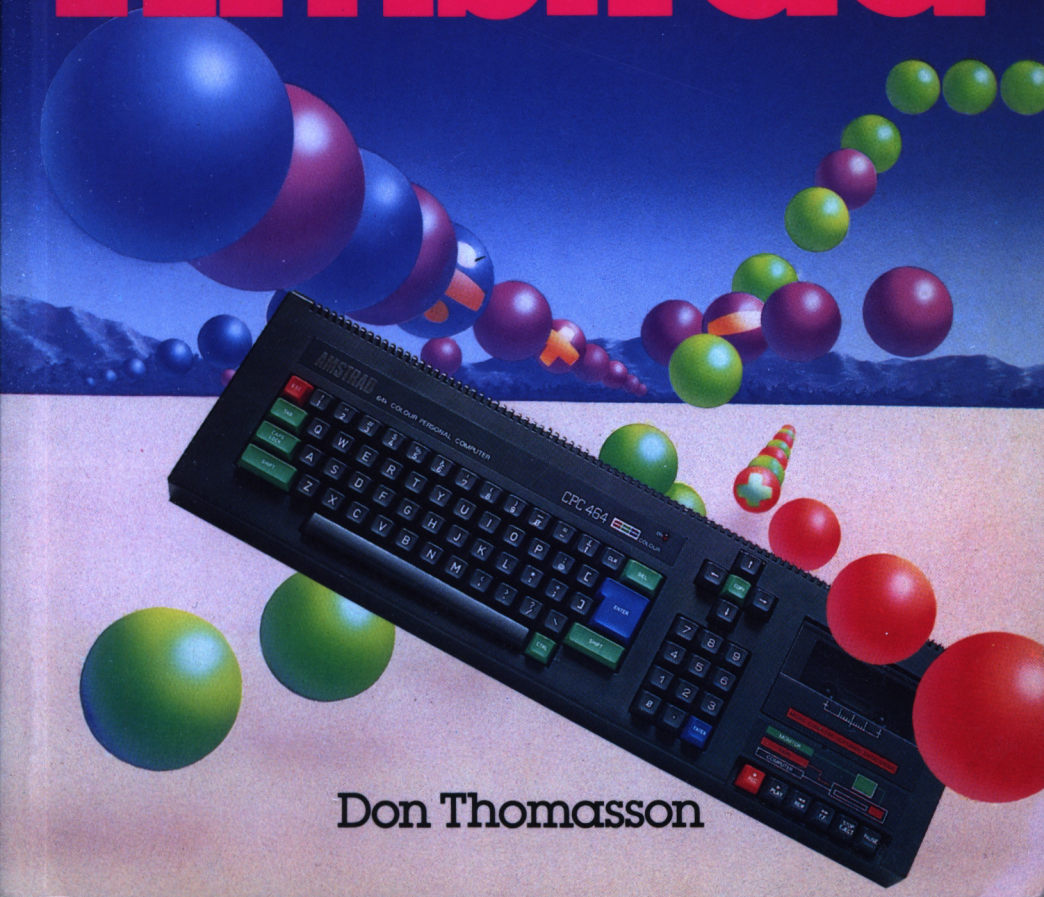
Amstrad
CPC 464

Melbourne House

The Ins & Outs of the Amstrad



Don Thomasson

The Ins & Outs of the Amstrad

CPC 464

The Ins & Outs of the Amstrad

CPC 464

Don Thomasson

MELBOURNE HOUSE
PUBLISHERS

IN THE UNITED KINGDOM —
Melbourne House (Publishers) Ltd
Castle Yard House
Castle Yard
Richmond, TW10 6TF

IN THE UNITED STATES OF AMERICA —
Melbourne House Software Inc.
347 Reedwood Drive
Nashville TN 37217

IN AUSTRALIA —
Melbourne House (Australia) Pty Ltd
2nd Floor, 70 Park Street
South Melbourne, Victoria 3205

ISBN 0 86161 190 X

Edition: 7 6 5 4 3 2 1
Printing: F E D C B A 9 8 7 6 5 4 3 2 1
Year: 90 89 88 87 86 85 84

CONTENTS

INTRODUCTION

The greater part of an iceberg lies hidden, accessible only to those who have suitable equipment. Much the same could be said of the capabilities of the AMSTRAD CPC464. Superficially it may appear to be just another games-playing machine. However there is a lot more to it than that. As with the iceberg, special equipment is needed to explore the system capabilities in full, and the object of this book is to provide such equipment.

Even at first glance, it is clear that the CPC464 is a little out of the ordinary. That it has a built-in cassette recorder is by no means unique, although the provision of a monochrome or colour monitor, incorporating the system power supply, is a welcome change, and other features are useful, if less obviously so.

For those who confine themselves to BASIC, the friendliness of the machine will be sufficient to recommend it, but for the 'hacker', the user who wants to delve deeper into the mysteries of the system, there is much to be explored.

Here we are mainly concerned with the way in which the CPC464 will work with external equipment. To deal with that effectively, we must first study and understand the internal system, for that determines how add-on bits and pieces can be controlled.

The book is therefore divided into two main parts, the 'Ins' dealing with the internal system, and the 'Outs' dealing with external additions.

It is inevitable that frequent reference will have to be made to machine code routines. Those who need additional data on this subject will find **Programming the Z80** a useful aid. It is listed in the Bibliography at the end of the book.

It should be said that the book is based on the cassette system version of the CPC464, and some points may be modified in the disc system version.

The Ins

THE INS

System Overview

The general arrangement of the internal hardware of the CPC464 is shown in Fig. 1. Note the directional arrows indicating the flow of data or control information. They are important.

The Z80 sits at the centre of the system, and it controls — directly or indirectly — everything that the system does. It communicates on a two-way basis with 64K of RAM, and on a read-only basis with 32K of ROM, which is addressed in two 16K blocks. Selection of ROM or RAM, at an address where both exist, is controlled by signals from the Video Gate Array.

Through its I/O interface, the CPU communicates directly with the CRT Controller, which works with the Video Gate Array to maintain the display; with the PPI (Parallel Peripheral Interface); and with the Printer Port. This is the primary peripheral area.

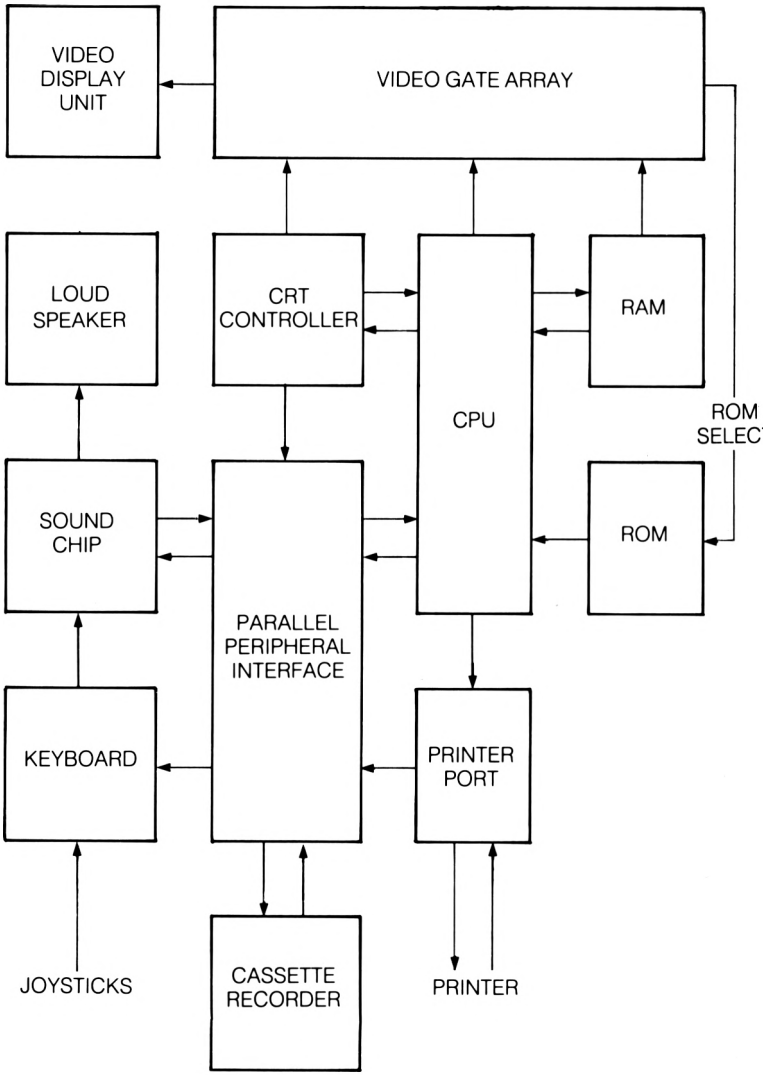


FIGURE 1: INTERNAL HARDWARE SYSTEM

Outside that, in system terms, lies the secondary peripheral area. This has no direct contact with the CPU, but communicates with the primary peripheral area. The elements in this outer area include the cassette recorder, which communicates with the PPI, the keyboard, and the sound chip. The latter contains a pair of data ports, one of which is used to communicate with the keyboard output.

The Z80 CPU is driven by a 4 MHz clock, but in order to interlace the memory accesses for the processor and display system it is necessary to delay processor accesses, effectively reducing the clock to 3.3 MHz.

The overall system is remarkably economical, in hardware terms, and you may wonder how it can compete in performance with systems that have three times as many components. The answer lies largely in the system firmware, which is more thoroughly organised than usual, and which displays some interesting ingenuities. True, the bigger system can do things that the CPC464 cannot, but many of these can be added as externals, which means that you only pay for the ones you want. Many users will see such facilities as unnecessary frills, while wanting features beyond the scope of either machine.

The system programs are divided into nine groups:

a. **Key Manager**

   Scans the keyboard, generates characters, implements the function key expansions, tests for break, scans joysticks.

b. **Text VDU**

   Puts characters on the screen, handles the cursor, and responds to control codes.

c. **Graphics VDU**

   Plots and tests points, draws lines.

d. **Screen Pack**

   Interfaces b. and c. to the screen hardware and executes common screen functions.

e. **Sound Manager**

Deals with sound generation.

f. **Cassette Manager**

Deals with cassette functions.

g. **Kernel**

The heart of the operating system, handling interrupts, events, ROM selection and program execution.

h. **Machine Pack**

Handles printer and general hardware control.

i. **Jumper**

Controls access to routines via jumpblocks.

This covers the operating system, mainly held in the lower ROM. The BASIC interpreter, if used, is held in the upper ROM, but can be replaced by alternative programs in external ROMs.

Access to the system routines should be via the nominated 'jumpblocks', which are tables of jumps giving access to the various entry points. These tables will remain unaltered in any future version of the system, but the direct access points may change.

The Memory System

Fig 2 shows the memory map in outline, but there is a lot of fine detail which cannot be shown in that way. (Note that all memory addresses are given in hexadecimal form.)

The overlap of ROM and RAM in the top and bottom quarters of the map is a salient feature. Since bank-switching schemes have run into problems in the past, the implications of this must be examined carefully.

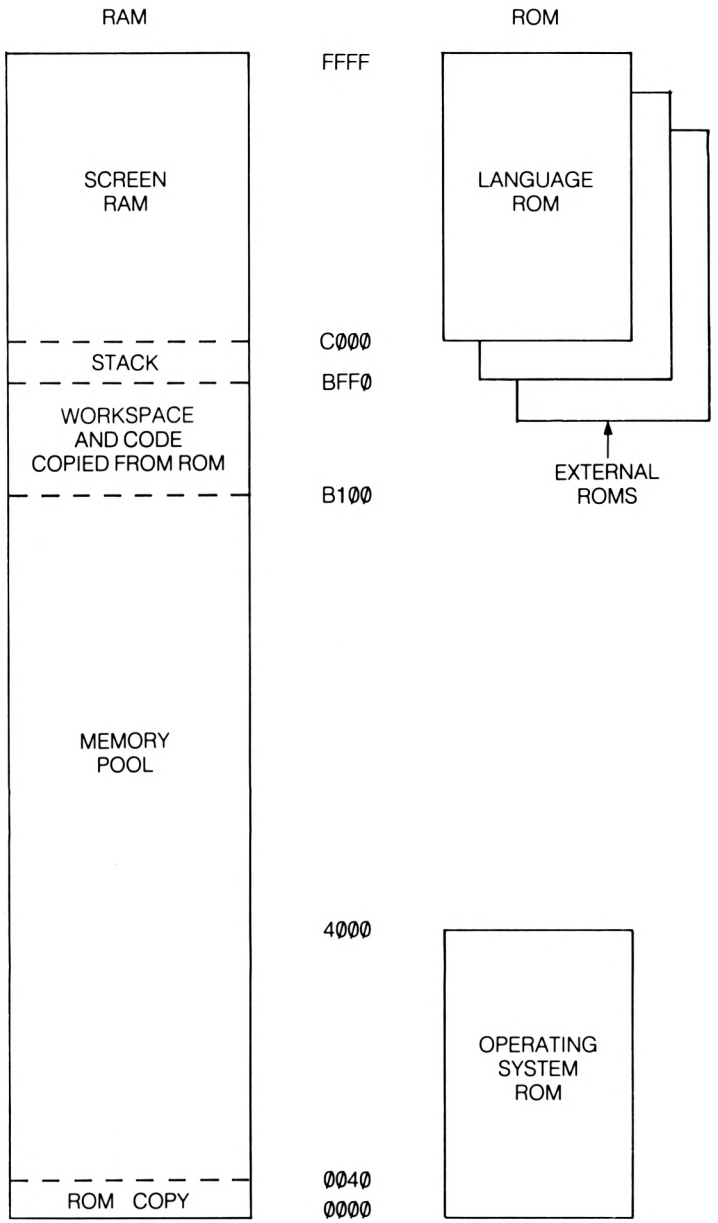


FIGURE 2: OUTLINE MEMORY MAP

All memory writes go to RAM. It's no use writing to ROM. It doesn't answer . . .

If a ROM is disabled, a memory read draws on the RAM in that area. Either or both of the ROM blocks can be enabled independently, in which case they supply the data for appropriate read addresses.

The two ROM areas are in fact implemented by a single component, a 32K ROM, the address lines of which are fiddled as follows:

If address lines A14 and A15 are both false, address bit A14 to the ROM is false, and the lower ROM (ØØØØ-3FFF) is addressed if it is enabled.

If address lines A14 and A15 are both true, address bit A14 to the ROM is true, and the upper ROM (CØØØ-FFFF) is addressed if it is enabled.

If the states of address lines A14 and A15 differ, ROM is inactive.

The ROM enable lines come from the Video Gate Array, which must be able to gain access to the screen memory between CPU memory accesses. The relevant data is output to the Video Gate Array directly, in bits 2 (lower ROM) and 3 (upper ROM) of a control word. However, it will be seen that direct access is both unwise and unnecessary. The other bits of the control word have important meanings, and must be maintained in the correct state.

The RAM is implemented by eight 64K × 1 components.

Turning to fine detail of memory use, the ØØØØ-ØØ3F area holds the same data in ROM and RAM, the latter copied from ROM during the initialisation process. This is necessary as the Z80 RST instructions jump into this area, and must find code available, whether the lower ROM is enabled or not. A number of other short routines are packed into the same region, and they must also be available in both ROM and RAM.

We will see later that certain of the RAM locations in this area can be changed by the user for particular purposes, but this must be done with care, adhering to strict rules.

From ØØ4Ø upwards, ROM and RAM diverge, the ROM containing operating system routines, while the RAM holds workspace data. BASIC sets its stored program from Ø17Ø upwards, for example.

The cental area of RAM is known as the Memory Pool. Only that part from 4ØØØ to CØØØ can be guaranteed to be available for immediate access. It would be foolish, for example, to position the machine stack outside that area. It normally sits very comfortably from BFFF downwards.

The upper part of the central area contains system variables and more program that must always be accessible. In the extreme, the block from A4ØØ up to the stack may be occupied, if SYMBOL AFTER Ø has been called to copy all character patterns to RAM. If the 'soft' patterns remain at the standard level (From &FØ to &FF), AB4Ø is the first location occupied.

Much of the area above AB4Ø is taken up by 'jumpblocks', which provide standard entries to many important routines. The entry addresses to the routines may change, but the jumpblock allocations will remain unaltered. This is important: It is wise to make use of the jumpblock path, even when the actual access point is known, because programs will then work with later marks of operating system.

From CØØØ upwards, the RAM serves the screen, and ROM implements the BASIC interpreter (Though some interpreter routines are in the lower ROM). External ROMs may replace the BASIC ROM, providing other 'language' implementations.

This rapid tour of the memory should have revealed the general pattern. No doubt it has encouraged thoughts of a more

detailed examination, in which case you may feel that the following program would serve your purpose:

```
100 CLS
110 INPUT "Start Address"; A
120 A=A-65536*(A<0)
130 N%=A-8*INT(A/8)
140 PRINT HEX$(A,4);
150 B=PEEK(A)
160 PRINT TAB(6+3*N%);HEX$(B,2)
170 A = A + 1
180 N%=N%+1
190 IF N%<8 THEN 150
200 N%=0
210 PRINT
220 GOTO 140
```

Line 120 is needed because input of a hexadecimal address above 7FFF makes A negative, and in that event 65536 must be added to give a positive value, or subsequent calculations will give trouble. The variable N% puts the dumped values in the correct column of the display, and starts a new line as appropriate.

But the results of running this program are disappointing. Acres of zeroes are seen, with a limited amount of code here and there. We are looking at RAM, because PEEK and POKE only access RAM, ignoring ROM. Oh, dear! We need a different approach.

Scanning the bulky Firmware Manual, we discover the following calls, which will solve our problem:

B900: Enable upper ROM

B903: Disable upper ROM

B906: Enable lower ROM

B909: Disable lower ROM

For these four calls there are no entry conditions, but the previous ROM status is returned in the A register. Flags are cor-

rupt, but other registers are unaltered. The value returned in A can be used to restore the Previous ROM state by use of:

B90C: Restore ROM state defined by contents of A register. The return is with AF corrupt, but other registers are unaltered.

Conscience compels an admission that the statement that other registers are unaltered is not entirely correct. Alternative register C' is changed, because it holds the updated state of the data sent to the Video Gate Array. However, you are required not to use the alternative registers, since they hold vital information that may be needed at any moment.

Leave the program above set up, and overwrite it with the following entries:

```
 90 GOSUB 400
150 GOSUB 300
300 Q=INT(A/256)
310 POKE &7019,Q
320 POKE &7018,(A-256*Q)
330 CALL'&7000
340 B=PEEK(&7020)
350 RETURN
400 FOR X=&7000 TO &7012
410 READ Y
420 POKE X,Y
430 NEXT
440 RETURN
450 DATA &2A,&18,&70,&CD,&00,&B9,&F5
460 DATA &CD,&06,&B9,&7E,&32,&20,&70
470 DATA &F1,&CD,&0C,&B9,&C9
```

With these changes, the program will dump ROM, upper and lower. To see why, we must consider the machine code routine set up by lines 400-470:

```
7000  2A 18 70  LD    HL, (7018)   ;Get byte address
7003  CD 00 B9  CALL  B900         ;Enable upper ROM
7006  F5        PUSH  AF           ;Save previous ROM
                                    state
7007  CD 06 B9  CALL  B906         ;Enable lower ROM
```

```
700A 7E          LD    A,(HL)      ;Read byte
700B 32 20 70    LD    (7020),A    ;Store byte
700E F1          POP   AF          ;Recover previous
                                   ROM state
700F CD 0C B9    CALL  B90C        ;Restore previous
                                   ROM state
7012 C9          RET               ;Return to BASIC
```

Lines 300-320 in BASIC set the value of A in 7018/9, lower byte first, as usual. The subroutine in machine code is called, and the value of A is picked up in HL. B900 is called to enable the upper ROM, and the value it returns in the A register is pushed on to the stack out of harms way. B906 enables the lower ROM, and then the byte pointed to by HL is read into the A register and thence into 7020. The previous ROM state is recovered from the stack, and a call to B90C restores the ROMs to their previous state.

Back in BASIC, PEEK(&7020) recovers the byte, and the program then proceeds as the original did.

Simple though this machine code is, purists may object that parameters could be passed by an extension of BASIC line 300, but that is scarcely worth while for such a short routine. The number of parameters (in this case one, the variable A) would be given in the A register, and register IX would point to the parameter value, which would have to be transferred to HL, converting it from floating point form in the process. The facility for passing parameters is useful, but only when it is fully justified.

Putting that aside, you should now have a reasonably clear picture of the memory system in your mind. To make good use of the knowledge you will need to know other things, but even a tyro may be tempted to experiment. There is plenty of room . . .

Talking of room, it is interesting to consider the statement that 42K of memory is available to the user. In hexadecimal terms, this is A800, so the claim is not exaggerated. It would almost be true to say 43K . . . However, the exact usage of workspace are not too clear.

The Inner Peripherals

The I/O Address Map

Address	Output	Input
ØØXX to 7EXX	Do not use	Do not use
7FXX	Video Gate Array	Do not use
8ØXX to BBXX	Do not use	Do not use
BCXX	CRTC Register Select	Do not use
BDXX	CRTC Data	Do not use
BEXX	Do not use	Reserved (CRTC Status)
BFXX	Do not use	CRTC Data
CØXX to DEXX	Do not use	Do not use
DFXX	Expansion ROM Select	Do not use
EØXX to EEXX	Do not use	Do not use
EFXX	Printer Latch	Do not use
FØXX to F3XX	Do not use	Do not use
F4XX	PPI Port A Data	PPI Port A Data
F5XX	PPI Port B Data	PPI Port B Data
F6XX	PPI Port C Data	PPI Port C Data
F7XX	PPI Control	Undefined
F8XX	Expansion Bus	Expansion Bus
F9XX	Expansion Bus	Expansion Bus
FAXX	Expansion Bus	Expansion Bus
FBXX	Expansion Bus	Expansion Bus
FCXX to FEXX	Do not use	Do not use
FFXX	Not used	Not used

The arrangement of I/O addresses, summarised in the table given here, appears even more complex than the memory map, but the hardware decoding is relatively simple.

*If address bit A15 is low, the Video Gate Array is selected.

*If address bit A14 is low, the CRT Controller is selected.

*If address bit A13 is low, the expansion ROM number must be set.

*If address bit A12 is low, the printer latch is selected.

*If address bit A11 is low, the PPI is selected.

*If address bit A1Ø is low, an expansion channel is implied.

Not more than one of bits A1Ø to A15 may be low in a given address, which accounts for the large number of 'do not use' restrictions. It is especially important that this rule is observed for inputs, since physical damage could otherwise occur.

In the case of the CRT Controller and the PPI, bits A8 and A9 select a particular function of the device.

Since all the address bits used as above are in the upper byte of the address, the Z80 instruction OUT (N),A must not be used. It defines the upper address byte from the contents of the A register, which must also hold data. IN A,(N) might be used, if the required upper byte is first defined in the A register, but its use is not encouraged.

The correct form of I/O instruction is OUT (C),r or IN r,(C), where r is an eight-bit register. In this case, the I/O address is defined by the contents of the BC register. The 'repeating' instructions INIR, INDR, OTIR, OTDR must not be used, as they employ the B register as a counter, and would generate changing addresses.

There is a further limitation for user peripherals, arising from the way the lower byte of the address is used.

*If address bit 7 is low, the disc system is selected.

*If address bit 6 is low, a reserved function is selected.

*If address bit 5 is low, a communications channel is selected.

For user peripherals, all these bits must be high, but a low byte of &FF also has a special meaning, F8FF calling for all expansion devices to be reset. The addresses available to the user are thus:

F8EØ — F8FE

F9EØ — F9FF

FAEØ — FAFF

FBEØ — FBFF

This provides for free use of bits Ø, 1, 2, 3, 8 and 9, giving plenty of scope, providing simplified decoding is not used. If all these bits, plus bit 1Ø low, are taken into account, 64 I/O channels

can be defined. If each bit were related to a particular channel, no more than six channels could be used.

It is worth re-emphasising that the use of illegal I/O addresses could cause damage in input operations, since more than one data source could attempt to drive the highway.

The Video Gate Array

The Video Gate Array, on address 7FXX, has three types of data, distinguished by the state of bits 6 and 7 of the data. The bit allocations are as follows:

7 6 5 4 3 2 1 Ø
1 Ø X X X X X X

Mode & ROM setting.

MODE Control: ØØ Mode Ø
Ø1 Mode 1
1Ø Mode 2
11 Illegal

Lower ROM: Ø enable
1 disable

Upper ROM: Ø enable
1 disable

Clear raster 52 divider if 1

Reserved. Must be Ø.

7 6 5 4 3 2 1 Ø
Ø Ø Ø X X X X X

Palette Pointer Register.

Palette pointer number.

7 6 5 4 3 2 1 Ø
Ø 1 Ø X X X X X

Palette Memory.

7 6 5 4 3 2 1 Ø
1 1 X X X X X X

Colour number

Reserved.

For mode and ROM setting, it is essential to supply all the active bits in the correct state, and this calls for a copy of the last

output byte to be maintained. Such a copy is held in alternative register C' of the Z80. However, reference to that copy is automatic when the standard jumpblock entries are used to make changes, so there should be no need to refer directly to the register contents.

For colour setting, the palette pointer determines the ink to be set, and the output to the palette memory determines the colour, which is thereafter used to interpret data read from screen memory.

The main function of the Video Gate Array is to obtain data from the screen memory, on the basis of addresses supplied by the CRT Controller, and rearrange it into a suitable form for transmission to the Video Display Unit.

The need for user access to the Video Gate Array is limited, since the system takes care of all necessary actions, but there could be special circumstances in which extremely fast response was needed that could only be implemented by direct access.

The CRT Controller

The HD6845S CRT Controller is a standard component of considerable complexity, and it can only be described here in relatively superficial terms. It can be set up to control screen displays of varying sizes and characteristics, generating the screen reference addresses, the sync pulses, and putting up a synthesised cursor character on demand.

The action of the chip is controlled by the contents of a number of registers, eighteen in all, of which the present system accesses sixteen. To set up these registers, it is necessary to output the relevant register number on address BCXX, then the data on BDXX. The standard settings are as follows:

R0 **Horizontal Total:** 63

This determines the horizontal scan period as 64 character periods, which are in turn determined by the clock frequency, in this case 8 MHz, giving one character width per microsecond.

R1 **Horizontal Displayed:** **4Ø**

This determines that 4Ø characters per line will be displayed, the flyback blanking occupying 24 character periods.

R2 **H Sync Position:** **46**

The horizontal sync pulse is generated 6 character periods after the end of displayed characters.

R3 **Sync Width:** **142**

This is a compound number, determining the width of the vertical and horizontal sync pulses. The vertical sync pulse is given as having a duration of 8 scan lines, while the horizontal sync pulse has a duration of 14 character periods. (8*16+14=142)

R4 **Vertical Total:** **U.K. 38. U.S.A. 31**

The total number of character lines (minus 1) has to be different for 5Ø and 6Ø Hz scans. Each character line uses 8 scan lines, so the precise figures would be 39.Ø625 and 32.8125, remembering that only one frame is used, not two in interlace. The adjustment necessary is made by R5.

R5 **Vertical Total Adjust:** **U.K. Ø. U.S.A. 6**

This adds six scan lines to the vertical total in the 60 Hz case. The 50 Hz total is 39, while the 60 Hz total is 32.75. These are acceptably close for practical purposes.

R6 **Vertical Displayed:** **25**

Twenty-five character lines are specified.

R7 **Vertical Sync Position:** **U.K. 3Ø. U.S.A. 27**

Here again, there are two different standards. For 5Ø Hz, the vertical sync begins 5 character line periods after the displayed area, whereas only two character lines delay can be allowed in the 6Ø Hz case. In the 5Ø Hz case the vertical sync ends after 38 scan lines,

while in the 60 Hz case it ends at the full 32 character line count, and is cut from 8 to 7 character line periods.

R8 **Interlace Mode and Skew:** 0

A zero entry calls for no interlace, no skew.

R9 **Maximum Scan Line:** 7

This determines the number of scan lines per character line, less one. There will be eight scan lines per character.

R10 **Cursor Start:** 0

No cursor is generated, because:

R11 **Cursor End:** 0

The cursor synthesised by the chip has zero height.

R12 **Start Address (H):** 48

R13 **Start Address (L):** 0

This determines the start address for screen memory scan as 48 * 256 = 12288 (3000 hex). The address is modified by the Video Gate Array.

R14 **Cursor Register (H):** 192
R15 **Cursor Register (L):** 0

This sets up the cursor address as &C000.

It need scarcely be said that these register values are so closely interlinked that any change will destroy the display. Any changes need to be carefully worked out in advance, taking into account the fact that the Video Gate Array will modify the results in some instances. For Mode 2, only one bit is needed to define a pixel, only one byte is needed to define a row of a character pattern. Eighty bytes must be read to define a screen line. For Mode 1, twice as many bytes are needed, but there are half as many characters per line, so the total number of bytes read per scan line is still 80, though they are processed in a different way. This can be repeated for Mode 0.

The addresses provided by the CRT Controller therefore have to be processed before use. Only 40 addresses are provided by the CRTC per scan line, but 80 bytes have to be read. The necessary conversion is simple enough, but needs to be remembered by anyone indulging in screen adventures. In general, such activities are best forgotten. The CRT Controller can usually be regarded as a fixed-performance device, though changes to the start address may have uses.

One function remains to be mentioned, and that concerns the Light Pen. This uses a pin connected to the expansion interface. A low to high transition on this pin causes the current screen address to be copied into R16 (high byte) and R17 (low byte) whence it may be read. The correct interpretation of this address is best obtained by experiment, owing to the action of the Video Gate Array in modifying addresses. The following outline of the screen RAM layout will assist in this respect.

In all modes, the screen memory occupies 16K of RAM, normally from C000 upwards. By use of the call SCR SET BASE the screen can be moved to 4000 upwards, but other positions are unacceptable.

The screen memory is divided into eight 2K blocks. Each scan line of the display uses 80 consecutive bytes, but the first scan line draws data from block 0, the second draws data from block 1, and so on. Hence the first character in the top left corner is defined by bytes at addresses C000, C800, D000, D800, E000, E800, F000, F800, each giving one row of the character pattern or, in Modes 0 and 1, part of the information for such a row. Mode 2 needs only one byte to define a character row, Mode 1 needs two, and Mode 0 needs four.

For Mode 2, the pixels in a character pattern row are defined by the bits of the byte read from screen memory.

For Mode 1, the pixels from left to right are determined by bits 3 and 7, 2 and 6, 1 and 5, 0 and 4, and then the same bits of the next byte continue the pattern row.

For Mode 0, the first pixel is defined by bits 1, 5, 3 and 7 of the first byte, the next by bits 0, 4, 2 and 6, then by the corresponding bits of the three subsequent bytes.

None of this need worry you in the ordinary way, since the firmware takes care of all the necessary conversions and interpretations, but it will at least explain the odd byte patterns which you get if you dump screen RAM.

As a final complication, an offset is used, controllable by the call SCR SET OFFSET (See the tabulation of operating system calls given later). Changing the offset by &5Ø rolls the screen up or down by one line. Now, this is an area where a great deal of experiment is justified, but be wary. Remember that the rules for each mode are different, so there will be different consequences if you make small changes in the offset. Mode 2 will respond to unit changes, but Mode 1 must have changes which are a multiple of two, and Mode Ø requires a multiple of four.

Approached with suitable caution, the screen offers a number of interesting possibilities, but rash fiddling will create chaos. Look through the list of operating system calls, and use them after careful thought.

The Parallel Peripheral Interface

The Parallel Peripheral Interface is another standard component. It implements three ports, which can be set up for input or output. Each port has its own address, and the control register, which determines port action, has a fourth address.

In the CPC464, the standard configuration is:

Port A: Input or output as necessary. Bidirectional communication with the data lines of the sound generator chip. Output is used to set up sound: Input senses keyboard output.

Port B: Input only, with the following allocations:

Bit Ø: Frame flyback pulse
Bit 1: Ø if link LK1 fitted These links determine
Bit 2: Ø if link LK2 fitted the name which the
Bit 3: Ø if link LK3 fitted CPC464 gives itself!
Bit 4: Ø if link LK4 fitted (6Ø Hz frame scan)
Bit 5: Ø if expansion port active
Bit 6: 1 if printer busy
Bit 7: Cassette data

Port C: Output only, with the following allocations:

Bit Ø: Keyboard input Ø
Bit 1: Keyboard input 1
Bit 2: Keyboard input 2
Bit 3: Keyboard input 3
Bit 4: Cassette motor control
Bit 5: Cassette data
Bit 6: BC1 to sound chip
Bit 7: BDIR to sound chip

These allocations are essential, but as a matter of interest the chip configuration is set up by sending a control word to I/O address F7XX. This word has two formats, one setting up the chip mode, the other setting or resetting bits of Port C, of no great interest in this context.

The format for mode setting is:

```
7 6 5 4 3 2 1 Ø
1 X X X X X X X

Port C low
Port B
Mode for above
Port C high
Port A
Mode for above
```

Mode Ø is basic input/output, mode 1 is strobed input/output, mode 2 is bi-directional bus. If a port bit is set to Ø, the port works in output mode, if the bit is set to 1 the port works in input mode. The byte sent by the CPC464 control system is &82, setting mode Ø, Port B input, Ports A and C output. (Port C is implemented in two halves, which can be set up independently.)

Being deeply embedded in the hardware system, the PPI offers little scope for experimentation or change.

$\overline{\text{STROBE}}$	1 19	GROUND
DØ	2 2Ø	GROUND
D1	3 21	GROUND
D2	4 22	GROUND
D3	5 23	GROUND
D4	6 24	GROUND
D5	7 25	GROUND
D6	8 26	GROUND
D7	9 27	GROUND
NC	1Ø 28	GROUND
BUSY	11 29	NC
NC	12 3Ø	NC
NC	13 31	NC
GROUND	14 32	NC
NC	15 33	GROUND
GROUND	16 34	NC
NC	17 35	NC

Note: A 34-way connector is used, but the numbering is adapted to suit the 36-way printer connector, on which pins 18 and 36 are unused.

FIGURE 3: PRINTER CONNECTOR

The Printer Port

The 'Centronics Latch', addressed by EFXX, is quite simple, in fact a little too simple for some users, since it carries only seven data bits, plus a strobe. There are a number of printers which will find their performance limited, though only by a loss of the block graphics of the type associated with Tandy systems. The upper control codes, beginning at &8Ø, are also lost, but while that could be an embarrassment on some machines, due to conflict of the lower control codes with VDU codes, the separation of VDU and printer data into different streams makes the loss less important.

Note, however, that bit 7 of any codes in the upper half of the ASCII range will be lost, which may on occasion produce unexpected results.

If an attempt is made to drive the printer directly, the following rules must be applied:

The data bits must be firm at least 1 μS before the start of the strobe pulse, and must remain firm until at least 1 μS after the end of the strobe pulse.

The duration of the strobe pulse must be 1 to 5ØØ μS.

The strobe pulse must not be transmitted when the Busy signal from the printer is true. This bit can be sensed by reading bit 6 of the input to Port B of the PPI.

The need to consider these rules can be avoided by using facilities provided by the operating system:

CALL BDF1: MC WAIT PRINTER

The character code to be sent is in the A register on entry. If the character was sent correctly, the routine returns with carry true, but carry false indicates that the port timed out. Registers A and BC are corrupt on return, other registers unaltered.

It is possible to divert this call to user code, in which case other related calls are of interest.

CALL BD28: MC RESET PRINTER

The normal indirection of BDF1 is restored. There are no entry conditions, but AF, BC, DE and HL are corrupt on return.

CALL BD2B: MC PRINT CHAR

This calls MC WAIT PRINTER, even if the indirection has been changed.

The arrangement of these calls allows the user to get the codes being passed to the printer and apply conditional changes.

The timeout period is approximately Ø.4 sec.

Other related calls are:

CALL BD2E: MC BUSY PRINTER

No entry conditions. The return is with carry true if the printer is busy (or not fitted), false if the printer is idle. Flags are corrupt, but registers are preserved.

CALL BD31: MC SEND PRINTER

The character in the A register is passed to the printer. The return is with carry true, A corrupt, and other registers preserved. This call may only be used if the printer is not Busy.

These are examples of the facilities which lie in wait in the operating system for those who are able to use them. They need to be called via machine code, since parameters have to be set or flags sensed, but the routines required are simple enough for a tyro to attempt.

There is one other point to watch. The CPC464 outputs both CR and LF codes, but it also earths printer line 14, which makes some printers add another LF action. To obtain single-spaced text, it is necessary to disconnect line 14, which over-rides the setting of the internal DIL-switches. The problem does not arise with the AMSTRAD DMP1 printer, which has a slightly different interface.

The Outer Peripherals

The Programmable Sound Generator

The AY-3-8912 Programmable Sound Generator is an interesting device. It has three independent sound outputs, which can be set up to generate square-wave tones or pink noise. The volume level on each channel can be set to sixteen values, from silent to maximum, and there is a relatively simple form of internal envelope control. There are also two I/O ports, of which the CPC464 uses one.

A fully comprehensive description of this component would take up a disproportionate amount of space, but the following notes will cover the aspects special to the CPC464 system.

The PSG has sixteen control registers, and the method of accessing them is similar to that used with the CRTC: A register is selected first, and data can then be read from or written to it. The transfer modes are selected by three control lines, BDIR, BC1 and BC2. In the CPC464 BC2 is tied high, while BC1 and BDIR are controlled by bits 6 and 7, respectively, of Port C of the PPI. This gives the following modes:

BDIR	BC1	
Ø	Ø	Inactive
Ø	1	Read from PSG
1	Ø	Write to PSG
1	1	Latch address. (Select register)

The registers may be summarised thus:

RØ-R5: Taken in pairs, determine the tone on channels A, B and C respectively. Even-numbered registers contain low 8 bits of tone data, odd-numbered registers contain upper four bits of tone data, giving 12-bit data in all.

R6. Noise Period. Five bits.

R7. Enable control. A 1 state disables, Bits Ø-2 control tone for the three channels, Bits 3-5 control noise for the three

channels, bits 6 and 7 control the two ports. (1 for output, Ø for input.)

R8-R1Ø. Channel amplitude. Five bits. Values Ø to &F give fixed levels. &1X gives envelope control.

R11-12. Envelope period. 16-bit word.

R13 Envelope type. Sixteen options.

R14-R15. I/O channels.

The tone period data set in the first three pairs of registers is defined by:

Period data = 125ØØØ/Required frequency.

The clock fed to the said generator is at 1MHz.

Access to the chip is simplified by the use of a call:

MC SOUND REGISTER: CALL BD34

A must hold register number.

C must hold data to be sent.

On exit, AF and BC are corrupt.

Use of this direct access may prove to be more convenient than control of the chip by the complex BASIC commands. The full envelope control and automatic tone envelope control are not available, but the envelope control within the chip is quite versatile. The options are chosen by a four-bit word as follows:

Ø to 3:	Loud attack, fading to silence.
4 to 7:	Build-up to full, then silence.
8:	As Ø-3, but repeated.
9:	As Ø-3.
10:	Starts loud, fades, rises, and repeats.
11:	Loud attack, fade to silence, then full.
12:	Soft attack, builds to maximum, repeats.

13: Soft attack, builds to maximum, holds.

14: As 10 inverted.

15: As 4.

The contents of registers R11-R12 determine the time scales of envelope action.

The simplest way to experiment with the sound chip is to use a BASIC routine to set up the parameters and then call the following short block of machine code:

```
7000 C5          PUSH   BC
7001 E5          PUSH   HL        ;Save registers
7002 F5          PUSH   AF
7003 21 10 70    LD     HL,7010   ;Set HL as pointer
7006 7E          LD     A,(HL)    ;Read register number
7007 23          INC    HL        ;Advance pointer
7008 4E          LD     C,(HL)    ;Read data
7009 CD 34 BD    CALL   BD34      ;Call MC SOUND
                                  REGISTER
700C F1          POP    AF
700D E1          POP    HL        ;Restore registers
700E C1          POP    BC
700F C9          RET              ;Return to BASIC
```

The BASIC program must set up the above code, as was demonstrated in the dump program earlier, and poke the register number into &7010, the data into &7011. In some instances it will be necessary to set two registers to cover the complete data, and since individual ideas on the nature of the BASIC program are bound to differ it will not be given in full. Working out what is wanted will be a good exercise for a beginner, and will not trouble the more experienced.

The sound output of the three channels is combined to feed the internal loudspeaker, but split on the stereo output, channel B feeding both sides, channels A and C one side each.

The square-wave tone is broadly characteristic of the clarinet, but two or more tones of this type do not always blend well. It is worth experimenting with low-pass filters in the stereo output lines to obtain a more rounded tone.

Considerably more could be said about this interesting chip, but enough data has been provided to allow and encourage experiment.

The Keyboard

The keyboard is of the usual matrix-connected type, using ten input rows and eight output columns. The rows are driven by bits Ø-3 of channel C, and the columns are sensed via the PSG port and Port A of the PPI.

The key number allocated to a given key is given by adding the column number to eight times the row number. Thus the key connected to the third row wire and the second column wire is number 17, the wires being numbered from Ø.

The keyboard is checked fifty times a second, and if any key is depressed its number is entered in a buffer, and an entry is made in a 'bit map'. The key remains marked as depressed until it has been released for two successive scans, this being a protection against contact bounce. However, the system is vulnerable to the simultaneous depression of three keys at the corners of a rectangle of the matrix, the key at the fourth appearing to be depressed as well. This does not seem to create problems in practice.

It is only when a key number is removed from the keyboard buffer that it is translated into a code, by reference to three tables that define the key/code relationship for different shift states. The shift state is also determined by buffer contents, so there is no risk of translation error arising because the shift state has changed since the key was pressed.

In addition to the Caps Lock state, toggled into and out of use by the Caps Lock key, there is also a Shift Lock state, which

can be toggled on and off by pressing the Caps Lock key with the Control key pressed, but direct transition from Shift Lock to Caps Lock and vice versa is not possible. The normal state must intervene.

For certain code values, expansion into strings is possible, and the system is arranged so that either the code value or the expansion can be obtained, according to which operating system call is used.

The joystick connector links directly into the keyboard matrix, and is sensed by the same mechanism.

The Cassette Recorder

The cassette recorder has a three-wire interface with the rest of the system, the input and output data paths and the motor control. It is a welcome fact that fast forward wind and rewind are not affected by motor control, and can be used at any time.

The recording method used employs square-wave cycles, those for a high bit having twice the duration of those for low bits. The mean frequency can be varied between 700 and 2500 baud, though 1000 and 2000 baud are regarded as standard. During playback, the system deduces the recording speed used by examination of the leader tone, which consists of a long sequence of high-state cycles.

The overall format of a record is:

A leader.

N segments.

A trailer.

The leader consists of a pre-record gap, 2 048 high-state bits, a zero bit, and a sync byte. The sync byte is &2C for a header record, &16 for a data record.

Each segment consists of 256 data bytes and two CRC (Cyclic redundancy check) bytes. The CRC polynomial used is:

$x^{15} + x^{12} + x^5 + 1$

The initial seed is &FFFF. The first CRC byte is the high byte.

The trailer consists of 32 high-state bits.

The complete file begins with a header record containing 64 bytes which determine the way in which the remaining records in the file are read and interpreted. The make-up of the header is shown in the following table.

HEADER BLOCK FORMAT FOR CASSETTE RECORDING

Bytes 0/15: Filename, padded out with null codes.

Byte 16: Block number.

Byte 17: Non-zero if last block.

Byte 18: File type:

Bit 0: True for protection.

Bits 1-3: 0: Internal BASIC

1: Binary

2: Screen Image

3: ASCII

4-7: Unallocated

Bits 4-7: Version (1 for ASCII, else 0)

Bytes 19/20: Data length. (Of record)

Bytes 21/22: Data location. (Start address when recorded)

Byte 23: Non-zero if first block.

User fields:

Bytes 24/25: Number of bytes in file.

Bytes 26/27: Execution address for machine code.

Bytes 28/63: Unallocated.

The operating system calls associated with the cassette recorder are somewhat more comprehensive than those available in BASIC. CAS NOISY will disable prompt messages, CAS RETURN puts a character back into the read buffer, so that it can be examined but read again later, CAS OUT ABANDON allows an output file to be discarded, while CAS CHECK compares a record on tape with the contents of store, giving a Verify facility. These calls will be found listed in the tabulation of operating system calls.

In general, the normal cassette handling facilities will suffice for most purposes, but it is useful to know that there are other possible modes of working. These, however, are likely to be purely local, and would not produce tapes that could be used on other CPC464 machines, and that is a deterrent.

An important point to note is that normal interrupt service is suspended while the cassette recorder is in use, since interrupt handling could ruin the signal timings. The elapsed time clock does not advance, and other interrupt-controlled functions are left to their own devices.

However, at whatever level it is used, the cassette system has its advantages. There is one mystery about it which has not been solved, though. After text data has been recorded or recovered, there is sometimes a long wait for the prompt messages. It is suspected that a 'garbage disposal' session is involved, the time being about right . . .

The Operating System

Having examined the internal hardware of the CPC464, we must now look at the firmware. This is not a simple matter. There are more than 200 entry points, and all are in RAM, so that they can be altered if you wish, accessing your own code instead of — or in addition to — the normal routines.

For each entry point there is a function to be performed, there are entry conditions, there are exit conditions, and there is a list of the registers and flags that will have been corrupted. All this is tabulated at the end of this section, in concise form to minimise the space consumed.

Some of the routines, which have no entry conditions and which return no data, are accessible from BASIC, but as we have already seen there is usually a need to pass parameters from BASIC to machine code and back.

The standard method of passing parameters is effective but a little complex. Anyone using it is immediately confronted by the complications of the different ways in which data is stored by BASIC:

Integers are held in pairs of bytes as straight binary, low byte first.

Strings involve a string descriptor and a string body. The string descriptor uses three bytes, the first to define the length of the string, then the address of the actual string. Only the descriptor need be passed as a parameter.

Real numbers are stored in five-byte floating point, the mantissa first (lowest byte first), and the exponent byte last. Converting that to integer form, or attempting to perform calculations with it would be far from easy.

Fortunately, the parameters of a CALL are passed in simplified form. The integer is passed as a signed two-byte word, effectively unchanged. A real number is forced into unsigned integer form. A variable is passed as its address, and the address of a string descriptor is given.

Each parameter is therefore passed as a two-byte word, and the words are stored in a block to which the IX register points, while the number of parameters is set in A. To complicate matters, IX initially points to the last parameter.

So, suppose you want to call GRA LINE ABSOLUTE, which draws a line to a point defined by the contents of DE and HL, as the X and Y coordinates of the end point. You cannot call the entry address, BBF6, directly, because DE and HL must be set up first. What you will have to work from is:

A = 2 (Two parameters)

(IX + 2) = Parameter 1 (X)

(IX) = Parameter 2 (Y)

You need a routine which will copy (IX) to HL, and (IX+2) to DE. That is fairly simple:

```
7000 DD 6E 00  LD  L,(IX)
7003 DD 66 01  LD  H,(IX+1)
7006 DD 5E 02  LD  E,(IX+2)
7009 DD 56 03  LD  D,(IX+3)
```

Then you can call BBF6, following it by a return code (C9).

Notice that the count of parameters in A was not used. We assumed that it was correct.

Perhaps you can see now why it was considered easier to pass parameters in the earlier BASIC/machine-code routines by poking them into store locations. Not as neat on the BASIC side, perhaps, but easier overall. It would be possible to create a general routine for picking up parameters, in which case the eventual call, if any, would be one of the parameters, but poking and peeking are usually simpler.

No, it is not an easy matter to make use of operating system routines by BASIC alone. To get the most out of the CPC464, you must tangle with machine code.

As noted earlier, there are a few operating system entries which can be used from BASIC without trouble. Many of these, however, are already accessible via BASIC commands. Scan through the tabulation, and you will soon pick out those that are usable but not already covered.

One final point on this theme: BASIC seems able to recover after a CALL without difficulty, but it may sometimes be useful to save the main registers on the stack before calling an operating system routine and restore them afterwards, just in case . . .

The RST Area

Mention has been made of the routines packed into the 'RST Area', and you will need to know about these if you are to take full advantage of the facilities offered. The available entries are:

ØØØØ Startup entry, also accessed by op-code &C7. Complete reinitialisation follows use of this entry.

ØØØ8 Accessed by &CF, which has been converted into a pseudo op-code. The two bytes following &CF are picked up as data, using the standard low-byte-first convention. Bits Ø to 13 define an address in the bottom quarter of memory. Bit 14 controls lower ROM, a 1 state disabling, and bit 15 similarly controls the upper ROM. It is therefore possible to switch banks and jump in response to a single call, the actions appearing simultaneous. For example, CF F2 87 will

jump to 07F2 with the lower ROM enabled and the upper ROM disabled.

ØØØB Similar to ØØØ8, but the two-byte definition is in HL.

ØØØE Jump to an address defined in BC.

ØØ1Ø Accessed by &D7. This is known as SIDE CALL. The defining bytes follow &D7. Bits Ø-12 are added to CØØØ to define an address in upper ROM, while bits 13-15 define a 'sideways ROM in terms of an offset from the ROM select address of the main foreground ROM. This allows direct access to a point in a given sideways ROM.

ØØ13 Known as SIDE PCHL. As ØØ1Ø, but the two defining bytes are in HL.

ØØ16 Known as PCDE. Jump to an address defined in DE.

ØØ18 Accessed by &DF. Known as FAR CALL. The two bytes which follow &DF are the address of a three-byte definition, in which the first two bytes give the required memory address, the third byte YY selecting a sideways ROM on the following basis:

YY = Ø to &FB: Select ROM YY: Enable upper, disable lower.

YY = &FC No ROM change: Enable upper, enable lower.

YY = &FD No ROM change: Enable upper, disable lower.

YY = &FE No ROM change: Disable upper, enable lower.

YY = &FF No ROM change: Disable upper, disable lower.

ØØ1B FAR PCHL. As ØØ18, but address in HL, YY in C.

ØØ1E PCHL: Jump to address defined in HL.

ØØ2Ø RAM LAM. Equivalent to LD A,(HL), but always accesses RAM, whatever the ROM state. Accessed by &E7.

ØØ23 FAR ICALL. HL holds the address of a three-byte definition, of which the first two bytes define an address, and the third has the significance of YY as listed above.

ØØ28 FIRM JUMP. Accessed by &EF. The two bytes following &EF define a location in lower ROM or central RAM. Note that this is a jump, not a call, and a return address must be supplied on top of the stack.

All the above are copied from ROM into RAM during initialisation, and must not be altered. We now reach a point where 'patching' is permissible.

ØØ3Ø If this point is entered (by &F7) with the lower ROM enabled, the following actions are executed:

Disable interrupt

Copy current ROM state to ØØ2B

Disable lower ROM

Enable interrupt

Go to ØØ3Ø (In RAM)

It is therefore possible to patch user code in the ØØ3Ø-ØØ37 area with the assurance that it will be executed if ØØ3Ø is entered, whether the lower ROM is enabled or not. Location ØØ3Ø in RAM is initialised to &C7 to cover a situation where no patching is carried out. The standard operating system does not call ØØ3Ø.

The Z80 is set for operation in interrupt mode 1, so it responds to an interrupt signal by calling an entry to ØØ38. Since either ROM or RAM may be active when interrupt occurs, both store banks must hold a jump to the interrupt handler. If the handler fails to recognise the source of the interrupt, it calls ØØ3B, which usually holds a return op-code in both ROM and RAM. The call is made with ROM disabled, however, so the user can patch his own code in RAM between ØØ3B and ØØ3F. The code will normally be a jump to a handling routine stored elsewhere.

The entry at ØØ38 could — but must not — be called by op-code &FF.

Making full use of these facilities calls for careful thought and a familiarity with the Z80 instruction set. However, the worst consequences of mistakes can be put to right by a system reset.

Jumpblock Entries

At the end of this section, a list of more than two hundred operating system calls is given. These should normally be accessed by way of the 'jumpblock' entries, which are held in RAM and can therefore be modified. This provides a powerful programming tool.

Suppose, for example, that you wish to alter the routine that sends code to the printer. This is accessed at BD2B, the MC PRINT CHAR entry. At BD2B there is the following code:

CF F2 Ø7

As noted earlier, this performs a jump to Ø7F2 in lower ROM, where the printer driver routine is stored. However, the link can be patched to give a jump to an entirely different user routine, which is executed in place of the standard routine. Details of this procedure are given in the latter part of the book, in relation to an alternative set of hardware forming an eight-bit printer interface.

Since the revised code will be held in RAM, and preferably in the centre half of the address range, the CF command is not necessary, and a normal &C3 jump code will be adequate.

It will be evident that some operating system calls should not be altered, since the consequences of change would be too extensive, but cautious experiment will always be in order.

Interrupts and Events

The CPC464 uses four standard timed interrupts:

*The Fast Ticker, occurring 300 times a second.

*The Sound Timer, occurring 100 times a second.

*The Frame Flyback, occurring at vertical scan frequency.

*The Ticker, occurring 50 times a second.

These are all normal interrupts of the maskable type. No provision is made for dealing with Non-Maskable Interrupts (NMI), which would cause a jump to ØØ66. That might be in ROM or RAM, so the consequences would be unpredictable.

Events are short routines which may be triggered by interrupts or by permission of the current foreground program. Those triggered by interrupt are called 'asynchronous events', those triggered by program action are called 'synchronous events'. Normally, a queue of event calls is used, with separate queues for the two kinds of event, but 'express events' are serviced as soon as they are called. Express events should be as short as possible, and may not enable interrupt or corrupt the IX and IY registers.

Events are defined by 'event blocks', which are set up by the operating system call KL INIT EVENT on the basis of data supplied. This data includes the address and ROM number of the routine to be executed, and the location of the event block, which is seven bytes long. The class of the event must also be stated, using a byte defined as follows:

Bit Ø is true if the event involves a 'near address', false if a 'far address' applies. (see notes on the RST area.)

Bits 1 to 4 are only relevant to synchronous events, for which they define a priority level.

Bit 5 is always zero.

Bit 6 is set to 1 for an express event.

Bit 7 is set to 1 for an asynchronous event.

KL INIT EVENT sets up an event block of the form:

Bytes Ø,1: Chain Link to next block.

Byte 2: Count of events outstanding

Byte 3: Class of event

} Reversed in CPC464 documentation.

Bytes 4,5: Address of event routine.

Byte 6: ROM select state required.

Bytes 7 onward may be set by the user to provide parameters.

To allow the event routine to pick these up, it is supplied with the address of the event block.

When an event is 'kicked', the count in byte 3 is incremented, and the count is decremented when the event is serviced. An event can be disabled by setting the count byte to – 64. If the count is zero it is not decremented, and if it is &7F it is not incremented.

Operating system calls are available to arm and disarm events. The foreground program must check at regular intervals for outstanding synchronous events.

For asynchronous events, the event block is tagged on to a header block appropriate to the type of interrupt involved. For Ticker interrupts:

Bytes Ø,1: Tick chain link.

Bytes 2,3: Tick count.

Bytes 4,5: Recharge count.

Bytes 6 on: Event block.

The count is set from the recharge count when an event is 'kicked', and is then decremented for each appearance of the Ticker interrupt. When the count reaches zero, the event is again kicked, and the count is set to the recharge value. Thus, if the recharge count is set to 5, the event wil be kicked on every fifth Ticker interrupt, or ten times a second.

For Frame Flyback and Fast Ticker interrupts, only the Chain Link is prefixed to the event block.

Note that event and interrupt blocks are set automatically by operating system calls, and should not be modified by the user except where otherwise indicated.

On first acquaintance, the interrupt and event system is of daunting complexity, but its potential is enormous. Using it to display the time of day in a corner of the screen, while a normal program proceeds, is merely scratching the surface of the possibilities. A really inspired user might well fill the entire area of middle RAM with event blocks and their routines, but that could lead to a situation in which the computer was asked to achieve the impossible. There is only one processor, and it can only do

a given amount in a given time. Each event pulls it away from its foreground task, using up execution time and perhaps reducing foreground time to nearly zero.

In particular, the Fast Ticker needs to be used with caution. If the events it calls take, say, 33Ø microseconds to execute, they will absorb a tenth of the processing time, slowing down foreground execution in the same proportion.

All event and interrupt blocks should be held in the centre half of RAM, so that they are always accessible. A 'safe area' can be defined by using the BASIC command MEMORY, which can be used to create a protected area of store. This will be examined in more detail later.

Operating System Calls

The operating system calls identified by AMSOFT are listed here in as concise a manner as possible. Further details can be found in the AMSTRAD CPC464 Firmware book. (See bibliography.)

Each call is identified by a name and the address of its jumpblock entry. Each jumpblock entry defines the address of the actual routine, and that address could be used if speed is vitally important, but the correct ROM state must then be set, whereas the jumpblock entry sets the state required automatically.

It should also be noted that AMSTRAD guarantee the jumpblock entries, but not the direct routine entries, which may change in later versions of the ROM.

For one section of the jumpblock, no call names or functions are described by AMSOFT. Investigation of some of these entries showed a converter from integer variables to floating point and other useful functions. These are used by BASIC, but a full definition of their characteristics is not feasible, since they involve a knowledge of the internals of the BASIC interpreter.

It can be assumed that registers not mentioned under Exit conditions are preserved. However, it should be noted that this does not include the 'alternate' registers, which are not available for the user.

KM INITIALISE: BBØØ

Initialises Key manager completely, all variables, buffers and special indirections being lost.

No entry conditions.

On exit AF, BC, DE and HL are corrupt.

KM RESET: BBØ3

Resets Key Manager indirections and buffers.

No entry conditions.

On exit AF, BC, DE and HL are corrupt.

KM WAIT CHAR: BBØ6

Waits for a character from the keyboard buffer or expansion string.

No entry conditions.

On exit the character code is in A and carry is true. Other flags are corrupt.

KM READ CHAR: BBØ9

Takes a character from the keyboard buffer or expansion string if one is immediately available. Does not wait.

No entry conditions.

On exit, if a character is available the code is in A and carry is true, else carry is false and A corrupt. Other flags corrupt.

KM CHAR RETURN: BBØC

A character is re-inserted in the keyboard buffer, so that it can be read again later, allowing the character to be checked without being lost.

Only one character can be 'put back' at a time. The character put back must be read before another is put back.

On entry, A holds code of character to be put back.

On exit, all flags and registers are preserved.

KM SET EXPAND: BB0F

Sets up an expansion string related to a given token code.

On entry, B holds the expansion token to be used, and C holds the length of the string to be set. HL points to the string.

On exit, carry is true if the setting was successful, else false. A, BC, DE, HL and other flags are corrupt.

Note: The string must lie in RAM, not ROM.

KM GET EXPAND: BB12

Read a character from an expansion string. The first character is number 0.

On entry, A holds the expansion token and L holds the character number.

On exit, if the get was successful carry is set and the character code is in A, else carry is false and A is corrupt. DE is corrupt, and so are flags other than carry.

KM EXP BUFFER: BB15

A buffer for expansion strings is allocated, and initialised with the default expansion strings.

On entry, DE holds the address of the buffer, HL holds its length.

On exit, carry is true unless the buffer is too short. A, BC, DE, and HL are corrupt, so are flags other than carry.

KM WAIT KEY: BB18

Waits for a key from the key buffer.

No entry conditions.

On exit, carry is true and A holds the character or expansion token. (Tokens are not expanded.)

KM READ KEY: BB1B

Takes a key from the key buffer if one is available.

No entry conditions.

On exit, if a key was available carry is true and A holds the character code or expansion token, else carry is false and A is corrupt. Other flags are corrupt.

KM TEST KEY: BB1E

Tests whether a specified key is pressed.

On entry A holds a key number.

On exit, if the key is pressed the zero flag is false, else true. Carry is always false, C contains the current shift and control state, A, HL and other flags corrupt.

KM GET STATE: BB21

Checks for Caps and Shift LOCK states.

No entry conditions.

On exit, L holds Shift Lock state, H holds Caps Lock state. If the lock is on, the state is &FF, else Ø. AF is corrupt.

KM GET JOYSTICK: BB24

Check joystick state.

No entry conditions.

A and H contain state of joystick Ø. L contains state of joystick 1. Flags are corrupt.

The reported bytes have the following form:

Bit Ø UP

Bit 1 DOWN

Bit 2 LEFT

Bit 3 RIGHT

Bit 4 FIRE 2

Bit 5 FIRE 1

Bit 6 Spare Button

Bit 7 Ø

KM SET TRANSLATE: BB27

Set code or token generated by given key.

On entry, A holds key number, B holds code or token.

On exit, AF and HL are corrupt.

The following token values have special use:

80 - 9F	Expansion tokens.
EO - FC	Edit system codes.
FD	Caps Lock.
FE	Shift Lock.
FF	Ignore. (Key has no meaning.)

KM GET TRANSLATE: BB2A

Check translation of key with neither Shift nor Control.

On entry, A holds a key number.

On exit, A holds the translation. HL and flags corrupt.

KM SET SHIFT: BB2D

Set translation of key with Shift, not Control.

On entry, A holds a key number, B holds translation.

On exit, AF and HL are corrupt.

See KM SET TRANSLATE for special code meanings.

KM GET SHIFT: BB3Ø

Check translation of key with Shift, not Control.

On entry, A holds a key number.

On exit, A holds the translation. HL and flags are corrupt.

KM SET CONTROL: BB33

Set translation of a key with Control.

On entry, A holds key number, B holds translation.

On exit, AF and HL are corrupt.

See KM SET TRANSLATE for special code meanings.

KM GET CONTROL: BB36

Check translation of key with Control.

On entry, A holds a key number.

On exit, A holds the translation, HL and flags are corrupt.

KM SET REPEAT: BB39

Determine whether a key is allowed to repeat.

On entry, A contains the key number; B contains &FF if the key is to be allowed to repeat, else Ø.

On exit AF, BC and HL are corrupt.

KM GET REPEAT: BB3C

Check whether a key is allowed to repeat.

On entry, A holds a key number.

On exit, the zero flag is false if the key is allowed to repeat, else true. Carry is false, A, HL and other flags corrupt.

KM SET DELAY: BB3F

Set key repeat delay and period.

On entry H holds start delay, L holds repeat period. Values are in fiftieths of a second. A zero value counts as 256.

On exit, AF is corrupt.

KM GET DELAY: BB42

Check key repeat delay and period.

No entry conditions.

On exit, H holds start delay, L holds repeat period. Values are in fiftieths of a second. AF is corrupt.

KM ARM BREAKS: BB45

Enable break events.

On entry DE holds the address of the break event routine, C holds the ROM select address for the routine.

On exit AF, BC, DE and HL are corrupt.

KM DISARM BREAK: BB48

Disable break events. (This is the default state.)

No entry conditions.

On exit AF and HL are corrupt.

KM BREAK EVENT: BB4B

Generate a break if break events enabled. Disable break event.

No entry conditions.

On exit AF and HL are corrupt.

A break event token (&EF) is placed in the key buffer. This generates a break event when read from the buffer.

TXT INITIALISE: BB4E

Full initialisation of the Text VDU, including all variables and indirections.

No entry conditions.

On exit AF, BC, DE and HL are corrupt.

TXT RESET: BB51

Resets the Text VDU indirections and control table only.

No entry conditions.

On exit AF, BC, DE and HL are corrupt.

TXT VDU ENABLE: BB54

Allow screen display for currently-selected stream.

No entry conditions.

On exit AF is corrupt.

TEXT VDU DISABLE: BB57

Bar screen display for currently selected stream (including cursor).

No entry conditions.

On exit, AF is corrupt.

TXT OUTPUT: BB5A

Output a code to the Text VDU.

On entry A holds the code to be sent.

On exit all flags and registers are preserved.

TXT WR CHAR: BB5D

Print a character at the cursor position of the current stream. Control codes are printed, not obeyed. VDU must be enabled.

On entry A holds the character code.

On exit AF, BC, DE and HL are corrupt.

TXT RD CHAR: BB60

Read a character from the screen at the cursor position of the current stream.

No entry conditions.

On exit, if a recognisable character was found; Carry is true and A holds the character code, else carry is false and A holds zero.

Other flags are corrupt.

TXT SET GRAPHIC: BB63

Enable or disable graphics character option.

On entry, A holds zero to turn on option, non-zero to turn it off.

On exit, AF is corrupt.

TXT WIN ENABLE: BB66

Set up window for current stream.

On entry, D and H contain edge columns, the smaller being the left edge. E and L contain edge rows, the smaller being the top.

On exit AF, BC, DE and HL are corrupt.

TXT GET WINDOW: BB69

Check the boundaries of the window for the current stream.

No entry conditions.

On exit H contains left column, D contains right column, L holds top row, E holds bottom row. Carry is false if the window covers the whole screen.

In the last two calls, column and line positions count from Ø, not from 1, as in BASIC.

TXT CLEAR WINDOW: BB6C

Set the text window of the current stream to the paper ink for that stream. Cursor to top left corner.

No entry conditions.

On exit AF, BC, DE and HL are corrupt.

TXT SET COLUMN: BB6F

Move the cursor for the current stream to a specified column.

On entry A holds the required column number. (1 upwards.)

On exit AF and HL are corrupt.

TXT SET ROW: BB72

Move the cursor for the current stream to a specified row.

On entry A holds the required row number. (1 upwards.)

On exit AF and HL are corrupt.

TXT SET CURSOR: BB75

Move the cursor for the current stream to a specified row and column.

On entry H holds the column, L holds the row. (1 upwards.)

On exit AF and HL are corrupt.

TXT GET CURSOR: BB78

Check the location of the cursor for the current stream and the 'roll count'.

No entry conditions.

On exit H contains the column and L the row. A contains the roll count, which is decremented when the window is rolled up, and incremented when the window is rolled down. The cursor position given is not necessarily valid.

TXT CUR ENABLE: BB7B

Enables the cursor for the current stream.

No entry conditions.

On exit, AF is corrupt.

TXT CUR DISABLE: BB7E

Disables the cursor for the current stream.

No entry conditions.

On exit, AF is corrupt.

TXT CUR ON: BB81

Allows the cursor for the current stream.

No entry conditions.

All flags and registers preserved.

TXT CUR OFF: BB84

Prevents the cursor for the current stream.

No entry conditions.

All flags and registers preserved.

Note: BB7B/E are intended to be used generally. They override BB81/4 in respect of cursor suppression. BB81/4 are for use by system ROMs.

TXT VALIDATE: BB87

Check if cursor within current window area.

On entry H holds column and L holds row (1 upwards) of position.

On exit: If in window, carry is true and B corrupt. If window would roll up, carry is false and B holds &FF. If window would roll down, carry is false and B holds Ø. In all cases H and L hold the column and row at which the character would appear, A and other flags corrupt.

TXT PLACE CURSOR: BB8A

Display a cursor for the currently selected stream. This allows multiple cursors. It is unwise to call TXT PLACE CURSOR twice for a given screen position without an intervening TXT REMOVE CURSOR, as one cursor could persist.

No entry conditions.

On exit, AF is corrupt.

TXT REMOVE CURSOR: BB8D

Remove a multiple cursor from the screen.

No entry conditions.

On exit, AF is corrupt.

TXT SET PEN: BB9Ø

Define foreground ink for current stream.

On entry A holds ink number.

On exit, AF and HL are corrupt.

TXT GET PEN: BB93

Check foreground ink for current stream.

No entry conditions.

On exit, A holds the ink number. Flags are corrupt.

TXT SET PAPER: BB96

Define background ink for current stream.

On entry A holds ink number.

On exit AF and HL are corrupt.

TXT GET PAPER: BB99

Check background ink for current stream.

No entry conditions.

On exit A holds the ink number. Flags are corrupt.

TXT INVERSE: BB9C

Exchange foreground and background inks for current stream.

No entry conditions.

On exit, AF and HL are corrupt.

TXT SET BACK: BB9F

Determine whether background colour is displayed. It is displayed in opaque mode, not in transparent.

On entry, A holds Ø for opaque mode, a non-zero value for transparent mode.

On exit AF and HL are corrupt.

TXT GET BACK: BBA2

Check whether background colour is being displayed.

No entry conditions.

On exit, A holds Ø if opaque mode applies, else a non-zero value. DE, HL and flags are corrupt.

TXT GET MATRIX: BBA5

Get the address of a character pattern.

On entry, A holds code for character.

On exit, HL holds the pattern address. If it is in ROM, carry is false. If it is in RAM, carry is true. A and flags corrupt.

TXT SET MATRIX: BBA8

Set a character pattern.

On entry, A holds the character code, and HL holds the address of the matrix pattern to be set up.

On exit, carry is true if the character is user-definable, else false. A, BC, DE, HL and flags corrupt.

TXT SET M TABLE: BBAB

Set multiple character patterns. (Up to &FF.)

On entry, DE holds first character code, HL contains address of start or pattern source table.

On exit, if there was no user-defined table before, carry is false and A and HL are corrupt. Otherwise carry is true, and A holds the first character for the new patterns, HL holds the address of the old table. BC, DE and flags corrupt.

TXT GET M TABLE: BBAE

Check the address of the user-defined matrix table and the code for the first character in the table.

No entry conditions.

On exit, if there is no user defined pattern table carry is false and A and HL are corrupt, else carry is true, A holds the first character code in the table, and HL holds the start address of the table. Other flags are corrupt.

TXT GET CONTROL: BBB1

Gets the address of the control code table. This has three bytes per code. The first byte gives the number of parameters which the code requires, the other two bytes give the address of the related routine.

No entry conditions.

On exit, HL holds the address of the control table.

TXT STR SELECT: BBB4

Select stream.

On entry A holds stream to be set.

On exit HL and flags are corrupt.

TXT SWAP STREAMS: BBB7

Exchange two streams.

On entry B and C hold two stream numbers.

On exit AF, BC, DE and HL are corrupt.

GRA INITIALISE: BBBA

Initialise the graphics VDU. (All variables and indirections are set to default values.)

No entry conditions.

On exit AF, BC, DE and HL are corrupt.

GRA RESET: BBBD

Set Graphics VDU indirections to default values.

No entry conditions.

On exit AF, BC, DE and HL are corrupt.

GRA MOVE ABSOLUTE: BBC0

Move the graphics cursor to an absolute position.

On entry DE holds the required X coordinate, HL holds the Y coordinate.

On exit AF, BC, DE and HL are corrupt.

GRA MOVE RELATIVE: BBC3

Move the graphics cursor relative to its present position.

On entry DE holds the signed X offset, HL holds the signed Y offset.

On exit AF, BC, DE and HL are corrupt.

GRA ASK CURSOR: BBC6

Check location of graphics cursor.

No entry conditions.

On exit DE holds the user X coordinate, HL contains the user Y coordinate. AF is corrupt.

GRA SET ORIGIN: BBC9

On entry DE holds the standard X coordinate, HL holds the standard y coordinate.

On exit AF, BC, DE and HL are corrupt.

GRA GET ORIGIN: BBCC

Check position of origin.

No entry conditions.

On exit DE holds standard X coordinate, HL holds standard Y coordinate.

GRA WIN WIDTH: BBCF

Set right and left edges of graphics window.

On entry DE and HL hold the standard X coordinates for the edges.

The smaller value will determine the left edge.

On exit AF, BC, DE and HL are corrupt.

GRA WIN HEIGHT: BBD2

Set top and bottom of graphics window.

On entry DE and HL hold the standard Y coordinates for the edges.

The smaller value will determine the bottom edge.

On exit AF, BC, DE and HL are corrupt.

GRA GET W WIDTH: BBD5

Check right and left edges of graphics window.

No entry conditions.

On exit DE holds the standard X coordinate for the left edge, HL holds the standard X coordinate for the right edge. AF is corrupt.

GRA GET W HEIGHT: BBD8

Check top and bottom edges of graphics window.

No entry conditions.

On exit DE holds the standard Y coordinate for the top edge, HL holds the standard Y coordinate for the bottom edge.

GRA CLEAR WINDOW: BBDB

Clear the graphics window to graphics paper colour.

No entry conditions.

On exit AF, BC, DE and HL are corrupt.

Note: The graphic cursor moves to the origin.

GRA SET PEN: BBDE

Set graphics foreground colour.

On entry A holds ink number.

On exit AF is corrupt.

GRA GET PEN: BBE1

Check graphics foreground colour.

No entry conditions.

On exit A holds the ink number. Flags are corrupt.

GRA SET PAPER: BBE4

Set graphics background colour.

On entry A holds ink number.

On exit AF is corrupt.

GRA GET PAPER: BBE7

Check graphics background colour.

No entry conditions.

On exit A holds the ink number. Flags are corrupt.

GRA PLOT ABSOLUTE: BBEA

Display a pixel at an absolute position.

On entry DE holds user X coordinate, HL holds user Y coordinate.

On exit AF, BC, DE and HL are corrupt.

GRA PLOT RELATIVE: BBED

Display a pixel at a position relative to the graphics cursor position.

On entry DE holds a signed X offset, HL holds a signed Y offset.

On exit AD, BC, DE and HL are corrupt.

GRA TEST ABSOLUTE: BBFØ

Check ink of pixel at absolute position.

On entry DE holds user X coordinate, HL holds user Y coordinate.

On exit A holds ink number, BC, DE, HL and flags corrupt.

GRA TEST RELATIVE: BBF3

Check ink of pixel at a position relative to the graphics cursor.

On entry DE holds a signed X offset, HL holds a signed Y offset.

On exit A holds an ink number, BC, DE, HL and flags are corrupt.

GRA LINE ABSOLUTE: BBF6

Draw a line to an absolute position from the current cursor position.

On entry DE holds the user X coordinate, HL holds the user Y coordinate.

On exit AF, BC, DE and HL are corrupt.

GRA LINE RELATIVE: BBF9

Draw a line from the present cursor position to a position relative to the present cursor position.

On entry DE holds a signed X offset, HL holds a signed Y offset.

On exit AF, BC, DE and HL are corrupt.

GRA WR CHAR: BBFC

Place a character on the screen at the current graphics position.

On entry A holds the character code.

On exit AF, BC, DE and HL are corrupt.

SCR INITIALISE: BBFF

The Screen Pack is fully initialised. All variables and indirections are reset, as are screen mode and inks.

No entry conditions.

On exit AF, BC, DE and HL are corrupt.

SCR RESET: BCØ2

Resets Screen Pack indirections and colours, also flash rate and write mode.

No entry conditions.

On exit AF, BC, DE and HL are corrupt.

SCR SET OFFSET: BCØ5

Set the screen offset.

On entry HL contains required offset.

On exit AF and HL are corrupt.

SCR SET BASE: BC08

Set the screen base address. (C000 or 4000.)

On entry A contains the high byte of the base address.

On exit AF and HL are corrupt.

SCR GET LOCATION: BC0B

Check screen memory base and offset.

No entry conditions.

On exit A holds high byte of base address, HL holds offset. Flags are corrupt.

SCR SET MODE: BC0E

Sets screen mode.

On entry A holds required mode.

On exit AF, BC, DE and HL corrupt.

SCR GET MODE: BC11

Check current screen mode.

No entry conditions.

On exit, for Mode 0 carry is true, zero false, and A holds 0. For Mode 1 carry is false, zero true, and A holds 1. For Mode 2 carry and zero are both false, and A holds 2. Other flags corrupt.

SCR CLEAR: BC14

The screen is cleared to ink 0.

No entry conditions.

On exit AF, BC, DE and HL are corrupt.

SCR CHAR LIMITS: BC17

Check screen size in characters.

No entry conditions.

On exit, B holds last screen column, C holds last screen row.

AF is corrupt.

SCR CHAR POSITION: BC1A

Calculate screen address for top left corner of character position.

On entry H holds column, L holds row, in physical values (Ø up).

On exit HL holds screen address, B holds bytes per character.

AF is corrupt.

SCR DOT POSITION: BC1D

Calculate screen address for a pixel.

On entry DE contains X coordinate, HL contains Y coordinate.

On exit HL contains screen address, B holds pixels per byte minus one, C holds the mask for the pixel. AF and DE are corrupt.

SCR NEXT BYTE: BC2Ø

Find screen address for one byte to right of given address.

On entry HL holds a screen address.

On exit HL holds updated screen address. AF is corrupt.

SCR PREV BYTE: BC23

Find screen address for one byte to left of given address.

On entry HL holds a screen address. AF is corrupt.

On exit HL holds updated screen address.

SCR NEXT LINE: BC26

Find screen address for one line down from given address.

On entry HL contains given screen address.

On exit HL contains updated screen address. AF is corrupt.

SCR PREV LINE: BC29

Find screen address for one line above given address.

On entry HL contains given screen address.

On exit HL contains updated screen address. AF is corrupt.

SCR INK ENCODE: BC2C

Encode an ink for all pixels in a byte.

On entry A holds an ink number.

On exit A holds the encoded ink. Flags are corrupt.

SCR INK DECODE: BC2F

Decode an ink

On entry A holds an encoded ink mask.

On exit A holds the corresponding ink number. Flags are corrupt.

SCR SET INK: BC32

Set colours related to an ink number. If the two colours differ, they will alternate in the display.

On entry A holds the ink number, B holds the first colour, C holds the second colour.

On exit AF, BC, DE and HL are corrupt.

SCR GET INK: BC35

Check the colours related to a given ink.

On entry A holds the ink number.

On exit B holds the first colour, C holds the second colour. AF, DE and HL are corrupt.

SCR SET BORDER: BC38

Set border colours. If the colours differ, they will alternate.

On entry B holds the first colour, C holds the second colour.

On exit AF, BC, DE and HL are corrupt.

SCR GET BORDER: BC3B

Check border colours.

No entry conditions.

On exit B holds first colour, C holds second colour. AF, DE and HL are corrupt.

SCR SET FLASHING: BC3E

Set flash periods. (In frame scan periods.)

On entry H contains the first colour period, L contains the second colour period.

On exit AF and HL are corrupt.

SCR GET FLASHING: BC41

Check flash periods. (In frame scan periods.)

No entry conditions.

On exit H contains first colour period, L contains second colour period. AF is corrupt.

SCR FILL BOX: BC44

Fill a rectangular area of the screen with given ink.

On entry, A holds the encoded ink to be used, H contains the left column to be filled, L contains the top row to be filled, D contains the right column to be filled, E contains the bottom row to be filled.

Coordinates are from Ø upwards.

On exit AF, BC, DE and HL are corrupt.

Note: The current VDU write mode is ignored.

SCR FLOOD BOX: BC47

Fill a rectangular area of the screen with a given ink, to byte boundaries.

On entry HL contains the screen address of the top left corner of the area to be filled, D holds the unsigned width to be filled (in bytes). E contains the unsigned height of the area to be filled (in screen lines). C contains the encoded ink to be used.

On exit, AF, BC, DE and HL are corrupt.

Note: Graphics VDU write mode is ignored.

SCR CHAR INVERT: BC4A

Convert a character at a given position to reverse video form, by changing inks.

On entry, B and C contain encoded inks. H defines the text column, L defines the text row. (Coordinates from Ø upwards.)

On exit AF, BC, DE and HL are corrupt.

SCR HW ROLL: BC4D

Roll the whole screen up or down by eight pixel lines. The line brought on to the screen is cleared. Text roll is not changed.

On entry, B holds Ø to roll down, a non-zero value to roll up. A holds the encoded ink to which the cleared line will be set.

On exit AF, BC, DE and HL are corrupt.

SCR SW ROLL: BC5Ø

Roll a defined screen area up or down by eight pixel lines. The line brought in is cleared.

On entry B holds Ø to roll down, non-zero to roll up. A holds the encoded ink to be used for the cleared line, H holds the left column of the area to be cleared, D holds the right column, L holds the top row, E holds the bottom row. (Coordinates from Ø upwards.)

SCR UNPACK: BC53

Convert a character pattern to a set of pixel masks for the current screen mode.

On entry, HL contains the address of the character pattern, DE contains the address of the area which is to hold the result.

On exit AF, BC, DE and HL are corrupt.

SCR REPACK: BC56

Convert a displayed character to standard character form.

On entry A holds the encoded ink for the character, H holds the column position and L the row position. (Coordinates from Ø upwards.) DE holds the address of the area in which the conversion is to be set.

SCR ACCESS: BC59

Set the Graphics VDU mode.

On entry A holds a key to the required mode:

Ø: Absolute or Force mode.

1: XOR mode.

2: AND mode.

3: OR mode.

On exit AF, BC, DE and HL are corrupt.

SCR PIXELS: BC5C

Write a pixel to the screen, ignoring Graphics VDU mode.

On entry B holds the encoded ink to be used, C holds the pixel mask, HL contains the screen address.

On exit AF is corrupt.

SCR HORIZONTAL: BC5F

Plot a horizontal line.

On entry A holds the required encoded ink, BC holds the X coordinate for the end of the line, DE holds the X coordinate for the start of the line, HL holds the Y coordinate. (DE must be not greater than BC.)

On exit AF, BC, DE and HL are corrupt.

SCR VERTICAL: BC62

Plot a vertical line.

On entry A holds the required encoded ink, BC holds the Y coordinate for the end of the line, DE holds the X coordinate, and HL holds the Y coordinate for the start of the line. (HL must not exceed DE.)

On exit AF, BC, DE and HL are corrupt.

CAS INITIALISE: BC65

Set up cassette manager, closing streams, setting default write speed and turning on prompt messages.

No entry conditions.

On exit AF, BC, DE and HL are corrupt.

CAS SET SPEED: BC68

Determine the cassette write speed and precompensation.

On entry A contains precompensation, HL defines write period. Precompensation can be between Ø and 255 (microseconds), the larger values being inappropriate. The contents of HL give the half-period of a zero bit cycle in microseconds. Mean baud rate is found by 1Ø⁶/(3*HL) HL must lie between 13Ø and 48Ø.

Default values: Fast: HL = 167, A = 5Ø. Slow: HL = 333, A = 25.

CAS NOISY: BC6B

Control cassette prompts.

On entry A holds Ø to enable prompts, non-zero to disable.

On exit AF is corrupt.

CAS START MOTOR: BC6E

Turn on cassette motor and wait for speed to stabilise.

No entry conditions.

On exit A holds previous motor state. Carry is true unless action aborted by pressing Escape.

CAS STOP MOTOR: BC71

Turn off cassette motor.

No entry conditions.

On exit A holds previous motor state. Carry is true unless Escape pressed.

CAS RESTORE MOTOR: BC74

Restore previous motor state.

On entry A holds previous motor state.

On exit carry is true unless Escape pressed.

CAS IN OPEN: BC77

Open cassette input file.

On entry B holds length of filename, HL holds address of filename, DE contains the address of a 2K buffer area.

On exit:

If opened correctly, carry is true and zero is false. HL holds the address of the buffer containing the file header, DE holds the data location given by the header, BC holds the file length given by the header. A holds the file type.

If stream in use, carry and zero are false. A, BC, DE and HL are corrupt.

If Escape was pressed carry is false and zero is true, A, BC, DE and HL are corrupt.

In all cases IX and other flags are corrupt.

CAS IN CLOSE: BC7A

Close cassette input file.

No entry conditions.

On exit carry is true if action completed, false if stream was not open. A, BC, DE, HL and other flags corrupt.

CAS IN ABANDON: BC7D

Abandon cassette input file.

No entry conditions.

On exit AF, BC, DE and HL are corrupt.

CAS IN CHAR: BC8Ø

Read a character from a cassette input file.

No entry conditions.

On exit:

If action executed, A holds character code, carry is true, zero is false.

If end of file was found, A is corrupt, carry false, zero false.

If Escape was pressed, A is corrupt, carry false, zero true.

In all cases IX and other flags are corrupt.

CAS IN DIRECT: BC83

Read complete file to store.

On entry HL holds address of block to be set from the cassette input file.

On exit:

If action completed, HL holds entry address from the header, carry is true, zero false.

If file was not open, HL is corrupt, carry false and zero false.

If Escape was pressed, HL is corrupt, carry false, zero true.

In all cases A, BC, DE, IX and other flags corrupt.

CAS RETURN: BC86

Return last character read to cassette input file.

No entry conditions.

On exit all registers and flags preserved.

CAS TEST EOF: BC89

Test for end of file.

No entry conditions.

On exit:

If not EOF, carry is true and zero false.

If EOF, carry is false and zero false.

If Escape pressed carry is false and zero true.

In all cases A, IX and other flags corrupt.

CAS OUT OPEN: BC8C

Open a cassette output file.

On entry B holds length of filename, HL holds address of

filename, DE holds address of 2K buffer area.

On exit:

If stream already open carry is false and HL is corrupt.

If opening successful carry is true and HL contains address of header buffer.

In both cases zero is false, A, BC, DE and IX are corrupt.

CAS OUT CLOSE: BC8F

Close a cassette output file. (And write it to tape.)

No entry conditions.

On exit:

If successful, carry is true and zero false.

If file not open, carry is false and zero false.

If Escape pressed, carry is false and zero true.

In all cases A, BC, DE, HL, IX and other flags are corrupt.

CAS OUT ABANDON: BC92

Abandon cassette output file.

No entry conditions.

On exit AF, BC, DE and HL are corrupt.

CAS OUT CHAR: BC95

Write character to cassette output file.

On entry A holds character to be written.

On exit:

If successful carry is true and zero false.

If file not open, carry is false and zero false.

If Escape pressed, carry is false and zero true.

In all cases A, IX and other flags are corrupt.

CAS OUT DIRECT: BC98

Write complete file.

On entry HL contains the address of the data to be written, DE contains the length of the data to be written, BC holds the entry address to be placed in the header, and A contains the file type to be placed in the header.

On exit:

If action executed carry is true and zero false.

If file was not open, carry is false and zero false.

If Escape was pressed carry is false and zero true.

In all cases A, BC, DE, HL, IX and other flags are corrupt.

CAS CATALOG: **BC9B**

List files on tape.

On entry DE holds the address of a 2K buffer area.

On exit carry is true if all was well, false if the read stream was in use (i.e. input file open). In all cases zero is false and A, BC, DE, HL, IX and other flags corrupt.

CAS WRITE: **BC9E**

Write a record to tape.

On entry HL holds address of data to be written, DE contains length of data, A holds the sync character. (&2C for header, &16 for data.)

On exit carry is true and A is corrupt if all went well, otherwise carry is false and A holds an error code:

Ø: Escape pressed
1: Overrun (writing error).

In any case BC, DE, HL and IX are corrupt.

CAS READ: **BCA1**

Read a record from tape.

On entry HL holds the address at which the data is to be set, DE holds the length of the data, and A holds the expected sync code.

On exit, if all went well carry is true and A corrupt. Otherwise carry is false and A holds an error code:

Ø: Escape pressed.

1: Overrun. (Reading error.)

2: CRC check failure.

In any case BC, DE, HL, IX and other flags are corrupt.

Note: CAS READ and CAS WRITE start and stop the cassette motor.

CAS CHECK: BCA4

Compare a tape record with the contents of store (Verify).

On entry HL holds the address of data to be compared, DE holds the length of the data, A holds the expected sync character.

On exit, if the check was successful carry is true and A is corrupt. Otherwise carry is false and A contains an error code:

Ø: Escape pressed.

1: Overrun. (Read error.)

2: CRC check failed.

3: Data mismatch.

In any case BC, DE, HL, IX and other flags corrupt.

Note: CAS CHECK starts and stops the cassette motor.

SOUND RESET: BCA7

Reset the Sound Manager, silencing the sound chip and clearing all queues.

No entry conditions.

On exit AF, BC, DE and HL are corrupt.

SOUND QUEUE: BCAA

Add a sound to a sound queue, if possible.

On entry HL points to a block of 9 bytes defining the sound to be set up:

Byte Ø: Bit Ø: Channel A

Bit 1: Channel B

Bit 2: Channel C

Bit 3: Rendezvous with A
Bit 4: Rendezvous with B
Bit 5: Rendezvous with C
Bit 6: Hold until released
Bit 7: Flush queue.

Byte 1: Amplitude envelope select
Byte 2: Tone envelope select
Byte 3:
Byte 4: Tone period
Byte 5: Noise period
Byte 6: Initial amplitude
Byte 7:
Byte 8: Duration or repeat count.

On exit, if successful carry is true and HL corrupt. If not, carry is false and HL is preserved. A, BC, DE, IX and other flags corrupt.

SOUND CHECK: **BCAD**

Find whether there is space in a sound queue.

On entry A contains the bit indicating which channel to test. (See SOUND QUEUE above.)

On exit, A hold the channel status thus:

Bits Ø-2: Number of free slots in queue
Bit 3: Waiting for rendezvous with A
Bit 4: Waiting for rendezvous with B
Bit 5: Waiting for rendezvous with C
Bit 6: Held
Bit 7: Active (Sound being produced.)

BC, DE, HL and flags corrupt.

SOUND ARM EVENT: **BCBØ**

Establish an event to run when sound queue empty.

On entry A holds the bit indicating a channel (see SOUND QUEUE above), and HL holds the address of the event block to be executed.

On exit AF, BC, DE and HL are corrupt.

See KL INIT EVENT.

SOUND RELEASE: **BCB3**

Release a held sound.

On entry A holds the bit/s indicating the channel. (See SOUND QUEUE above.)

On exit AF, BC, DE, HL and IX are corrupt.

SOUND HOLD: **BCB6**

Stop all sounds at once.

No entry conditions.

On exit carry is true if sound was active. A, BC, HL and other flags are corrupt.

SOUND CONTINUE: **BCB9**

Restart sounds that have been held.

No entry conditions.

On exit AF, BC, DE and IX are corrupt.

SOUND AMPL ENVELOPE: **BCBC**

Set up an amplitude envelope.

On entry A holds the envelope number, HL holds the address of a data block made up as follows:

Byte Ø: Number of sections.
Bytes 1- 3: First section.
Bytes 4- 6: Second section.
Bytes 7- 9: Third section.
Bytes 10-12: Fourth section.
Bytes 13-15: Fifth section.

Within each section, the first byte gives step count, the second gives step size, and third gives pause time. Unused sections need not be set up. The data block must not lie in a ROM-masked area of RAM.

On exit, carry is true if the action was successful. HL holds the data block address plus &1Ø, A and BC are corrupt.

Note: The details of this call and the next can be very complex, but can be resolved by reference to the User Instruction Manual.

SOUND TONE ENVELOPE BCBF

Set up a tone envelope.

On entry conditions are as for the previous call, SOUND AMPL ENVELOPE, except that the step size refers to pitch instead of amplitude.

On exit, conditions are as for SOUND AMPL ENVELOPE.

SOUND A ADDRESS: BCC2

Find address of amplitude envelope data.

On entry A holds envelope number.

On exit carry is true if action was successful, HL holds the address of the data block, BC holds envelope length. If action failed carry is false, HL corrupt, BC is preserved. A always corrupt.

SOUND T ADDRESS: BCC5

Find address of tone envelope data.

Entry and exit conditions as for SOUND A ADDRESS above.

KL CHOKE OFF: BCC8

Clear all event queues, timer and frame flyback lists, clearing all pending synchronous events and time related functions except sound generation and keyboard scanning. (Prepares for MC BOOT PROGRAM, BD13.)

No entry conditions.

On exit B holds ROM select address for the current foreground ROM, C holds ROM select address for a RAM foreground program, DE holds the entry address for the current foreground ROM. AF and HL are corrupt.

KL ROM WALK: BCCB

Identify and initialise all backround ROMS.

On entry DE holds the address of the first usable byte in memory, HL holds the address of the last usable byte.

On exit DE holds the address of the new first usable byte, HL holds the address of the new last usable byte. AF and BC are corrupt.

KL INIT BACK: BCCE

Initialise a background ROM.

On entry C holds the ROM select address, DE holds the address of the first usable byte in memory, HL holds the address of the last usable byte in memory.

On exit DE holds the address of the new first usable byte, HL holds the address of the new last usable byte. AF and B are corrupt.

KL LOG EXT: BCD1

Set up a Resident System Extension. (Add to list of external command servers.)

On entry BC holds the address of the Resident System Extension's command table, HL holds the address of a four-byte area of RAM.

On exit DE is corrupt.

Note: Neither the command table nor the four-byte area may lie under ROM. The use of Resident System Extensions will be discussed in the latter part of the book.

KL FIND COMMAND: BCD4

Search for a command word.

On entry HL holds the address at which the command word is stored.

On exit carry is true if the command was found, and C then holds the ROM select address, HL holds the address of the related routine. If the command was not found carry is false

and HL and C are corrupt. In either case A, B and DE are corrupt.

Note: Only sixteen letters of the command are significant.

KL NEW FRAME FLY: BCD7

Initialise and add a block to the frame flyback list.

On entry HL holds the address of the block, B holds the event class, C holds the ROM select address for the event routine, DE holds the address of the event routine.

On exit AF, DE and HL are corrupt.

KL ADD FRAME FLY: BCDA

Add block to frame flyback list.

On entry HL contains the address of the block.

On exit AF, DE and HL are corrupt.

KL DEL FRAME FLY: BCDD

Remove a block from the frame flyback list.

On entry HL holds the address of the block.

On exit AF, DE and HL are corrupt.

KL NEW FAST TICKER: BCEØ

Initialised and add a block to the fast ticker list.

On entry HL holds the address of the block, B contains the event class, C holds the ROM select address for the event routine, DE holds the address of the event routine.

On exit AF, DE and HL are corrupt.

KL ADD FAST TICKER: BCE3

Add a block to the fast ticker list.

On entry HL holds the address of the block.

On exit AF, DE and HL are corrupt.

KL DEL FAST TICKER: BCE6

Remove block from fast ticker list.

On entry HL holds the address of the block.

On exit AF, DE and HL are corrupt.

KL ADD TICKER: BCE9

Add a block to the ticker list.

On entry HL holds the address of the tick block, DE holds the initial count value, BC holds the recharge value.

KL DEL TICKER: BCEC

Remove block from the ticker list.

On entry HL holds the address of the block.

On exit, if the block was found carry is true and DE holds the count remaining. Otherwise carry is false and DE corrupt. In either case A, HL and other flags are corrupt.

KL INIT EVENT: BCEF

Initialise an event block.

On entry HL holds the address of the event block. B holds the event class, C holds the ROM select address for the event, and DE holds the address of the event routine.

KL EVENT: BCF2

Activate an event block.

On entry HL holds the address of the event block.

On exit AF, BC, DE and HL are corrupt.

KL SYNC RESET: BCF5

Clear the synchronous event queue.

No entry conditions.

On exit AF and HL are corrupt.

KL DEL SYNCHRONOUS: BCF8

Remove a synchronous event from the event queue.

On entry HL holds the address of the event block.

On exit AF, BC, DE and HL are corrupt.

KL NEXT SYNC: BCFB

Get next event from the queue.

No entry conditions.

On exit, if there is an event awaiting processing carry is true, HL holds the address of the event block, and A holds the priority for the previous event, if any. Otherwise carry is false, A and HL are corrupt. DE is always corrupt.

KL DO SYNC: BCFE

Execute an event routine.

On entry HL holds the address of the event block.

On exit AF, BC, DE and HL are corrupt.

KL DONE SYNC: BDØ1

Complete processing an event.

On entry C holds the previous event priority, HL holds the address of the event block.

On exit AF, BC, DE and HL are corrupt.

KL EVENT DISABLES: BDØ4

Disable normal synchronous events.

No entry conditions.

On exit HL is corrupt.

KL EVENT ENABLE: BDØ7

Enable normal synchronous events.

No entry conditions.

On exit HL is corrupt.

KL DISARM EVENT: BDØA

Prevent an event from occurring.

On entry HL holds the address of the event block.

On exit AF is corrupt.

KL TIME PLEASE: BDØD

Find the elapsed time.

No entry conditions.

On exit DE/HL hold the four-byte time count, DE holding the two more significant bytes.

KL TIME SET: BD1Ø

Set the time count.

On entry DE/HL hold the four-byte time count, DE holding the two more significant bytes.

On exit AF is corrupt.

MC BOOT PROGAM: BD13

Load and run a program.

On entry HL contains the address of the routine to call to load the program.

There is no exit as such, but if the load fails the loader routine should return with carry false. For a correct load the return should be with carry true and HL holding the program entry link.

MC START PROGRAM: BD16

Run a foreground program.

On entry HL holds the start address and C holds the required ROM selection byte.

There is no exit, as such.

MC WAIT FLYBACK: BD19

Wait for frame flyback.

No entry conditions.

On exit all registers and flags are preserved.

MC SET MODE: BD1C

Set screen mode.

On entry A holds required mode.

On exit AF is corrupt.

MC SCREEN OFFSET: BD1F

Set the screen offset.

On entry A holds required screen base, HL holds required screen offset.

On exit AF is corrupt.

MC CLEAR INKS: BD22

Set all inks and border to the same colour.

On entry DE holds an ink vector. D holds the common ink colour, E holds the border colour.

On exit AF is corrupt.

MC SET INKS: BD25

Set colours for all inks.

On entry, DE holds the address of an ink definition block, in which byte Ø defines the border colour, and bytes 1 to 16 define the colours for inks Ø to 15.

On exit AF is corrupt.

MC RESET PRINTER: BD28

Restore the normal printer indirection.

No entry conditions.

On exit AF, BC, DE and HL are corrupt.

MC PRINT CHAR: BD2B

Offer a character to the printer port.

On entry A holds the character to send. (Bit 7 will be ignored.)

On exit carry is true if the action was completed, false if the timeout (approx Ø.4 sec) was completed with no action. In either case A and other flags are corrupt.

MC BUSY PRINTER: BD2E

Test whether printer port is busy.

No entry conditions.

On exit carry is true if the port is busy, else false. Other flags are corrupt.

MC SEND PRINTER: BD31

Send a character to the printer port. (No busy test.)

On entry A holds the character to be sent. (Bit 7 ignored.)

On exit carry is true and A corrupt.

MC SOUND REGISTER: BD34

Send data to sound chip.

On entry A holds the register number, C holds the data byte.

On exit AF and BC are corrupt.

JUMP RESTORES: BD37

Re-establish the standard jumpblock.

No entry conditions.

On exit AF, BC, DE and HL are corrupt.

That completes the main firmware jumpblock. Firmware indirect jumps follow, but there is an intervening continuation of the main jumpblock for which no definitions are available. (BASIC functions.)

Indirections

IND:TXT UNDRAW CURSOR: BDDØ

Remove the cursor from the screen.

No entry conditions.

On Exit AF is corrupt.

IND:TXT WRITE CHAR: BBD3

Put a character on the screen.

On entry A holds the character code, H holds the column, L holds the row. (Coordinates from Ø upwards.)

On exit AF, BC, DE and HL are corrupt.

IND:TXT UNWRITE: BBD6

Read a character from the screen.

On entry H holds the column and L the row marking the character to be read.

On exit, if a readable character was found carry is true and A holds the character code, else carry is false and A holds zero.

In either case BC, DE, HL and other flags are corrupt.

IND:TXT OUT ACTION: BDD9

Output a character or control code.

On entry A holds the character or control code.

On exit AF, BC, DE and HL are corrupt.

IND:GRA PLOT: BDDC

Plot a point.

On entry DE holds X coordinate and HL holds Y coordinate.

On exit AF, BC, DE and HL are corrupt.

IND:GRA TEST: BDDF

Test a pixel.

On entry DE contains the X coordinate and HL holds the Y coordinate.

On exit A holds the decoded ink for the point. BC, DE and HL and flags are corrupt.

IND:GRA LINE: BDE2

Draw a line. (From the current graphics cursor position.)

On entry DE holds the X coordinate of the endpoint, HL holds the Y coordinate.

On exit AF, BC, DE and HL are corrupt.

IND:SCR READ: BDE5

Read a pixel from the screen and decode its ink.

On entry HL holds the screen address of the pixel, and C holds the mask for the pixel.

On exit A holds the decoded ink for the pixel. Flags are corrupt.

IND:SCR WRITE: BDE8

Write a pixel or pixels on the screen.

On entry HL holds the screen address for the pixel/s, C holds the pixel mask, and B holds the encoded ink to be used.

IND:SCR MODE CLEAR: BDEB

Clear the screen to ink Ø.

No entry conditions.

On exit AF, BC, DE and HL are corrupt.

IND:KM TEST BREAK: BDEE

Test for Break and Reset, calling appropriate event.

On entry interrupt must be disabled, and C hold SHIFT and CTRL key states.

On exit AF and HL are corrupt.

IND:MC WAIT PRINTER: BDF1

Print a character or time out.

On entry A holds the character code to be sent to the printer.

On exit carry is true if the action succeeded, else false. A and BC are corrupt.

That completes the indirections, and we now come to the High Kernel Jumpblock, which accesses routines held in RAM.

High Kernel Jumpblock

HI:KL U ROM ENABLE: B900

Enable the upper ROM.

No entry conditions.

On exit A holds the previous ROM state.

HI:KL U ROM DISABLE: B903

Disable the upper ROM.

No entry conditions.

On exit A holds the previous ROM state.

HI:KL L ROM ENABLE: B906

Enable the lower ROM.

No entry conditions.

On exit A holds the previous ROM state.

HI:KL L ROM DISABLE: B909

Disable the lower ROM.

No entry conditions.

On exit A holds previous ROM state.

HI:KL ROM RESTORE: **B90C**

Set ROM state.

On entry A holds ROM state, as returned by above calls.

On exit AF is corrupt.

HI:KL ROM SELECT: **B90F**

Select an upper ROM.

On entry C holds the select address for the required ROM.

On exit C holds the previous select address, B holds the previous ROM state. AF is corrupt.

HI:KL CURR SELECTION: **B912**

Check which upper ROM is currently selected.

No entry conditions.

On exit A holds select address for current upper ROM.

HI:KL PROBE ROM: **B915**

Check class and version of an upper ROM.

On entry C holds the relevant ROM select address.

On exit, A holds the ROM class, L holds the mark number, and H holds the version number.

HI:KL ROM DESELECT: **B918**

Restore previously selected upper ROM.

On entry B holds previous ROM state, C holds previous select address.

On exit C holds the ROM select address now applicable, B is corrupt.

HI:KL LDIR: **B91B**

Copy store, with an incrementing pointer, ROMs disabled.

On entry BC holds length of data block, DE holds destination address, HL holds source address. (Initial values.)

On exit F, BC, DE and HL are as set by LDIR instruction.

Note: Do not use where original and copy areas overlap, with copy higher.

HI:KL LDDR: B91E

As KL LDIR above, but with decrementing pointer. For use where LDIR is inappropriate, as indicated above.

HI:KL POLL SYNCHRONOUS: B921

Check if an event with a higher priority than the current event is pending.

No entry conditions.

On exit carry is true if a higher priority event is pending. A and other flags are corrupt.

Note that this last entry is direct, and cannot be patched.

The Interface

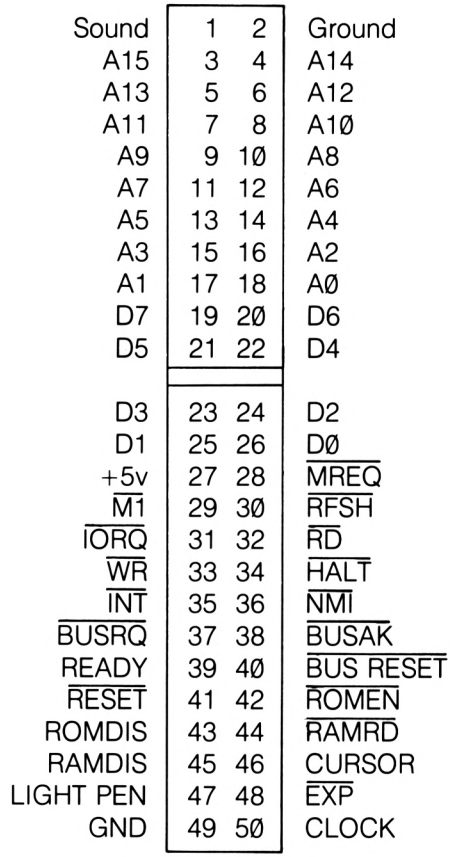


FIGURE 3a: EXPANSION PORT CONNECTOR

THE INTERFACE

The pin allocations of the 50-way expansion connector are shown in Fig 3a. All the central processor lines are brought out, plus a number of system control lines, but it should not be assumed that all the lines can be used.

Lines AØ-A15 are outputs defining a memory or I/O address. If the output $\overline{\text{MREQ}}$ is low, a memory access is required. If the output $\overline{\text{IORQ}}$ is low, an I/O transfer is required. The address should be interpreted accordingly, data being transferred in either case via the bidirectional data lines DØ-D7.

The $\overline{\text{M1}}$ line has caused a certain amount of confusion. It is an output which goes low when the central processor is fetching an op-code, which is the first byte of an instruction, except where the first byte is &CB, &DD, &ED, or &FD, in which case the op-code occupies the first two bytes. $\overline{\text{M1}}$ does not remain low while any subsequent bytes of the instruction are being fetched, so it

cannot be used to control a system which fetches instructions from one memory area and data from another.

If both $\overline{M1}$ and $\overline{IORQ}$ are pulled low, the processor is acknowledging an interrupt.

The output $\overline{RFSH}$ goes low when the lower seven address lines are carrying a refresh address for dynamic memory. The built-in refresh facility is one of the strong points of the Z8Ø processor, but needs to be used with care, since there are circumstances in which the refresh service is suspended, as explained below.

$\overline{HALT}$ is an output which goes low when the processor has executed a $\overline{HALT}$ instruction and is waiting for an interrupt. During this period the processor executes NOPs to maintain refresh action.

$\overline{INT}$ is the main interrupt input to the processor. It should be driven on an open-collector basis, being pulled low to call an interrupt. In this system, the processor responds by jumping to address ØØ38, the address of the instruction which would otherwise have been executed being pushed on to the stack, to allow return to that instruction when interrupt processing is complete. The Z8Ø interrupt mode 1 is used.

$\overline{NMI}$ is the non-maskable interrupt line. It should never be made low, since this would induce a jump to location ØØ66, which might be in RAM or ROM. The response is therefore unpredictable, but probably will be catastrophic.

$\overline{BUSRQ}$ is an input to the processor. When it is made low, it asks the processor to release control of all its lines other than $\overline{M1}$, $\overline{RFSH}$ and $\overline{HALT}$. Having completed the current instruction, the processor releases the lines and pulls $\overline{BUSAK}$ low to indicate that it has done so. The bus may now be controlled by an external system, which must replace the control actions normally executed by the processor. When $\overline{BUSRQ}$ is made high again, the system returns to normal.

The most common use for this facility is Direct Memory Access (DMA), which allows data to be transferred to and from memory or I/O channels without the need to execute program. Such transfers can be very fast, but it is unwise to keep $\overline{BUSRQ}$ low for too long, as this halts normal refresh action. The usual course is to

transfer one DMA byte at a time. This is called 'cycle stealing', because the external system takes over for one machine cycle.

READY is the processor signal WAIT, an input telling the processor that it should prolong its machine cycle by inserting wait states, usually because a peripheral or memory device is not ready for transfer.

That completes the processor signals. The processor outputs are not buffered within the CPC464, and should not be asked to drive more than one TTL load. Similarly, the 5v supply should not be asked to provide more than 100 mA. (Though a little more may be permissable if the cassette recorder is out of use.)

The lines which relate to the CPC464 system are as follows:

$\overline{\text{ROMEN}}$ is an output which goes low to select the external upper ROM. ROMDIS is an input which is made high to disable the internal ROM. Similarly, $\overline{\text{RAMRD}}$ goes low to enable external RAM, while the matching input RAMDIS goes high to disable internal RAM.

The timing of these signals is important. If two data sources are connected to the data bus at the same time, they could burn each other out. ROMDIS should therefore go high before $\overline{\text{ROMEN}}$ goes low, and so on.

CURSOR is the CURSOR signal of the CRT Controller, which goes high to indicate a valid cursor address in registers 1Ø and 11 of the Controller.

LIGHT PEN is the light pen input to the CRT Controller, which acts as described in the Ins section.

$\overline{\text{EXP}}$ is bit 5 of Port B of the PPI. This bit is identified as 'Reserved' in the documentation, and is normally set to the low state. Making use of this line involves reference to the C' register, which holds the state of the other Port B bits, and needs to be considered with care.

$\overline{\text{BUS RESET}}$ is pulled high through a 2K2 resistor. If it is grounded a complete reset, as at switch-on, will occur.

SOUND is connected via 10K resistors to the three tone outputs of the sound generator.

The CLOCK signal is at 4 MHz.

The above data is given in good faith, but it is based on rather scanty information, and needs to be used with care.

The Outs

THE OUTS

General Principles

Since you have access to almost all the central processor lines, there is virtually no limit, in theory, to the variety of external equipment that can be tacked on to the CPC464. The first stage will always involve the creation of a parallel interface, which, in turn, can be made to drive other devices, or be driven by them. The transfer of input and output data is the key process, but it may need to be supported by the passage of control data to set modes, ensure correct synchronism, sense external conditions.

Once the parallel interface is established, the data it handles can be converted to or from serial data. Similarly, conversion to or from analogue data is possible, opening up an entirely new range of possibilities. Finally, the CPC464 allows scope for program extensions, but this is a rather more complex matter. Information regarding it will be given, but a complete treatment would fill more than one book of this size.

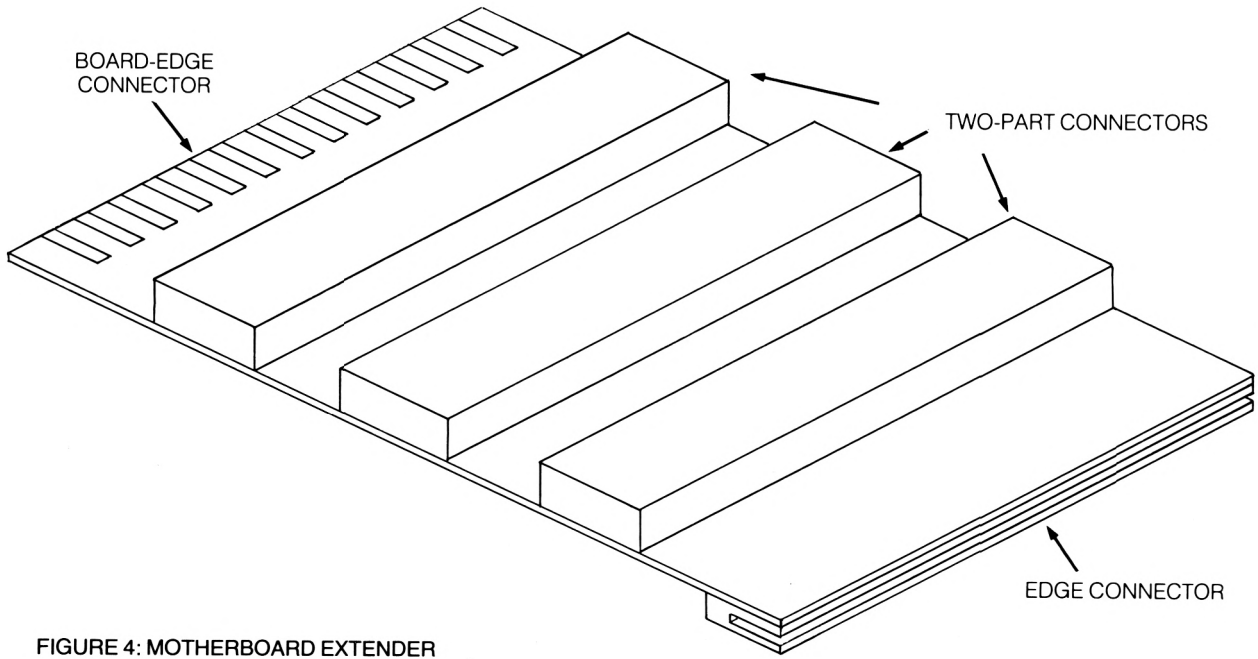


FIGURE 4: MOTHERBOARD EXTENDER

It must be noted that the hardware extensions must share facilities with the disc system and other AMSTRAD extensions. It is therefore worth using a 'motherboard' system, the parent unit plugging into the external interface, providing facilities for connecting the disc system as well as other peripherals. The scheme shown in Fig 4 could provide a convenient answer. The motherboard is fitted with two-part connectors into which daughter boards can be plugged to stand vertically, while the outer end of the motherboard mirrors the original external connector of the CPC464.

Avoid the temptation to work in 'bird's nest' fashion, with wires running here and there at random, but be equally careful not to use closely bunched cables. The signals you are dealing with have sharp edges, and can easily set up crosstalk in adjacent lines. Ribbon cable is useful, especially if alternate wires are earthed, and the twisted-pair form is particularly valuable — if you can find it.

It is important to watch the loading on the interface lines, not least on the 5v power supply. Where possible, load limits have been specified, but not all are known with certainty. Buffering is advisable, but remember that buffers introduce time delays, which can upset the action of critical systems.

British constructors will probably turn automatically to the Veroboard system of construction, but it appears that this system is not as widely known in other parts of the world. Briefly, it provides pre-punched insulating boards, the holes typically at Ø.1″ pitch, with copper overlay strips connecting rows of holes. The strips can easily be divided into shorter lengths by cutting away the copper. There are also specialised boards with hole and strip patterns particularly suited to integrated circuit systems.

Details can be obtained from Vero Electronics Ltd, Industrial Estate, Chandler's Ford, Hampshire, U.K.

Parallel Interfaces

Interface Rules

In the final analysis, all interfaces between a computer and the outside world are initially parallel in essence. Conversion to or

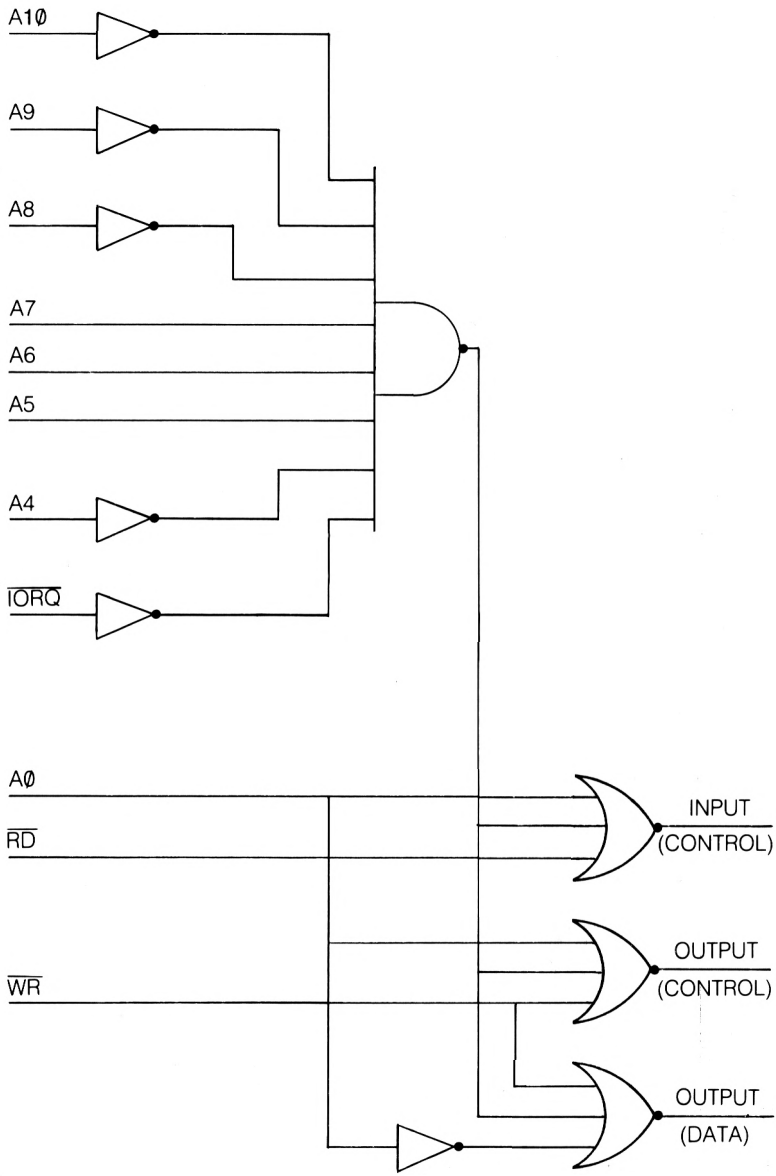


FIGURE 5: CONTROL FOR PARALLEL PORT

from serial or analogue form is a matter for external circuitry. The computer will only talk in parallel terms.

The exception to this rule arises when a special serial adaptor is fitted as an internal part of the computer. In the case of the CPC464, such an interface adaptor is envisaged as an external extension.

For data output, the state of the data lines DØ-D7 must be copied into a set of latches, since the data is only available briefly on these lines. Copying must occur when the following conditions are satisfied:

*The relevant I/O address is recognised as being present on the lines AØ-A15, though it may not be necessary to take all the lines into consideration.

*The $\overline{\text{IORQ}}$ line is low, indicating an I/O transfer.

*The $\overline{\text{WR}}$ line is low, indicating an output (write) transfer.

For data input, the data sources must normally allow the data lines to 'float', and this requires the use of a 'tri-state' source device, which can pull the lines high, pull them low, or exert no influence at all on them, showing a high impedance. Only one such source may be taken out of the high impedance state at a given time, the required conditions being:

*The relevant I/O address is recognised.

*The $\overline{\text{IORQ}}$ line is low.

*The $\overline{\text{RD}}$ line is low.

As noted earlier, there are rules limiting the choice of address for user peripherals. All permissible addresses have bit A1Ø low, which can be used to simplify decoding. Suppose, for instance, that it is decided to use addresses F8EØ and F8E1, the first for input and output of control data and the second for data output. The circuit shown in Fig 5 will generate the required control signals.

The eight-input NAND gate is supplied with the signals $\overline{\text{IORQ}}$, A4, A5, A6, A7, A8, A9, A1Ø. Its output will go low when an I/O address of the form XXXX XØØØ 111Ø XXXX is generated. If the address limitation rules are observed, this will be in the range F8EØ-F8EF.

The NAND gate output is taken to three three-input NOR gates. The first is also fed by AØ and $\overline{RD}$. Its output will be high if the NAND gate output is low, AØ is low, and $\overline{RD}$ is low. It will therefore signal an input for an even-numbered address in the range stated above. This includes F8EØ, but also covers F8E2, F8E4, etc. If these other addresses were to be used, additional decoding would be necessary.

The second NOR gate receives the NAND gate output, AØ, and $\overline{WR}$. It will give a high output for an even-numbered address output transfer. The addresses which satisfy this are as listed above.

The third NOR gate receives the NAND gate output, AØ, and WR. Its output will go low for an output transfer on an odd address in the F8E1-F8EF range.

The three NOR gate outputs control the remainder of the I/O system.

An Alternative Printer Port

To illustrate how the above circuit is put to use, let us consider the provision of a full eight-bit printer port. We will need an eight bit latch to accept output data and drive the printer. We will need a single line control output to generate the strobe signal, which tells the printer to take the data. We will need a single-line control input to sense the state of the BUSY signal, which warns that the printer is unable, for the moment, to accept further data.

Some systems use the ACK signal instead of BUSY, and a comment on this is needed. BUSY goes high when the printer accepts an input, and remains high until a further input can be accepted. It may remain high while a line of stored data is printed, or just long enough for a single character code to be stored.

ACK, on the other hand, is a low-going pulse a few micro-seconds long, with its trailing edge coinciding with the transition of the BUSY signal from high to low. It would be difficult to be sure of seeing this pulse in a read transfer, so it must be used to clear a bistable set by $\overline{STROBE}$, so generating a local form of BUSY.

Quite apart from this, the use of BUSY allows detection of a situation in which the printer is not fitted, in which case the BUSY

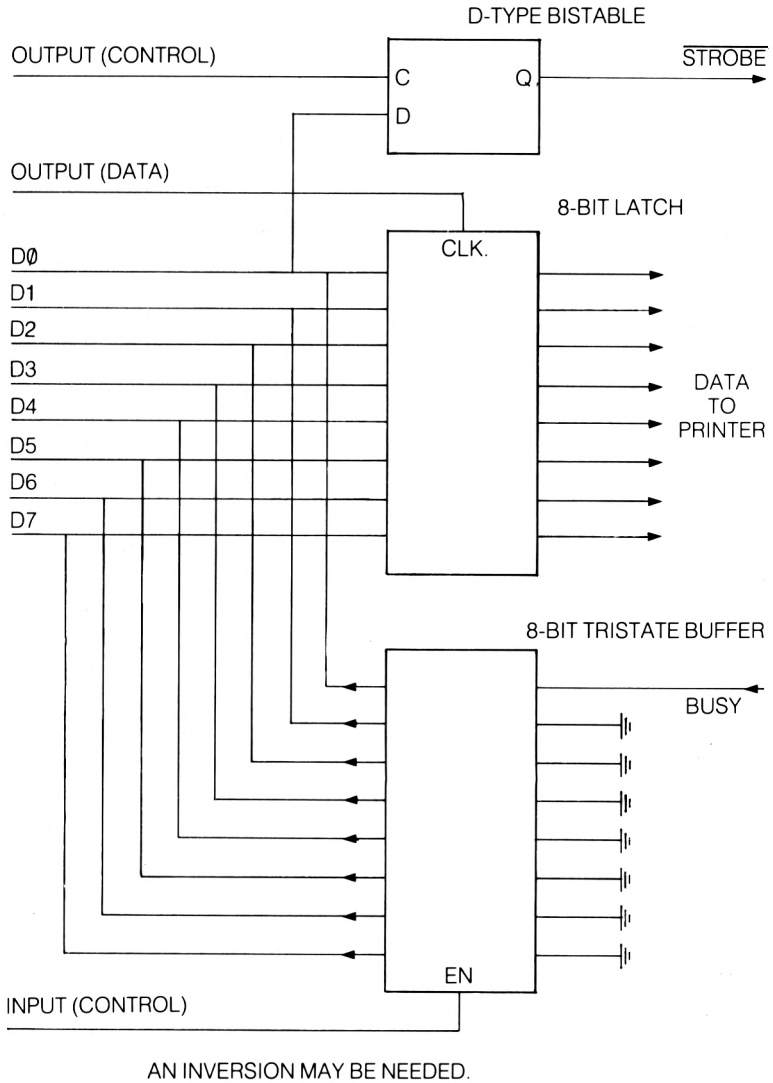


FIGURE 6: PARALLEL PRINTER PORT

line will be high continuously. To discover this, the BUSY line must be checked at a time when no strobe has been output for several seconds.

It might appear that the control output signal from the second NOR gate could be inverted to provide $\overline{\text{STROBE}}$, and this method has been used successfully, but the $\overline{\text{STROBE}}$ pulse then lasts no more than half a microsecond, and that is too short for some printers. The NOR gate output must set a bistable to a state determined by a data line.

The control input signal, on the other hand, must be used to enable an element with 'tristate' output. Such elements are not easy to find in single-channel form, so the added circuit shown in Fig 6 uses an eight-bit element. This can be useful in providing for other input control lines, and control output can similarly be made eight bits wide for the same reason.

It should be noted that some printers provide extra output lines signalling their internal state, and also accept additional inputs which control that state. The system shown will accommodate this, though additional software support will be needed to handle the additional data.

Software Support

The original printer firmware of the CPC464 is of no use where this external printer port is concerned, since it is specialised to the port allocations of the internal hardware. A set of revised routines must be provided, and the jumpblock entries must be changed to access those routines. There are five calls relevant to the printer, and while one would serve for normal purposes it is best to retain the layout of the original, so that the calls remain valid. The five calls may be summarised thus:

BD2B: MC PRINT CHAR: Calls MC WAIT PRINTER with BC preserved.

BDF1: MC WAIT PRINTER: Enters MC SEND PRINTER if MC BUSY PRINTER returns with carry clear.

BD31: MC SEND PRINTER: Sends a byte to the printer.

BD2E: MC BUSY PRINTER: Returns with carry clear if BUSY is low.

BD28: MC RESET PRINTER: Resets the jumpblock entry for MC WAIT PRINTER.

The required routines are defined here in source code. The question of their position in store will be examined later.

```
MPC     PUSH    BC          ;MC PRINT CHAR
        CALL    MWP         ;Call MC WRITE PRINTER
        POP     BC
        RET

MWP     LD      BC,&0032    ;MC WRITE PRINTER. Set timeout
                            count.
L1      CALL    MBP         ;Call MC BUSY PRINTER
        JR      NC,MSP      ;If BUSY low, output data.
        DJNZ    L1          ;Loop up to 50 times.
        DEC     C
        JR      NZ,L1       ;Repeat 256 times.
        OR      A           ;Clear carry: Action failed on
                            timeout.
        RET

MSP     PUSH    BC          ;MC SEND PRINTER.
        LD      BC,&F8E1    ;Data address
        OUT     (C),A       ;Set up data
        DEC     BC          ;Control address
        XOR     A
        OUT     (C),A       ;Set STROBE
        LD      A,1
        OUT     (C),A       ;Terminate STROBE
        POP     BC
        SCF                 ;Set carry: Action succeeded.
        RET

MBP     PUSH    BC          ;MC BUSY PRINTER
        LD      BC,&F8EØ    ;Control address
        IN      C,(C)       ;Read BUSY
        RR      C           ;Bit Ø to carry
        POP     BC
        RET
```

```
RES   LD    A,&C3        ;This routine resets the jumpblock
      LD    (&BD2B),A
      LD    HL,MPC
      LD    (&BD2C),HL
      LD    (&BDF1),A
      LD    HL,MWP
      LD    (&BDF2),HL
      LD    (&BD31),A
      LD    HL,MSP
      LD    (&BD32),HL
      LD    (&BD2E),A
      LD    HL,MBP
      LD    (&BD2F),HL
      RET
```

Note that the entries made are straight jumps, rather than the &CF calls normally used for these links. The routines must be in the 'Memory Pool', not under the ROMs, so the ROM selection state is irrelevant.

But where should the programs be stored? BASIC may use any location above A3FF, and will need to go that far down only if SYMBOL AFTER Ø has been called, copying all character patterns into RAM. If we enter MEMORY &AØØØ, we will have reserved 1K of RAM in a protected state. This is important, because all locations from LOMEM to HIMEM are cleared when a BASIC program is loaded or NEW is called.

There is no need to reserve a full kilobyte for the programs we are concerned with at the moment, but the reserved area can be set as we wish, the call MEMORY &AØØØ being merely an example.

By the way, once you have reset the upper memory limit, you will be unable to use SYMBOL AFTER, as the character pattern copies would then be split, instead of being continguous.

One other reservation is necessary. If any 'sideways ROMs' are fitted, the store area committed to other purposes may change. To be strictly correct, the revised printer driver routines should be relocatable, to allow for this. However, it will usually be possible to put them in a predictable position.

Communicating Computers

There are occasions when it would be convenient to be able to pass data from one computer to another. If the computers are of the same type, transmission via cassette tape may be convenient, and in the early days of personal computers, this was seen as the only way for computers — even computers of different types — to talk to each other, despite the need for a lot of hardware and software to achieve compatibility.

More recently, serial transmission has become popular for this purpose, but it can be seen as the older tape method without the tape. There may still be problems of incompatibility of baud rate and data format.

Parallel transmission was shunned because it involved direct connections between the communicating computers, and that was felt to be risky. The advent of opto-isolators has removed that worry, and it is possible to arrange relatively simple transfer of data between totally different computers, given that the receiving computer has a suitable input interface.

Opto-isolators consist of a light-emitting diode and a photocell mounted in a single unit. The input to them lights the diode, and the photocell provides an output in response, yet there is absolutely no connection between input and output. One opto-isolator is needed in each data line and each control line.

The circuit for the parallel input interface is almost identical with that for the parallel output interface, but differs in the following ways:

*The output data latch is replaced by a tri-state buffer similar to that used in the output interface for control input.

*A bistable must be added to generate the BUSY signal. It is set by the incoming $\overline{\text{STROBE}}$ signal and cleared by a pulse generated by the controlling software.

*The software must be revised to store data, rather than read it, and to produce the ACK pulse when storage is complete.

Rather than enlarging on what must be reasonably obvious, we must turn from the hardware to consideration of the software implications.

If the source computer has a facility for generating an ASCII listing of a BASIC program, the resulting data could be applied to the destination computer as if it came from the keyboard. It would not be possible to run the program so transferred without some adjustment. A program transferred from a BBC Computer to a CPC464 would probably need to have a lot of spaces inserted, and the sound and graphics commands would need to be changed, but that would be a lot less tedious than transferring the program by hand.

Transfer of data is usually straightforward, but machine code is only transferable if the computers use the same processor. It might be possible, in theory, to convert code for one processor to matching code for a different processor, but the converter might leave little room for anything else. Emulators, which execute 'foreign' machine code, are a different matter, but they are rather slow.

So transferring the data from one computer to another is not the whole story. Once transferred, conversion of some sort is almost certainly necessary to put the code or data into intelligible form.

But we are in danger of straying into an area of great complexity. Suffice it to say that the idea of transferring programs between different computers is not as impossible as it may seem. Those who wish to experiment in this area may care to reflect that an intermediate jumpblock might be used to convert operating system entries to match the 'foreign' code . . .

Serial Interfaces

A serial interface caters for a need to send data at a limited rate over a single pair of wires. Parallel transmission is faster and simpler, but the multiple wires it entails may be inconvenient.

Serial interfaces can take various forms, but it is usual nowadays to transmit data in separate bytes, rather than on a continuous basis, since this allows the elimination of a transmitted clock. The system relies on separate clock generators at the transmitting and receiving ends of the line, and these must generate the same frequency to within one or two percent. For each byte transmission, there is an added 'start bit', which is used to bring the clock at the receiving end into temporary synchronism with the transmitter clock, and this synchronism lasts long enough to allow proper clocking of the subsequent data bits. There may also be a 'stop bit', which serves as a buffer between one byte transmission and the next.

The actual clock rate may, in theory, be anything up to about 9000 Hz or so, perhaps up to 20,000 Hz in favourable conditions, but there are preferred values at 110, 150, 300, 1200, 2400, 4800, 9600 and 19,200 Hz. (The 110 Hz case is a left-over from mechanical teleprinters, which are still with us in odd corners.)

Serial interface converters come in two main varieties. There are the slave types, which need the help of a computer to function properly, and the master types, which make all their own decisions. The slave types have to be set up by computer outputs, and can be made to perform in a variety of formats and baud rates, whereas the master type have these options preset. If there is no computer at one of the communicating stations, a master component must be used there.

The CPC464 has no built-in serial interface, but an external interface is planned. This will have its own controlling ROM, and hence raises no problems of associated software. Since the creation of a similar system would entail a great many difficulties, it is suggested that no more than a dual master system be considered for a DIY project. The details would depend on the component chosen.

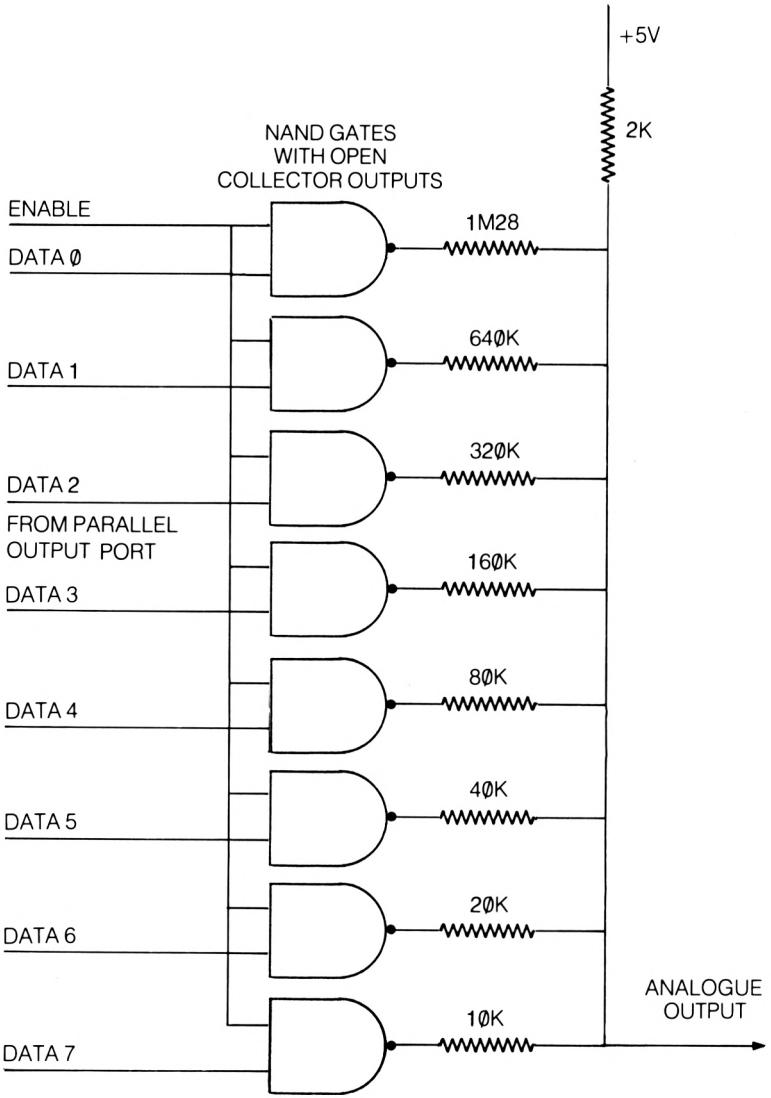


FIGURE 7: SIMPLE DIGITAL TO ANALOGUE CONVERTER

Analogue Interfaces

The simplest way of generating an analogue output from a computer system is shown in Fig 7. This has been used with great success in the generation of sounds from stored waveform data. Indeed, the flexibility of such a system is considerable, and the tone quality can be much superior to that generated by square waves.

In such an application there is no need for precise accuracy in the relation between data output and the voltage it produces. A little deviation may actually improve the sound quality! The method used to generate the data involves setting up a table of values representing instantaneous levels within one cycle. The table can be established by a short BASIC program, and may be a sinewave or may include harmonics.

The different pitches are obtained by scanning through the table at differing rates. The actual sample rate can remain constant, but the increments of the pointer can vary. If the whole table is scanned in 1/440th of a second, the note produced is the A above middle C.

By using multiple pointers, and adding the results, it is possible to generate four-part harmony. However, as with all music generators, the chore of setting up the necessary controlling data is a deterrent, though the result can be surprisingly rewarding.

This system was pioneered by one Howard Arrington, of Boise, Idaho, and its full ramifications cannot be pursued here.

For a more precise output, the arrangement in Fig 8 may be used. This is a 'ladder' system, and has the advantage that the resistors have only two values.

Analogue input is a different matter. In concept, it is possible to turn an analogue output system round, so to speak, driving the digital side from a counter and stopping when the generated voltage matches the input voltage, but this is rather slow and crude. At the opposite extreme there are proprietary components which can give a fabulous performance — at a cost — and in between there are many different levels of price and performance. Most devices will need a certain amount of software support, especially those which serve multiple channels.

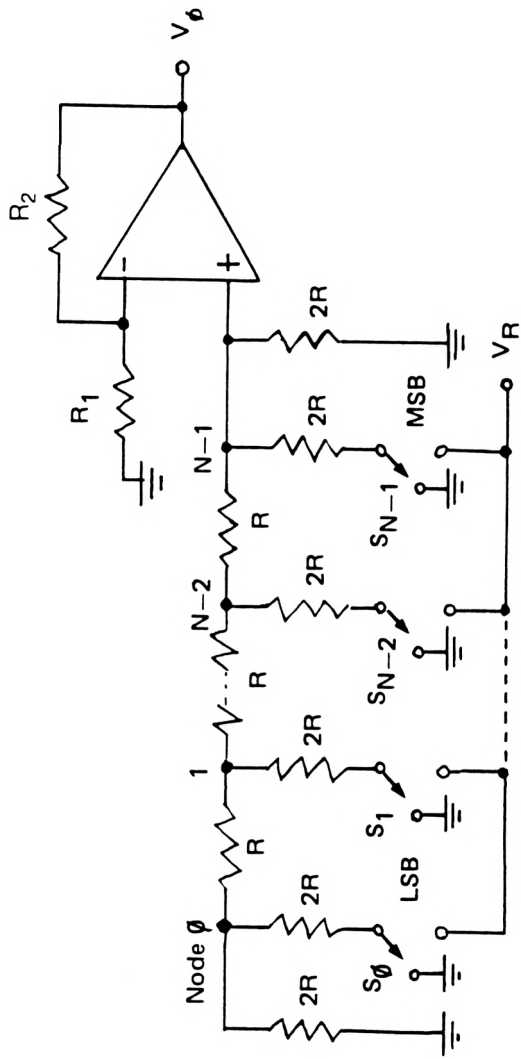


FIGURE 8: D/A CONVERTER USING AN R,2R LADDER

A simple demonstration of the operation of an analogue to digital converter is provided by the program below:

```
100  INPUT D%
110  A%=128
120  B%=0
130  FOR X%+1 TO 8
140  B%=B%+A%
150  C%=D%-B%
160  IF C% < 0 THEN B%=B% XOR A%
170  PRINT A%;TAB(6);B%;TAB(12);C%
180  A%=A%/2
190  NEXT
200  PRINT B%
210  GOTO 100
```

Input a number in the 1 to 255 range, and you will see that B% is built up to equal it.

The program can be adapted to perform an actual analogue input. Delete lines 100 and 160, and add lines as follows:

160 Output B% to a port driving a simple digital to analogue converter.

163 Input a bit from a port, the bit depending on the state of a comparator driven on one side by the D/A converter output, and on the other side by the voltage to be checked. Take the input as C%, which is 0 if the external voltage is lower than the D/A converter output.

166 IF C = 0 THEN B%=B% XOR A%

The details have been left general: The hardware has all been defined, and you may want to choose your own port addresses.

This arrangement is faster than the simple counter approach, which may require 256 comparisons, instead of 8. It is advisable to check that the D/A converter is reasonably linear in its output,

otherwise there may be some odd results. To check linearity, the simplest method is to output the numbers Ø to 255 to the converter, pausing on each number. A meter connected to the converter output should show a steadily increasing reading, without hesitations or set backs.

To obtain the best results from analogue conversions in either direction, precision devices are really essential, but the above experiment will produce quite useful results, given a little care.

Sideways ROMs

It is almost inevitable that any external system added to the CPC464 will require some software support, and the overall system concept allows that support to be provided by ROMs that are mounted in unit with the rest of the external hardware. In the extreme, no less than 252 such ROMs could be added, but in practice the method is the same for one ROM or more.

First, each ROM is allocated a number, and output of that number on I/O channel &DFXX must enable the ROM. It will then replace the internal upper ROM as far as the system is concerned. Even external ROM Ø will do that, though the internal ROM is also, nominally, ROM Ø.

Once the external ROM is selected, the routines which it contains can be executed. It is possible to call or enter routines in other external ROMs, using the facilities provided in the 'RST area', such as FAR CALL.

Since programs held in external ROMs will need workspace, each ROM should contain an initialisation program, which is called at power on or system reset by the main operating system. This program will define an amount of RAM space which it requires, and the RAM limit pointers are adjusted automatically.

A program held in an external ROM can have its own set of command words, which form an extension of the normal reserved words, and which will be implemented by a call to a particular routine.

The external ROMs must conform to a specified format, and their use involves a certain degree of complexity. The details

which follow are as complete as possible, but some grey areas remain in the available data. However, it is believed that all the key points have been covered.

The main difficulty in implementing the extensions may well be the creation of the necessary ROMs. However, it is not impossible to use RAM, perhaps loaded by external means and sustained by battery power.

ROM Types and Formats

ROM programs may be of the 'Foreground' or 'Background' type. Foreground programs are in overall control, while background programs supply supporting functions as called up by the foreground program currently in use. For example, a background program might provide additional sound control facilities in response to a call from the current foreground program, but would not be able to run a complete program on its own. (In some senses, the operating system can be seen as a background program, in that it provides services in response to requests from, say the BASIC interpreter, but it can also act as a foreground program in some circumstances.)

Most peripheral extensions would require a background program to support them. The program might be as simple as that given earlier for the alternative printer port, or might amount to a complete disc operating system.

Up to seven background ROMs may be used, and their reference numbers must lie in the range 1 to 7. External ROMs in general may have reference numbers in the range Ø to 251. The ROM numbers must be consecutive from Ø or 1 upwards. If the reference Ø is used, the on-board ROM will be available at the first reference number not used by external ROMs.

Up to four ROMs with consecutive reference numbers may be used to hold a single foreground program.

All of which may sound rather complicated, but the fog will clear in due course.

A given external ROM will occupy addresses CØØØ to FFFF, and may be up to 16K in size. The first six locations must be set up as follows:

C000: ROM type. 0 for a foreground ROM
1 for a background ROM
2 for an extension ROM.

(The onboard ROM is given type &80)

C001: ROM Mark number

C002: ROM Version number

C003: ROM Modification level.

C004/5: Address of external command table.

At C006 onward there must be a jumpblock, beginning with the entry to the initialisation routine and continuing with jumps that match the external command words.

The external command table must list the command words, with &80 added to the code for the last letter in each word. It is advisable to use upper case letters for command words.

An external command is distinguished by being prefaced by a vertical divide character. It is implemented by KL FIND COMMAND, at BCD4, which is entered with HL pointing to the command string, and returns with the ROM select address in C and the address of the required routine in HL. The command word table is terminated by '0'.

Use of the external command table is optional. The format used for the BASIC interpreter ROM is:

C000 80 01 00 00 4C C0

This defines the ROM as onboard; Mark 1; Version 0; Mod level 0, and places the command table at C04C.

At C04C:

C04C 42 41 53 49 C3 00 : BASIC

The final zero shows that there is only this one external command word, so instead of a jumpblock starting at C006 it is possible to start the initialisation routine at that point.

Some distinctions between foreground and background ROMs can now be examined in more detail.

At initial start-up, the peripherals and firmware managers are reset, and ROM 0 is then entered at C006 with HL = ABFF (top

of available RAM), DE = ØØ4Ø (bottom of available RAM) and BC = BØFF (highest usable byte). The stack pointer is reset to CØØØ.

The ROM Ø initialisation program reserves space by adding Ø12F to the value held in DE, moving the lower limit of free-use RAM up to Ø16F. (The stored BASIC program begins at Ø17Ø.) The new limits are stored in workspace.

As noted earlier, the upper limit of free RAM can be modified by the BASIC command MEMORY, and this protects an area of store in which routines can be held in RAM.

If MC START PROGRAM is called with HL holding ØØØØ, initialisation as above is repeated, but if the entry is called with HL holding any other value the ROM defined in the C register is entered at the address defined in HL, once again with BC, DE and HL holding the memory limits. The program entered must claim any workspace areas which it requires by modifying the contents of DE and/or HL appropriately, and storing them for reference.

Note that when the ROM entered by MC START PROGRAM returns, a full system reset occurs, so the onboard ROM is re-selected as at power-up.

The foreground program set up in this manner may choose to activate one or more background programs which it requires as support. It may do so by calling KL ROM WALK, which calls KM INIT BACK for all available background ROMs, or it may call INIT BACK for particular ROMs which it requires. The BASIC ROM calls KL ROM WALK, initialising any ROMs that may be available.

The background programs respond by modifying the available RAM limits held in DE and HL, so reserving areas which they can use for workspace. Now, the actual location of these areas may depend on the amount of memory reserved by the foreground program, so the background programs must note where the area begins, and access the area on a displacement basis. This can be done by using IX as a base. If IX is always set when the program is entered, then a given location in the workspace can be accessed by LD A,(IX+n), or similar instructions.

We thus have a system which allows the use of any one of a number of foreground ROMs, each of which may contain a

number of different programs called by external command words, plus up to seven background ROMs, which can supply support routines. Once we are in a foreground ROM we must stay there, except for excursions to background ROMs, because a return from a foreground ROM results in a general reset.

Applications

The possible ways of using the sideways ROM system are too numerous to examine in detail, but it may be useful to point out some ideas on the subject.

The provision of alternative languages, such as Pascal or FORTH, is a fairly obvious starting point, while other foreground programs might implement word processors or spreadsheets. A considerable advantage is that the use of on-board RAM can be minimal, only workspace being required.

As mentioned earlier, the use of battery-supported RAM instead of ROM would extend the scope even further, since large amounts of data could be stored externally, being brought into main RAM as and when required. There would be no need to implement 'virtual memory' techniques, which make a given amount of RAM look much larger than it really is, since the external ROM system effectively does much the same thing.

Background ROMs could be used to extend existing systems, offering a total of 112K of program space. It is probable, however, that some of this space will be pre-empted by official add-ons, such as a disc system or serial interface.

A word of caution is necessary. With such a complex system, some time may pass before it is fully explored and any hidden limitations are exposed. Some aspects of the system may yet change in minor ways. Very few 'bugs' have been discovered, and those that have emerged are trivial, such as the insertion of a line feed if a text line looks like overrunning the screen width. (This may be deliberate, but it is not popular!)

We are now far down the iceberg, and the image of the CPC464 as a simple BASIC machine is getting more and more remote. It can be used in that way, but it will do far more, given proper persuasion. The level of knowledge needed to take full advantage of this is far from trivial, and there is a lot to be studied.

External ROM Hardware

The actual hardware needed to implement an external ROM is not very complex. First, output addresses of the form DFXX must be recognised, and the associated data byte must be transferred to a latch. The latched data must be compared with the ROM select number for the ROM in question, and if the two match the ROM must be enabled, but only if the $\overline{\text{ROMEN}}$ line of the interface is low. The full address is available to select a ROM location, and a read transfer should be expected, with $\overline{\text{RD}}$ low.

The relevant address can be recognised with the aid of an eight-input NAND gate and an inverter. The latch can be a standard eight-bit latch. The ROM can be of almost any type of suitable size. Fig 9 sketches a possible configuration, using comparators to detect the ROM select number.

If RAM is used instead of ROM, it needs to be of the static type, to avoid the need for refresh. While refresh action could be arranged, it is an inconvienient complication.

There is no need, of course, to mount each ROM on a separate daughterboard. The address recognition elements and the latch could be common to more than one ROM, separate comparators being used to detect which ROM is required.

As in other areas covered in this book, it seems a pity to be too specific, since the possible variations are so vast. Merely breadboarding circuitry to try out some of the ideas that come to mind would take a very long time. Without that, it is only possible to indicate what those ideas are . . .

A Second Processor

The technique of using the BBC Computer as an 'intelligent terminal' serving an external second processor system has led to enquiries regarding the application of a similar technique to the CPC464. Some comments are therefore offered, though a complete response would be far too complex to include here.

First, the need to add a second processor is much reduced by the greater capabilities of the CPC464, by its extra RAM space, and by its use of the Z8Ø processor, which makes it directly

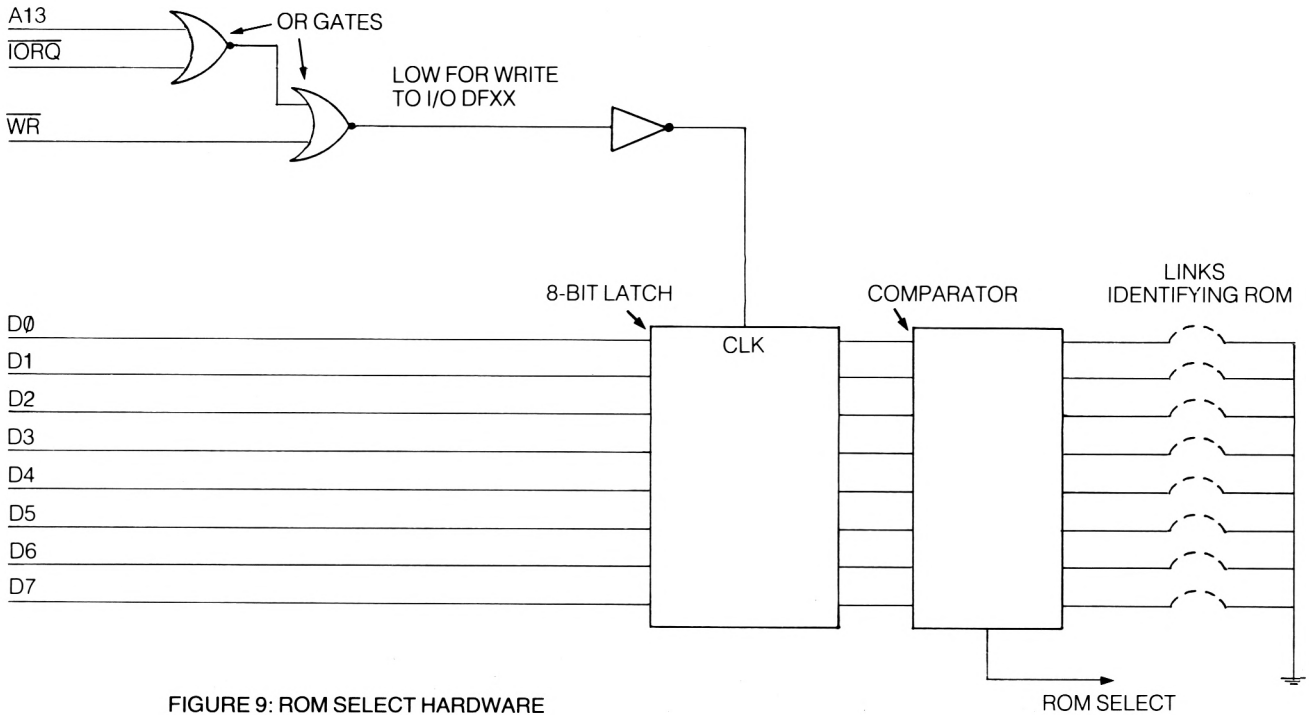


FIGURE 9: ROM SELECT HARDWARE

compatible with CP/M. However, it is possible to envisage situations in which it would be useful to carry on an external process simultaneously with that being run in the CPC464 itself. (The CPC464 can cope with 'concurrent' execution of two processes, alternating between them, but 'simultaneous' operation implies that both processes are running all the time.)

In any logical system, it is important to distinguish between a 'master' element and 'slave' elements. There can only be one master at a time, though the master role may be transferred from one element to another in particular system states. The master determines overall system action, exercising control over the slaves, which may ask for assistance by raising interrupts, but which otherwise must do what they are told.

By definition, any external processor using the CPC464 as an intelligent terminal must be the master, but if both the internal and external processors are working independently most of the time they can both be masters, until intercommunication is required. This is where the external processor must exert its authority, by raising an interrupt. That forces the internal processor to respond, and the response must take the form of a transfer of data. This could be executed by a background program, called by an 'event'. The external processor could pass data defining the required action.

The routine servicing the external processor interrupt would have to be brief, since normal internal action must be maintained.

An alternative approach, which may have advantages where only data is to be passed, is based on the $\overline{\text{BUSRQ}}$ and $\overline{\text{BUSAK}}$ lines. The external processor takes over the internal bus for one memory cycle, and passes a data byte to the internal memory. This can be repeated to send a complete 'message', the internal processor being allowed to continue execution between data transfers, while the external system picks up another data byte.

The message can be interpreted by a foreground program, and could instruct the display of text or a change in the action of the internal foreground program. This might be implemented in response to a synchronous event called by the foreground program, which would ensure that a partially-transferred message was not used prematurely.

Enough has been said to indicate the possibilities, and to show that appreciable effort would be needed to detail the hardware and software systems. A beginner would be in much the same case as an average motorist whose car has broken down miles from anywhere. He may have all the tools needed for a repair, but he lacks the detailed knowledge of how the tools should be used.

The CPC464 provides a vast array of tools. Learning to use them properly could take time.

Overview

The CPC464 has been revealed as a wolf in sheep's clothing. To the superficial observer, it looks like another simple games machine, though with some rather nice characteristics. Exploration reveals more and more that is hidden under that innocent exterior.

Making use of all that has been discovered is not a simple matter. Involvement with machine code is virtually essential, and even a simple external add-on may call for careful hardware design.

However, it is a machine which will serve a developing user well. It will support him as he learns to walk, yet keep pace when he aspires to run. As his confidence grows, it is expected that machine enhancements will appear, mainly as external additions, and these will help him on his way.

BIBLIOGRAPHY

Programming the Z80, Rodney Zaks (Sybex Inc.)
Amstrad BASIC, AMSOFT
Amstrad CPC464 Firmware, AMSOFT
Amstrad DEVPAC for CPC464, AMSOFT

Enhanced Dump Program for print or display, ROM or RAM

```
10Ø GOSUB 37Ø
11Ø CLS
12Ø PRINT "MODE Ø,Display ROM
13Ø PRINT "MODE 1,Print ROM
14Ø PRINT "MODE 2.Display RAM
15Ø PRINT "MODE 3,Print RAM
16Ø INPUT "MODE";Y
17Ø X=Y MOD 2
18Ø Z=Y\2
19Ø INPUT "Start Address";A
2ØØ A=A-65536*(A<Ø)
21Ø NL%=A-8*INT(A/8)
22Ø PRINT #(X*8),HEX$(A,4);
23Ø ON Z+1 GOSUB 31Ø,46Ø
24Ø PRINT #(X*8),TAB(6+3*NL%);HEX$(B,2);
25Ø A=A+1
26Ø NL%=NL%+1
27Ø IF NL%<8*(X+1) THEN 23Ø
28Ø NL%=Ø
29Ø PRINT #(X*8)
3ØØ GOTO 22Ø
31Ø Q=INT(A/256)
32Ø POKE &7Ø19,Q
33Ø POKE &7Ø18,(A-256*Q)
34Ø CALL &7ØØØ
35Ø B=PEEK(&7Ø2Ø)
36Ø RETURN
37Ø FOR X=&7ØØØ TO &7Ø12
38Ø READ Y
39Ø POKE X,Y
4ØØ NEXT
41Ø RETURN
42Ø DATA &2A,&18,&7Ø,&CD,&ØØ,&B9
43Ø DATA &F5,&CD,&Ø6,&B9,&7E
44Ø DATA &32,&2Ø,&7Ø,&F1,&CD,&ØC
45Ø DATA &B9,&C9
46Ø B=PEEK(A):RETURN
```

Index

Ins and Outs of the Amstrad CPC 464

Customer Registration Card

Please fill out this page (or a photocopy of it) and return it so that we may keep you informed of new books, software and special offers. Post to the appropriate address on the back.

Date .....19....

Name .....

Street & No. .....

City .....Postcode/Zipcode .....

Model of computer owned .....

Where did you learn of this book:

☐ FRIEND ☐ RETAIL SHOP

☐ MAGAZINE (give name) .....

☐ OTHER (specify) .....

Age? ☐ 10-15 ☐ 16-19 ☐ 20-24 ☐ 25 and over

How would you rate this book?

QUALITY: ☐ Excellent ☐ Good ☐ Poor

VALUE: ☐ Overpriced ☐ Good ☐ Underpriced

What other books and software would you like to see produced for your computer?

.....

.....

.....

EDITION 7 6 5 4 3 2 1

Melbourne House addresses

Put this Registration Card (or photocopy) in an envelope and post it to the appropriate address:

United Kingdom

Melbourne House (Publishers) Ltd
Castle Yard House
Castle Yard
Richmond, TW10 6TF

United States of America

Melbourne House Software Inc.
347 Reedwood Drive
Nashville TN 37217

Australia and New Zealand

Melbourne House (Australia) Pty Ltd
2nd Floor, 70 Park Street
South Melbourne, Victoria 3205

Amstrad CPC 464

Melbourne House

The AMSTRAD CPC464 is a unique and very special computer in very many ways.

One of the most important features of this computer is the ease with which all major software functions can be accessed by simple calls to the operating system. This means that you can make the maximum use of your Amstrad computer whether you are a first-time user or a professional programmer.

Don Thomasson explores all this in the INs chapters of this book in a clear, well-structured manner that will allow you to write more powerful professional programs. Features such as screen plotting, cassette input/output and so on, can be reduced to simple calls to subroutines.

Other unique features of the Amstrad CPC464 are its abilities to interface with the outside world, and the option of external ROMs which vastly expand the potential of this computer. Don Thomasson details all of this in the OUTs chapters of this book.

A comprehensive description of how to add external devices to the Amstrad CPC464 through the use of the expansion and printer ports is given. Hardware enthusiasts will find this book indispensible.

INS AND OUTS OF THE AMSTRAD CPC464 shows you how to best explore that part of the Amstrad computer which is otherwise hidden.

Melbourne House Publishers

ISBN 0-86161-190-X

9 780861 611904

AMSTRAD CPC

MÉMOIRE ÉCRITE
MEMORY ENGRAVED
MEMORIA ESCRITA

https://acpc.me/