)
2	(24,2)	10	(142,2)	18	(37,3)	26	(186,3)
3	(38,2)	11	(164,2)	19	(62,3)	27	(209,3)
4	(51,66)	12	(180,2)	20	(85,3)	28	(219,3)
5	(59,98)	13	(196,2)	21	(112,3)	29	(232,3)
6	(63,2)	14	(217,2)	22	(135,35)	30	(247,3)
7	(83,82)	15	(234,130)	23	(141,3)	31	(6,4)

Standard Set:

ASCII	Character	Offset
32	Space	(27,4)
33	!	(27,84)
34	"	(32,20)
35	#	(48,4)
36	$	(79,4)
37	%	(96,4)
38	&	(123,4)
39	'	(155,84)
40	(	(162,116)
41	)	(168,68)
42	*	(174,20)
43	+	(187,20)
44	,	(192,84)
45	-	(201,4)
46	.	(205,84)
47	/	(210,36)
48	0	(223,36)
49	1	(233,52)
50	2	(239,4)
51	3	(4,5)
52	4	(23,5)
53	5	(42,5)
54	6	(53,5)
55	7	(62,5)
56	8	(77,5)
57	9	(94,5)
58	:	(107,85)
59	;	(112,85)
60	<	(121,37)
61	=	(128,5)
62	>	(132,37)
63	?	(139,5)
64	@	(152,5)
65	A	(178, 5)
66	B	(202,5)
67	C	(215,5)
68	D	(224,5)
69	E	(233,5)
70	F	(244,5)
71	G	(1,6)
72	H	(20,6)
73	I	(34,70)
74	J	(39,6)
75	K	(50,6)
76	L	(68,6)
77	M	(77,6)
78	N	(92,6)
79	O	(108,6)
80	P	(120,6)
81	Q	(129,6)
82	R	(149,6)
83	S	(169,6)
84	T	(184,6)
85	U	(203,6)
86	V	(221,6)
87	W	(237,6)
88	X	(0,7)
89	Y	(19,7)
90	Z	(38,7)
91	[	(62,39)
92	\	(65,39)
93	]	(78,39)
94	Up	(81,71)
95	_	(86,7)
96	acc	(90,103)
97	a	(93,7)
98	b	(110,7)
99	c	(119,7)
100	d	(126,7)
101	e	(136,7)
102	f	(144,23)
103	g	(157,135)
104	h	(178,7)
105	i	(189,39)
106	j	(198,7)
107	k	(213,7)
108	l	(233,39)
109	m	(238,7)
110	n	(255,7)
111	o	(14,24)
112	p	(20,136)
113	q	(34,136)
114	r	(48,8)
115	s	(57,8)
116	t	(67,8)
117	u	(82,8)
118	v	(92,8)
119	w	(109,8)
120	x	(128,8)
121	y	(143,136)
122	z	(174,8)
123	{	(199,104)
124	bar	(208,104)
125	}	(209,40)
126	~	(218,40)
127	0	(227,8)
Table end		(244,8)

END

SWITCHING MEMORY BANKS

This corrected and updated account of block and bank switching is taken more or less verbatim from the 3rd edition of *PCW Machine Code*. Readers of earlier editions may be interested in the corrections.

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 No.*

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 '8256/512') is as follows:

The Memory Manager

```
FD21  push hl          229           Save HL
      ld (FEA0h), a    50 160 254    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  jump 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 (see the note in Interrupts), 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. 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.
(Scrn Envment	128	129	130	0	1	2	7	)

Note that only one block is changed for Bank Nos larger than 1. The Screen Environment cannot be accessed through the Memory Manager, but I include a list of its blocks for completeness.

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; 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 (240), a      211 240
```

When doing any block switching by use of 'out' instructions, it is necessary to disable the interrupts first, and to re-enable them later.

```
di                243           Disable the interrupts
ld a, BLOCK       62  N2        Select block No
out (R), a        211 R         Switch it in
...               ...           Your
...               ...             procedure
ld a, BLOCK       62  N1        Restore the
out (R), a        211 R           original 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. (Also see page 10.)

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

MEMORY MAP OF PRESSED KEYS

Whilst a key is being pressed this will be indicated by the fact that a particular bit in block 3 will be set. Sixteen addresses are involved starting at (240,191), and finishing at (255,191). For ease of location, the table below lists the keys on the top line of the keyboard, followed by those on the second line, etc. The entries are all in the form of two numbers; the first is the offset counting from (240,191), and the second is the bit number in that address. Hence the pressing of the 'STOP' key will set bit No 2 of address (248,191). The bit is reset when the key is released (but see the note on CAPS LOCK later).

Some keys also set bits of offsets 14 & 15 as well as their 'individual' bit; these I have marked with " * ". f1 & f2 also set bits of offset 12. The SPACE bar sets no less than five bits (bit 4 of offsets 12, 14, and 15, and bit 5 of offset 13) !

Top line

Key	Offset Bit	Key	Offset Bit	Key	Offset Bit	Key	Offset Bit
STOP	8 2	1	8 0	2	8 1	3	7 1
4	7 0	5	6 1	6	6 0	7	5 1
8	5 0	9	4 1	0	4 0	-	3 1
=	3 0	DEL→	2 0	DEL←	9 7	CAN	10 2
CUT	1 2	COPY	1 3	PASTE	0 3		

2nd Line

Key	Offset Bit	Key	Offset Bit	Key	Offset Bit	Key	Offset Bit
TAB	8 4	*q	8 3	*w	7 3	*e	7 2
*r	6 2	t	6 3	y	5 3	u	5 2
i	4 3	*o	4 2	*p	3 3	*[	3 2
*]	2 1	RETURN	2 2	f7	10 4	FIND	2 4
PAGE	1 4	PARA	0 4				

3rd Line

Key	Offset Bit	Key	Offset Bit	Key	Offset Bit	Key	Offset Bit
C-LOCK	8 6	*a	8 5	*s	7 4	*d	7 5
*f	6 5	*g	6 4	*h	5 4	*j	5 5
k	4 5	l	4 4	;	3 5	§	3 4
*#	2 3	RETURN	2 2	f5	10 0	EOL	0 5
*↑	1 6	CHAR	0 5				

4th Line

Key	Offset Bit	Key	Offset Bit	Key	Offset Bit	Key	Offset Bit
*SHIFT	2 5	*z	8 7	*x	7 7	*c	7 6
*v	6 7	*b	6 6	*n	5 6	*m	4 6
,	4 7	.	3 7	/	3 6	1/2	2 6
*SHIFT	2 5	f3	0 0 and 12 0	*←	1 7	*SCREEN	0 7
*→	0 6						

Bottom Line

Key	Offset Bit	Key	Offset Bit	Key	Offset Bit	Key	Offset Bit
ALT	10 7	EXTRA	10 1.	[+]	2 7	*SPACE	5 7
[-]	10 3	PTR	1 1	EXIT	1 0	f1	0 2 and 12 1
RELAY	0 1	*↓	10 6	ENTER	10 5		

OTHER SET BITS

The above table indicates which bits are specific to which particular keys, but other bits also are set by various key combinations, as follows.

Offset 10: Bit 1 is set for EXTRA, and Bit 7 is set for ALT.
Offset 13: Bit 7 is always set. Other bits may also be set.
Offset 14: Bit 5 is set for SHIFT. Other bits may also be set.
Offset 15: Bits 6 & 7 toggle regularly and are independent of key-presses
Bit 5 is set for SHIFT.

The three 'shift' keys may be detected by:

SHIFT: Bit 5 of offsets 2, 14, & 15.
ALT: Bit 7 of offset 10.
EXTRA: Bit 1 of offset 10.

While it is held down CAPS LOCK sets bit 6 of offset 8, and additionally bit 6 of offset 13 stays set whilst in the CAPS LOCK mode (until the key is pressed again , or the SHIFT key is pressed).

For multiple key presses the combination of set bits is additive.

END.

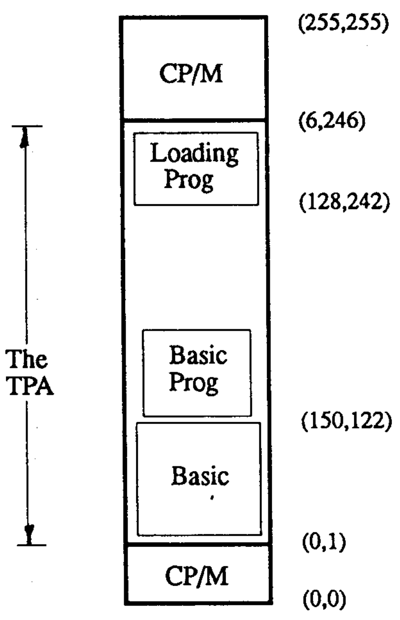
CONTENTS OF MEMORY

The contents of the memory blocks used by CP/M are briefly as follows:

Block 0	Interrupt address & ISR location
	A BIOS jump block
	The 'Set-Up' routines
Block 1	The lower part of the screen pixel data
Block 2	The upper part of the screen pixel data
	The Character Matrix RAM
	Roller RAM
Block 3	BIOS & BDOS routines
	Key-board map
Blocks 4 to 6	Most of the TPA
Block 7	The top of the TPA plus some BIOS & BDOS (Common Memory)
Block 8	Console Command Processor
	Disc Hash Tables
	Parts of BIOS
	Data buffers
	The printer fonts
Blocks 9 & up	The Memory Disc (drive M:)

Bank 1

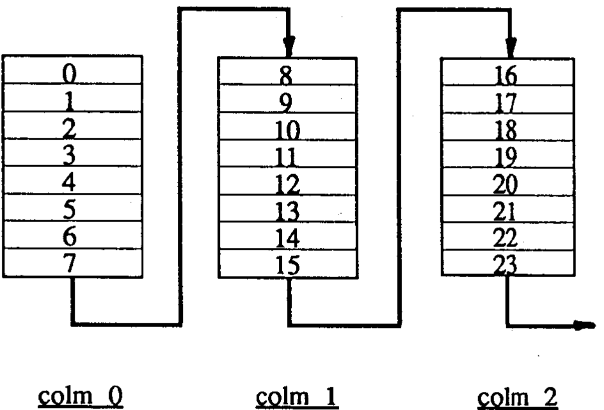
The Addresses within bank 1, which includes the TPA, are as indicated to the right.

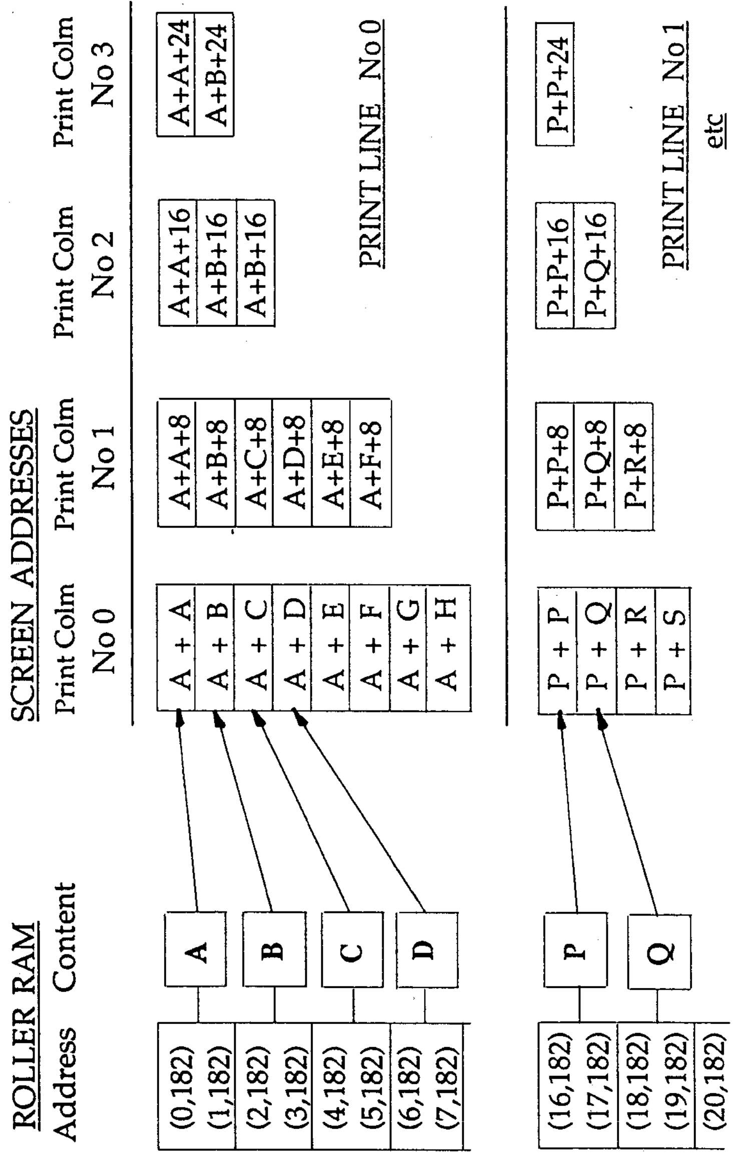


SCREEN ADDRESSING

The connection between screen addresses and the corresponding entries in Roller-Ram is indicated by the diagram on the next page.

The sequence of addresses in a print line is as indicated below:





The contents of Roller-RAM mapped onto the screen

IM 2 PROGRAMMING

Iain Stirzaker has assiduously worked through the pitfalls associated with using the very handy *im* 2 interrupt mode on the PCW, and has come up with the following way of dealing with it.

This example purposely gets itself into an endless loop consisting of repeated jumps back to the printing of the same message. However, by use of *im* 2, it allows escape from this (or any other) loop by pressing the ALT key. On such enterprises it is important to make sure that all (and I mean **all**) of the registers are preserved.

This demonstration program is for insertion at (0,1), as would be the case for a COM file. Its first job is therefore to move its important bits up into Common Memory.

```
0100
F0FF VBASE   EQU #F0FF
     VPAGE   EQU #F0
     BDOS    EQU 5
             ORG #0100

0100         ld de, VECBASE   17 255 240   Transfer
0103         ld hl, START     33 7 3         the programming
0106         ld bc, FINISH-VECBASE
                              1 85 0         to higher
0109         ldir             237 176        memory
```

The preparation

Then it saves all the registers in sight:

```
010B         push af          245          Save
010C         push bc          197            the
010D         push de          213            normal
010E         push hl          229            registers
010F         push ix          221 229
0111         push iy          253 229
0113         ex af, af'       8            And
0114         exx              217            then
0115         push af          245            the
0116         push bc          197            alternates
0117         push de          213
0118         push hl          229
```


and then the stack pointer and replaces it with our home-grown variety, and then saves the address to which BDOS calls are directed.

```
0119         ld ( OLDST), sp  237 115 156 2
011D         ld sp, NEWST     49 154 2
0120         ld hl, (#0006)   42 6 0
0123         ld ( OLDBD), hl  34 159 2
```

Next, BDOS calls are redirected to our 'trap' that will toggle the flag that tests whether BDOS is being used:

```
0126         di               243
0127         ld hl, TRAP      33 136 1
012A         ld (#0006), hl   34 6 0
```

followed by initialisation of *im* 2:

```
012D         ld a, VECPAGE    62 240       High byte of vector
012F         ld i, a          237 71
0131         im 2             237 83
0133         ei               251
```

Now start the endless loop:

```
0134         call LOOPS       205 2 161
```

The Recovery

The routine that handles the return to sanity (and announces the fact) follows on from the above and so is located at (55,1). It consists of the following:

```
0137 SAFE    di               243          Back to im 1
0138         im 1             237 86
013A         ld hl, (OLDBD)   42 159 2     Recover the normal
013D         ld (#0006), hl   34 6 0         BDOS vector
0140         ei               251

0141         ld sp, (OLDST)   237 123 156 2  Original stack
0145         pop hl           225          Recover
0146         pop de           209            all
0147         pop bc           193            the
0148         pop af           241            registers
0149         exx              217
```

```
014A          ex af, af '        8
014B          pop iy             253  225
014D          pop ix             221  225
014F          pop hl             225
0150          pop de             209
0151          pop bc             193
0152          pop af             241

0153          ld de MSG          17  M  M     Print the
0156          ld c, 9            14  9          victory
0158          call BDOS          205  5  0      message
```

With everything nicely under control again, the programming ends with the usual hook for DEVPAC 80:

```
015B          RST 0              199
```

The victory message at address (92,1) consists of:

```
015C MSG      DEFB               13  10  10
015F          DEFM               "We have regained control  -  thanks
                                 to  IM 2 $"
```

The BDOS trap

Because we are not able to come out of *im* 2 if the machine happens to be processing a BDOS call, we need a flag to tell us whether such a call is being dealt with or not. This flag task has been given to address (158,2). If it contains a zero then we are clear to return, but if it contains anything else (which can only be '255' in fact) we are not. As indicated in Chapter 4, the 'halt' instruction terminates the execution of code and awaits an interrupt thereby greatly increasing the chance of the key being detected quickly. The trap is set up by:

```
0188 TRAP     ld a, #FF          62  255      Set flag
018A          ld (FLAG ), a      50  158  2
018D          ld hl, (OLDBD)     42  159  2   Save BDOS jump
0190          call CALLJP        205  153  1    vector
0193          xor a              175          Reset flag to
0194          ld (FLAG), a       50  158  2     signal BDOS end
0197          halt               118          Pause for key
0198          ret                201
0199 CALLJP   jp (hl)            233
```



```
Addresses:
019A  (154,1)                DEFS   256     New stack area
029A  (154,2)   NEWST        DEFW   2       New stack addr
029C  (156,2)   OLDST        DEFW   2       Old stack addr
029E  (158,2)   FLAG         DEFB   0       BDOS Flag
029F  (159,2)   OLDBD        DEFW   #0000   Old BDOS addr
```

The endless loop

At (161,2) is the simple loop that keeps printing the same message. It consists of:

```
02A1 LOOPS    ld c, 9            14  9
02A3          ld de, HAHA        17  171  2
02A6          call BDOS          205  5  0
02A9          jr LOOPS           24  246
```

The message at (171,2) is:

```
02AB HAHA     DEFM               "You are now in an endless loop.
                                 To exit, hold down the ALT key"
              DEFB               13  10  10  36
```

The ISR

The ISR is to be located at (1,241), and it will be pointed to by the Interrupt Vector which is to be conveniently located just in front of it at (255,240). The vector will therefore consist of the address (1,241). The code from (255,240) upwards is the section that is pushed up into high memory by the 'ldir' operation.

```
0307 START    EQU  $
F0FF          ORG  VECBASE
F0FF          DEFW  ISR          1  241       The interrupt vector

F101 ISR      push af            245          Start of the ISR
F102          push bc            197          Save 'em all again
F103          push de            213
F104          push hl            229
F105          push ix            221  229
F107          push iy            253  229
F109          ex af, af '        8
```

```
F10A          exx            217
F10B          push af        245
F10C          push bc        197
F10D          push de        213
F10E          push hl        229

F10F          ld a, (FLAG )  58  158  2    BDOS flag into A
F112          and a          167           If the flag is not zero,
F113          jr nz  ESCP    32  16          do not test for a key

F115          ld a, #83      62  131       Else switch in
F117          out (#F2), a   211  242        block 3
F119          ld hl, N N     33  250  191  And test for
F11C          bit 7 (hl)     203  126        the ALT key
F11E          jp nz  GOTK    194  55  241  If found goto GOTK

F121          ld a, '6'      62  134       If not then restore
F123          out (242), a   211  242        the TPA
F125  ESCP    pop hl         225             and put
F126          pop de         209             all the
F127          pop bc         193             registers
F128          pop af         241             back
F129          exx            217             as
F12A          ex af, af '    8               you
F12B          pop iy         253  225        found
F12D          pop ix         221  225        them
F12F          pop hl         225
F130          pop de         209
F131          pop bc         193
F132          pop af         241
F133          rst  #0038     255      **    Housekeep
F134          ei             251           Re-enable
F135          reti           237  77       Leave the ISR

F137  GOTK    ld a, #86      62  134       As above
F139          out (#F2), a   211  242
F13B          pop hl         ....

      (repeat from ESCP  down to **)

F149          ....           255
F14A          inc sp         51            Remove the im 2
F14B          inc sp         51              return address
F14C          push hl        229           Save HL
```



```
F14D          ld hl  SAFE    33  55  1     Put the safe restart
F150          ex (sp), hl    227             addr onto the stack

F151          ei             251           Re-enable
F152          reti           237  77       Leave the ISR
F154  FINISH  EQU  $
F154          END
```

This last section of code from F14A to F150 merely removes from the stack the usual address to which the ISR would return and replaces it by our chosen restart address, which is (55,1). The 'reti' therefore causes a return to (55,1).

Operation

Everthing is now prepared. The registers have been saved, the ISR, the test for BDOS, and the strings, etc. are in place, and *im 2* is in operation. We therefore get ourselves into an endless loop (and even boast about it !), just to prove we can get out again. The loop is established by the code at (161,2), which just keeps printing the same message; the one that announces that the loop has been formed.

If the ALT key is now pressed and held until it is detected during an interrupt, the interrupt will not return to the program location at which it originated, instead it will be redirected to address (55,1).

Assembly

The program was assembled by I Stirzaker using DEVPAC 80. The decimal bytes were added after transcription for printing to assist direct insertion into memory if required.

END

"PCW Machine Code" *by Mike Keys*

A good primer that sets out the principles and practice of Z80 machine code programming. It contains many program examples that are specific to controlling all aspects of the PCWs, both simple and advanced. Well recommended by the press and by readers.

Published: Spa Associates
Spa Croft, Clifford Road, Boston Spa, LS 23 6 DB

"Streamlined Basic" *by Geoffrey Childs*

An excellent description of how to get the best out of Mallard Basic. Written specifically for PCW Basic programmers but with lots of interesting m/c routines.

Published: PCW World,
Cotswold Ho, Cradley Heath, Warley, BN64 7NF

"The Amstrad CP/M Plus" *by D Powys-Lybbe & A Clarke*

Lots of detail on CP/M and assembly language in the PCWs, but not very readable.

Published: MML Systems Ltd.
11 Sun Street, London, EC2M 2PS

"An Introduction to Z80 Machine Code" *by R & J Penfold*

An excellent booklet listing the Z80 instruction set and the effect of each instruction on the flags. Opcodes in hex.

Published: Bernard Babarni Publishing Ltd.
Shepherds Bush Road, London, W6 7NF

INDEX

A

address
equivalent_ 182
character_ 93
jump-USERF 11
_ranges 196
screen_ 78+, 87, 202
ADDRS 119
animation 152, 159
arrows 135

B

bank
_No 1 map 200
_switching 9, 194+
bars
_mods 55
printer_ 51
_repeated 55
BCD 24
BDOS 69
fnc 1 69, 71, 136
fnc 2 69, 71
fnc 9 69, 72
fnc 10 69, 72, 130
fnc 104 29
fnc 105 24
fnc 110 72
fnc 111 69, 72
eeps 127
BIOS 10
BITADR 88
BLEN 91
blocks
_fitted 13
memory_ 9, 194, 197
books 210
buffer tables 13
buffered
_keying 124
_selection 130+

C

CALCDIG 34
case
_upper/lower 129
character
_addresses 93
_designing 115
_information 56
_inversions 110
_large 111
_making 114
_manipulations 107+
_rotations 108
CHDES 118
clear screen 143
codes
control_ 21, 70
ID 131
comma 60
COMPARE 163
compare
_sequences 163
_strings 163
co-ordinates 87
copyright message 40
CSCAN 140
CURSC 136
cursor
_menus 124, 134+
_on/off 73
_position 14, 129
printed_ 134
_scanning 139
screen data_ 138

D

decimal 8
_points 170, 171
deletes 73
descenders 54
dis-assembly 164
discing screens 148
draft data 184
_fonts 52+
drives
_facilities 22+
_fitted 13

E

'E' shape 61, 67
emphasis 73
equals sign 59, 65
equivalent addresses 182
escape sequences 70
EVAL 173
expansion token 19+

F

FADDR 86
FASC 93
files 148
FILL 34
FIND 56
FOLIO 133
fonts
_bytes 184+
draft_ 51+, 184
_data 46, 184
NLQ 64, 188
redesigning_ 62, 67, 68
Locoscript_ 68
frills 123

H

halt 32, 48
HANDL 85
help screens 122
housekeeping 37

I

im 0, 1, 2 38
im 1 39+
im 2 45+
INIT 166
in-line parameter 11
input 168+
number_ 169
_strings 175
instructions
user_ 122
INTDIG 172
interface fitted 13
interrupts 37+, 204+
_address 39
_modes 38
_register 45
_timer 42
_vector 45
interval timing 32
inverse video 137
inversions
character_ 110
screen_ 105
ISR 39, 49
exit_ 48

J

jump USERF 11
jump table 10

K

key
_buffered 124, 130
_detect 15
_memory map 16, 198
_number 15
_put 16
_release 18
set_ 18
keyboard
_input 71
Keying
_buffered 124, 130
_non-buffered 126
_routines 15+

L

labels
changing_ 58
_data 184, 185
font_ 53, 63, 65
redesigned 63, 67, 68
larger than 59, 66

LARGE 112
LINE 154
LINE2 156
listing numbers 164
Load 166
loading screens 142+
CP/M_ 144

M

MABLK 95
machine
_config 13
_type 13
making chars 114
manuals 8
MAX 177
MCHAR 114
memory 9, 200, 202
_banks/blocks 10
contents of_ 200
_disc 145, 197
_key map 198
_manager 9, 194
menu 121+
_cancellation 125
_keypress 124, 126+
_cursor 124, 134+
_single key 127
_types 124
MESPR 163
message printing 162
modular pattern 11, 84
motor on/off 23
movement 152, 158

N

NLQ
_data 188
_fonts 604
NNLQ 67
NUMACC 170
number
decimal_ 175
_input 169
_listing 164

O

offsets
list of_ 186, 192
font_ 52, 64
_mods to 54
redesigned_ 62
OK 169

P

panning 102
PANLEFT 103
PANRT 104
parameter
inline_ 11, 12
passwords 132, 163
pauses 30+
_timing 31
PCHK 92
picture building 152, 156
pixel 80
polite 122
ports 143, 151, 194+
print
_position 70
printing
_columns 100
_customised 96, 124
_from table 97
_lines 99
_messages 162
sequence_ 98
printer operation 51
program technique 28, 92
proportional spacing 62, 68
PRLIN 100
PRSEQU 99
PRTABL 97
PROOF 90

R

red-biro 8
repeat
_font bars 56
_speed 18

REVBAR 138
Roller-Ram 81+, 151+, 203
RRAM 151
RRCHAR 108
rotations 107+

S

safe areas 39
SBARS 144
SCINV 105
screen
_addresses 85, 202
artificial_ 150
_blocks 94+
_clearing 143
_co-ordinates 87
_data 14, 143, 146
_environment 80, 196
_hatching 143
_inversion 105
loading_ 142+
_manipulations 94+
_map 80, 146, 203
_on/off 143
_panning 102
pre-composed_ 145
_printing 69+
_re-establishing 147
reset 14
routines 14+,
_saving 146, 148
_scrolling 101
size of_ 14
_speckling 157
Screen Run Routine 14, 84
scrolling 101
SCRUP 101
set
_expand 19
_key 18
_time 29
set-up 9+
SHOW 58
speckling 157
SRES 152
SSINV 110
stack 10, 79

status line 83
_ask/set 14
STORE 147
string printing 72
syntax 8

T

TBINV 110
text control 70
time 24+
TIMER 34
timing
precise_ 42
TOD block 25
tokens 19+
_table 20
TPA 10, 78, 200
truth table 170

U

underlining 73
USERF 10, 93

V

version number 13

W

WCLS 143

Z

z80 7