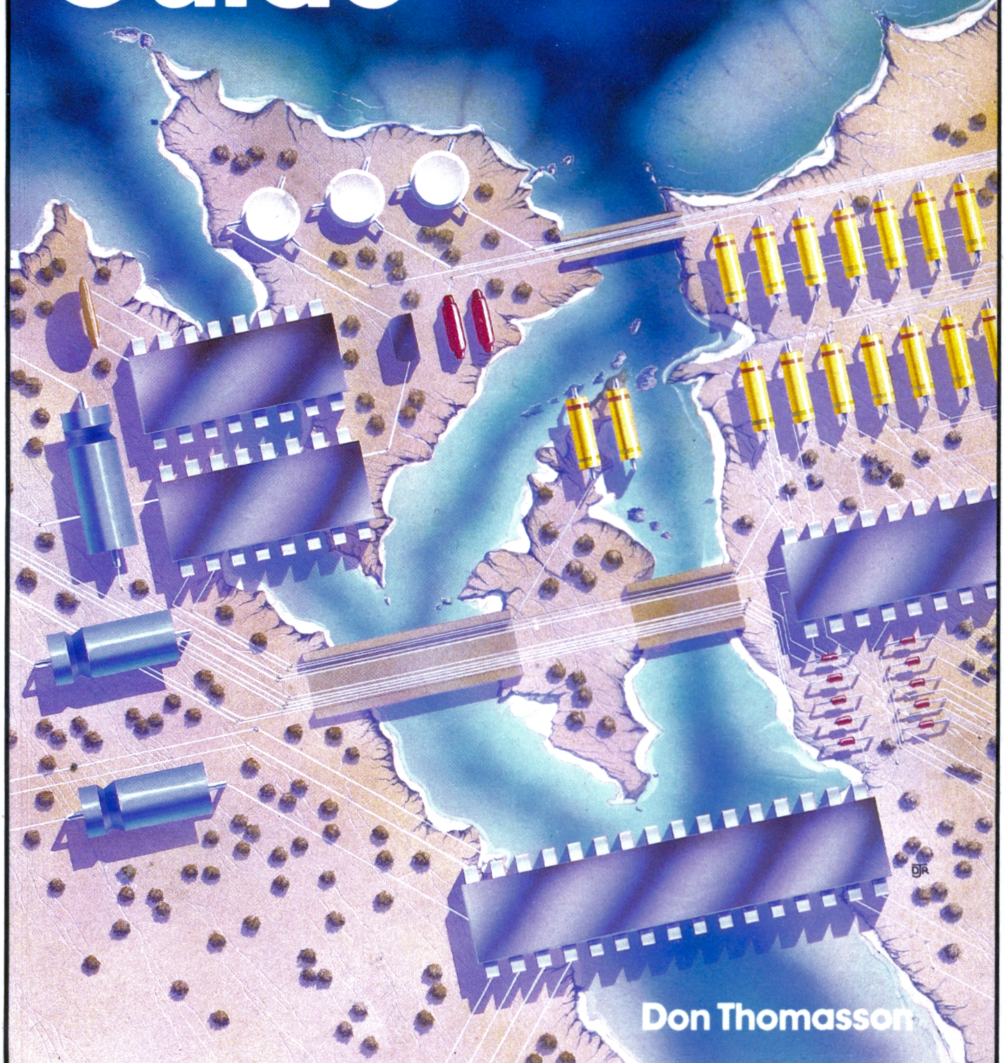
Amstrad CPC 464 Whole Memory Guide



Don Thomasson

Amstrad CPC464 Whole Memory Guide

Don Thomasson

MELBOURNE HOUSE
PUBLISHERS

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

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

ISBN 0 86161 199 3

Printed and Bound by Short Run Press Ltd, Exeter

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

Contents

Introduction

The Amstrad/Arnold/Schneider CPC464 is a fascinating machine in many ways, but it can be infuriating if you lack some of the essential items of information regarding its inner workings. Even with a complete set of official documentation, which can run to several large volumes, there may be points that remain obscure.

The operating system, for example, has more than 25Ø entry points, each related to a specific function, but some fifty of these entries are not formally defined, because they are primarily intended as extensions of the BASIC interpreter. Some of the other entry points are defined in such a way that their function is not immediately obvious. Broader, less formal explanations are needed to complete the picture, and this book seeks to meet that need.

A fully detailed analysis of the operating system would be very long and tedious, and might still fail to provide answers to all the questions that are likely to arise. What is attempted is an analysis of the more important functions, the rest being covered by shorter descriptions.

It is assumed that the reader has some knowledge of machine code. The inclusion of a complete tutorial on Z8Ø programming would leave little or no room for anything else. For those who need such help, the well-known book "Programming the Z8Ø", by Rodnay Zaks, is recommended. However, a study of the various operating system routines in relation to the descriptions given hereafter may prove enlightening, even to the merest tyro.

A key difficulty in this connection is that some disassemblers will only access code in ROM. A program given in the Appendix provides a solution, since it will work from ROM-borne code, while another program in the Appendix provides convenient means for calling the various functional routines and checking their action.

The book is based on version 1.Ø of the ROMs, but the comments can largely be applied to other versions by working from the jumpblock entry points, which should remain at the addresses given, even though they access different entry points in the ROM code.

Chapter 1
GENERAL SYSTEM ARRANGEMENT

Superficially, the CPC464 is a typical Z8Ø-based system, with an unusually economical arrangement of peripheral devices. By making full use of the capabilities of these devices, a performance level has been obtained which is higher than the limited chip count might suggest. One consequence of this is that the operating system is especially complex, a fact which is offset by the comparative ease of user access to the various functions. The word 'comparative' is necessary, because knowledge of machine code is needed, which may be a difficulty for some users, but once they have come to terms with machine code a wide range of possibilities opens up.

Among other ingenuities, the way in which a minimum of 96K of memory has been packed into a 64K memory map is especially noteworthy, and this aspect of the system will be studied first.

The Memory Map

The whole of the 64K byte memory is occupied by RAM, to which any writes to memory will be directed. This makes sense, since there is no point in trying to write to ROM. Reads from addresses in the middle half of memory will also access RAM, there being no ROM in this area. For addresses in the top and bottom quarters of memory, however, both ROM and RAM are present, and it is possible to read from either at will. A BASIC peek will always access RAM, so a special bit of machine code is needed to obtain the contents of ROM.

The memory arrangement is complicated by the fact that the top quarter of RAM is dedicated to use as screen memory, and must be immediately accessible at regular intervals while data is being passed to the display. For this purpose, two bytes are read every microsecond.

The processor is put into a wait state while the pairs of bytes are being transferred, the transfer being made directly from memory to the Video Gate Array, using an address generated by the CRT Controller chip. This means that the main processor can only make one memory access per microsecond, and although its clock runs at 4 MHz the actual processing speed is slightly reduced, a point to watch when calculating execution times.

The Video Gate Array handles the switching between ROM and RAM for this purpose, so it is natural that it is also used to control ROM selection in general. The instructions for switching between ROM and RAM are given by outputs to bits 2 and 3 of port 7FXX. A 1 disables, a Ø enables, while bit 2 applies to the lower ROM and bit 3 to the upper ROM. Incidentally, there is only one ROM component, some address fiddling dividing it into two 16K blocks as far as the system is concerned.

As in any bank-switching memory system, the key problem is the need to jump and switch banks simultaneously, or to appear to do so. The CPC464 achieves this by using routines held in central RAM. These are always accessible, whatever the ROM selection state. In addition to simple switching between ROM and RAM, these routines allow the selection of alternative upper ROMs, extending the available memory still further. In the extreme, it would nominally be possible to address a total of 4128K bytes of memory, but few systems are likely to approach that ultimate limit.

The complexities of the memory system can be evaded by putting machine code into the central half of the memory map, which contains only RAM, but this is neither essential nor always feasible.

The I/O Map

The selection of peripheral channels is largely determined by making one of the bits of the upper byte of the 16-bit I/O address low, which means that the older I/O instructions of the form IN A,(N) and OUT (N),A cannot be used, because they draw the upper byte from the contents of the A register. Instructions which set the I/O address from the contents of the BC register are mandatory, and there are strict limits regarding the contents of the B register, because no more than one of the six upper bits may be low in any given address. (Making more than one of these bits low in an input instruction invites physical damage, because two data sources may fight for control of the bus, while it is rarely sensible to send the same output to two different ports at the same time.)

The I/O addresses can be summed up as follows:

* If address bit A15 is low, the Video Gate Array is selected. This port is for output only. The address must be 7FXX.

* If address bit A14 is low, the CRT Controller is selected. Address bits A8 and A9 are used to select four different transfer modes;

BCXX	Output to Register Select
BDXX	Data Output
BEXX	Status Input
BFXX	Data Input

* If address bit A13 is low, ROM select data is being output. The address must be DFXX.

* If address bit A12 is low, the printer channel is selected for output only. The address must be EFXX.

* If address bit A11 is low, the Parallel Peripheral Interface (PPI) is selected. Here again, bits A8 and A9 are used to select four sub-channels;

F4XX	Port A (I/O)
F5XX	Port B (I/O)
F6XX	Port C (I/O)
F7XX	Control (Output only)

* If address bit A1Ø is low, an expansion channel is selected. In this case, bits A5 - A7 have special significance;

A5 low selects a communication channel.
A6 low selects a reserved function.
A7 low selects the disc system.

* Address F8FF is a general reset for expansion channels.

The above allocations restrict the user to the following address ranges for any special I/O functions he may require;

F8EØ-F8FE: F9EØ-F9FF: FAEØ-FAFF: FBEØ-FBFF

Outer Peripherals

The devices mentioned above are the 'inner peripherals', which are accessed directly from the main processor. Further devices, classed as the 'Outer Peripherals', are accessed by the inner peripherals. They include the Programmable Sound Generator, accessed by the PPI; the Keyboard, accessed by the PPI and the

Sound Generator; the Cassette Recorder, accessed by the PPI; and the Loudspeaker, driven by the Sound Generator.

For further details of the hardware system, consult 'The Ins and Outs of the Amstrad', which gives additional information on the coding and action of these devices.

System States

At switch-on, a number of initialisation procedures are executed, and control then passes to upper ROM Ø. If there is no external ROM of this number, the internal BASIC interpreter takes charge as the 'foreground' program.

Once a foreground program has been entered, it remains in charge until a return at entry level is executed, when a full reset is performed, and ROM Ø is again put in charge. However, the foreground program can call on 'background' programs for assistance, and these, in turn, can call other programs. There is thus - nominally - one foreground level, but there can be several background levels.

A ROM other than Ø, or a program in RAM, can be selected as the foreground program. This can be done by a RUN " " command which reads a machine code program that has a defined start address, or by a machine code routine. It may be more convenient to leave the BASIC interpreter nominally in charge and run a program CALLed from BASIC as if it was a foreground program. This has the advantage that a full reset is not inevitable in response to a return at entry level. Instead, the interpreter is re-entered.

Using BASIC in this way has other advantages. HIMEM can be checked and adjusted quite easily, putting it below the area in which machine code is to reside, and other system variables can be set up. The BASIC program will use some RAM, particularly from Ø17Ø upwards, but this is likely to be a negligible drain on the large RAM area available.

One point to watch is that if extension systems are added, such as a disc drive, speech facility, or the MAXAM assembler in ROM form, HIMEM is lowered, because the extensions have claimed workspace for their own use. Some commercial programs are incompatible with a disc drive , because they trespass on the disc workspace. Protected or not, they cannot be transferred to disc.

As a guide, HIMEM is AB7F in typical circumstances, but drops to A67B with disc drives connected, and may go even lower with AMSDOS active.

The official advice is that machine code programs should be relocatable, but that is not always feasible. It has been noted, however, that it is possible to set short routines in the BFØØ area, and these survive a reset, which is useful....

Jumpblock Entries

The RAM area from BBØØ to BDC9 holds instructions accessing the principal operating system entries. Special jumps are used which select the required ROM automatically. Entries beginning &CF do this and no more, but entries beginning &EF also disable the relevant ROM when the routine returns. (See the section on 'The RST Area'.)

The jumpblock entries should be accessed by subroutine calls, so that a return address is available on the stack.

From BDCD to BDF3 the entries are simple jumps, beginning &C3. These are 'indirections', which do not enable the appropriate ROM and should only be called when it is known that the ROM is already selected.

The intention is that the jumpblock addresses should not change, though they may access different entry points with different system versions. However, for ease of reference each routine description is headed by the jumpblock address and the associated destination in the operating system. The latter will change with the system version, as in the CPC664, and the new entry points must be determined by checking the jumpblock instructions.

Summary

This quick tour of the main system features should have served as a useful introduction to the system. We must now begin to look more closely at detail, beginning with the routines held in RAM.

Conventions

Two-digit hexadecimal numbers will are prefaced by '&', four-digit hexadecimal numbers are not. Brackets round a four-digit number indicate the contents of the location at the address identified by the number. Where the brackets contain a range of numbers, e.g in the form (ØØFA/D), the joint contents of the locations specified are indicated.

Chapter 2
THE RAM ROUTINES

During initialisation, two areas of RAM are set up by copying from ROM. The result provides code which can be executed whether the ROMs are enabled or not, though that is only part of the story.

The area ØØØØ-ØØ3F is set from the corresponding ROM locations. This is the 'RST Area', which contains a number of special entry points that need to be effective at all times. An interesting aspect of this is that initial entry at switch-on is at ØØØØ, but at that time the RAM copy has not been set up, and the lower ROM must be entered. This is assured by hardware initialisation in the Video Gate array.

Some points within the RST Area can be accessed by the Z8Ø RST instructions, but the meaning of these has been changed in the CPC464 system, by making them call routines in the other RAM routine area, from B9ØØ to BAE8. This area serves a number of purposes;

```
  B9ØØ-B92Ø holds a jumpblock accessing routines in BA4A-BAB1
  B921-B938 holds KL POLL SYNCHRONOUS. (See Event Routines)
  B939-B97B holds the main interrupt handler.
  B97C-BA49 holds the routines implementing RST Area entries.
  BA4A-BAB1 holds ROM control and copy routines.
  BAB2-BAE8 holds RAM read routines.
```

To simplify explanation, the action of the routines in B97C-BAE8 will be described first in functional terms, the coding being examined in detail later, for those who want to know more about their operation.

The RST Area

The Z8Ø RST instructions take the form &C7+X, performing a subroutine call to location X, a return address being left on the stack. X can be Ø,8,&1Ø,&18,&2Ø,&28,&3Ø or &38.

The CPC464 system extends the meaning of these instructions by making them access RAM routines which modify their effect considerably;

RSTØØ (Code &C7) enters location ØØØØ, and - as at initial start-up - a complete reset is performed. The immediate code sets the Video Gate array by an output of &89 to 7FXX, then there is a jump to Ø58Ø to perform the remainder of the initialisation. (See Machine Pack.)

RSTØ8 (Code &CF) enters location ØØØ8, where there is a jump to B982 in the RAM routines. The two bytes which follow the &CF code are read as a 16-bit word, which is interpreted as follows;

Bits Ø-13:	An address in the ØØØØ-3FFF range.
Bit 14:	Ø to enable lower ROM, 1 to disable it.
Bit 15:	Ø to enable upper ROM, 1 to disable it.

The specified ROM condition is set, and a jump to the given address is performed. This function, called LOW JUMP, is one of the secret weapons that make the system of bank-switching practicable, since the jump and ROM change appear to occur simultaneously.

Entry at ØØØB accesses a jump to B97C in the upper RAM routines. This is PCHL, which is similar to LOW JUMP, except that the 16-bit qualifying word is held in the HL register.

Entry at ØØØE accesses a jump to the address defined by the contents of the BC register. This is PCBC, which resembles JP (HL) in function.

Neither of these two entries is accessible by an RST instruction, but they can be accessed in the usual way by a jump or call.

RST1Ø (&D7) enters location Ø1ØØ, where there is a jump to BA16 in the upper RAM routines. This implements the SIDE CALL function. The two bytes following the &D7 code are read as a 16-bit word and interpreted as follows;

Bits Ø-13: CØØØ is added to give an address in the CØØØ-FFFF range.

Bits 14-15: A value in the Ø-3 range. This is added to the number of the current foreground ROM to determine the number of the ROM which is to be accessed.

Upper ROM is enabled, lower ROM is disabled, the required upper

ROM is selected, and a jump to the specified address is performed. This function simplifies cross-access within a group of up to four sideways ROMs with consecutive numbers, allowing for extension programs up to 64K in size.

It should be noted that whereas LOW JUMP leaves no return address, and is a true jump, not a call, SIDE call does preserve a return address pointing to the location following the qualifying bytes, and is a call, rather than a jump. (The return addresses left by the RST instructions are used to locate the qualifying bytes.)

Entry at ØØ13 accesses a jump to BA1Ø in the upper RAM routines. This is SIDE PCHL, which resembles SIDE CALL, except that the 16-bit qualifying word is held in the HL register.

Entry at ØØ16 accesses a jump to the address defined by the contents of register DE. (PCDE)

These two entries are not accessible by means of RST instructions.

RST18 (&DF) enters location ØØ18, where there is a jump to B9BF in the upper RAM routines. The two bytes following the &DF code are read as a 16-bit word, which is an address pointing to a three-byte qualifier. The first two bytes of the qualifier give an entry address, and the third byte is interpreted as follows:

&ØØ to &FB: Select ROM of this number: Upper ROM enabled, lower ROM disabled.

&FC: ROM unchanged. Upper and lower ROMs enabled.

&FD: ROM unchanged. Upper ROM enabled, lower ROM disabled.

&FE: ROM unchanged. Upper ROM disabled, lower ROM enabled.

&FF: ROM unchanged. Upper and lower ROMs disabled.

This is FAR CALL, a versatile function that can access almost anything.

Entry at ØØ1B accesses a jump to B9B1 in the upper RAM routines. This is FAR PCHL, which resembles FAR CALL, except that the address part of the qualifier is held in the HL register, while the third byte is in the C register.

Entry at ØØ1E accesses a jump to the address defined in the HL register (PCHL).

These two entries are not accessible by means of RST instructions.

RST2Ø (&E7) enters location ØØ2Ø, where there is a jump to BACB in the upper RAM routines. This is RAM LAM, which executes LD A=(HL) with ROMs disabled. It can therefore be used to read RAM at any time. The previous ROM state is restored after the read.

Entry at ØØ23 accesses a jump to B9B9 in the upper RAM routines. This is FAR ICALL, which resembles FAR CALL, except that the addresss of the qualifier is held in the HL register.

RST28 (&EF) enters location ØØ28, where there is a jump to BA2E in the RAM routines. This is FIRM JUMP. It resembles the usual C3XXXX instruction, but lower ROM is enabled before the jump and disabled after the return.

SIDE CALL, SIDE PCHL, FAR CALL and FAR ICALL enter the called routine with IY pointing to the RAM data area reserved for the selected ROM.

RST3Ø (&F7) is the entry for USER RESTART. If it is used with lower ROM enabled, the current contents of C', which contain the current ROM select bits, are copied to (ØØ2B) in RAM, the lower ROM is disabled, and the action returns to ØØ3Ø, but now in RAM. If the lower ROM is already disabled, this procedure is unnecessary.

The area ØØ3Ø-ØØ37 in RAM can be patched to access a special routine to meet the user's requirements. As initialised, ØØ3Ø in RAM holds &C7, and entry to ØØ3Ø invokes a full reset.

RST38 (&FF) is the equivalent of the response to interrupt, and is not available to the user.

Entry at ØØ3B is part of the interrupt handling procedure. If an interrupt lasts too long to be of internal system origin, ØØ3B is called. It normally contains &C9, a return instruction, but RAM from this point may be patched to access a user interrupt handler.

Some of the functions which have been described will rarely be needed by a typical user, though they constitute a vital part of the CPC464 system, which would not work without them. Their action should be studied with care.

The RAM Routine Jumpblock

The jumpblock at the start of the upper RAM routines gives access to eleven functions;

U ROM ENABLE: B900,BA5E

The currently-selected upper ROM is enabled. The routine returns with the A register holding the previous ROM state.

U ROM DISABLE: B903,BA68

The currently-selected upper ROM is disabled. The routine returns with the A register holding the previous ROM state.

L ROM ENABLE: B906,BA4A

Lower ROM is enabled. The routine returns with A holding the previous ROM state.

L ROM DISABLE: B909,BA54

Lower ROM is disabled. The routine returns with A holding the previous ROM state.

These four routines are almost identical, taking the general form;

```
Disable Interrupt

Select alternate registers

A = C'

Modify C'

OUT (C'),C'

Select normal registers

Enable interrupt

Return
```

The modification to C' affects bit 2 for the lower ROM, bit 3 for the upper ROM. The relevant bit is zeroed to enable, set to disable. The output is to the Video Gate Array. Note that it is assumed that B' contains &7F, the upper byte of the required I/O address. The contents of B' may only be changed, on a temporary basis, while interrupt is disabled and no operating system calls are made.

ROM RESTORE: B90C,BA72

On entry, A must hold the required ROM state bits, as defined above, and supplied by the previous four routines. This state is set.

The routine is similar to the first four, except that bits 2 and 3 of A are copied to C', the remaining bits of which are unaltered.

ROM SELECT: B90F,BA7E

On entry, C holds the number of the required ROM. This ROM is selected, and Upper ROM is enabled. When the called routine returns, C holds the number of the previously-selected ROM and B holds the previous ROM enable state.

The routine calls U ROM ENABLE, then jumps to BA92, where an output of C to DFXX selects the required ROM. The ROM number is copied in (B1A8), which maintains a note of the upper ROM in current use.

CURR SELECTION: B912,BAA2

The A register is set from (B1A8) to the current ROM number.

PROBE ROM: B915, BAA2

On entry, C holds a ROM select address. One exit, A holds the ROM class, H holds the ROM version number, and L holds the ROM mark number. The class byte is interpreted thus;

Ø:	Foreground ROM
1:	Background ROM
2:	Extension Foreground ROM
&8Ø:	On-board ROM

The routine calls ROM SELECT, to bring the selected ROM into action, then A = (CØØØ), HL = (CØØ1/2). ROM DESELECT follows to restore the previously selected ROM.

ROM DESELECT: B918,BA8C

On entry, C holds the required ROM number, and B holds the required ROM enable state. These will normally have been obtained by ROM SELECT. The specified ROM and state are selected, using ROM RESTORE and the routine from BA92 used by ROM SELECT.

LDIR: B91B, BAA6
LDDR: B91E, BAAC

These routines allow copies to be made from RAM to RAM with ROM temporarily disabled. On entry, BC,DE and HL should be set as for a normal LDIR or LDDR.

The routines are tortuous, and can only be followed by noting how the stack contents change. The LDIR routine is given here. The LDDR routine is almost identical.

```
BAA6 CALL BAB2                 Stack: X
BAA9 LDIR                      Stack: X,BC',BABF
BAAB RET                       To BABF

BAB2 DI                        Stack: X,BAA9
     EXX
     POP HL'                   Stack: X      HL'=BAA9
     PUSH BC'                  Stack: X,BC'
     C'=C' OR &ØC
     OUT (C'),C'               Disable ROMs
     CALL BAC7                 Stack: X,BC'
BABF DI                        Stack: X,BC'
     EXX
     POP BC'                   Stack: X
     OUT (C'),C'               Restore ROMs
     EXX
     EI
     RET                       To X (set by calling routine.)

BAC7 PUSH HL'                  Stack: X,BC',BABF,BAA9
     EXX
     EI
     RET                       To BAA9
```

RST AREA EXTENSIONS

The routines used to implement the RST Area functions are complex and convoluted, but it is advisable to examine them in some detail so that their action is clearly understood. They will be examined in the order in which they appear in the upper RAM routines.

LOW PCHL: ØØØB,B97C

```
B97C DI
     PUSH HL
     EXX
     POP DE'
     JP B988                   See over.
```

The qualifier is transferred from HL to DE'

LOW JUMP: RST08,B982

```
B982 DI
     EXX
     POP HL'                     Return address.
     DE' = (HL')                 Qualifier in DE'.
B988 EX AF/AF'
     A'=D'                       Upper byte of qualifier.
     D'=D' AND &3F               DE' holds address only.
     RLCA
     RLCA                        Bits 6,7 to Ø,1
B99Ø RLCA
     RLCA                        Bits Ø,1 to 2,3
     XOR C'
     AND &ØC                     Bits 2,3 of A' isolated.
     PUSH BC'                    Save previous ROM state.
     CALL B9A8                   See below.
     DI
     EXX
     EX AF/AF'
     A'=C'
     POP BC'
     AND 3
     C'=C' AND &FC
     A'=A' OR C'
     JP B9A9

B9A8 PUSH DE'                    Set link to called routine.
B9A9 OUT (C'),C'                 Set enables.
     CLEAR CARRY'
     EX AF/AF'
     EXX
     EI
     RET
```

The manipulations of return addresses are somewhat similar to those noted earlier in LDIR/LDDR. The crucial point is whether the last block is entered at B9A8 or B9A9. At B9A8, the required entry address is put on to the stack, so the block 'returns' to the called routine. Entry at B9A9 leaves the overall return address on top of the stack. The same block therefore does two entirely different things.

The routine is complicated by the need to preserve the other bits of C' while bits 2 and 3 are manipulated to select the ROM state. On the other hand, bits Ø and 1 may be changed during execution of the called routine, and they must be incorporated into the previous state of the other bits.

FAR PCHL: ØØ1B,B9B1

```
B9B1 DI
     EX AF/AF'
     A'=C                     ROM number.
     PUSH HL                  Routine address.
     EXX
     POP DE'                  Routine address to DE'
     JP B9CE                  See below.
```

FAR ICALL:ØØ23,B9B9

```
B9B9 DI
     PUSH HL                  Pointer to qualifier
     EXX
     POP HL'                  Pointer to HL'
     JP B9C8                  See below.
```

FAR CALL: RST18,ØØ18,B9BF

```
B9BF DI
     EXX
     POP HL'                  Return Link.
     DE'=(HL')                Pointer.
     HL=HL+2
     PUSH HL'                 Modified return link.
     EX DE'/HL'               HL' holds pointer.
B9C8 DE'=(HL')                Address
     HL'=HL'+2
     EX AF/AF'
     A'=HL'                   ROM number.
B9CE IF A')&FB THEN B99Ø
B9D2 B'=&DF
     OUT (C'),A'              Select ROM
     B'=(B1A8)                Previous ROM number.
     (B1A8)=A'                New ROM number.
     PUSH BC'
     PUSH IY
     A'=A'-1
     IF A')6 THEN B9F2        Not background ROM
     HL'=B1AC + 2*A'
     IY =(HL')                Address of workspace.
B9F2 B'=&7F                   Restore normal value.
     A'=C'AND &F3
     CALL B9A8                Enter called routine.
     POP IY                   Restore previous contents.
     DI
     EXX
     EX AF/AF'
     E'=C'                   Save current value.
     POP BC'                 Restore old value.
     A'=B'                   Previous ROM.
     B'=&DF
     OUT (C'),A'             Restore previous ROM.
     (B1A8)=A'               Note current ROM.
     B'=&7F                  Restore normal value.
     A'=E'                   ROM which was called.
     JP B99F                 See above.
```

SIDE PCHL: ∅∅13,BA1∅

```
BA1∅ DI
     PUSH HL                 Qualifier
     EXX
     POP DE'                 Qualifier to DE'
     JP BA1E                 See below.
```

SIDE CALL: RST1∅,∅∅1∅,BA16

```
BA16 DI
     EXX
     POP HL'                 Return Link.
     DE'=(HL')               Qualifier.
     HL'=HL'+2
     PUSH HL'                Modified return link.
BA1E EX AF/AF'
     A'=D'                   Upper byte of qualifier.
     D'=D' OR &C∅            Form address in C∅∅∅-FFFF.
     A'=A' AND &C∅           Isolate ROM select bits.
     RLCA
     RLCA                    Bits 6,7 to ∅,1
     A'=A'+ (B1AB)           Add foreground ROM number.
     JP B9D2                 See above.
```

FIRM JUMP: RST28,∅∅28,BA2E

```
BA2E DI
     EXX
     POP HL'                 Qualifier pointer.
     DE'=(HL')               Qualifier.
     C'=C' AND &FB           Zero bit 2
     OUT (C'),C'             Enable lower ROM.
     (BA3F/4∅)=DE'           Modify instruction.
     EXX
     EI
BA3E CALL XXXX               Address defined above.
     DI
     EXX
     C'=C' OR 4              Set bit 2.
     OUT (C'),C'             Disable lower ROM.
     EXX
     EI
     RET
```

This routine involves the creation of an instruction at run-time, which is not approved by all programmers, but it works. However, it can give confusing results in disassembly...

RAM LAM: RST2Ø,ØØ2Ø,BACB

```
BACB DI
     EXX
     E'=C'                   Enable state.
     E'=E' OR &ØC
     OUT (C'),E'             Disable upper and lower.
     EXX
     A=(HL)                  Read from RAM
     EXX
     OUT (C'),C'             Restore enable state
     EXX
     EI
     RET
```

The final routine is unnamed, but is a variant of RAM LAM:

```
BADC EXX
     A=C' OR &ØC
     OUT(C'),A               Disable ROMs
     A=(IX)'                 Read RAM
     OUT (C'),C'             Enable ROMs
     RET
```

This form does not corrupt E', and uses IX as a pointer instead of HL.

Comment

One experienced programmer, glancing through a draft for this chapter, shook his head in amazement. "What a kerfuffle!" was his initial reaction, but after further study he came to the conclusion that every routine was necessary to implement the storage system to full effect. A user who takes no note of detail will find that the routines make everything simple, and that is the key point.

For example, while the BASIC interpreter is running the upper ROM must be enabled, but the BASIC program is stored from Ø17Ø upwards, under the lower ROM. RAM LAM allows direct access to this area of RAM, with a minimum of fuss and bother. If a function in lower ROM is needed, it can be called through the jumpblock by RST8 or RST28.

The routines which have been described form the groundwork of the CPC464 operating system, a foundation on which the rest of the system is built. We can now go on to examine the higher and more directly interesting parts of the edifice.

Chapter 3
THE MACHINE PACK

Broadly speaking, the Machine Pack is reponsible for the control of hardware peripherals, but it will be convenient to include the main initialisation processes under this heading, since they are largely concerned with peripheral setting-up.

Several of the Machine Pack routines depend on the action of other routines to set up data. To understand this data in full, you need to read 'The Ins and Outs of the AMSTRAD CPC464', which gives full details of the peripheral codes. Only the more essential codes will be defined here.

Main Reset

At switch-on, or in response to instruction code &C7, location ØØØØ is entered. At switch-on, lower ROM is enabled, but the ROM routines are later copied to the corresponding RAM locations in this area, so the enable state of the lower ROM is then unimportant. However, the first action of the reset routine is to output &89 to the Video Gate array on I/O address 7FXX, and this enables lower ROM, disables upper ROM, and also sets up Mode 1. There being no further room in the RST Area, the routine jumps to Ø58Ø to continue reset action.

Interrupt is disabled, and &82 is output to F7XX. This sets the PPI (Parallel Peripheral Interface) to output on ports A and C, input on port B. Zero outputs to F4XX and F6XX clear ports A and C, while an output of &7F to EFXX initialises the printer port. Bit 7 is low, the other bits are high.

The CRT Controller is then set up. There are two alternative sets of values for this, one for 5Ø Hz frame scan and the other for 6Ø Hz. The set to be used is determined by reading port B, bit 4. If this bit is true, 5Ø Hz values are used, while the 6Ø Hz values are used if the bit is false, this being determined by the presence of Link 4 on the main printed circuit board.

The tables are read backwards, which can be a little confusing at first, and the outputs alternate between BCXX, which selects the register to be set, and BDXX, which performs the actual setting.

Then MC START PROGRAM is entered at Ø6ØE with DE=Ø65C and HL=ØØØØ. The contents of DE point to the display routine for the main title, which is called at an appropriate point. The zero value in HL means that ROM Ø will be entered at CØØ6. This will normally invoke the BASIC interpreter, unless an external ROM responds to Ø. CØØ6 is the standard upper ROM entry point.

Before discussing MC START PROGRAM, it will be convenient to look at a program which calls it, having first loaded the necessary data:

MC BOOT PROGRAM: BD13,Ø5DC

On entry to this function, HL must hold the address of a loading routine, which must be designed to return with carry set and the program start address in HL if the load is successful, or with carry clear if the load fails.

The stack is reset by SP=CØØØ, this being the normal stack position, and sound RESET is called to silence the Sound Generator. Interrupt is disabled, and &FF is output to port F8FF, requiring that all external peripherals devices should be reset.

KL CHOKE OFF is called to clear the B1ØØ-B1BF area to zeroes, though the previous contents of (B1A9/B) are first saved. (B1A9/A) holds the last-used foreground ROM entry address, which is copied to DE, while (B1AB) holds the last-used foreground ROM number, which is copied to B. (Note that the number of the ROM in current use is held in (B1A8), which is not preserved here.) If (B1AB) holds &FF the routine returns with C, D and E all zeroed.

Back in the main BOOT routine, HL is restored to its value on entry and DE, BC and HL are pushed. KM RESET is called to initialise the Key Manager, TXT RESET is called to initialise the text screen, assisted by a call to SCR RESET, and U ROM ENABLE is called to bring the upper ROM into action.

HL is popped, and the loading program it defines is entered, using an odd little subroutine that consists solely of the JP (HL) instruction. BC and DE are popped.

If the loader returned with carry set, MC START PROGRAM is entered at Ø6ØB. Otherwise, DE and HL are exchanged, putting the address obtained by KL CHOKE OFF into HL, C=B, and MC START PROGRAM is entered at Ø6ØE with DE=Ø6E8, the entry address of a routine that reports 'LOAD FAILED'. The previously-selected ROM is entered.

MC START PROGRAM: BD16,Ø6ØB

If the normal entry to this function, at Ø6ØB, is used, DE is set to Ø726 (pointing to a Return instruction), but it is also possible to enter at Ø6ØE, with DE pointing to a subroutine to be run during the latter part of the START PROGRAM routine. In either case, HL must hold the entry address to be used, and C must hold the number of the ROM to be employed, though the contents of C may be irrelevant if HL points to a RAM area.

Interrupt is disabled, and interrupt mode 1 is selected. The alternative BC, DE and HL registers are brought into action.

An output of Ø to DFXX selects upper ROM Ø, and an output of &FF on I/O address F8FF should reset external peripherals. Workspace in the B1ØØ-B8FF range is zeroed, and the Video Gate Array receives an output of &89 on address 7FXX. (Mode 1, enable lower, disable upper.) The normal BC, DE and HL registers are re-selected. XOR A zeroes A and clears carry, and EX AF,AF' exchanges AF registers. This sets up the initial conditions required by the interrupt system.

The stack pointer is again set to CØØØ, its normal base, and HL, BC and DE are pushed. A series of calls then performs the main initialisation;

To ØØ44, copying the RAM routines from RAM, with KL CHOKE OFF following.

JUMP RESTORE resets the jumpblock entries.

KM INITIALISE resets the Key Manager.

SOUND RESET initialises the Sound system.

TXT INITIALISE initialises the Text VDU.

GRA INITIALISE initialises the Graphics VDU.

CAS INITIALISE initialises the Cassette Manager.

MC RESET PRINTER standardises the printer system.

SCR INITIALISE initialises the Screen Pack.

The details of these routines will be examined in the appropriate place, but it can be said that everything - or nearly everything - is brought to a standard state. This can be annoying to someone who likes to set up non-standard conditions, but it has the great advantage that every program starts on the same basis.

Interrupt is now enabled, and the routine defined in DE on entry is called. This may be the initial title display, or a 'load failed' report, as defined below. The main program then pops BC and HL and jumps to ØØ77, which is the actual entry routine. This key routine is not accessible via the Jumpblock, which might be useful, because the system concept requires a full reset before a program is entered.

If HL holds ØØØØ, the default entry to CØØ6 in ROM Ø is executed, but otherwise the ROM is defined in A and the entry address in HL.

```
ØØ77 If HL=ØØØØ, HL=CØØ6,A=Ø              Default Values.
     (B1A8)=A                             ROM number.
     (B1AB)=A                             Part of qualifier.
     (B1A9/A)=HL                          Qualifier address.
     HL=ABFF                              Initial HIMEM.
     DE=ØØ4Ø                              Initial LOMEM.
     BC=BØFF                              Top of usable memory.
     A FAR CALL DF A9 B1 enters the specified routine.
     On return, a full reset from ØØØØ is executed.
```

This completes the MC START PROGRAM routine, apart from the routines called near the end:

Ø65C This calls Ø712, which reads port B, picking up bits 1-3, which are determined by links. According to the links set, the display announces that the name of the machine is one of the following:

Arnold	Amstrad	Orion
Schneider	Awa	Solovox
Saisho	Triumph	Isp

The actual output to the display is handled by Ø6EB, which is also called with HL=Ø66D to output the rest of the title, and finally with HL=Ø693 to complete the display.

Ø6E8 HL=Ø6F4, pointing to '***PROGRAM LOAD FAILED***'

```
Ø6EB A=(HL)
     HL=HL+1
     IF A=Ø THEN RETURN
     CALL TXT OUTPUT
     JP Ø6EB
```

Printer Routines

Access to the printer port is obtained via a small group of closely related routines;

MC RESET PRINTER: BD28,07E6

The indirection for entry to MC WAIT PRINTER at BDF1 is reset to access Ø7F8. This cancels any change which has been made to bring an alternative printer driver into use.

MC PRINT CHAR: BD2B,Ø7F2

BC is saved on the stack while MC WAIT PRINTER is called at BDF1.

MC WAIT PRINTER: BDF1,Ø7F8

BC is set to ØØ32, a delay count. MC BUSY PRINTER is called, and if it returns with carry clear the routine jumps to MC SEND PRINTER. The printer is not busy.

If the return is with carry set, the routine loops back to repeat the call. In all, the call is executed 12,8ØØ times before giving up and returning with carry clear to indicate failure, which should give you ample time to put the printer on line if you have forgotten to do so.

MC SEND PRINTER: BD31,Ø8Ø7

BC, holding the delay count, is pushed, and A AND &7F is output to the printer port on address EFXX. This sets strobe low. Then A OR &8Ø is output to the same address, making strobe high. Finally, A AND &7F is again output to bring strobe low. During the last pair of outputs interrupt is disabled, to avoid any risk of lengthening the strobe duration. BC is popped, carry is set, and the routine returns.

MC BUSY PRINTER: BD2E,Ø81B

BC, holding the delay count, is pushed, and A is copied to C. Then A is set by an input from port B on F5XX, and bit 6, the printer Busy line, is copied to carry. A is restored from C, BC is popped, and the routine returns.

This is a good point at which to remind you of the key difference between the main jumpblock entries and the indirections. Apart from the preservation of BC in the first case, BD2B and BDF1 appear to have the same effect, but BD2B enables lower ROM, and BDF1 does not. It is useful to make BDF1 an indirection, to simplify calling alternative drivers, but calling it with lower ROM disabled could cause chaos. If the indirection has been altered to call code in RAM, however, this does not arise.

Other MC Routines

MC CLEAR INKS: BD22,Ø786

BC and DE are pushed, and BC=7F1Ø, forming a Video Gate Array address. Ø7AB is called. (See below)

At Ø79Ø, Ø7AB is called again. DE is decremented, and if Ø7AB returned NZ the routine loops to Ø79Ø. Otherwise, BC and DE are popped, and the routine returns.

Since Ø7AB increments DE, the contents of this register remain the same during the execution of the loop.

MC SET INKS: BD25,Ø799

This is identical with MC CLEAR INKS, except that the decrement of DE is omitted. The increment of DE in Ø7AB is therefore allowed to stand.

At Ø7AB OUT (C),C sets the palette pointer of the Video Gate Array. Then A=(DE) AND &1F OR &4Ø is output to the Video Gate Array to set the palette entry. DE and C are incremented, and if C=&1Ø the zero flag is set. The routine returns.

The above routines require entry with DE pointing to an entry in the colour tables, which will be discussed later. MC CLEAR INKS sets all the palette entries to the same colour, MC SET INKS sets them from the colour table.

MC WAIT FLYBACK: BD19,Ø7BA

AF and BC are pushed, and B=&F5, ready to access port B. An input from the port is taken, and carry is set from bit Ø of the result. The input is repeated until carry is true, which shows that frame flyback has occurred. BC and AF are popped, the routine returns.

This routine allows screen action to be delayed until frame flyback occurs.

MC SCREEN OFFSET: BD1F,Ø7C6

On entry, A must hold the upper byte of the required screen base, which is set in (B1CB) by SCR SET BASE, and HL must hold the required screen offset, as held in (B1C9/A).

```
BC is pushed
C=A/4 AND &3Ø
A=H/2 AND 3 OR C
Output of &ØC to BCXX selects CRTC register 12
Output of A to BDXX sets the register
Output of &ØD to BCXX selects CRTC register 13
HL=HL/2
Output of L to BDXX sets the register
BC is popped
Return
```

This is an example of an MC routine that helps to implement a routine elsewhere, by setting hardware to match the software settings. The values set may appear strange until the section on the screen has been read.

MC SET MODE: BD1C,Ø776

On entry, A must hold the number of the mode required. If A exceeds 2, the routine drops out.

Bits Ø,1 of A are copied to the corresponding bits of C', and C' is then output to 7FXX, the Video Gate Array.

Serious confusion could result if this function was not executed after a software mode change. The hardware and software must be kept in step.

MC SOUND REGISTER: BD34, Ø826

This routine passes data to the Sound Generator registers. The data is passed via port A of the PPI, while the interpretation of the data is controlled by bits 6 and 7 of port C thus:

Bit 6	Bit 7	
Ø	Ø	Inactive
Ø	1	Write to register
1	Ø	Read from register
1	1	Select register

On entry to MC SOUND REGISTER, A must hold the register number and C must hold the data.

```
Interrupt is disabled.
A is output to F4XX            Port A specifies register.
A is set from F6XX             Port C input.
A=A OR &CØ                     Set bits 6,7.
A is output to F6XX            Select register.
A=A and &3F                    Zero bits 6,7.
A is output to F6XX            Inactive.
C is output to F4XX            Port A specifies data.
C=A
A=A OR &8Ø                     Set bit 7.
A is output to F6XX            Write to register.
C is output to F6XX            Inactive.
Interrupt is enabled
Return
```

This routine only handles outputs to the Sound Generator. Inputs needed in scanning the keyboard are handled elsewhere.

That completes the Machine Pack routines.

Chapter 4
THE KERNEL

The KERNEL routines deal with Interrupts and Events. They are somewhat complex and tortuous, but need to be understood by anyone who wishes to use the CPC464 system to full advantage. It will be best to begin by making a rapid tour of the system.

The Video Gate Array generates an interrupt pulse every 1/3ØØth of a second, or to be more precise, every 52 horizontal scans of the screen system, which gives an interval of 64*52=3328 microseconds.

The processor responds to the interrupt pulse by jumping to ØØ38, where there is a jump to the primary interrupt handler in the RAM routines at B939. The primary handler calls a secondary handler at ØØB1 in ROM, and this first deals with the time counter update, then with the Frame Flyback Events, if any, and then calls the sound system interrupt routine. These are the functions of the 'Fast Ticker' inetrrupt.

In five cases out of six, the secondary routine then drops out, and the handling process is complete, but on every sixth entry the action continues, to service the slower 'Ticker' interrupt, which nominally occurs every 1/5Øth of a second. In this case the main handler calls a further secondary handler at Ø1ØA in ROM.

Since the interrupts occur so frequently, it is essential that they are handled as quickly as possible, since the handling time is stolen from the running time of the main routines. Even so, there may not always be time to complete outstanding actions before the next interrupt occurs, so provision is made for noting actions which are left over.

There is also provision for a special user interrupt handler, but more about that later.

Most of the actions induced by interrupt are Events, each of which is identified by an Event Block that defines its characteristics and the address of the routine which implements it. Events may be tied to the Fast Ticker, Ticker, or Frame Flyback interrupts, or may be activated by the main program.

The Interrupt Handler

MC START PROGRAM selects interrupt mode 1, which means that the processor reponds to interrupt by putting the address of the next instruction to be executed on to the stack and jumping to ØØ38, where there is a jump to B939 in the RAM Routines.

The first requirement for an interrupt handler is that it should preserve the contents of the processor registers so that the interrupted routine can continue when the handler has completed its task. This is commonly achieved by pushing the register contents on to the stack, but the CPC464 uses the faster method of switching to the alternative main registers, at least initially. This leads to some interesting gymnastics.

First, AF' is selected. If carry' is found to be set, the routine jumps to B97Ø. The system is already in the interrupt path, and special action is needed, as explained below. Otherwise, BC', DE' and HL' are brought into use, A'=C' preserves the current ROM enable state, and carry' is set.

Interrupt is now enabled briefly while AF is re-selected. This is the only brief period during execution of the handler when a further interrupt can intrude. It occurs 44 clock cycles after the interrupt took effect - say 13 microseconds. If the original interrupt is still active, it did not come from the Video Gate Array, so it must be a user interrupt. Since carry' is set, the jump to B97Ø mentioned above will be taken. We will look at the consequences of that later.

AF is now pushed, to preserve its contents, and bit 2 of C' is zeroed so that the subsequent OUT (C'),C' will enable lower ROM. It is then permissible to call ØØB1, the first secondary handler.

At ØØB1, the Time count in (B187/B) is incremented. Then an input is taken from F5XX (port B). If bit Ø is true, it is Frame Flyback time, and if there are any events on the Frame Flyback list they will be serviced by calling Ø153 with HL=(B1BC). (See Events)

If there are any events on the Fast Ticker list, they are now serviced by calling Ø153 with HL=(B18E). This completes the actions that occur 3ØØ times a second.

The count in (B192) is now decremented, and if the result is not zero the routine returns. Otherwise, (B182) is reset to 6, and the Ticker actions are executed.

First, the keyboard scan routine is called (see Keyboard Manager). Then the Ticker event list is checked. If it is not empty, bit 6 of (B1Ø4), a flag byte, is set. The routine returns.

Back in the main handler, a little bit of juggling is performed. Carry is cleared, and is then made carry' by EX AF/AF'. A now holds the previous ROM state copied earlier from C' to A'. C'=A' and B=&7F, its usual value.

If (B1Ø4)=Ø, or (B1Ø4) is negative, the routine now jumps to B96A. Otherwise, A = C' AND &ØC, and AF is pushed. Bit 2 of C' is reset. The normal BC, DE and HL registers are selected, and Ø1ØA is called to execute Ticker events. Then BC', DE' and HL' are brought back into action again. POP HL sets H to the value pushed from A, then C'=C' AND &F3 OR H, forming the correct value to be used to restore the ROM status as it was before interrupt.

We have now, by one route or another, reached B96A. OUT (C'),C' restores the previous ROM status, the normal BC,DE and HL registers are finally reselected, and AF is restored from the stack. Interrupt is enabled, and the routine returns, the interrupted action being resumed.

But what about that special action taken if carry' is found to be set? The routine at B97Ø looks a little strange;

```
B97Ø EX AF/AF'               Cancel earlier change.
     POP HL                  HL', to be accurate.
     PUSH AF
     SET 2,C'
     OUT (C'),C'             Disable lower ROM.
     CALL ØØ3B               User Interrupt Entry.
     GOTO B94B               Follows call to ØØB1.
```

The alternate registers are in use, having been selected by the main handler. POP HL' removes the return address for the second interrupt, as it is not needed. PUSH AF saves the normal AF. After calling ØØ3B in RAM the main routine is entered immediately after the call to ØØB1.

A user handler must not use EXX or EX AF/AF', since the main registers not in use are preserving data for the interrupted program. Of the registers in active use, HL' has already been corrupted, while the contents of BC' must be preserved. DE' does not appear to contain critical data, but if IX or IY are used they should be preserved on the stack first.

To complicate matters, there is an official suggestion that user handlers should be 'nested', each calling another if it finds that the interrupt is none of its business. This allows for external units to set up their handlers in any order they wish, but if matters get to that stage the situation must be almost out of hand.

A specific and important requirement is that the handler should clear the interrupt source.

The Event System

The key to the Event System is the Event Block, which has the following format;

Bytes Ø,1;	Chain Link
Byte 2;	Count
Byte 3;	Class
Bytes 4,5;	Routine Address
Byte 6;	ROM number
Bytes 7 on	User Field

Chain Link is used to combine a number of event blocks into a list, by setting each link to point to the next event block. The last block in the list has byte 1 = Ø. This will be examined in more detail later.

Count is a record of outstanding requests for execution. In that role, it may have any value from Ø to 127. When an event is 'kicked' Count is incremented (but not beyond 127), and when the routine is executed Count is decremented. If, however, Count has a negative value the event is disabled, and Count remains unaltered by kicks or executions.

Class defines the type of event. The coding used is;

Bit Ø	Ø; The routine can be reached by a 'near address' i.e. by a simple jump. 1; A 'far address' is involved, including ROM selection.
Bits 1-4	Priority (synchronous events only).
Bit 5	Ø

Bit 6	Ø; Normal event.
	1; Express event.
Bit 7	Ø; Synchronous Event.
	1; Asynchronous Event.

The routine entry address and the ROM number provide access to the routine which must be called to implement the event, the ROM number being ignored if a 'near address' is specified.

For normal events, a kick is marked by incrementing Count, but Express events are executed immediately. It is important that they should be as brief as possible. Asynchronous events are tied to interrupts, while Synchronous events are called from the main program.

The event block must be in RAM, so that its contents can be adjusted, and it must be in the central half of RAM, so that it is accessible whenever it is needed.

The user field beginning at byte 7 may be used to hold parameters which are relevant to the event function.

An event block is set up by;

KL INIT EVENT: BCEF, Ø1D2

On entry, HL must hold the address at which the event block is to be set up, DE must hold the entry address for the associated routine, B must hold the class byte, and C must hold the number of the ROM which contains the associated routine.

Chain link is not set up at this stage, and Count is set to Ø. The rest of the entries are set up from the given data.

It is advisable to keep a written note of event block addresses, as they are not easy to locate, once they have been set up.

The block having been established, must be linked into the system. This can be done in various ways;

KL EVENT: BCF2, Ø1E2

If KL EVENT is called with HL holding the address of the event block, the event is 'kicked', subject to checks on its status.

If Count is negative, the routine drops out, taking no action. If Count is in the range Ø-126, it is incremented, but if it is 127 the routine drops out, there being too many outstanding requests already. (As a diagnostic aid, a case of Count=127 is a clear indication that the interrupt system is overloaded.)

If the routine has not dropped out, Class is checked. A synchronous event is linked into the synchronous list, a normal event is added to the 'kicked' list, and an express event is implemented immediately.

The events on the 'kicked' list are executed at the next Fast Ticker interrupt. The list is constructed by setting each chain link to point to the next event block, the first link being held in (B1ØØ/1). For the synchronous list the first link is in (B193/4). If the upper byte of a first link is zero, the list is empty.

There are further event lists associated with the Fast Ticker, Ticker and Frame Flyback interrupts, the relevant functions being:

KL NEW FRAME FLY: BCD7, Ø163

On entry, HL must contain the address at which the Frame Flyback block is to be set up: It consists of an event block preceded by two bytes used as a chain link. The normal event block chain link is not used. The other parameters required for KL INIT EVENT apply here, as the event block is first created, then linked to the Frame Flyback list.

The first link of the Frame Flyback list is held in (B18C/D)

KL ADD FRAME FLY: BCDA, Ø16A

On entry, HL must hold the address of an existing event block, less two. The event concerned is added to the Frame Flyback list.

KL DEL FRAME FLY: BCDD, Ø17Ø

On entry, HL must hold the address of an event block less two. The event is deleted from the Frame Flyback list, if it is there in the first place.

Once an event has been tied to an interrupt list, it is kicked every time the related interrupt occurs.

The three calls related to the Fast Ticker interrupt are directly analogous to those for the Frame Flyback interrupt:

KL NEW FAST TICKER: BCE0, Ø176

KL ADD FAST TICKER: BCE3, Ø17D

KL DEL FAST TICKER: BCE6, Ø183

The first link of the Fast Ticker list is held in (B18E/F)

Ticker blocks are more complex, requiring six bytes prefaced to a normal event block;

Bytes Ø,1	Ticker Chain Link
Bytes 2,3	Tick Count
Bytes 4,5	Count Recharge

The Ticker interrupt nominally occurs 5Ø times a second. The Tick Count is then decremented, but no further action is taken until the count reaches 1, when the associated event is kicked and the count is reset from Count Recharge. If Count Recharge is zero, the event is only kicked once. A zero count disables the event. Otherwise, the event can be called at intervals of up to rather more than 21 minutes.

The first link of the Ticker list is held in (B19Ø/1)

KL ADD TICKER: BCE9, Ø1B3

On entry, HL must hold the event block address less six, DE must hold the initial count, and BC must hold the count recharge. The event is added to the Ticker list. (There is no function to both create an event and add it to the Ticker list.)

KL DEL TICKER: BCEC, Ø1C5

On entry, HL must hold the event block address less six. The event is removed from the Ticker list.

Note that deletion of an event from any list leaves the event block intact, and it can be put back on the list later, if necessary.

KL DISARM EVENT: BDØA, Ø28E

On entry, HL must hold the address of an event block. The Count byte in the event block is set negative, so that the event is disabled.

Synchronous Events

Synchronous events are handled in a rather different way, the events being set in the synchronous list in priority order. The first link of the list is held in (B193/4) and the current priority level is held in (B195).

KL SYNC RESET: BCF5, Ø228

This zeroes (B194/5), marking the list as empty and setting the current priority level as zero.

KL DEL SYNCHRONOUS: BCF8, Ø285

On entry, HL must hold the address of an event block which is to be removed from the synchronous list. KL DISARM EVENT is called to make the Count byte negative, and the chain link pointing to the specified event block is changed to point to the next event on the list. If there is no subsequent event, the upper byte of the link is zeroed.

For non-express synchronous events, bits 5-7 of Class are zero, so the magnitude of Class depends on the priority bits 1-4 and the address type bit Ø. When KL EVENT finds that it is dealing with a synchronous event, the 'kick synchronous' subroutine is called. This scans the synchronous list until either the end of the list is reached or the Class byte of a listed event is smaller than the Class byte of the new event, which means that the new event has a higher priority.

Calling the listed event N, the previous event, N-1, will have a chain link pointing to the event block for N. This is changed to point to the new event, while the chain link of the new event is set from the previous chain link of N-1. The new event block is thus inserted in the list at a point appropriate to its priority.

KL DEL SYNCHRONOUS reverses this process, changing links to bypass the event block to be deleted.

KL NEXT SYNC: BCFB, ∅256

This function searches the synchronous list for an event with a higher priority than that set in (B195). If no such event is found, the routine returns with carry clear.

If a suitable event is identified, the routine returns with carry set, HL holding the event block address, and A holding the event priority (Class), which is also stored in (B195). The event is removed from the synchronous list.

When KL NEXT SYNC finds a suitable event, it is processed by;

KL DO SYNC: BCFE, ∅21A

On entry, HL must point to an event block, as provided by KL NEXT SYNC. The event routine is called and executed. To complete the action, it is then necessary to call:

KL DONE SYNC: BD∅1, ∅277

On entry, HL must point to the relevant event block. The address is not provided by KL DO SYNC, so the address returned by KL NEXT SYNC must be saved on the stack while KL DO SYNC is executed. Similarly, A must hold the previous event priority, also provided by KL NEXT SYNC but not by KL DO SYNC. (CPC464 formal documentation specifies C instead of A, but the code uses A...)

The priority level in (B195) is set from A, and the count in the event block is decremented. If the count is then positive, non-zero, the event is returned to the synchronous list.

KL POLL SYNCHRONOUS: B921

This is a routine in RAM, entered directly to allow a quick check to be made of the first item on the synchronous list. If this has a higher priority than the current priority in (B195), the routine returns with carry true.

KL EVENT DISABLE: BD∅4, ∅295

Bit 5 of (B195) is set to 1, indicating an impossibly high priority.

KL EVENT ENABLE: BD∅7, ∅29B

Bit 5 of (B195) is zeroed.

The above description is nominally correct, but it leaves some questions unanswered. For example, how can (B195) be set by a named call? Is there any need to so set it?

Let us look over the system on a broader basis.

The intention is that the foreground program should make regular checks for outstanding synchronous events. This can be done by calling KL POLL SYNCHRONOUS. If the return is with carry set, the sequence;

```
L1    CALL  KL NEXT SYNC
      JR    NC,EXIT
      PUSH  HL
      PUSH  AF
      CALL  KL DO SYNC
      POP   AF
      POP   HL
      CALL  KL DONE SYNC
      JP    L1
```

can be run. This will process all events at the priority level. The level will initially be ∅, because initialisation clears (B195) to zero, but as soon as KL NEXT SYNC finds a top priority event, the current priority is set to that level, and all events of lower priority are barred.

What is needed here is an extension to the above routine. When EXIT is reached, if (B195) is not zero it is decremented, and the routine is re-entered. For those who do not relish the task of checking whether (B195) is the correct location in their system version, it is possible to set (B195) by calling KL DONE SYNC with the required value in A and HL pointing to a dummy event block...

Comment

The Event system is unlikely to be mastered completely in an afternoon. Since the lengths of the various lists are virtually unlimited, it would be possible to go berserk and create so many events that the system would have no time to attend to anything else, so a cautious approach is advisable.

It is possible to write quite complex programs without making use of events, but once the system is understood it provides enormous scope for ingenuity.

Because the contents of the lists are changing rapidly all the time, it can be difficult to trace exactly what is happening. The events provide a powerful tool, but - like all powerful tools - it needs to be used with care.

Other Kernel Routines

Some of the Kernel routines are not accessible through the Jumpblock. We have already met the program entry routine at ØØ77, and a routine at ØØ44 that copies code to RAM. There are also routines for adding an event to a list, or deleting an event, but these are not suitable for use in isolation. Two more Kernel routines, accessible through the Jumpblock, will suffice here;

KL TIME PLEASE: BDØD, ØØ99

The contents of the time counter are set in DE (upper word) and HL (lower word).

KL TIME SET: BD1Ø, ØØA3

The time counter is set from DEHL, with (B18B)=Ø.

The very compact routine for incrementing the time count is worth quoting here;

```
L1  HL=B187
L2  INC (HL)
    INC HL
    IF HL=Ø THEN L2
    RET
```

If a byte is incremented from &FF to Ø, a carry is required to the next byte. This could involve a spillover into (B18B) when FFFFFFFF is incremented. That would occur roughly once in 4ØØØ hours, but (B18B) would not return to zero for around a million hours. The system would then fail - if you can wait that long!

Four further Kernel routines are so closely linked with the external ROM system that they are best dealt with in that context.

Kernel Data Area

B1ØØ-B1Ø1	Kicked List Base
B1Ø2-B1Ø3	Kicked List End
B1Ø4	Flag byte
B1Ø5-B1Ø6	SP Hold
B1Ø7-B186	Special stack
B187-B18B	Time count
B18C-B18D	Frame Fly List Base
B18E-B18F	Fast Ticker List Base
B19Ø-B191	Ticker List Base
B192	Ticker Count
B193-B194	Sync List Base
B195	Current Priority
B196-B1A5	Command Word Copy
B1A6-B1A7	Command Chain Base
B1A8	Current ROM
B1A9/B	Far Address Qualifier
B1AC-B1B8	Data Area Pointers

Chapter 5
THE DISPLAY SYSTEM

In all, the Display System takes up some 4ØØØ bytes of code and fixed data in ROM, and its workspace spans about 38Ø bytes of RAM, not to mention the 16K byte screen RAM. There are more than 1ØØ entry points. Fortunately, the system divides into three main parts:

* The Screen Pack deals directly with screen handling, colour selection and screen read and write.

* The Text VDU handles matters relating to text display, including the implementation of stream selection. It also deals with the control codes and their parameters.

* The Graphics VDU handles the graphic display.

Each of these parts requires a chapter to itself, but it will be useful to offer some general information first.

The Screen RAM

In theory, the Screen RAM could be any 16K byte area of memory starting at a multiple of 4ØØØ, but the 8ØØØ-BFFF block would overwrite workspace and RAM routines, while ØØØØ-3FFF would overwrite the RST Area, so the choice narrows to 4ØØØ-7FFF or CØØØ-FFFF, and it is usually more convenient to adopt the latter area, leaving the central half of RAM free for other purposes.

The Screen RAM is accessed by the Video Gate Array on a basis of addresses supplied by the CRT Controller, but the addresses are not used in a straightforward manner. The CRT Controller embodies two counters. One, output on RAØ-RA4, is incremented after each line of the display has been scanned. When this count reaches the value set for the number of scan lines in the character height it is zeroed, and the second counter, output on MAØ-MA13, is incremented. This counter is initialised to the Start Address set in the CRT Controller, which is 3ØØØ when the CØØØ-FFFF area is in use. These outputs are used as follows;

* Address bits A14,A15 are driven from MA12,MA13. Since the MA counter works from a Start Address of 3ØØØ, both these bits are true.

* Address bits A11-A13 are driven from RAØ-RA3

* Address bits A1-A1Ø are driven from MAØ-MA9

* Address bit AØ is driven from the CRT Controller clock.

The scan line takes 4Ø microseconds to traverse the visible part of the display, and during each microsecond the Video Gate Array requires two bytes of screen data. These are transferred directly from RAM to the Video Gate Array, the processor being meanwhile held in Wait. The process is so timed that the CRTC clock changes state between the two transfers. Once all the bytes have been read, the normal processor action is allowed to continue.

The bytes are used in different ways in the three screen modes.

In Mode 2, each byte defines one row of a character pattern matrix, each bit determining which of two colours should be given to a pixel, and eighty characters are displayed in each screen row.

In Mode 1, two bytes are required to define each matrix row, two bits being used to give each pixel one of four colours. Successive pixels are defined by bits 3,7; 2,6; 1,5; and Ø,4. This sequence is repeated in the second byte From each pair of bits, the Video Gate Array determines which palette entry should be used, and sets the colour accordingly. Since each matrix row requires two bytes, only forty characters can be displayed per screen row.

In Mode Ø, four bits are required to define one of sixteen colours for each pixel. This means that four bytes are required for each matrix row. The first pixel is defined by bits 1,5,3,7 of the first byte, the second by bits Ø,4,2,6 and so on. Twenty characters can be displayed in each screen row.

The way the CRT Controller counts are used complicates the calculation of screen addresses. Numbering columns and rows from Ø:

Address=Base + Offset + N*Column + 8Ø*Row + 2Ø48 per scan line.

where N is the number of bits per pixel in the current mode.

For a given scan line, the bits are taken in sequence. The next scan line is located by increasing the addresses by Ø8ØØ.

Fortunately, the system will work out screen addresses on the basis of column, line, base and offset.

Observant readers may notice a slight anomaly. If N*Column = 79, and Row = 25, the column and row terms in the above equation total 1999, so there are 48 locations in each scan line that are spare. The MA counter in the CRTC does not address them.

However, there is the Offset term to be taken into account. Making offset = &5Ø moves the screen up one line. Making the offset Ø8ØØ would move the display up by one scan line, but offset is limited to Ø7FF by the routine normally used to set it. When Offset is used, the Start Address in the CRTC is modified, and the missing 48 bytes may then come into play. There is a lot of scope for gentle experiment here.

Streams

The system provides for the definition of eight 'streams' of screen data, each with its own independent parameters, which are:

Window
Cursor Position
Pen and Paper
Cursor Enable
Screen Enable
Opaque or Transparent
Text or Graphics Write
Roll Type

All eight sets of parameters are held in store, the set in current use being copied into a common area.

Each stream may reserve for itself a rectangular window. If two windows overlap, the streams may overwrite each other in the overlap area. Any area not so reserved may be accessed by stream Ø, which is the default if no stream is specified.

Parameters

A certain amount of care is needed in dealing with screen parameters, as their definition can vary. A distinction is made between 'physical' and 'logical' values, the former numbering columns and rows from Ø upwards, while the latter start at 1. There are also distinctions between absolute and relative values.

Similar distinctions arise with Graphics parameters, user coordinates being relative to the origin set by the user, while standard coordinates are relative to the default origin.

Workspace

As many workspace locations are common to more than one section of the display system, the addresses for the whole screen workspace in Version 1.Ø are given here

Screen Pack

B1C8	Mode
B1C9-B1CA	Offset
B1CB	Base (high byte)
B1CC-B1CE	Jump instruction
B1CF-B1D6	Pixel masks
B1D7	Flash Time 2
B1D8	Flash Time 1
B1D9-B1E9	Colour Table 2
B1EA-B1FA	Colour Table 1
B1FB	Table select flag
B1FC	Flash count
B1FD	Colour time
B1FE-B2Ø6	Event Block
B2Ø7	Bits/pixel, negated.

Graphics

B328-B329	X Origin
B32A-B32B	Y Origin
B32C-B32D	X Position
B32E-B32F	Y Position
B33Ø-B331	Window Left
B332-B333	Window Right
B334-B335	Window Top
B336-B337	Window Bottom
B338	Encoded Pen
B339	Encoded PAPER
B33A-B341	Matrix copy
B342-B343	X Hold
B344-B345	Y Hold

Text VDU

B2ØC	Current Stream
B2ØD-B21B	Stream Ø data
B21C-B22A	Stream 1 data
B22B-B239	Stream 2 data
B23A-B248	Stream 3 data
B249-B257	Stream 4 data
B258-B266	Stream 5 data
B267-B275	Stream 6 data
B276-B284	Stream 7 data
B285	Current Row
B286	Current Column
B287	Current Roll Type
B288	Current Top
B289	Current Left
B28A	Current Bottom
B28B	Current Right
B28C	Current Roll Count
B28D	Current Cursor Flag
B28E	Current Screen Enabl
B28F	Current Pen
B29Ø	Current Paper
B291-B292	Link for print mode
B293	Graphic Write Flag
B294	1ST RAM Matrix Code
B295	Matrix Flag
B296-B297	Address of RAM Matri
B298-B2B7	Pattern Hold
B2B8	Parameter Count
B2B9	Control code
B2BA-B2C2	Control Parameters
B2C3-B322	Control Jump Table

Chapter 6
THE SCREEN PACK

The Screen Pack routines occupy the ØAAØ-1Ø6E area of ROM, and provide 34 defined entry points and three indirections. The routines deal with screen Mode selection, address calculations, colour control, and similar matters. We will begin by looking at the initialisation routines.

SCR INITIALISE: BBFF, ØAAØ

MC CLEAR INKS is called with DE=1Ø4D, clearing all palette entries to &Ø4. Screen Base is set to CØØØ, and SCR RESET and SCR CLEAR follow.

SCR RESET: BCØ2, ØAB1

SCR ACCESS is called with A=Ø to select normal write mode. The indirections SCR READ, SCR WRITE and SCR MODE CLEAR are reset to the default addresses. ØCD2 is called to copy default colour data from 1Ø4D-1Ø6E to the two colour tables at B1D9-B1E9 and B1EA-B1FA. Flash times are set to one fifth of a second. The colour select flag in (B1FB) is zeroed.

SCR CLEAR: BC14, ØAF2

Mode 1 is selected, with appropriate mask settings, and SCR MODE CLEAR follows.

SCR MODE CLEAR: BDEB, ØAF7

This is an indirection, and must not be called when lower ROM is disabled.

ØD4F is called to disable the flash system, which will be examined later. SCR OFFSET is called with HL=ØØØØ to standardise the screen map, then the screen RAM is cleared to zero entries, using LDIR. The routine exits via ØD3C to re-activate the flash system.

Mode Control

SCR SET MODE: BCØE, ØACA

On entry, A must hold the number of the mode required. If the number is outside the Ø-2 range the routine returns immediately.

Otherwise, ØD4F is called to disable the flash system, then 1ØB7 is called to initialise the streams, this being a routine in the Text VDU area. 15D6 in the Graphics VDU is called to set graphics pen and paper. The Mask Table is then set up as shown below.

The Mode number is set in (B1CB) and MC SET MODE is called to reset the Video Gate Array. SCR MODE CLEAR is called, then GRA INITIALISE at 15B6, following the call to GRA RESET. The final exit is via 1ØD5, which leads into TXT STR SELECT.

MASK TABLE

	Mode Ø	Mode 1	Mode 2
B1CF	&AA	&88	&8Ø
B1DØ	&55	&44	&4Ø
B1D1	*	&22	&2Ø
B1D2	*	&11	&1Ø
B1D3	*	*	&Ø8
B1D4	*	*	&Ø4
B1D5	*	*	&Ø2
B1D6	*	*	&Ø1

Asterisks indicate entries which are set up but not used.

The table picks out the bits of the screen data bytes which are to be used to define individual pixels. For example, in Mode 1 the first pixel is defined by bits 3 and 7, so the mask is &88.

SCR GET MODE: BC11, ØAEC

A is set from (B1C8) and compared with 1. This gives flags C,NZ for Mode Ø, NC,Z for Mode 1, and NC,NZ for Mode 2. There are a number of internal calls to this routine, and the flag state is more often used than the number in A.

Addresses

SCR SET OFFSET: BC∅5, ∅B3C

On entry, HL must hold the required screen offset. H=H AND 7, and then (B1C9/A)=HL. SCR GET LOCATION is called, and the routine exits via MC SCREEN OFFSET to reset the CRT Controller.

Officially, the given offset is limited to an even number in the range ∅-∅7FE, which is desirable, since odd numbers could cause confusion, but the routine does not zero bit ∅, so the user must attend to the limitation.

SCR SET BASE: BC∅8, ∅B45

The upper byte of the required screen base must be held in A on entry. It is masked by A=A AND &C∅, allowing the base to be ∅∅∅∅, 4∅∅∅, 8∅∅∅ or C∅∅∅, and the result is set in (B1CB). SCR GET LOCATION is called, and the routine exits via MC SCREEN OFFSET to reset the CRT Controller.

SCR GET LOCATION: BC∅B, ∅B5∅

HL=(B1C9/A), Offset, and A=(BC1B), Base upper byte.

SCR CHAR LIMITS: BC17, ∅B57

SCR GET MODE is called. For all modes C=&18 (screen rows less one), while B is set to the number of screen columns less one. These values match the 'physical coordinates', which start at ∅.

SCR CHAR POSITION: BC1A, ∅B64

On entry H must hold a physical column number and L a physical row number. The corresponding screen address is calculated and returned in HL, and the number of bits per pixel for the current mode is returned in B.

SCR DOT POSITION: BC1D, ∅B95

This is really a graphics function. On entry, DE must hold the X coordinate of a pixel, and HL must hold the Y coordinate, both being expressed in terms of absolute displacement from the bottom left corner of the screen. The screen address of the byte relating to the pixel is returned in HL, B holds bits/pixel less one, and C holds a bit mask identifying the relevant bits of the specified screen byte.

We now come to four routines which are by no means easy to follow. In each case an address in HL is modified to point to a byte in an adjacent screen position.

SCR NEXT BYTE: BC2∅, ∅BF9

L is incremented, and if the result is non-zero the routine returns.

Otherwise, a carry to H is required, so H is incremented, but if this gives H AND 7 = ∅, H=H-&∅8. The end of a block has been reached, and correction is required.

SCR PREV BYTE: BC23, ∅C∅5

L is decremented, and if it was not previously zero the routine returns.

Otherwise, H is decremented, and if previously H AND 7 < > ∅ the routine returns. Otherwise H=H+&∅8 to apply the necessary correction.

SCR NEXT LINE: BC26, ∅C13

H=H+8, moving to the corresponding byte in the next scan line. If H AND &38 < > ∅, the routine returns. Otherwise, the address has gone out of range, and H=H-&4∅, L=L+&5∅, taking the address to the other end of screen RAM and then forward one line. Finally, if H AND 7 = ∅, then H=H-8.

SCR PREV LINE: BC29, ∅C2D

H=H-8. If H AND &38 < > &38, the routine returns. Otherwise, H=H+&4∅, L=L-&5∅. If H AND 7 = ∅ then H=H+8.

It is useful to picture the screen RAM as being divided into eight sections, each of which deals with one particular matrix row for all characters. It may help to draw out a map of part of the screen - but use a large sheet of paper!

Inks and Flashing Colours

The colour system involves some disconcerting translations, but its otherwise fairly straightforward.

SCR INK ENCODE: BC2C, ØC86

The ink number held in A on entry is converted to an ink mask, which is returned in A.

First, ØCC2 is called to interchange bits 1 and 2 of A if Mode Ø is in use. Then an eight-iteration loop is entered with E initially holding the ink number, original or modified, and C holds (B1CF), the first colour mask.

Bit Ø of E is copied to bit Ø of A, E being rotated right and A being shifted left in the process. C is shifted right, and if the bit which passes from C into the carry is Ø, E is rotated left, restoring its previous contents. The routine loops.

For Mode 2, C holds &8Ø, so the contents of E remain unaltered until the last iteration. Bit Ø of E is set in all locations of A, each of which relates to one pixel.

For Mode 1, C holds &88. Bit Ø of E is set in bits 4-7 of A, and bit 1 of E is set in bits Ø-3 of A.

For Mode Ø, bearing in mind the bit exchange, the bits of A are set as follows:

Bit of A:	7	6	5	4	3	2	1	Ø
Bit of E:	Ø	Ø	1	1	2	2	3	3
Colour bit:	Ø	Ø	2	2	1	1	3	3

SCR INK DECODE: BC2F, ØCAØ

The above process is reversed, an encoded ink in A on entry being converted to an ink number in A on exit.

The above two routines preserve BC, DE and HL.

SCR SET INK: BC32, ØCEC

SCR SET BORDER: BC38, ØCF1

These two entries share a common routine. On entry, B and C must hold colour numbers, which should be the same for no flash, different for flash. For SCR SET INK A must hold an ink number, but for SCR SET BORDER A is zeroed. For SCR SET INK, A=A AND &ØF + 1, giving the range 1 to &1Ø.

One might expect that the subsequent process, common to both entries, would be quite simple, the colour numbers being entered in the locations of the two colour tables indicated by the ink numbers, but an additional process is required, the colour numbers being converted by reference to the following table:

&ØØ	&14	&Ø8	&ØD	&1Ø	&Ø7	&18	&ØA
&Ø1	&Ø4	&Ø9	&16	&11	&ØF	&19	&Ø3
&Ø2	&15	&ØA	&Ø6	&12	&12	&1A	&ØB
&Ø3	&1C	&ØB	&17	&13	&Ø2	&1B	&Ø1
&Ø4	&18	&ØC	&1E	&14	&13	&1C	&Ø8
&Ø5	&1D	&ØD	&ØØ	&15	&1A	&1D	&Ø9
&Ø6	&ØC	&ØE	&1F	&16	&19	&1E	&1Ø
&Ø7	&Ø5	&ØF	&ØE	&17	&1B	&1F	&11

The colour number on the left of each pair becomes the colour number on the right of the pair. Note that only the first half of the table is used.

The converted numbers are entered in the two colour tables, and then (B1FC)=&FF to warn the colour system that the colours have changed.

SCR GET INK: BC35, ØD14

SCR GET BORDER: BC3B, ØD19

Here again, a common routine is used, except that SCR GET INK requires an ink number to be held in A on entry, whereas SCR GET BORDER sets A to zero. The ink is read from the colour tables, and then converted by reverse reference to the table above. The results are returned in B and C., with the first colour in B.

SCR SET FLASHING: BC3E, ØCE4

The contents of HL on entry are set in (B1D7/8). The two bytes give the colour flash periods in fiftieths of a second. The time for the first colour is given by the second byte...

SCR GET FLASHING: BC41, ØCE8

HL=(B1D7/8): See previous routine.

The Flash System

Colour flashing is executed automatically by an event on the Frame Flyback list. The event details are:

Event Block Address:	B2ØØ
Class: Asynchronous, Near Address	
Routine Address:	ØD5B

The actual event routine is one of a group of small routines which are closely interlinked. They are described below in address order:

Called by SCR CLEAR

ØD3C The event is removed from the Frame Flyback list, ØD6D is called, and the event is returned to the list.

ØD4F The event is removed from the list. ØD81 sets DE and A and the routine exits via MC SET INKS.

Event Routine

ØD5B The current flash count in (B1FD) is decremented, and if the result is zero ØD6D is called to change colours. If (B1FC)()Ø, indicating that colours have been reset, ØD81 is called to set DE and A, and MC SET INKS is called. Finally, (B1FC)=Ø.

Called by ØD3C and ØD5B

ØD6D ØD81 is called to set DE and A, and (B1FD)=A, setting the current flash count. MC SET INKS is called, the colour select flag in (B1FB) is complemented, and (B1FC)=Ø.

Called by ØD4F, ØD5B and ØD6D

ØD81 DE is set to point to a colour table, and A is set to a flash count. If (B1FB)=Ø, the first colour is selected, the data being B1EA, (B1D8). Otherwise, the second colour is represented by B1D9, (B1D7).

None of these routines are accessible via the Jumpblock.

General Routines

SCR FILL BOX: BC44, ØDB3

SCR FLOOD BOX: BC47, ØDB7

The difference between these two routines lies in the way the input parameters are expressed. SCR FILL BOX calls a parameter conversion routine at ØB95, and then executes SCR FLOOD BOX.

On entry to either routine, A must hold the encoded ink to be used. For SCR FILL BOX, H=left column, L=top row, D=right column, E=bottom row. These are physical coordinates, from Ø upwards.

ØB95 calculates E=(E-L+1)*8, the number of scan lines in the height of the box. Then D=(D-H+1), the number of characters in the box width. HL is preserved. SCR CHAR POSITION is then called to return in HL the address of the top left corner of the box (defined in HL). B is also set, to give bits/pixel for the current mode, and this allows the calculation D=D*B to be made, giving the number of bytes in the box width. C=A.

The entry conditions for SCR FLOOD BOX are precisely the same as the exit conditions for SCR FILL BOX; HL must hold the screen address for the top left corner of the box, D must hold bytes in box width, E must hold scan lines in box height, and C must hold encoded ink.

A loop is entered at ØDB7. HL is pushed, A=D, and ØEE8 is called to check whether the line addresses are in straightforward sequence. If they require no corrective action, ØEE8 returns with carry clear, in which case a simple LDIR routine can be used to set the bytes in a line. This is the faster method, but it cannot be used in all cases.

If ØEE8 returns with carry set, the sequence (HL)=C:SCR NEXT BYTE is repeated D times.

In either case, SCR NEXT LINE is called, and the routine loops back to ØDB7 E-1 times, clearing all the scan lines in the box.

SCR CHAR INVERT: BC4A, ØDDF

On entry, B and C hold different encoded inks, H holds a physical column number, and L holds a physical row number. The colours of the character at the indicated position are interchanged.

C=B XOR C, forming a mask which indicates the bits which are different in the two colours. The process (HL)=(HL) XOR C is applied to all the bytes forming the character.

SCR HW ROLL: BC4D, ØDFA

The screen can be rolled by simply changing Offset if the window area covers the whole screen. This is called Hardware Roll.

The contents of B on entry determine the direction of the roll. B=Ø gives a downward roll, otherwise the roll is upwards.

First, the 48 unused screen RAM locations are cleared to background colour. They will form part of the line which is brought into the visible screen area. The routine then calls MC WAIT FLYBACK before changing the offser by +/- &5Ø and clearing the remaining 32 locations in the new line.

SCR SW ROLL: BC5Ø, ØE3E

If the current stream has a window defined which is smaller than the size of the full screen, Software Roll has to be used, allowing areas outside the window to remain unaffected. As with Hardware Roll, the contents of B on entry determine the roll direction, B=Ø giving roll down.

There are separate routines for the two directions of roll, but the principle is the same for both. The line which is to go off screen is overwritten by copying the next line over it, and the process is repeated for the remaining lines. Finally, the last line is cleared, using SCR FLOOD BOX.

Where possible, LDIR is used for the copying process, but - as with SCR FLOOD BOX - this is not possible if block or line boundaries have to be crossed. It is worth noting that bulk screen manipulations are very much faster when Offset is zero and there are no windows.

SCR UNPACK: BC53, ØEF3

A character pattern matrix defines the character shape, but only in a manner directly suited to Mode 2. For the other modes, the matrix has to be spread out over 16 bytes for mode 1, 32 bytes for Mode Ø. This process is known as 'unpacking' the matrix.

On entry, HL points to the start of a character matrix block, which may be in ROM in the 38ØØ-3FFF area or in a user defined-area of RAM. DE must point to an area of RAM large enough to hold the unpacked matrix.

Separate routines are provided for each mode. In Mode 2, the matrix is copied directly into the receiving area without modification.

For Mode 1, the conversion is as follows;

Matrix Byte;	abcdefgh
1st Unpacked Byte;	abcdabcd
2nd Unpacked Byte;	efghefgh

The conversion is determined by reference to the colour masks in B1CF-B1D2.

For Mode Ø, the colour masks in B1CF-B1DØ are used, and the conversion is:

Matrix Byte;	abcdefgh
1st Unpacked Byte;	abababab
2nd Unpacked Byte;	cdcdcdcd
3rd Unpacked Byte;	efefefef
4th Unpacked Byte;	ghghghgh

The general method is that the original matrix byte is shifted left bit by bit, and if the bit transferred to carry is true A=A OR (Colour Mask). The colour mask is changed for each shift, and the process is repeated for all eight bytes of the matrix.

SCR REPACK: BC56, ØF49

The reverse process takes the unpacked matrix from Screen RAM, so it is necessary to specify a character position, H holding column and L holding row, both in physical coordinates. Colour must also be taken into account, so A must hold encoded ink. DE points to an eight-byte area of RAM in which the basic matrix can be reconstructed.

C=A, and SCR CHAR POSITION is called to convert HL to a screen address, after which the routine divides for the three modes:

For Mode 2, the following action is repeated for each matrix byte;

```
  A=(HL) XOR C
  Complement A
  (DE)=A
  DE=DE+1
  CALL SCR NEXT LINE
```

For the other modes, A=(HL) OR C, and the result is compared with the colour masks for the mode. If A AND (Colour Mask)=Ø, a 1 is shifted into the output byte, otherwise a Ø is shifted in, the byte being shifted left through carry.

SCR ACCESS: BC59, ØC49

SCR WRITE can work in four different modes, and SCR ACCESS determines which mode is to be used. A jump destination is determined by the contents of A on entry to SCR ACCESS. The four modes, with B=Encoded Ink and C=Pixel Mask, are given below, with the values of A which will cause SCR ACCESS to select them.

```
  FORCE MODE (A=Ø);

  A=(HL)

  A=A XOR B

  A=A OR C

  A=A XOR C

  A=A XOR B

 (HL)=A
```

The implications of this are best shown by a truth table;

(HL)	B	C	XOR B	OR C	XOR C	XOR B
Ø	Ø	Ø	Ø	Ø	Ø	Ø
Ø	Ø	1	Ø	1	Ø	Ø
Ø	1	Ø	1	1	1	Ø
Ø	1	1	1	1	Ø	1
1	Ø	Ø	1	1	1	1
1	Ø	1	1	1	Ø	Ø
1	1	Ø	Ø	Ø	Ø	1
1	1	1	Ø	1	Ø	1

When C=1, the sequence OR C, XOR C produces a zero state, whereas when C=Ø the previous state is unaltered. In the latter case XOR B ,XOR B makes no difference to (HL), so bits can only be changed where C=1, when the final state depends solely on the initial state of B.

XOR MODE (A=1);

A=B AND C
(HL)=A XOR (HL)

Bits that are true in both B and C are reversed in (HL).

AND MODE (A=2);

A=C complemented.
A=A OR B
(HL)=A AND (HL)

Bits in (HL) that correspond to zero bits in B coupled with true bits in C are zeroed.

OR MODE (A=3);

A=B AND C
(HL)=(HL) OR A

Bits which are true in both B and C are made true in (HL).

Perhaps the best way to understand the implications of these modes is to try them out.

SCR PIXELS: BC5C, ØC6B

This is identical to SCR WRITE in FORCE mode, except that it is not an indirection.

SCR HORIZONTAL: BC5F, ØFC4

All graphic lines are made up of horizontal or vertical segments, and this routine draws the horizontal segments. On entry A must hold an encoded ink, DE the X start coordinate, BC the X end coordinate, and HL the Y coordinate. All coordinates are absolute displacements from the bottom left corner of the screen.

The routine is lengthy but fairly straightforward. HL determines the scan line to be used, and DE and BC determine the end points. Note that these are not affected by the last plotted point.

SCR VERTICAL: BC62, 1Ø2F

This is similar to SCR HORIZONTAL, but is much simpler, because a given dot position has to be set in a series of scan lines. On entry, A must hold an encoded ink, DE the X coordinate, HL the Y start coordinate, and BC the Y end coordinate.

Two Indirections remain to be covered:

SCR READ: BDE5, ØC82

On entry, HL holds the screen address of a byte, and a mask in C identifies pixels covered by the byte. These can be obtained from SCR DOT POSITION. On exit, A holds the decoded ink for the pixel. The routine consists of A=(HL), followed by the relevant part of SCR INK DECODE.

SCR WRITE: BDE8, ØC68

On entry, HL holds the screen address of a character position, C holds a pixel mask, and B holds an encoded ink. A jump to B1CC finds a further jump set up by SCR access, and this leads to one of the four write routines already examined.

Comment

The Screen Pack routines should not be judged solely in isolation. They are workhorses designed to serve higher level functions, and it is these higher level functions that are most likely to be used. For example, SCR WRITE sets only one screen byte, since it may be needed to set a single pixel, whereas the display of a character requires the setting of up to 32 bytes.

That is the business of the Text VDU, which also has to make sure that the data calls for character display, and not a control action.

The Screen Pack deals with operations that involve the screen, largely as a servant of the Text VDU and Graphics VDU, but there are times when it can be accessed directly with advantage. In any case, it is worth careful study, because it needs to be understood as a basis for study of the VDU routines.

Chapter 7
THE TEXT VDU

The Text VDU occupies 1Ø78-15AØ in ROM, and provides 36 entry points and five indirections. Its main task is the placement of characters on the screen, but it also handles control codes.

TXT INITIALISE: BB4E, 1Ø78

TXT RESET is called, then (B295)=Ø indicates that there is no user matrix table. 113D is called with HL=ØØØ1 to set:

(B28D)=3	Cursor disabled and off.
(B28F)=H	Paper Ø
(B29Ø)=L	Pen 1
(B293)=Ø	Not Graphic Write
(B291/2)=1391	Opaque Mode

Window=Full screen

VDU ENABLE

This sets the current stream, and the routine exits via 1ØA3 to copy this stream data to all the streams.

TXT RESET: BB51, 1Ø88

The five indirections are set to default addresses. The link table for control codes is copied from 146B-14CA in ROM to B2C3-B322 in RAM

Screen and Cursor Control

TXT VDU ENABLE: BB54,1451

TXT CUR ENABLE is called, (B28E)=&FF, (B2B8)=Ø

We now have to deal with a group of routines related to cursor positioning. They are complicated by the fact that user coordinates are expressed in 'logical' terms, counting from 1, while the stored data is in 'physical' terms, counting from Ø.

TXT SET COLUMN: BB6F,115E

On entry, A must hold the required logical column number. Adding window left and subtracting 1 gives the absolute column. H=A, L=(B285), which is the row number, and TXT UNDRAW CURSOR is called. Then (B285/6)=HL, and the routine exits via TXT DRAW CURSOR.

Note that the cursor has to be removed before the coordinates are changed.

TXT SET ROW: BB72, 1174

This is similar to the above, except that A must specify a required row. L=A+(window top)-1, H=(B286), column, and the cursor sequence follows.

TXT GET CURSOR: BB78,118Ø

Nominally, this routine returns the cursor position relative to the current window limits, but the validity of the position is not checked, and if it lies outside the window the position may be changed before it is used.

```
HL=(B285/6) sets H to absolute column, L to absolute row.
H=H-(B298)+1 converts to user column coordinate.
L=L-(B288)+1 converts to user row coordinate.
A=(B28C) picks up the roll count.
```

The roll count is decremented at roll up, incremented at roll down. It only serves to indicate whether a roll has occurred.

Cursor enable control is dependent on bits Ø and 1 of (B28D). Bit Ø is controlled by the user (ENABLE/DISABLE), while bit 1 is controlled by the system (ON/OFF). If either bit is true, the cursor does not appear.

TXT CUR ENABLE: BB7B, 1289

TXT UNDRAW CURSOR is called. Then (B28D)=(B28D) AND &FE, zeroing bit Ø. The routine exits via TXT DRAW CURSOR.

TXT CUR DISABLE: BB7E, 129A

As the above routine, except that (B28D)=(B28D) OR 1, setting bit Ø.

TXT CUR ON: BB81, 1279

As TXT CUR ENABLE, except that bit 1 of (B28D) is zeroed.

TXT CUR OFF: BB84, 1281

As TXT CUR DISABLE, except that bit 1 of (B28D) is set.

The latter two routines preserve AF.

We now encounter an oddity, two jumpblock entries which call the same entry point, though their action is equal and opposite. The cursor is made to appear, if it is absent, or disappear if present, by inverting the character at the cursor position.

TXT PLACE CURSOR: BB8A, 1268

TXT REMOVE CURSOR: BB8D, 1268

BC, DE and HL are preserved. A subroutine at 11AB sets HL from (B285/6) to column and row for the cursor position, checks the validity of the position in relation to the current window, using subroutine 11DA (see TXT VALIDATE), and sets (B285/6) from the resulting contents of HL, which may have been changed to bring the position within the window. If carry is set, the 11AB subroutine returns. Otherwise, roll is required. If B=Ø, A=1, while if B=&FF, A=&FF. A is added to the roll count, B indicates the required direction of roll.

TXT GET WINDOW is called. If it returns with carry set, SCR SW ROLL is called, otherwise SCR HW ROLL. In either case, A=(B29Ø), paper.

Back in the main routine, B=Pen, C=Paper, and SCR CHAR INVERT is called to invert the character at the cursor position.

The old cursor should have been removed first, using the same routine with the old cursor position.

Linked with this dual-purpose routine we have a pair of indirections:

TXT DRAW CURSOR: BDCD, 1263

TXT UNDRAW CURSOR: BDDØ, 1263

If (B28D)<>Ø, the routine returns, taking no action. Otherwise, TXT PLACE CURSOR follows.

TXT VALIDATE: BB87, 11CE

This routine checks whether the cursor position is inside the current window, and adjusts the position if it is outside the window. On entry, H holds column and L holds row, in logical coordinates counted from 1. The coordinates are converted to absolute coordinates by adding window left-1 and window right-1. A subroutine at 11DA then checks and adjusts as follows:

If H>window right, H=window left, L=L+1

If H<window left,H=window right, L=L+1

If L<window top, L=window top, B=Ø, and the routine returns with carry clear and B=Ø to call for a roll down Otherwise, if L(window bottom, return with carry set. Barring that, L=window bottom, and the routine returns with carry clear and B=&FF to call for a roll up.

HL is reconverted to logical coordinates. If carry is clear, roll is executed.

Colour

TXT SET PEN: BB9Ø, 12A9

TXT SET PAPER: BB96, 12AE

Apart from setting HL=B28F for pen, and HL=B29Ø for paper, these entries use a common routine. On entry, A holds the ink to be set. TXT UNDRAW CURSOR is called, then SCR INK ENCODE. (HL)=A, and the routine exits via TXT DRAW CURSOR.

The cursor has to be removed, since the change of colour would otherwise upset the inversion process.

TXT GET PEN: BB93, 12BD

TXT GET PAPER: BB99, 12C3

For pen, A=(B28F), for paper A=(B29Ø). SCR INK DECODE follows.

TXT INVERSE: BB9C, 12C9

Pen (B289) and paper (B29Ø) are interchanged.

TXT SET BACK: BB9F, 137A

This determines whether background colour should be written (opaque) or left unaltered (transparent). The action is determined by setting a link address:

If A=Ø, (B291/2)=1391, a link to Opaque.

If A=1, (B291/2)=139F, a link to Transparent.

The chosen routine is called by TXT WRITE CHAR, but it will be convenient to describe the action here. C holds the matrix byte, and DE holds the screen address;

Opaque

```
HL=(B28F/9Ø), pen and paper.
B=H AND (C complemented), the paper mask.
A=C AND L, then pen mask.
C=&FF, calling for all pixels to be set.
B=A OR B, the combined mask.
EX DE/HL puts the screen address in HL
SCR PIXELS follows.
```

Transparent

```
B=(B28F), pen
EX DE/HL
SCR PIXELS follows.
```

Note that only FORCE mode is available.

TXT GET BACK: BBA2, 1387

If (B291/2)+EC6F=Ø, the link set is for opaque, and the routine returns with A=Ø. If the link is set for transparent, A<>Ø.

Windows

TXT WIN ENABLE: BB66, 12ØC

On entry, D and H must hold physical columns for the window side positions, the larger being used to define the right side. Similarly, E and L hold the top and bottom rows, the larger defining the bottom row. The coordinates are checked for validity, and adjusted if they are outside the screen area. The resulting values are set in (B288/B) (see next routine).

If the window covers the whole screen, (B287)=Ø, this being the flag which selects hardware or software roll. The cursor position is moved to the top left corner of the window.

TXT GET WINDOW: BB69, 1256

L=(B288)	Top
H=(B289)	Left
E=(B28A)	Bottom
D=(B28B)	Right

If (B287)=Ø, the routine returns with carry clear, permitting hardware roll. If carry is set, software roll is needed.

TXT CLEAR WINDOW: BB6C, 154Ø

The cursor is 'undrawn', (B285/6)=(B288/9), setting the 'home' position at the top left of the window, HL and DE are set as for TXT GET WINDOW above, and SCR FILL BOX is called. If enabled, the cursor is restored.

The multiple streams, each of which can have its own set of parameters, complicate a number of the above routines. The cursor manipulations are also more complex than might be expected.

On the other hand, the flexibility of the screen system justifies the complexity, and makes a close study of the system worthwhile.

Streams

Much use has been made of the current screen data in (B285-B293). At any time, the data for another stream may be copied into this area from the copies held in (B2ØD-B284). The previous current data is saved in its copy area.

TXT STREAM SELECT: BBB4, 1ØE8

On entry, A holds the number of the required stream. A=A AND 7 ensures that a number Ø-7 is used. If A=(B2ØC), the current stream number, the routine returns without taking action.

Otherwise, BC and DE are pushed, C=(B2ØC), (B2ØC)=A, changing the current stream number. B=A, A=C. DE=B2ØD+15*A sets the address of the copy area for the current stream, and HL=B285, BC=ØØØF prepares for LDIR to save the current parameters to the copy area.

A is then set to the new stream number, and the copy process is repeated, though with an interchange of DE and HL to reverse the transfer direction. A is set to the number of the previous current stream.

TXT SWAP STREAMS: BBB7, 11Ø7

This routine involves some interesting sleight of hand. On entry, B and C hold the numbers of two streams, the data for which is to be interchanged.

A=(B2ØC) notes the current stream number, and AF is pushed.

TXT STR SELECT is called with A=C to store the current stream data and bring stream C data into the current area. Then (B2ØC)=B, and DE is set to B2ØD+15*B. DE is pushed, A=C, and DE=B2ØD+15*C. The address pushed from DE is popped into HL, and the copy routine transfers stream B data into the stream C area. AF is popped, and TXT STR SELECT is called to copy the old stream C data (in the common area) into the stream B area (because (B2ØC)=B). The stream defined in A, which was current before this process began, is brought into the current area.

Matrix Data

Standard character matrix patterns are held in ROM at 38ØØ-3FFF, but special patterns can be set up in RAM by the user. When such RAM patterns are set, the matrix flag in (B295) is non-zero, and (B294) holds the code for the first character in the user table. The address of the start of the RAM table is held in (B296/7).

When a pattern is wanted, it may be in ROM or RAM, and the choice is made by:

TXT GET M TABLE: BBAE, 132A

HL=(B294/5), putting the matrix flag in H and the code for the first user-defined character in L. If H<>Ø, carry is set, then HL=(B296/7), the start of the user-defined table, if any. If carry is clear, HL is ignored, but in any case A=L.

TXT GET MATRIX: BBA5, 12D3

On entry, A holds a character code.

12D3 DE is pushed, E=A, and TXT GET M TABLE is called. If it returns with carry clear, the routine jumps to 12E3. There is no user table. If, otherwise, E is less than the value of A set by TXT GET M TABLE, the code is below the range of the user table, and the routine again jumps to 12E3. Otherwise, 12E6 follows, with E=E-A.

12E3 HL=38ØØ, the base of the ROM table.
12E6 HL=HL+8*E, forming the address of the required matrix. DE is popped, and the routine returns.

To create a user matrix table, the RAM area must first be defined and the relevant data must be set up.

TXT SET M TABLE: BBAB, 12FD

On entry, E holds the code for the first character in the table. and D holds Ø. However, if D<>Ø the existing table is cancelled. HL holds the start address of the table.

12FD HL is pushed, saving the start address of the RAM table. If D< >Ø the routine jumps to 131D with D=Ø, which zeroes the matrix flag. Otherwise, D=&FF and DE is pushed. C=E, and DE and HL are interchanged.

13Ø8 A=C, and TXT GET MATRIX is called. If, as might be expected, the matrix flag is clear, the address of the matrix in ROM which corresponds to the character code in E on entry is set in HL. If HL=DE, the routine jumps to 131C, else:

1314 BC is pushed, and eight bytes are copied from (HL) to (DE) by LDIR. BC is popped, and C is incremented to point to the next character. If C<>Ø. the routine loops to 13Ø8.

131C DE is popped.

131D TXT GET M TABLE is called, then DE is copied to (B294/5), the matrix flag being set from D and the first character code from E. The start address of the new table is then popped and set in (B296/7)

This prepares for the user table entry, the table consisting of a copy of the relevant part of the ROM table. The user matrices must now be set up;

TXT SET MATRIX: BBA8, 12F1

The matrix is set up by copying from a specified source, which could be another existing matrix. On entry, A holds the code for the matrix and HL points to the copy source. A return with carry clear reports failure, carry true means success. Failure is an indication that the character is not within the defined user table, or that no table has been defined.

DE=HL, and TXT GET MATRIX is called. If it returns NC, the routine drops out. Otherwise DE and HL are exchanged, and LDIR copies the matrix into its place.

Text Output

The text output system is complicated by the interlinking of the relevant calls. TXT OUTPUT uses TXT OUT ACTION (An indirection). TXT OUT ACTION handles control codes, but passes character codes to TXT WR CHAR or the Graphics system. TXT WR CHAR calls TXT WRITE CHAR, another indirection.

This complication can have advantages. For example, Graphic Write does not respond to control characters, so it displays them, which can be annoying. By altering the indirection TXT OUT ACTION, it is possible to intercept control codes and treat them more usefully.

The routines will be described in reverse order, from the end of the chain backwards towards the beginning, since this will make the action clearer.

TXT WRITE CHAR: BDD3, 134A

On entry, A holds an ASCII code, H holds a physical column, and L holds a physical row. (From Ø upwards)

134A HL is pushed, and TXT GET MATRIX is called to get the address of the required matrix. DE=B298, the start of the pattern hold area. DE is pushed, and SCR UNPACK is called to expand the matrix in the hold area. DE and HL are popped, and SCR CHAR POSITION is called to set a screen address. C=8.

135C BC and HL are pushed. This is an outer loop point.

135E BC and DE are pushed. This is an inner loop point. DE and HL are exchanged, C=(HL), and a call to 1376 accesses either opaque or transparent mode. (See TXT SET BACK) SCR NEXT BYTE is called, and DE is popped and incremented. BC is popped, and a DJNZ to 135E follows. This loop sets one scan line. (Note that B is set by SCR CHAR POSITION to the number of bytes per character width.) When the DJNZ drops out, HL is popped, SCR NEXT LINE is called, BC is popped, C is decremented, and if C<>Ø the routine loops to 135C, otherwise returning, all eight scan lines having been set.

TXT WR CHAR: BB5D, 1334

On entry, A holds an ASCII code. B=A. If (B28E)=Ø, the routine returns, display being disabled. Otherwise, BC is pushed, and a routine at 11A8 is called. This removes the cursor, checks validity, and if necessary corrects the column and line values to bring them within the current window. (B285)=H+1, setting the next column position, and AF is popped, recovering the code pushed from B. TXT WRITE CHAR is called, then TXT DRAW CURSOR.

TXT OUTPUT: BB5A, 14ØØ

TXT OUT ACTION is called with BC, DE, HL AND AF saved on the stack.

TXT OUT ACTION: BDD9, 14ØC

On entry, A holds a value which may be a character code, a control code, or a parameter following a control code. The routine must decide which it is.

If the parameter count in (B2B8)=Ø, the value is not a parameter, and must be an ASCII code. If it is &2Ø or more, it is a character code, otherwise it must be a control code.

If the graphic write flag in (B293)<>Ø, action passes to GRA WR CHAR, whether the value is a control code or not. This explains why graphic write insists on displaying control code symbols in such a disconcerting way.

The following description should be read in conjunction with the Control Code Table given hereafter. This details and interprets data held in RAM from B2C3 on.

The first action is to copy A to C and check the graphic write flag, jumping to GRA WR CHAR with A=C if the flag is non-zero.

The parameter count in (B2B8) is then checked, and if (B2B8) holds a number greater than 9 the routine drops out with (B2B8)=Ø. Something has gone astray, since no control code requires more than nine parameters. If (B2B8)=Ø and A holds a value greater than &1F the routine jumps to TXT WR CHAR.

Otherwise, B=(B2B8)+1, and an address is formed as B2B8+B. The value in A is stored at this address. Note that if the value is less than &2Ø, (B2B8) may hold zero, and the control code is stored at (B2B9), but if (B2B8)<>Ø the value is a parameter, and will be stored at (B2B9) to (B2C1).

The contents of (B2B9), the control code, are used to form an address B2C3+3*(B2B9). This picks out one of the three-byte groups in the Control Code Table. The first byte of the group specifies the number of parameter bytes required, and if the required number have not been found the routine drops out.

Otherwise, the routine indicated by the second and third bytes of the group is called. Then (B2B8)=Ø, and the overall routine returns.

CONTROL CHARACTER TABLE

Code	Parameters	Link	Function
&ØØ	Ø	14E2	Immediate Return. No action.
&Ø1	1	1334	TXT WR CHAR (Writes parameter)
&Ø2	Ø	139A	TXT CUR DISABLE
&Ø3	Ø	1289	TXT CUR ENABLE
&Ø4	1	ØACA	SCR SET MODE
&Ø5	1	1945	GRA WR CHAR
&Ø6	Ø	1451	TXT VDU ENABLE
&Ø7	Ø	14D8	SOUND QUEUE WITH HL=14CF (BEEP)
&Ø8	Ø	15ØA	CURSOR LEFT
&Ø9	Ø	15ØF	CURSOR RIGHT
&ØA	Ø	1514	CURSOR DOWN
&ØB	Ø	1519	CURSOR UP
&ØC	Ø	154Ø	TXT CLEAR WINDOW
&ØD	Ø	153Ø	Column=Window left
&ØE	1	12AE	TXT SET PAPER
&ØF	1	12A9	TXT SET PEN
&1Ø	Ø	154F	DELETE
&11	Ø	158E	CLEAR WINDOW LEFT TO CURSOR
&12	Ø	1584	CLEAR CURSOR TO WINDOW RIGHT
&13	Ø	156D	CLEAR WINDOW START TO CURSOR
&14	Ø	1556	CLEAR CURSOR TO WINDOW END
&15	Ø	144B	TXT VDU DISABLE
&16	1	14E3	TXT SET BACK
&17	1	ØC49	SCR ACCESS
&18	Ø	12C9	TXT INVERSE
&19	9	15Ø4	TXT SET MATRIX
&1A	4	14F8	TXT WIN ENABLE
&1B	Ø	14E2	Inmediate return. No action.
&1C	3	14E8	SCR SET INK
&1D	2	14F1	SCR SET BORDER
&1E	Ø	152A	HOME CURSOR
&1F	2	1538	LOCATE

In many cases, the routines are described elsewhere, but where parameters are involved the routines may not be entered directly, the parameters first being picked up in the appropriate registers. For cursor movements the cursor is removed, column and/or row are modified, and the cursor is restored. For Delete and bulk clearances, SCR FILL BOX is used. Note that the table is in RAM, so alterations can be made quite easily, giving the control codes new meanings.

The actual routines implementing the control codes are, in general, quite simple, and it seems unnecessary to examine them in detail here.

A closely allied routine is:

TXT GET CONTROLS: BBB1, 14CB

HL is set to B2C3, the start of the Control Code Table.

Other Text Routines

For the Edit system, it is necessary to be able to read a character from the screen and derive the corresponding ASCII code. Two routines are provided for this;

TXT READ CHAR: BB6∅,13AB

HL, DE and BC are pushed, and TXT UNDRAW CURSOR is called, so that the character at the cursor is not inverted, which would complicate matters. TXT UNWRITE is called with HL=(B285/6), row/column. AF is pushed, and TXT DRAW CURSOR is called. AF, BC, DE and HL are popped.

TXT UNWRITE: BDD6,13C∅

This is an indirection. On entry, H must contain a physical column and L a physical row, counted from ∅ upwards. The unpacked matrix is transferred from the screen position so defined to the hold area beginning at B298, using SCR REPACK entered with A=(B28F), pen. A comparison routine is called, and if it returns with carry set the whole routine drops out, with A holding the required ASCII code.

The comparison compares the repacked matrix with all the source matrices in turn. If no match is found, the comparison returns NC,Z, and in this case SCR REPACK is called again, this time with A=(B29∅), paper. The resulting matrix is inverted, and the comparison routine is re-entered. This allows for detection of characters in an unexpected ink.

One last entry point remains.

TXT SET GRAPHIC: BB63, 137A

(B293)=A. If A=Ø graphic write is disabled, otherwise it is enabled.

Comment

The text routines are perhaps a little more complicated than might have been expected, though most of the action is fairly straightforward. The use of inversion to create the cursor is not a complete success, since the resulting character is sometimes difficult to read, but enough information has been given to allow and encourage experimentation with alternatives, such as a simple underline. The necessary routines can be accessed via TXT DRAW CURSOR and TXT UNDRAW CURSOR.

Chapter 8
THE GRAPHICS VDU

The graphics VDU occupies the area 15BØ-19DC in ROM, and provides 23 entry points and 3 indirections.

Since the various ways in which coordinates are expressed can be confusing, some prelimiary explanation will be useful.

The current graphics position is stored in (B32C/D) for X and (B32E/F) for Y. The values are given in 'user coordinates'.

The Origin is initially at Ø,Ø - the bottom left corner of the screen - but can be altered at will. The coordinates of Origin are held in (B328/9) for X and (B32A/B) for Y.

An Absolute form of routine sets the current position from the contents of DE (X) and HL (Y) on entry. A Relative form of routine adds the current position coordinates to DE and HL, the Absolute form then being entered. The addition is performed by a routine at 1657.

For internal purposes, the coordinates have to be modified, the X coordinate being divided, in integer fashion, by the number of bits which represent one pixel for the current mode. If the original number is negative, an increment is applied. The absolute Y coordinate is halved, since it must point to one of 2ØØ scan lines.

Prior to these calculations, user coordinates are added to the Origin coordinates. Subroutine 161D performs this conversion, entry at 161A first calling GRA ASK CURSOR.

We can now turn to the initialisation routines.

GRA INITIALISE: BBBA,15BØ

GRA RESET is called, then Paper=Ø, Pen=1, Origin=Ø,Ø and the graphics window is set to the whole screen.

GRA RESET: BBBD, 15DF

The indirections for GRA TEST, GRA LINE, and GRA PLOT are set to the default addresses.

Setting Up

GRA SET ORIGIN BBC9,16Ø4

On entry, DE must hold the required X coordinate, HL the Y coordinate. These are absolute values. The origin coordinates in (B328/B) are set from DE and HL, which are then zeroed. GRA MOVE ABSOLUTE follows, which zeroes the current position coordinates in (B32C/F), effectively moving the cursor to the new origin.

GRA WIN WIDTH: BBCF, 1734

On entry, DE and HL must hold the standard (absolute) coordinates for the left and right edges of the graphic screen. The larger of the two will define the right edge. The values are limited so that they lie within the screen boundaries, and are converted to internal coordinates before being stored in (B33Ø/1), left, and (B332/3), right.

GRA WINDOW HEIGHT: BBD2, 1779

This routine is similar to the last, except that DE and HL hold the top and bottom coordinates of the window. Limited and converted, the coordinates are stored in (B334/5), top, and (B336/7), bottom.

GRA CLEAR WINDOW: BBDB, 17C5

GRA GET WINDOW WIDTH (see below) is called to get the coordinates of the left and right edges of the window, and from these the window width is determined. The number of scan lines in the window height is similarly calculated. SCR DOT POSITION is called, A is set from (B339), paper, and SCR FLOOD BOX is called to perform the clearance. Home cursor follows.

GRA SET PEN: BBDE, 17F6

SCR INK ENCODE is called to encode the ink set in A on entry, and the result is set in (B338).

GRA SET PAPER: BBE4, 17FD

As the previous routine, but the result is set in (B339).

Checking Values

GRA ASK CURSOR: BBC6, 15FC

DE is set from (B32C/D) to give the current X coordinate, and HL is set from (B32E/F) to give the Y coordinate.

GRA GET ORIGIN: BBCC, 1612

DE is set from (B328/9), X origin, and HL is set from (B32A/B), Y origin.

GR GET W WIDTH: BBD5, 17A6

DE is set from (B33Ø/1), left, and HL from (B332/3), right. The values are adjusted according to mode, giving absolute coordinates.

GRA GET PEN: BBE1, 18Ø4

A=(B338), and SCR INK DECODE is called.

GRA GET PAPER: BBE7, 18ØA

A=(B339), and SCR INK DECODE follows.

Main Functions

So far, the routines have been relatively trivial, though none the less essential. We now reach the main function routines.

GRA MOVE ABSOLUTE: BBC0, 15F4

GRA MOVE RELATIVE: BBC3, 15F1

For the relative version, subroutine 1657 is called, then the absolute version is entered. (B32C/D)=DE establishes the X coordinate, and (B32E/F) sets up the Y coordinate. The new values will take effect when the next major function is called.

GRA PLOT ABSOLUTE: BBEA, 1813

GRA PLOT RELATIVE: BBED, 1810

GRA PLOT: BDDC, 1816

For the relative version, subroutine 1657 is called, then the absolute version follows. This immediately calls GRA PLOT, which is an indirection.

16FC is called to adjust the coordinates in DE and HL for Mode, using 161D, and then perform a validation function by comparing the requested position with the window limits. If the position is not valid, the routine drops out. Otherwise, SCR DOT POSITION is called, B=(B338), and SCR WRITE follows.

Note that the ultimate action involves setting the bits which relate to a particular pixel, but the normal write action can be used.

GRA TEST ABSOLUTE: BBFØ, 1827

GRA TEST RELATIVE, BBF3, 1824

GRA TEXT: BDDF, 182A

These routines are related in the same way as the previous set, the indirection GRA TEXT leading to the main action. 16FC is called to check for validity of the requested position, and if it returns NC, indicating an invalid position, the routine exits via GRA GET PAPER.

Otherwise the routine exits via SCR DOT POSITION and SCR READ.

GRA LINE ABSOLUTE: BBF6, 1839

GRA LINE RELATIVE: BBF9, 1836

GRA LINE: BDE2, 183C

This is where the graphics system has to work for its living. The routines are too complex to examine in detail, but the gist of the action is this;

On entry, DE holds the X coordinate of the end point, and HL holds the Y coordinate. The start point is the present cursor position. The first step is to calculate the X and Y spans, the amount by which the two coordinates must change while the line is being drawn. Suppose the X span, AX, is the larger. Then AX is divided by AY, the Y span, and the result indicates how many X steps must be taken for each Y step. The calculation is on an integer basis, using a routine discussed under BASIC Support, but it is repeated after each Y step to minimise any resulting errors. Suppose that the result of the calculation is 5. Then a short horizontal line five units long is wanted, for a start. This is drawn by SCR HORIZONTAL.

AX is then reduced by 5, AY by 1, and the process is repeated to take the line a stage further, until AY=Ø, AX=Ø.

GRA WR CHAR: BBFC, 1945

On entry, A holds an ASCII code, not necessarily a normal character code. IX is saved on the stack, and TXT GET MATRIX is called. DE=B33A, the start of the copy matrix area, and IX=DE. The matrix is copied into the copy area.

The current screen address is adjusted according to mode, and the validity of the position is checked by 16FF. A return NC, showing that the position is not valid, leads to corrective action.

What follows is similar to the action of TXT WR CHAR, except that graphic coordinates are used. The routine exits via GRA MOVE ABSOLUTE to position the cursor at the top left corner of the next character position.

Comment

Grouping some of the functions together has made the graphics system look relatively simple, but that hides a great deal of complexity. The lack of circle and fill functions will be regretted in some quarters, but programs to add such facilities are available for those who need them.

A point which is sometimes missed is that the effective number of pixels in the screen width is 64Ø for Mode 2, 32Ø for Mode 1, and only 16Ø for Mode Ø. This can lead to disappointment if an attempt is made to combine intricate patterns with a wide range of colours.

Some programmers, especially some who live in Spain, have found means for creating displays of very great complexity that involve an apparently prodigal use of colours, and if an opportunity arises for a study of their methods it can be very revealing. Lacking that, it is worth trying out various experiments, using the details which have been given here as a guide.

Chapter 9 THE KEY MANAGER

The Key Manager occupies 19E∅-1E62 in ROM, and uses workspace in the B34C-B548 area. There are 26 defined entry points and one indirection.

The system uses three 'bit maps' to register key actions, and these are summarised in a table at the end of this section. Each individual bit relates to a given key, and the maps are updated 5∅ times a second by a routine called by the interrupt handler. This means that the map contents can only be examined coherently while interrupt is disabled. Note that interrupt is always disabled when the cassette system is in action, so the keyboard is effectively dead during that period.

When a key is depressed, the corresponding bit in map 1 is zeroed. The matching bit in map 2 is set to 1 if the map 1 bit changes from ∅ to 1 or is zero. This holds the map 2 bit true if the map 1 bit is dithering due to contact bounce.

Map 3 is then checked. If it holds ∅ for a particular key, and map 2 holds 1 for that key, data is placed in the keyboard buffer, and the map 3 byte is set from the map 2 byte.

This process is applied to all the ten map bytes in turn, taking the bits of each byte in order of increasing significance, so it is quite possible that more than one entry may be made during a single keyboard scan, the entries being made in ascending key number order. (Should you examine the relevant code, you may be puzzled by the function B=A AND -A: This sets one bit in B, corresponding to the least significant true bit in A.)

The keyboard buffer data bears no relation to ASCII codes, and only an indirect relationship to the key number. Each entry in the buffer occupies two bytes, and there is room for forty bytes, so up to twenty key depressions can be held at a given time. The upper byte is a one-bit mask, indicating which bit of a map byte is involved. The lower byte is compound. If Shift is pressed, bit 5 is true, and if Control is pressed, bit 7 is true. The number of the map byte involved (Ø to 9) is added. All the necessary data is covered, but it needs decoding.

As an illustration, consider what happens if you press key '@'. This is key 26 = 3*8 + 2, so we are dealing with bit 2 of the third map byte. If neither Shift nor Control is pressed, the keyboard buffer entry will be Ø4Ø3. With Shift pressed, the entry would be Ø423.

The keyboard buffer is of the 'circular' type, using two pointers. The input pointer is in (B53D) and the output pointer is in (B53F). They are single-byte displacement pointers. When an entry is added to the buffer, the input pointer is incremented, and when an entry is removed the output pointer is incremented. When either pointer reaches &14 it is reset to zero. Think of the pointers as the hands of a watch chasing each other round the dial, the buffer in use lying between them.

(B53C) is initialised to &15, and is decremented when an attempt is made to make a buffer entry, and incremented when an entry is removed. If the decrement gives zero, the attempt to make an entry is aborted, as the buffer is full, and (B53C) is reset to 1. It thus holds available buffer space (in words) plus one.

(B53E) is initialised to 1, and is incremented when a new entry is made, decremented when removal of an entry is attempted. If the decrement gives zero, the buffer is empty. The attempt to read an entry is abandoned, and (B53E) is reset to 1. It therefore holds the number of buffer entries (in words) plus one.

(B54Ø) is initialised to Ø, and is incremented when an entry is made, decremented when an entry is removed. It therefore gives a check on the current number of buffer entries, which can be useful.

The buffer access addresses are calculated by adding twice the pointers to B514. Since (B53D) and (B53F) work over the range Ø to &13, the buffer occupies B514-B53B.

An important point is that the shift states stored in the buffer are determined from bits 5 and 7 of the lower byte of the buffer entry, which are dependent on the shift state at the time the entry was made. The shift states at the time the entry is read are irrelevant. However, shift lock states have to be implemented separately. Note that information regarding changes of state is interleaved with other keyboard inputs.

Keyboard Routines

The above general description of the keyboard system has brought to light some important points regarding its operation, such as the length of the keyboard buffer, but the system can be used without worrying too much about the way in which it works. However, an understanding of the system will help to explain how the system calls function.

It will be best to begin with two routines that restore the system to a standard state, which can be valuable if you are tempted to experiment and get unexpected and unwelcome results.

KM INITIALISE: BBØØ, 19EØ

This is the major reset function, which affects everything:

* The key/code tables and part of the repeat control table are reset. by copying (1D69)-(1E62) to (B34C)-(B445) (see table).
* The keyboard Caps and Shift Lock states in (B4E7/8) are zeroed.
* Repeat speed in (B4E9) is set to 2
* Repeat delay in (B4EA) is set to &1E.
* Map 2, at (B45F)-(B4FE) is set to &FF entries.
* Map 3, at (B4EB)-(B4F4) is set to zero entries.
* (B541/2) is set to B34C, base of key/code table 1.
* (B543/4) is set to B39C, base of key/code table 2.
* (B545/6) is set to B43C, base of the repeat control table.
* The routine exits via KM RESET.

KM RESET: BBØ3, 1A1E

This lesser reset is still fairly drastic.

```
* (B53C)=&15 (Buffer free space plus 1).
* (B53D)=Ø   (Buffer input pointer).
* (B53E)=1   (Buffer entries plus 1).
* (B53F)=Ø   (Buffer output pointer).
* (B44Ø)=Ø   (Number of buffer entries).
* (B4EØ)=&FF ('Put back' character = 'ignore').
```

A key-string buffer of 152 bytes (&98) beginning at B446 is allocated, and the default strings are set up. (Unfortunately, this overlaps the repeat control table!)

The KM TEST BREAK indirection at BDEE is set to the default address.

The routine exits via KL DISARM BREAKS.

The amount of resetting involved hints at the scope for alteration...

Input Routines

The fact that there are four main data input routines can be confusing, but their inter-relationship is important.

KM READ KEY: BB1B, 1B5C

If there is an entry in the keyboard buffer, the routine exits with the appropriate code in A, carry being set. If the buffer is empty, the routine returns with carry clear. Registers other than AF are preserved.

KM WAIT KEY: BB18, 1B56

KM READ KEY is called repeatedly until it returns with carry set. This can be used for 'press any key to continue', whereas KM READ KEY just checks the buffer in passing to see if a key has been depressed since the last check. Registers other than AF are preserved.

KM READ CHAR: BBØ9, 1A42

First, the 'put back' character in (B4EØ) is checked. If it is not &FF, the character is returned in A with carry set, and (B4EØ)=&FF. (see KM CHAR RETURN)

Next, (B4DE/F) is checked. If (B4DF)<>Ø, a key string has been partially output, and routine returns with the code for the next character in the string set in A, with carry set.

If there is no put-back character or key string, KM READ KEY is called to look for a keyboard buffer entry. If the code returned in A is a key string token, output of the string is initiated, otherwise the routine returns with carry set and the code in A.

Failing a put-back character, a key-string character or a code from the buffer, the routine returns with carry clear.

In all cases, registers other than AF are preserved.

KM WAIT CHAR: BBØ6, 1A3C

KM READ CHAR is called repeatedly until it returns with carry set.

The difference between these routines should now be clear. The WAIT versions loop until a code is found, whereas the READ forms take one quick look. Only the CHAR versions look at the put-back and strings.

KM CHAR RETURN: BBØC, 1A77

This allows you to have your cake and eat it. When a character has been read, it has been taken from the keyboard buffer, and is no longer available. Calling KM CHAR RETURN transfers the code from A to (B4EØ), whence it can be read by KM READ CHAR, as if it had just come from the keyboard buffer. Only one character can be 'put back' at a time, as there is only the one hold location.

All registers are preserved.

Key Strings

The standard key-string buffer is at B446-B4DD, which overlaps the repeat control table, but an alternative buffer can be defined anywhere in RAM. Of the 152 locations in the standard buffer, 49 are set by default, leaving 1Ø3 for other definitions.

The buffer format is simple. Each string is prefaced by its length in bytes, so string n can be found by jumping forward n times from one length byte to the next. The default settings, which cover the keypad, are;

```
  B446                   Ø1 3Ø Ø1 31 Ø1 32 Ø1 33 Ø1 34
  B45Ø Ø1 35 Ø1 36 Ø1 37 Ø1 38 Ø1 39 Ø1 2E Ø1 ØD Ø5 52
  B46Ø 55 4E 22 ØD
```

The first ten strings are one character long, and give codes for the numbers Ø to 9. The eleventh string gives a full stop (&2E), the twelfth gives an Enter code, while the thirteenth string has five bytes, being RUN", followed by Enter.

A user buffer, with the above default strings initially set, can be set up by using:

KM EXP BUFFER: BB15, 1A7B

On entry, DE must hold the start address of the new buffer, HL must hold the buffer length, which must be &31 or more (not 44, as stated in the formal documentation.) If the buffer has been established, the routine returns with carry set, carry clear indicating failure. In case a string is in the process of being output, (B4DF) is zeroed, which stops the output immediately.

The next step is to set up the strings you want. This is done by:

KM SET EXPAND: BBØF, 1ABD

On entry, B must hold the relevant expansion token, C must hold the string length, and HL must point to the source of the new string. A return with carry set indicates success, carry clear showing failure, perhaps because the buffer was not large enough to hold the string.

First, the position of the string in the buffer is determined, using a routine which sets a pointer in HL, initially to the buffer start, reads the length byte, adds the length plus one to HL, uses the result to read the next length byte, and so on. When this has been done a number of times equal to the length token minus &8Ø, HL points to the string position. All entries above that are then moved to leave the necessary space for the new string, which can then be copied into place. (Note that the movement may be up or down, dependng on whether the new string is longer or shorter than the string it replaces.)

Compared with some key string implementations, this procedure is delightfully simple.

To read a string, you need:

KM GET EXPAND: BB12, 1B2E

On entry, A must hold the expansion token, while L holds the number of the character within the string to be read. On exit, success is shown by carry set, the character code being in A. If the string output process is complete, carry is clear and A is corrupt. In either case, DE is corrupt.

This call is used by KM READ CHAR, and would not normally be used independently, since it needs to be associated with a routine which will update relevant flags and the L pointer.

To find whether a particular key is pressed, without identifying the related code, you can use:

KM TEST KEY: BB1E, 1CBD

On entry, A must hold a key number. A return NZ means that the key is pressed.C holds the current shift and control states (see below), the lock states being ignored. A and HL are corrupt.

The information supplied comes from map 3. The shift and control bits in (B4ED) are isolated and set in C. The key number is then converted to byte and bit pointers, which are used to isolate the bit relating to the specified key.If the key is pressed, the bit is 1.

The state of Caps Lock and Shift Lock can be checked by:

KM GET STATE: BB21, 1BB3

On return, this routine sets L=&FF if Shift Lock is effective, H=&FF if Caps Lock is effective, the registers otherwise holding zero. All other registers are preserved, since the routine merely sets HL=(B4E7/8), locations which are toggled when the relevant lock codes are read from the key buffer.

Completing the keyboard read processes:

KM GET JOYSTICK: BB24, 1C5C

On exit L=(B4F1) AND &7F, giving the data for joystick 1 (keys 48-54) and H=(B4F4 AND &7F), giving the data for joystick Ø (keys 72-78). A=H. All other registers are preserved.

Key/Code Tables

The relationship between the keys and the codes they generate is determined by the three key tables in the B34C-B43B area. (see table.) Since the tables are in RAM, any key can be made to produce any desired code, with the restriction that codes in the &8Ø-&9F range will be treated as string tokens by KM READ CHAR (but not by KM READ KEY).

Three calls are available for changing the key table entries:

KM SET TRANSLATE: BB27, 1D52 Table 1

KM SET SHIFT: BB2D, 1D57 Table 2

KM SET CONTROL: BB33, 1D5C Table 3

On entry, A must hold a key number, and B the code which the key is to generate. Once the appropriate table base has been read, the routine is common to all three entry points. The call is rejected if A exceeds &4F, returning NC. Otherwise, A is added to the table base to form a pointer for setting the contents of B in the table.

Table 1 relates to no shift, no control. Table 2 is used when Shift is effective, and Table 3 applies when control is effective.

A corresponding set of entries allow a code to be read:

KM GET TRANSLATE: BB2A, 1D3E Table 1

KM GET SHIFT: BB30, 1D43 Table 2

KM GET CONTROL: BB36, 1D48 Table 3

The format is much the same as that for the previous three calls, except that the key number in A on entry is used to read a byte from the appropriate table and return it in A. HL is corrupt.

The keyboard lock state is ignored, being taken into account by KM READ KEY.

Repeat Action

The repeat action of a key can nominally be changed by:

KM SET REPEAT: BB39, 1CAB

On entry, A holds a key number. If B holds Ø, the key will be allowed to repeat, while if B holds &FF repeat will be barred. A key number greater than &4F is rejected, otherwise B is copied into the repeat control table.

The snag is that if A exceeds 9, B will also be copied into the standard key-string buffer, causing chaos unless you have set up a different buffer of your own...

To read the repeat table;

KM GET REPEAT: BB3C, 1CA6

On entry, A holds a key number. If that key can repeat, the routine returns NZ. A and HL are corrupt.

The repeat constants are handled by:

KM SET DELAY: BB3F, 1C6D

On entry, H must hold the delay factor and L the speed factor. The default values are &1E, giving a delay of 3Ø/5Ø=Ø.6 second, and L=2, giving a speed of 5Ø/2 repeats per second. The routine returns with AF corrupt.

KM GET DELAY: BB42, 1C69

The delay factor is returned in H and the speed factor in L. AF is corrupt.

Break Functions

To understand the Break calls, you need to have read the section on Events.

KM DISARM BREAK: BB48, 1C82

(B5ØC) is zeroed to mark the disarmed condition. KL DEL SYNCHRONOUS is called with HL=B5ØD to remove the break event from the synchronous list. The routine returns with AF and HL corrupt.

KM ARM BREAK: BB45, 1C71

On entry, DE holds C45E, the address of the break event routine, and C holds the relevant ROM number. (&FD, which means ROM unchanged, enable upper, disable lower ROM) This accesses the Break Routine in the BASIC interpreter, but other entry data can be used to access an alternative routine.

KM DISARM BREAK is called to establish a known state. Then KL INIT EVENT is called with HL=B5ØD and B=&4Ø to set up an event block at B5ØD-B513. The class is Far Address, Express, Synchronous. The routine address and ROM are as specified in DE and C. (B5ØC)=&FF marks the armed condition.

KM BREAK EVENT: BB4B, 1C9Ø

If (B5ØC)=Ø, the routine returns. Otherwise, (B5ØC)=Ø, and KL EVENT is called with HL=B5ØD to kick the break event. &EF is set in the keyboard buffer to mark the point at which the event was kicked.

KM TEST BREAK: BDEE, 1C9Ø

On entry to this indirection interrupt must be disabled to inhibit keyboard action, and lower ROM must be enabled. C must contain the Shift/Control key state, as found after interrupt was disabled.

If Shift and Control are not both pressed, the routine exits via KM BREAK EVENT.

Otherwise, it appears that a reset is being requested, but to make quite sure the bytes in map 3 are added up. If only Shift, Control and Escape are pressed, the total should be &A4, and if this total is found a jump to ØØØØ follows. Otherwise, the routine exits via KM BREAK EVENT.

Summary

There is really very little that you can do with a keyboard other than ask it to provide data. Most of the calls listed above are mainly relevant to other calls which make use of the data obtained. However, there are a few tricks worth mentioning.

You can empty the keyboard buffer by changing the pointers, but you must change them all. There is a routine (at 1CED in version 1.Ø) which does just this. (It is the destination of the first call in KM RESET, so should be easy to find in other versions.)

It is worth while to set up an alternative key-string buffer, so that the repeat table can work.

You can set up a special key/code table of your own, bringing it into action when necessary. This would allow extra codes to be generated by keyboard action.

A particularly important consideration arises if you are using a foreground program of your own, and it concerns Break. The C45E address suits the BASIC interpreter, but may not suit your program at all, but if you are using a sideways ROM, it will be entered at C45E in response to pressing ESCAPE.

In some respects the Keyboard Manager is one of the less satisfactory sections of the operating system, but it works reasonably well, despite that. Providing you know its quirks, they should not worry you.

Bit Maps

						Bit				
Map1	Map2	Map3	Ø	1	2	3	4	5	6	7
B4F5	B4FF	B4EB	Ø	1	2	3	4	5	6	7
B4F6	B5ØØ	B4EC	8	9	1Ø	11	12	13	14	15
B4F7	B5Ø1	B4ED	16	17	18	19	2Ø	21	22	23
B4F8	B5Ø2	B4EE	24	25	26	27	28	29	3Ø	31
B4F9	B5Ø3	B4EF	32	33	34	35	36	37	38	39
B4FA	B5Ø4	B4FØ	4Ø	41	42	43	44	45	46	47
B4FB	B5Ø5	B4F1	48	49	5Ø	51	52	53	54	55
B4FC	B5Ø6	B4F2	56	57	58	59	6Ø	61	62	63
B4FD	B5Ø7	B4F3	54	65	66	67	68	69	7Ø	71
B4FE	B5Ø8	B4F4	72	73	74	75	76	77	78	79

The numbers in the table are key numbers.

Key/Code Tables

Key No	Ø	1	2	3	4	5	6	7	8	9	1Ø	11	12	13	14	15
Table 1	FØ	F3	F1	89	86	83	8B	8A	F2	EØ	87	88	85	81	82	8Ø
Table 2	F4	F7	F5	89	86	83	8B	8A	F6	EØ	B7	88	85	81	82	8Ø
Table 3	F8	FB	F9	89	86	83	8C	8A	FA	EØ	87	88	85	81	82	8Ø

Key No.	16	17	18	19	2Ø	21	22	23	24	25	26	27	28	29	3Ø	31
Table 1	1Ø	5B	ØD	5D	84	FF	5C	FF	5E	2D	4Ø	7Ø	3B	3A	2F	2E
Table 2	1Ø	7B	ØD	7D	84	FF	6Ø	FA	A3	3D	7C	5Ø	2B	3A	3F	3E
Table 3	1Ø	1B	ØD	1D	84	FF	1C	FF	1E	FF	ØØ	1Ø	FF	FF	FF	FF

Key No.	32	33	34	35	36	37	38	39	4Ø	41	42	43	44	45	46	47
Table 1	3Ø	39	6F	69	6C	6B	6D	2C	38	37	75	79	68	6A	6E	2Ø
Table 2	5F	29	4F	49	4C	4B	4D	3C	28	27	55	59	48	4A	4E	2Ø
Table 3	1F	FF	ØF	Ø9	ØC	ØB	ØD	FF	FF	FF	15	19	Ø8	ØA	ØE	FF

Key No.	48	49	5Ø	51	52	53	54	55	56	57	58	59	6Ø	61	62	63
Table 1	36	35	72	74	67	66	62	76	34	33	65	77	73	64	63	78
Table 2	26	25	52	54	47	46	42	56	24	23	45	57	53	44	43	58
Table 3	FF	FF	12	14	Ø7	Ø6	Ø2	16	FF	FF	Ø5	17	13	Ø4	Ø3	18

Key No.	64	65	66	67	68	69	7Ø	71	72	73	74	75	76	77	78	79
Table 1	31	32	FC	71	Ø9	61	FD	7A	ØB	ØA	Ø8	Ø9	58	5A	FF	7F
Table 2	21	22	FC	51	Ø9	41	FD	5A	ØB	ØA	Ø8	Ø9	58	5A	FF	7F
Table 3	FF	7E	FC	11	E1	Ø1	FE	1A	FF	FF	FF	FF	FF	FF	FF	7F

Key Manager Workspace

B34C-B39B	Key/code table 1: No shifts
B39C-B3EB	Key/code table 2: Shift
B3EC-B34B	Key/code table 3: Control
B34C-B49B	Repeat Control Table
B446-B4DD	Keystring buffer (Note overlap)
B4DE	Keystring pointer
B4DF	Current string token
B4EØ	Put back character
B4E1/2	Keystring buffer start
B4E3/4	Keystring buffer end
B4E5/6	Start of free space in keystring buffer
B4E7/8	Keyboard lock state
B4E9	Repeat Speed
B4EA	Repeat Delay

B4EB-B4F4	Map 3
B4F5-B4FE	Map 1
B4FF-B5Ø8	Map 2
B5Ø9	Repeat Count
B5ØA	Map byte number
B5ØB	Map bit mask
B5ØC	Break Armed Flag
B5ØD-B513	Break event block
B514-B53B	Keyboard buffer
B53C	Buffer free space + 1
B53D	Input pointer
B53E	Buffer entries + 1
B53F	Output pointer
B54Ø	Buffer entries
B541/2	Pointer to key/code table 1
B543/4	Pointer to key/code table 2
B545/6	Pointer to key/code table 3
B547/8	Pointer to repeat control table.

Chapter 10 THE CASSETTE MANAGER

The Cassette Manager occupies 237Ø-2A91 in lower ROM, and uses B8ØØ-B8D4 in RAM as workspace. In addition to this, 2K byte buffer areas are used for intermediate storage during input and output. There are 22 defined entry points.

The recordings are based on square wave cycles, those representing high bits being twice as long as those representing low bits. The nominal mean frequency can be varied between 7ØØ and 25ØØ Hz, but the default frequency is 1ØØØ Hz, which makes the low bit cycle 666 microseconds long, with the high bit cycles 1332 microseconds long.

'Precompensation' is used to unbalance the duration ratio of the square waves, the default value being 25 microseconds. This is added to the high bit cycle and subtracted from the low bit cycle, emphasising the difference between them. Precompensation should be increased as the mean frequency is increased.

A useful feature of the system is that it adjusts automatically to the correct frequency during playback. This is based on reading the 'leader' block, which is made up thus;

Pre-record gap.
2Ø48 high-bit cycles.
One low-bit cycle.
A sync byte.

For data, the sync byte is &16, for a header it is &2C.

The overall structure of a file consists of a header record followed by a data record. The header block nominally contains 64 bytes, but most of these are unallocated. During recording, the header is taken from the B8Ø7-B846 area of RAM. During playback, it is set in the B88C-B8CB area. By reference to the header read during playback, the system can decide whether it should load the subsequent data block, or merely report it as 'found', which allows recovery from a read error by asking for the tape to be rewound if a block has been missed.

The system is remarkably tolerant to wow and flutter, though it can be defeated in extreme cases.

Data blocks can handle up to 65536 bytes, but it is more usual to limit them to 2Ø48 bytes, to keep buffer size within reasonable bounds. The blocks are divided into segments of 258 bytes each, two of these bytes being used to provide a cyclic redundancy check (CRC).

Since the recording and playback processes must be continous, the interrupt system is disabled while the recorder is in use. For the same reason, blocks which are built up from bytes created at intervals, or which may be read on a byte-by-byte basis, are handled via an intermediate buffer. In recording, the buffer is set up as the bytes become available, and the contents are recorded when the buffer is full, or when the buffer is closed. Similarly, in playback the buffer is filled from tape initially, and is refilled when all the data has been taken.

An exception to this arises when a block of data, such as a BASIC program, is already established, and can therefore be recorded directly. The same applies when data of this type is played back. The intermediate buffer is still used, but transfers to and from it are automatic.

The calls for the buffer method are CAS IN CHARACTER and CAS OUT CHARACTER, whereas for the more direct method the calls are CAS IN DIRECT AND CAS OUT DIRECT.

Some of the calls offer facilities not available via BASIC. CAS CHECK provides a verification facility, while CAS NOISY allows prompt messages to be suppressed. CAS RETURN resets the intermediate buffer pointers so that an entry can be read more than once, and the ABANDON calls allow buffer contents to be discarded.

For those who fancy experimenting with recording systems of their own, perhaps seeking compatibility with other computer recording systems, the primitives CAS READ and CAS WRITE are accessible. They deal with single records, handling data of given length starting at a given place in store.

The timing system used involves the Z8Ø Refresh Register, an unusual application of it which gives very precise short-term timings.

Messages

The prompt messages are stored in compressed form, with spaces omitted, but invoked by adding bit 7 to the preceding character code. There are four relevant error reports:

Read Error a:	Bit too long.
Read Error b:	CRC check failed.
Read Error d:	Block too long.
Write Error a:	Frequency too high.

Read Error a may occur if the tape is halted during playback, and has also been seen in a case of extreme wow which slowed the tape down from time to time. Read Error b is more common, and implies a defect in the tape surface. The other errors have not been seen, though they could be induced deliberately.

The ESCAPE key provides an exit from the cassette routines in limited circumstances. It is sensed directly, not through the keyboard buffer, which is ineffective in the absence of interrupt. This means that it may be necessary to hold the key down for some time before it takes effect.

Filenames may have up to 16 characters. Any additional characters will be ignored, while shorter names are padded out to 16 bytes with spaces.

The Routines

CAS INITIALISE: BC65,237Ø

CAS IN ABANDON and CAS OUT ABANDON are called, and CAS NOISY is called with A=Ø to enable prompt message display. Then CAS SET SPEED is called with HL=Ø14D (333) and A=&19 (25) to set the default speed and precompensation values.

CAS SET SPEED: BC68, 237F

On entry, HL must hold the length of half a low-bit cycle in microseconds, and A must hold the required precompensation in microseconds. HL is multiplied by 64. A is divided by 4 and added to HL, the result being set in (B8D1/2).

BC and DE are preserved.

CAS NOISY: BC6B: 238E

The contents of A are set in (B8ØØ). If A=Ø, prompts are enabled, else prompts are disabled.

All registers are preserved.

CAS START MOTOR: BC6E, 2A4B

There are no entry conditions. If the action was completed, the routine returns with carry set, but if Escape was pressed carry is clear. On exit, A holds the previous state of PPI port C, bit 4 of which controls the motor. (See CAS RESTORE MOTOR below).

After the motor has been switched on, there is a two-second delay before the routine returns, to give the motor time to get up to speed, unless the motor was already running.

CAS STOP MOTOR: BC71, 2A4F

This is identical with CAS START MOTOR, except that the motor is turned off instead of on, and there is no delay before return.

CAS RESTORE MOTOR: BC74, 2A51

On entry, A must hold the byte returned in A by the above two routines. The previous motor state is restored.

The above calls are invoked automatically by higher level routines, and are only of interest to the user if he wishes to move the tape without recording or playing back. However, the PLAY key must be locked down to allow this. Such an arrangement might be used to find a given recording automatically, by running the motor for a calculated length of time.

CAS IN OPEN: BC77, 2392

This routine sets up an input buffer tagged with the given filename, and initiates the procedures necessary to fill the buffer from tape and make the data available to program action.

On entry, B must hold the number of bytes in the filename, HL must contain the address of the filename start, and DE must point to the 2K byte area which is to be used as a buffer. As the buffer is read by RAM LAM, it may lie under a ROM.

The B8Ø2-B846 area is then initialised, largely by a routine shared with CAS OUT OPEN. The unallocated bytes are set to zero. The buffer is then set up by reading from tape, the read action being called automatically.

If all goes well, the routine returns with carry set and zero false. HL points to the stored file header, DE holds the data location which is specified in the header, BC holds the file length specified in the header, and A holds the file type.

If an input file was already open, the routine returns NC,NZ, while a return with NC,Z indicates that ESC was pressed.

If this routine has been executed successfully, up to 2K bytes can be read from the buffer by repeated use of CAS IN CHAR.

CAS IN CLOSE: BC7A, 23FC

CAS IN ABANDON: BC7D, 2401

The difference between these calls is that CAS IN CLOSE returns NC without doing anything if no input file is open. Otherwise, it runs on into CAS IN ABANDON, which is executed unconditionally.

There are no entry conditions. (B8Ø2) is zeroed to mark the file as closed. The buffer start address is copied from (B8Ø3/4) to DE. A is set to (B8CC) XOR 1, and if A=Ø (B8CC) is zeroed. (This is the prompt flag.) The contents of DE can be used to re-establish the buffer.

CAS IN CHAR: BC8Ø, 2435

This reads the next byte from the intermediate buffer into A, subject to validity checks. There are no entry conditions, all action being dependent on internal pointers.

Unless (B8Ø2) holds 1 or 2, the routine drops out NC,NZ without further action. (1 indicates buffer open, no data taken, and 2 shows that CAS IN CHAR has been used at least once.)

Otherwise, (B8Ø2)=2. If the count of available bytes in (B81A/B)=ØØØØ, all the data has been taken, and an attempt is made to read a further file from tape. If the end of the available data has been reached, the routine returns NC,NZ.

If the buffer still holds data, (B81A/B) is decremented, and HL=(B8Ø5/6), the buffer pointer. RAM LAM reads the next byte into A, and (B8Ø5/6) is incremented. The routine exits C,NZ, BC, DE and HL being preserved.

CAS IN DIRECT: BC83, 24AB

This call must be preceded by CAS IN OPEN, and CAS IN CHAR must not be called thereafter, since that would debar execution of CAS IN DIRECT. On entry HL must point to the start of the data area to be set.

If (B8Ø2) does not hold 1 or 3, the routine drops out NC,NZ, since either the buffer is not open or CAS IN CHAR has been used. Otherwise, (B81C/D)=HL, setting the data destination address, and the block is copied into position. The copy process is 'intelligent', LDIR or LDDR being used, as appropriate.

CAS RETURN: BC86, 249A

The bytes-in-buffer count in (B81A/B) is incremented, and the buffer pointer in (B8Ø5/6) is decremented. This makes the character last read from the buffer available again. However, there can be problems if the buffer has just been refilled....

CAS TEST EOF: BC89, 2496

CAS IN CHAR is called, and the routine drops out NC,Z if end of file is found. Otherwise, the routine exits via CAS RETURN to make the character read available again.

All registers except AF are preserved.

CAS OUT OPEN: BC8C, 23AB

On entry, B must hold the number of bytes in the filename, HL must hold the address of the filename, and DE must point to the start of a 2K byte area to be used as a buffer.

The (B847-B88B) area is set up, largely by a routine common to CAS IN OPEN. If an output file is already open, the routine returns NC. If ESC is pressed the routine returns NC, NZ. A return with carry set indicates success, and HL holds the address of the header buffer. The other registers (including IX) are corrupt.

CAS OUT CLOSE: BC8F, 2415

If (B847)=4, CAS OUT ABANDON is executed. If (B847)=Ø, the routine returns NC, there being no open output file. Otherwise, (B85D)=&FF, this being the end of file flag. If (B85F/6Ø)=ØØØØ, the buffer is empty, and CAS OUT ABANDON follows. Otherwise, a write to tape is attempted, and if this fails the routine drops out, otherwise CAS OUT ABANDON is called. It should be noted that if ESC is pressed during recording, the file is not closed.

CAS OUT ABANDON: BC92, 242E

No checks are made. (B847)=Ø. DE=(B848/9). the buffer address. If (B8CC)<>2 the routine returns with carry set. Otherwise (B8CC)=Ø, A=&FF, and the routine returns C, NZ.

CAS OUT CHAR: BC95, 245B

On entry, A must hold a byte to be added to the output file.

If (B847)<>1 or 2, the routine returns NC,NZ. The file status is incorrect. Otherwise, (B847)=2, indicating that a byte entry has been made. If the buffer is full, its contents are written to tape. Pressing ESC during the recording causes the routine to return NC, NZ. The output byte is set at the location determined by the buffer pointer in (B84A/B), and the pointer is then incremented. (B85F/6Ø), bytes in buffer, is also incremented.

BC,DE and HL are preserved, AF and IX are corrupt.

CAS OUT DIRECT: BC98, 24EA

Like CAS IN DIRECT, this routine handles data in bulk. On entry, HL must hold the address of the data, DE must hold the data length, BC must hold the start address, if any, and A must hold the file type.

An output file must be open, and it must be closed after the call to CAS OUT DIRECT.

If (B847)<> 1 or 3, the routine returns NC,NZ, the file status being incorrect. Either the file is not open or CAS OUT CHAR has been used. Otherwise, the specified data is copied into the intermediate buffer in 2K byte blocks, each block being recorded separately. The normal exit is C,NZ. An exit NC,NZ indictates incorrect file status, and NC,Z indicates that ESC was pressed.

Miscellaneous Calls

CAS CATALOG: BC9B, 2528

On entry, DE must point to a 2K byte area of RAM to be used as a buffer. If (B8Ø2)<>Ø, the routine drops out, as an input file is open. Otherwise (B8Ø2)=5, indicating catalog status. The buffer address is set in (B8Ø3/4), CAS NOISY is called with A=Ø to ensure that messages will be displayed, and tape is read, block by block, until ESC is pressed, when CAS IN ABANDON follows.

CAS WRITE: BC9E, 283F

This is the 'primitive' used by other routines to record tape. On entry, HL must hold the data address, DE the data length, while A holds the sync character, (&16 or &2C). Note that DE=ØØØØ would specify that 65536 bytes were to be written.

In case of failure or abort, the routine returns NC, A holding Ø if ESC was pressed, other errors being indicated by A=1.

CAS READ: BCA1, 2836

This is the corresponding 'primitive' for reading tape. On entry HL must hold the data address, DE must hold data length, and A must hold the expected sync character. This information can be determined by reference to the header: Data for the header is predictable.

CAS CHECK: BCA4: 2851

This is a verify function. On entry, HL must hold the data address, DE must hold data length, and A must hold the expected sync character. If the check is successful, the routine returns with carry set. Error is shown by NC, with an error code in A.

File Types

File type is more important than may be apparent, since it determines the way in which a file is handled. The file type byte is made up as follows;

```
  Bit Ø      If 1, the file is protected.
  Bits 1-3   ØØØ: BASIC
             ØØ1: Binary
             Ø1Ø: Screen Image
             Ø11: ASCII
  Bits 4-7   ØØØ: Not ASCII
             ØØ1: ASCII
```

  Other combinations not defined.

Note that adding &24 to the byte produces character codes, e.g.;

```
  BASIC:             &ØØ + &24 = &24: "$"
  BASIC Protected:   &Ø1 + &24 = &25: "%"
  Binary:            &Ø2 + &24 = &26: "&"
  Binary Protected:  &Ø3 + &24 = &27: "'"
```

and so on. However, the bit 5 entry for ASCII seems to be ignored.

Comment

One of the main problems in using the cassette system is the selection and protection of suitable buffer areas. Otherwise, the calls can be used quite simply, without much need to worry about how they work, unless you want to do something naughty, like removing protection. No, the method will not be given here, though it would not be too difficult to work out from the data provided. Like most protection systems, it is rather fragile...

Cassette Workspace

B8ØØ	CAS NOISY Flag. (Ø enables messages)
B8Ø1	Display Column Count (for messages)

Input File Control Block

B8Ø2	File status
B8Ø3/4	Buffer Address
B8Ø5/6	Buffer Pointer
B8Ø7/16	Filename
B817	Block Number
B818	EOF Flag
B819	File Type
B81A/B	Bytes In Buffer
B81C/D	Data Write Address
B81E	First Block Flag
B81F/2Ø	Data Length
B821/2	Execution Address
B823/46	Unallocated

Output File Control Block

B847	File Status
B848/9	Buffer Address
B84A/B	Buffer Pointer
B84C/5B	Filename
B45C	Block Number
B45D	EOF Flag
B45E	File Type
B45F/6Ø.	Bytes In Buffer
B461/2	Data Read Address
B463	First Block Flag
B464/5	Data Length
B466/7	Execution Start Address
B468/8B	Unallocated

Header Copy

B88C/9B	Filename
B89C	Block Number
B89D	Last Block Flag
B89E	File Type
B89F/AØ	Data Length
B8A1/2	Data Address
B8A3	First Block Flag
B8A4/5	Total Data Length
B8A6/7	Execution Start Address
B8A8/CB	Unallocated

General

B8CC	Prompt Flag
B8CD	Sync Character
B8CE/F	Timing
B8DØ	Timing
B8D1	Precompensation
B8D2	Speed
B8D3/4	Timing

Chapter 11
THE SOUND MANAGER

The Sound Manager occupies 1E68-2363 in lower ROM and uses the B55Ø-B7F9 area as workspace. There are 11 defined entry points.

The sequence of actions needs to be clearly understood. First, a block of nine data bytes defining a sound must be set up in RAM. If SOUND QUEUE is called with HL pointing to the block the data is transferred, in slightly modified form, to a queue slot, providing that a slot is available. There are four slots in all.

If the relevant channel of the PSG (Programmable Sound Generator) is free, the slot data is expanded into the common buffer for that channel. This is done by an event routine.

Part of the Interrupt Handler updates the parameters in the common area every 1ØØth of a second, passing any necessary instructions to the PSG.

Once SOUND QUEUE has been called, the rest of the process is automatic, a complex sequence of actions that must be maintained with precision. Any attempt to short-cut or modify the procedure is likely to lead to chaos.

A particular problem arises when more than five successive sounds are to be set up for a given channel. The first sound goes into the common area, the next four into the slots. Thereafter, SOUND CHECK needs to be called repeatedly to discover when a further entry can be made, but that could debar any other action. Waiting too long to make a check could cause discontinuity in the sound pattern.

The intended solution involves the use of an event block and associated routine, provided by the user. Once that is set up and enabled, the event will call its routine whenever there is a free slot, and the next sound queue entry can then be made. This could be quite a complicated business, especially if more than one channel is in use. Once again, the action is entirely automatic, which is convenient in some ways, restricting in others.

However, it is always possible to act on the PSG directly, using MC SOUND REGISTER to set the register defined in A to the data defined in C, and this may provide an escape route, especially for experimental purposes.

The function of the fourteen PSG registers can be summarised thus:

RØ	Tone period, Channel A, bits Ø-7
R1	Tone perios, Channel A, bits 8-11
R2	Tone Period, Channel B, bits Ø-7
R3	Tone period, Channel B. bits 8-11
R4	Tone period, Channel C, bits Ø-7
R5	Tone period, Channel C. bits 8-11
R6	Noise period. bits 1-4
R7	Enables (Ø enables, 1 disables)
	Bit Ø: Channel A Tone
	Bit 1: Channel B Tone
	Bit 2: Channel C Tone
	Bit 3: Channel A Noise
	Bit 4: Channel B Noise
	Bit 5: Channel C Noise
	Bits 6/7 control I/O ports, Ø for input. 1 for output.
R8	Channel A Volume
R9	Channel B Volume Ø-&ØF sets level
R1Ø	Channel C Colume &1X gives envelope
R11	Envelope Period, bits 1-7
R12	Envelope Period, bits 8-15
R13	Envelope Type: Ø to &ØF
R14/15	I/O Ports.

Used directly, the PSG will produce a useful variety of sounds, but the driving program must keep track of time, since there is no feedback to show that a sound is complete.

System Calls

SOUND RESET: BCA7, 1E68

To all intents and purposes, the workspace area is zeroed, exceptions being:

The event block, which is of the asynchronous near-address type, set up at B555-B55B.

Location &1C of each channel buffer is set to 4, indicating that there are four free slots available.

Locations Ø-2 of each channel buffer are set as follows;

		A	B	C
Loc Ø	Channel Number	Ø	1	2
Loc 1	Channel Bit	1	2	4
Loc 2	Rendezvous Bit	&Ø8	&1Ø	&2Ø

Envelopes are unchanged.

All channels are silenced.

SOUND QUEUE: BCAA, 1F9F

On entry, HL must point to a block of nine bytes defining the required sound:

Byte Ø:	Bit Ø:	Select Channel A if 1
	Bit 1:	Select Channel B if 1
	Bit 2:	Select Channel C if 1
	Bit 3:	Rendezvous with A if 1
	Bit 4:	Rendezvous with B if 1
	Bit 5:	Rendezvous with C if 1
	Bit 6:	Hold if 1
	Bit 7:	Flush if 1

Byte 1	Amplitude envelope Ø-&ØF
Byte 2	Tone envelope Ø-&ØF
Byte 3	Tone period, bits Ø-7
Byte 4	Tone period, bits 8-11
Byte 5	Noise period, bits Ø-4
Byte 6	Initial Amplitude
Bytes 7-8	Duration or envelope repeat count.

In all cases, SOUND CONTINUE is called to release any sound held on any channel.

If no channel is specified, the routine returns with carry set.

If bit 7 of byte Ø is true, the specified channel or channels will be flushed, which has much the same effect as SOUND RESET.

A check is then made for empty queue slots in the buffers for the specified channel or channels. If the slots for any specified channel are all occupied, the routine returns NC.

Otherwise, the data is transferred to the first free slot. Byte Ø is set to the channel bit, byte 1 is set to (16*Amplitude Envelope Number + Tone Envelope Number), and bytes 2 to 7 are set from bytes 3 to 8 of the data. The slot pointers are updated.

The event calling for entry of further data when slots are available is disarmed (If it exists), and it must kick itself again to maintain continuity. (It may be assumed that SOUND QUEUE will be called by such an event if that approach is adopted.)

SOUND CHECK: BCAD, 2Ø6C

On entry, A holds 1 to check Channel A, 2 to check Channel B, 4 to check Channel C. Channel status is returned in A:

Bits Ø-2:	Number of free slots
Bit 3:	Waiting for Channel A
Bit 4:	Waiting for Channel B
Bit 5:	Waiting for Channel C
Bit 6:	Channel Held
Bit 7:	Channel Active

The user sound event is disarmed.

SOUND ARM EVENT: BCBØ, 2Ø89

On entry, HL must point to the user event block, which must have been set up by KL INIT EVENT, and A must hold the relevant channel bit.

The event block address is set in the channel buffer area, but if there is a free slot the upper byte of the address is zeroed, and KL EVENT is called to kick the event. Zeroing the upper byte disarms the event, which must re-enable itself when it is executed, or after execution of SOUND QUEUE or SOUND CHECK.

SOUND HOLD: BCB6, 1ECB

All channels are silenced. If any channel was active, the routine returns with carry set.

SOUND CONTINUE: BCB9, 1EE6

Active channels that are held are released if their channel bit is set in A.

SOUND RELEASE: BCB3, 2Ø4A

On entry, A must hold the channel bits for the channels to be released. SOUND CONTINUE is called, and flags and pointers are updated.

SOUND AMPL ENVELOPE: BCBC, 2338

On entry, A must hold an envelope number, and HL must point to a data block of up to 16 bytes, as follows:

Byte Ø:	Number of sections
Bytes 1-3:	Section 1
Bytes 4-6:	Section 2
Bytes 7-9:	Section 3
Bytes 1Ø-12:	Section 4
Bytes 13-15:	Section 5

If Byte Ø=Ø, the envelope calls for a constant-level sound held for two seconds.

Either 'software' or 'hardware' envelopes can be specified. For a 'software envelope', the bytes within a section are:

Byte 1:	Step Count (1-127)
Byte 2:	Step Size
Byte 3:	Pause Time

If Step Count = Ø, Step Size becomes a volume level.

For a 'hardware envelope';

Byte 1:	Envelope Shape + &8Ø
Byte 2:	Envelope Period (L)
Byte 3:	Envelope Period (H)

The data relates directly to PSG registers 11-13.

Providing the envelope number in A is valid, the envelope data is copied unaltered to the envelope storage area. The routine returns with carry set and HL holding the address of the next envelope storage area, carry clear indicating failure.

SOUND TONE ENVELOPE: BCBF, 233D

This is very similar to the previous routine, except in the meaning of the data. There are two formats for the sections:

Byte 1:	Step Count (Ø-239)
Byte 2:	Step Size (-127 to +127)
Byte 3:	Pause Time (Hundredths of a second)

The step size is added to the current volume level.

Alternatively;

Byte 1:	(24Ø to 255)
Byte 2:	-127 to +127
Byte 3:	Pause Time

The tone period set is 256*(Byte 1 - 24Ø) + Byte 2

If bit 7 of the first byte of the envelope data is set true, the envelope repeats.

SOUND A ADDRESS: BCC2, 2349

This converts an amplitude envelope number in A on entry to the address of the envelope data in HL on exit. Exit NC means the number in A was not valid.

SOUND T ADDRESS: BCC5, 234E

As the previous routine, but for tone envelopes.

Comment

The sound routines are very ingenious - perhaps a little too ingenious for their own good. The appreciable automatic element can be frustrating to anyone who wishes to experiment, unless it is bypassed completely.

This comment may be coloured by long-term experience with sound systems which rely almost entirely on software, though involving PSG devices as well. Writing the routines and data for such systems can be tedious, but the programmer can feel he is in charge of the situation, whereas with automatic actions he has less control.

Nevertheless, the CPC464 system is versatile - more versatile than is apparent to a user of BASIC alone - and can produce quite interesting sounds.

It should be appreciated that the generation of music requires less data then might be thought. Once an envelope shape has been established, the only relevant variables are pitch and duration. The volume level may differ between channels, to obtain balance, but can usually be left at a constant level. What is needed in these circumstances is a routine that will pick up the essential variables and insert them into the data blocks used to call up notes.

Sound Manager Workspace

B55Ø	Flags for new entries
B551	Flags for active channels held
B552	Flags for active channels
B553	Interrupt Control Counter
B554	Outstanding action count
B555-B55B	Event Block
B55C-B59A	Channel A buffer
B59B-B5D9	Channel B buffer
B5DA-B618	Channel C buffer
B619	PSG Enable Byte
B61A-B7Ø9	Amplitude Envelopes
B7ØA-B7F9	Tone Envelopes

Buffer Format

&ØØ	Channel No
&Ø1	Channel Bit
&Ø2	Channel Rendezvous Bit
&Ø3	Channel Status Flags
&Ø4	Calls for tone setting if bit Ø=1
&Ø5	Interrupt Count 1
&Ø6	Interrupt Count 2
&Ø7	Calls for amplitude setting if bit Ø=1
&Ø8/&Ø9	Duration negated
&ØA/&ØB	Amplitude Envelope Address
&ØC	Number of sections in Amplitude Envelope
&ØD/&ØE	Amplitude Envelope Section Address
&ØF	Volume
&1Ø	Envelope type (hardware envelope)
&11/&12	Tone Envelope Address
&13	Number of sections in Tone Envelope
&14/&15	Tone Envelope Section Address
&16/&17	Tone Period
&18	Step Count

&19	Current Slot Number
&1A	Count
&1B	Slots in use
&1C	Slots Free
&1D/&1E	Event Block Address
&1F/&26	Slot 1
&27/&2E	Slot 2
&2F/&36	Slot 3
&37/&3E	Slot 4

The above are relative addresses within the channel buffers.

Chapter 12 EXTERNAL ROMS

It has been necessary to mention external ROMs from time to time, but a broader view of the subject must now be taken. Only the more dedicated enthusiasts are likely to venture into this area, which calls for a combination of hardware and software expertise, but the system is nevertheless well worth examination.

Nominally, up to 252 external upper ROMs could be added to the system to provide additional program space, the limit being determined by the format of FAR CALL and FAR ICALL, which interpret ROM numbers &FC-&FD in a special way. In practice, the number of ROMs added is likely to be much smaller.

Each ROM must be given a number, and when that number is output on DFXX the ROM must become active. This does not mean that it should immediately offer data to the highway. It may only do that when it is active, ROMEN is true (low) and address bit A15 is true. In these circumstances, the ROMDIS line must be pulled high before the external ROM is enabled, ensuring that the internal ROM (upper and lower) goes into a high-impedance state before the external ROM tries to drive the highway. Failure to attend to this detail can cause physical damage.

The internal upper ROM is thus disabled when an external ROM is active, even if the external ROM has been allocated the number Ø, which is usually associated with the internal upper ROM. In fact, the internal upper ROM is enabled when a number has been output on DFXX which does not match that allocated to any external ROM.

Foreground ROMs may be given any number, and it is possible to accomodate up to 64K of homogenous foreground program by using four 16K ROMs with consecutive numbers. SIDE CALL allows for simple access between the ROMs in such a group.

Background ROMs must be given numbers in the 1 to 7 range, so that they can be initialised by KL ROM WALK, as described hereafter.

The distinction between foreground and background ROMs can now be brought out more clearly. Only one foreground ROM can be active at a time, since it takes control, and only one controlling agency is allowed to exist in a system at a given time. Background ROMs, on the other hand, can be called into action on a temporary basis to support a foreground routine.

Entered at CØØ6, a foreground ROM must modify the store address data passed to it in BC, DE and HL to claim an adequate amount of workspace. The BASIC interpreter claims two initial areas, a low area at ØØ4Ø-Ø16F, and a high area at ACØØ-BØFF. These form boundaries to the main working area, which must hold the BASIC program, variables and strings, and cassette buffers. This area is called the memory pool, and it needs to be shared out with care.

Having claimed its own workspace, the foreground program must invite any background ROMs which it intends to use to claim workspace for their own needs. KL INIT BACK will do this for a particular ROM, while KL ROM WALK will initialise workspace for all the background ROMs in the system.

MN BOOT PROGRAM allows a program to be loaded into RAM and treated as a foreground program, but there is also provision for programs in RAM to serve in the background role. Such programs are known as 'Resident System Extensions' (RSX) and are expected to reserve their own data areas.

Command Words

While any program location can be accessed by the jumps and calls provided, it is sometimes more convenient to use command words. The on-board upper ROM has, in fact, only one command word, BASIC, and this accesses the initialisation entry at CØØ6, but other ROMs may define a whole host of special commands. Each such command requires to be prefaced by '|' to distinguish it from a BASIC command.

The first few locations of an external ROM must conform to the following format:

CØØØ:	ROM Type
CØØ1:	ROM Mark Number
CØØ2:	ROM Version Number
CØØ3:	ROM Modification Number
CØØ4/5:	Address of Command Table Start
CØØ6 on	Command Jumpblock

ROM types are;

```
  Ø:   Foreground
  1:   Background
  2:   Foreground Extension
  &8Ø: On-board ROM
```

The ROM format needs illustration by an example. Suppose that a ROM provides an enhanced printer driver, and can generate code sequences which set up the printer in a particular working mode. In simplistic terms, the start of the ROM might look like this:

```
  CØØØ: Ø1 Ø1 Ø1 Ø1
  CØØ4: ØØ C1                     Command Table at C1ØØ
  CØØ6: C3 ØØ C2                  Initialise at C2ØØ
  CØØ9: C3 ØØ C3                  DOUBLE at C3ØØ
  CØØC: C3 ØØ C4                  EXTEND at C4ØØ
  CØØF: .....                     Other links


  Command Table:

  C1ØØ 49 4E 49 D4                INIT
  C1Ø4 44 4F 55 4C C5             DOUBLE
  C1ØA 45 58 54 45 4E C4          EXTEND
  C11Ø ....                       Other Command Names
```

The command words must consist of upper case alphabetic characters (though full stops appear to be permissible) and &8Ø must be added to the code for the last letter in each word. The Command Table must be terminated by a zero.

For initialisation, the ROM is entered at CØØ6, no command word being involved. However, if the word EXPAND is set up, and KL FIND COMMAND is called, the routine will return with the entry address CØØC in HL and the appropriate ROM number in C. FAR PCHL will then access the required routine.

It is best to make the command words simple, though up to sixteen characters can be significant.

Similar facilities are available for programs in RAM, but in this case there are no constraints on the location of the Command Table. Linkage to the table is via a four-byte reference block, which in the case of background ROMs is placed immediately below the workspace area allocated for the ROM.

With such comprehensive facilities for program extension, it is natural that there have been queries regarding the possibility of extending RAM in a similar manner. This, unfortunately, is not easy to achieve. When a write to memory is executed, RAM is selected., and the output buffer to RAM is enabled, but this enable is not brought out to the extension connector, and there is no provision for over-ruling it. A simultaneous write to internal and external RAM might be achieved, if we could decode the available signals to derive a suitable enable, but a signal line selecting this mode would still be necessary.

It is possible, of course, to set up an external RAM and read it as if it was a ROM, but that is a rather different matter. Such a procedure could prove useful while developing ROMs, or in linking with another computer system, but there we are moving into deep waters.

Routines

The four Kernel Routines relevant to the Command Word system are;

KL LOG EXT: BCD1, Ø2A1

On entry, HL must point to a four-byte block of RAM which is otherwise unused. For a background ROM B=Ø and C holds the ROM number, but for an RSX BC must hold the address of the Command Table.

```
PUSH HL
DE = (B1A6/7)
(B1A6/7) = HL
(HL) = E
(HL+1) = D
(HL+2) = C
(HL+3) = B
```

This sets up the four-byte area. The first two bytes point to the last such area set up, so by a simple scan it is possible to start at B1A6/7 and trace through all the areas in turn.

A slightly different procedure is used for background ROMs and RSXs. the latter applying if B<>Ø.

KL INIT BACK: BCCE, Ø332

On entry, C must hold the number of the ROM to be initialised, DE must hold the address of the lowest free byte in memory, and HL must hold the address of the highest free byte. The address information is passed to the foreground ROM by the routine at ØØ77, which calls the ROM, and the foreground ROM must pass the addresses to KL INIT BACK, perhaps modifying them first to claim its own workspace.

If the contents of C are outside the 1 - 7 range, the permitted numbers for background ROMs, the routine drops out. Otherwise, KL ROM SELECT is called with A = C. If (CØØØ) AND 3 <> 1, the ROM is not of the background type, and the routine drops out via KL ROM DESELECT.

Otherwise, BC is pushed, and the ROM is called at CØØ6, the entry point for initialisation. The ROM must modify the address pointers to claim the workspace which it requires.

The modified values are returned in DE (LOMEM) amd HL (HIMEM). DE is pushed and DE=HL+1. HL is then set to B1AA + 2*(B1A8), (B1A8) being the number of the currently-selected ROM. DE is set in the location so defined. This is the new HIMEM, marking the start of the reserved workspace.

KL LOG EXT is then called with HL=DE-4, B=Ø and C holds the ROM number. This sets up the four-byte block in the area immediately below the workspace.

HL is set to point to the location below the four-byte block, and DE and BC are popped. The routine drops out via KL ROM DESELECT.

Note that the table set up at B1AA + 2*(B1A8) gives the bottom of each data area, not including the four-byte block. The top of the data area is not defined in a similar way, but can be noted by the ROM initialisation procedure.

Note, also, that it is not certain that a background ROM will always be allocated the same data area, but some of the access routines pass the address of the workspace start in IX when the ROM is called. (See RAM Routines).

KL ROM WALK: BCCB, Ø329

This calls KL INIT BACK with successive values of C decreasing from 7 to 1, and therefore initialises all available background

ROMs. Entry conditions are as for KL INIT BACK, except that C need not be set.

KL FIND COMMAND: BCD4, 02B2

This is entered with HL pointing to a command word set up in RAM. The command can be under a ROM, because the first action is to copy sixteen bytes of the word to (B196-B1A5). Bit 7 of the last byte of the copy is set, then HL = (B1A6), A=L. The routine jumps to Ø2D5, and then to Ø2C5 if (HL)< >Ø.

Ø2C5 HL is pushed, and BC=(HL+2,HL+3). This is the Command Table address for an RSX or the number of a background ROM. Ø2F4 is called to process the data accordingly.

If Ø2F4 returns with carry set, the command word has been matched, and the routine returns with DE holding the chain link pushed from HL (to clear the stack), while HL holds the entry address and C holds the ROM number.

Otherwise, the chain link is passed to HL, and HL=(HL) picks up the next link in the chain, so that the next command table can be checked.

Ø2D5 If HL< >Ø, the routine loops to Ø2C5 for a further search. If HL=ØØØØ, the end of the chain has been reached, and other possibilities need to be checked. C=&FF, and a further loop is entered:

Ø2DA C=C+1. KL PROBE ROM is called to check the ROM type. If this is Ø or &8Ø, foreground or on-board, Ø2F4 is called. It it returns with carry set, MC START PROGRAM is entered - so a command word could select a new foreground ROM...

Otherwise, if the class is not &8Ø, or if C=Ø, the routine loops to Ø2DA to try another ROM. Barring that, the routine returns with carry clear.

Ø2F4 HL=CØØ4, pointing to the command table address in a ROM, but if B< >Ø an RSX table is to be accessed, and HL=BC, C=&FF.

KL ROM SELECT is called, BC is pushed, DE=(HL), HL=HL+2. DE and HL are exchanged, and the routine jumps to Ø321.

At this point, DE holds the address of the command link table, and HL holds the address of the Command Table.

Ø3ØA BC=B196, the start of the command word copy.

Ø3ØD A=(BC). If A<>(HL), the routine jumps to Ø319. There is a mismatch. Otherwise, HL and BC are incremented, and the routine loops to Ø3ØD until bit 7 of A is true, marking a word end. If this condition is reached, a match has been found. DE and HL are exchanged, and the routine jumps to Ø325 with carry set (from bit 7 of A). HL points to the jumpblock entry, and C holds the ROM number.

Ø319 A=(HL), HL=HL+1. The routine loops to Ø319 until bit 7 of A is true, marking a word end.

DE=DE+3, advancing to the next jumpblock entry.

Ø321 If HL<>ØØØØ then back to Ø3ØA to try the next word.

Ø325 POP BC, exit via KL ROM DESELECT

That completes the routines relevant to Command Word functions.

Chapter 13 BASIC SUPPORT

The entries to the 2A98-37FF section of the lower ROM are not officially defined, because they are more a part of the BASIC interpreter than of the operating system. However, the 49 entries hold a host of treasures, especially for those whose programs involve mathematics.

The first entry, via BD3A, accesses the EDIT system, which is rather too specialised to be of general interest, being mainly concerned with the interpretation of various key combinations and the modification of data held in a buffer, the start of which is defined in HL on entry.

The entries relating to floating point arithmetic are of much more general importance.

Floating Point

A five-byte floating point system is used. For example:

```
  86 65 2E EØ D3
```

The first byte is the exponent. Subtracting &8Ø gives 6, so the value of the exponent is 2↑6 = 64. The remaining bytes form the mantissa, and the overall value of the number is found by multiplying together the values of the exponent and the mantissa.

The most significant bit of the second byte is the sign bit. In this case it is Ø, so the number is positive. However, the true value of the bit in numeric terms is always 1, so the value of the last four bytes - the mantissa - is:

```
  E62EEØD3 = 3,845,Ø54,675
```

The mantissa, however, is expressed in 'fractional binary', which means that the most significant bit has a value of 1/2, the next bit a value of 1/4, and so on. To find the real value of the mantissa we must divide by 2↑32. Multiplying the result by 64 gives the overall value of the floating point number as 57.2957795 = 18Ø/PI

A zero value is a special case, the exponent being Ø and the mantissa irrelevant. This can be seen as the next step from an exponent of &Ø1, which has a value 2↑-127. Combined with the minimum mantissa value, which is Ø.5, an overall value of 2↑-128 can be represented. The maximum possible value that can be shown is a whisker under 2↑127.

Floating point numbers are stored with the bytes in reverse order, the least significant byte of the mantissa first and the exponent last. A number of examples can be found in ROM in the 2E18-37FF area. Some are placed in the middle of a section of code, which makes disassembly difficult. This is because the 'power series' routine picks up constants from the locations following the instruction which calls it. For example:

```
  CD A9 32          CALL 32A9
  Ø4                Four entries in table
  4C 4B 57 5E 7F    Ø.4342597
  ØD Ø8 9B 13 8Ø    Ø.5765815
  23 93 33 76 8Ø    Ø.96180Ø7
  2Ø 3B AA 38 82    2.8853901
```

The routine continues at the location following the table, which in this case is taken from the LOG routine.

When a mantissa is brought into registers, it normally occupies DEHLC, C serving to collect any carry bits in right shift operations. These may be needed for rounding purposes.

There are three defined hold locations in RAM for floating point numbers:

HOLD 1:	B8E5-B8EC
HOLD 2:	B8ED-B8F1
HOLD 3:	B8F2-B8F6

These are primarily for the use of the basic interpreter.

The floating point and integer arithmetic can be accessed without worrying too much about the detailed working, but some comments have been added to the following tabulation to assist those who wish to investigate the routines more closely. A BASIC program given in the Appendix will help such investigations.

The Entry Points

The jumpblock entries relevant to this area use the FIRM JUMP code (&EF) rather than the LOW JUMP code (&CF) used for the rest of the jumpblock. but the same rules apply: On jumping to an entry there must be a return address left on the stack to allow continuation after the called routine has been executed.

A special notation will be used for floating point data, FP(X) meaning a floating point number pointed to by the contents of register pair X.

The actual routine addresses are not given. They can be found easily enough by examining the jumpblock entries at the addresses stated.

BD3D DE and HL on entry point to two floating point numbers (or areas where floating point numbers may exist). FP(DE) is copied to FP(HL). On exit, A holds the exponent of the copied number. Carry is set. (Note that FP(HL) must be in RAM〉)

BD4Ø On entry, DE points to an FP number area in RAM, and HL holds an unsigned binary number in the range Ø - 65535. FP(DE) is set to the floating point equivalent of the number in HL. On exit HL=DE on entry, DE is corrupt, and A holds the most significant byte of the new mantissa.

BD43 On entry, HL points to a four-byte binary number in RAM. The number, treated as an integer, is overwritten by its FP equivalent. On exit, HL points to the new number, and A hold the most significant byte of its mantissa.

BD46 This call is used by the BASIC command CINT. On entry, HL points to an FP number in RAM, the number having a value within the range +/- 32767. The integer part of the number is set up in HL as a two's complement number rounded to the nearest whole number. On exit, A holds the sign byte of the FP number. Carry is set unless overflow occurred due to the number being too large.

BD49 On entry, HL points to an FP number in RAM. BD4C is called to convert the number to integer form. If the result leaves a remainder greater than Ø.5, or if the FP number was negative, the integer is incremented. On exit, C holds the number of non-zero bytes in the integer.

BD4C This call is used by the BASIC command FIX. On entry, HL points to an FP number in RAM. The number is truncated to signed integer form, the result overwriting the mantissa of the original number. On exit, C holds the number of non-zero bytes in the integer. A holds &FF for a negative number, Ø for a positive number.

BD4F This call is used by the BASIC command INT. It is almost identical to BD49, except that sign is sensed, remainder ignored.

The foregoing calls need little explanation, but the next is a different matter. It is used in preparation for decimal output, though it does not perform the actual output process.

An interesting algorithm is used to calculate the number of decimal places in the integer part of the number being processed. The real value of the exponent is found by subtracting &8Ø, and the result is multiplied by 77/256, which is a close approximation to log1Ø 2. The integer part of the result states the number of decimal places needed.

The calculation can be written;

$$\log_{10} N = (\log_2 N)*(\log_{10} 2)$$

where N is the number concerned.

Nine is subtracted from the number of decimal places, since up to nine can be displayed. If the result is non-zero, the number is multiplied or divided by powers of ten until it lies in the range 3125ØØ to 1Ø↑9. (3125ØØ - (1Ø↑7)/32).

BD52 On entry, HL points to an FP number in RAM. The number is processed as described above, and set in place of FP(HL). HL is then adjusted to point to the most significant byte.

This is the most difficult to use of all the BASIC support routines, and an alternative approach may be preferable.

BD55 On entry, A holds an index value, and HL points to an FP number in RAM. The number is multipled by 1Ø↑A. A may range from -127 to +127, but values outside the range +/- 76 will be meaningless. The result replaces FP(HL). On exit BC and DE are corrupt, and A holds the sign byte of the result mantissa.

BD58 On entry, DE and HL point to FP numbers, the latter in RAM) The calculation FP(HL)=FP(HL)+FP(DE) is performed. On exit BC and DE are corrupt, A holds the sign byte of the resulting mantissa.

BD5B As BD58, but FP(HL)=FP(HL)-FP(DE)

BD5E As BD58, but FP(HL)=FP(DE)=FP(HL)

BD61 As BD58, but FP(HL)=FP(HL)*FP(DE)

BD64 As BD58, but FP(HL)=FP(HL)/FP(DE)

BD67 On entry, A holds an index and HL points to an FP number in RAM. A is added to the exponent of the number, effectively multiplying the number by 2↑A. A may have any value between -127 and +127. On exit. A holds the new exponent.

BD6A On entry, DE and HL point to two FP numbers.

If FP(DE) = FP(HL), the routine exits NC,Z. A=Ø
If FP(DE) < FP(HL), the routine exits NC,NZ. A=1
If FP(DE) > FP(HL), the routine exits C,NZ. A=&FF

The FP numbers are unchanged.

BD6D FP(HL) is negated.

BD7Ø If FP(HL) = Ø, the return is with A=Ø.
If FP(HL) > Ø, the return is with A=1.
If FP(HL) < Ø, the return is with A=&FF.

BD73 Entered with A=Ø, this sets the RAD (radian) condition. Entry with A=1 sets the DEG (degree) condition.

BD76 FP(HL)=PI

BD79 FP(HL) is replaced by its square root. (SQR)

BD7C FP(HL) = FP(HL) ↑ FP(DE)

BD7F FP(HL) is replaced by its natural logarithm. HL is preserved.

BD82 As BD7F, but the logarithm is to base 1Ø.

BD85 FP(HL) = e↑(FP(HL)). (EXP)

BD88 FP(HL) = SIN(FP(HL)).

BD8B FP(HL) = COS(FP(HL)).

BD8E FP(HL) = TAN(FP(HL)).

BD91 FP(HL) = ATN(FP(HL)).

The limitations and rules for the BASIC versions apply to these functions in general.

BD94 This is similar to BD43, but works on a five-byte integer. The least significant byte of the integer is discarded, since it lies outside the system resolution limit.

BD97 The random seed is set to Ø76C6589.

BD9A The random seed is set as above, then XORed with FP(HL).

BD9D The random seed is updated and copied into FP(HL).

BDAØ The random seed is set from FP(HL).

BDA3 B=H. If H $<$ Ø it is negated. C=2, A=Ø.

BDA6 BC=ØØØ2, E=Ø.

The last two calls are only significant to the BASIC interpreter.

BDA9 If H $<$ Ø it is negated, and the routine returns. Otherwise, if B$>$Ø HL is negated.

We now reach the integer calculations.

BDAC HL=HL+DE

BDAF HL=HL-DE

BDB2 HL=DE-HL

BDB5 HL=HL*DE. Absolute values are used. If the signs of HL and DE differ, B is set negative. BDA9 follows.

BDB8 HL=HL/DE, remainder in DE.

BDBB DE=HL/DE, remainder in HL.

BDBE HL=HL*DE.

BDC1 HL=HL/DE, remainder in DE. Absolute values are used.

BDC4 If HL=DE the return is with A = Ø.

```
     If HL>DE the return is with A = 1.
     If HL>DE the return is with A = &FF.

BDC7 HL is negated.

BDCA If HL=Ø, the return is with A = Ø.
     If HL>Ø, the return is with A = 1.
     If HL<Ø, the return is with A = &FF.
```

Using the Maths Calls

You should not be deceived by the array of mathematical calls. Using them to full advantage can entail a lot of surrounding code. Nor should it be assumed that the descriptions given above cover all the possibilities. Use the program given in the Appendix to explore the details. (The descriptions are based on examination of the routines, and it is only too easy to miss odd points here and there...)

Notable omissions are routines for the input and output of numeric data, which is handled by the main interpreter. The input process can be complicated by the need to cover binary, decimal and hexadecimal bases, and by the fact that alphabetic data may also be involved. However, in broad terms the process involves;

(a. Checking that the data is numeric.
(b. Converting from ASCII to binary values.
(c. Multiplying the number already input by the number base.
(d. Adding the new digit.
(e. Looping to A.

Exit from the loop is usually dependent on a non-numeric being found at stage a.

The output process is more difficult, especially if exponent forms are to be included. It is often desirable to position the number with care, and for that you need to know the number of decimal digits and hence the number of leading zeroes. The process involves division by the number base, taking the remainder as a basis for the digit to be displayed, but this produces the last digit first, and a string of codes has to be assembled in reverse order before output can begin.

Access to floating point routines opens the door to many types of machine code program that would otherwise be much more difficult to write, but that does not mean that such programs become easy to create. A good deal of thought may be needed to get everything right, but the results can be very satisfying.

Chapter 14 THE BASIC INTERPRETER

If there are weaknesses in the CPC464 system, they lie mainly in the BASIC interpreter. Though it shares a ROM with the operating system, it conveys a somewhat different image. There are a few undeniable bugs, which fortunately have a minimal effect on overall performance, and it carries subroutine nesting to extreme lengths, which makes explanations difficult.

Most of the interpreter is concerned with the execution of the procedures relating to keywords, and the list of keywords, tokens and entry addresses which has been provided hereafter will simplify exploration, but these apply only to version 1.Ø. Extraction of the corresponding information for other versions is not easy, because the data is widely scattered.

First, there is the reserved word table, which is organised in an ingenious but confusing way. There are, in fact, a series of short tables, one for each letter of the alphabet. If, say, the keyword PRINT is to be found, the routine first branches to examination of the list for 'P', and then looks for RINT. The last character of this word is marked by the addition of bit 7, and the following byte gives the token for PRINT: &BF. The entry is, in fact, the first in the 'P' list, so it is found very quickly, but there are only nine words in the list, so even the last is found without much delay. Systems which use a single list would take much longer to find an entry.

Having found the tokens, you will need the entry addresses, and here a certain amount of patience is needed, because the relevant tables are scattered around somewhat, and they take different forms. For tokens &F4-&FD there is a form of jumpblock, recognisable by repeated &C3 entries. That falls at CF81 in version 1.Ø. Other links are established by special tables relating tokens to addresses, and other tables just give the link addresses, being accessed on a displacement pointer basis. I fear that the only answer is to search for areas of non-code entries, and then do a little detective work to find out what they mean.

On the whole the entry points within the interpreter are of limited interest to the machine code programmer, especially as he has direct access to the mathematical routines. A notable exception is the CALL function, which can pass parameters to a machine code program.

Each parameter is passed as a two-byte number, which may express an integer, an integer derived from a real (FP) number, or the address of a real number. Register A is set to the number of parameters passed, and IX points to the last parameter. so if there are N parameters then parameter X is stored at (N - X)*2 relative to the address in IX.

Despite the comments made at the start of this section, there is much of interest to be found by anyone who explores the interpreter, but to examine it all here would be too space-consuming. You have the tools needed for exploration, so why not use them?

Reserved Words in Token Order

Token	Word	Addr	Token	Word	Addr
&ØØ	ABS	FD85	&12	PEEK	F158
&Ø1	ASC	FA1Ø	&13	REMAIN	C99F
&Ø2	ATN	D53E	&14	SGN	FFØ2
&Ø3	CHR$	FA16	&15	SIN	D52F
&Ø4	CINT	FE8D	&16	SPACE$	FA57
&Ø5	COS	D534	&17	SP	D329
&Ø6	CREAL	FEEC	&18	SQR	D4EF
&Ø7	EXP	D52Ø	&19	STR$	F91E
&Ø8	FIX	FDE8	&1A	TAN	D539
&Ø9	FRE	FC2D	&1B	UNT	FEC2
&ØA	INKEY	D4Ø9	&1C	UPPER$	F842
&ØB	INP	F16D	&1D	VAL	FA77
&ØC	INT	FDED	&1E	-	FFØ6
&ØD	JOY	D423	$4Ø	EOF	C417
&ØE	LEN	FAØA	&41	ERR	DØDC
&ØF	LOG	D52A	&42	HIMEM	DØF4
&1Ø	LOG1Ø	D525	&43	INKEY$	FA24
&11	LOWER$	F834	&44	PI	D4ØB

Token	Word	Addr
&45	RND	D584
&46	TIME	DØE5
&47	XPOS	D1Ø7
&48	YPOS	D1ØE
&71	BIN$	F8BA
&72	DEC$	F8EA
&73	HEX$	F8C4
&74	INSTR	FAA1
&75	LEFT$	F93C
&76	MAX	D1EE
&77	MIN	DIEA
&78	POS	C276
&79	RIGHT$	F943
&7A	ROUND	D219
&7B	STRING$	FA36
&7C	TEST	C4E9
&7D	TESTR	C4EE
&7E	–	CEAB
&7F	VPOS	C262
&8Ø	AFTER	C971
&81	AUTO	CØDF
&82	BORDER	C221
&83	CALL	F1BA
&84	CAT	D246
&85	CHAIN	EA3C
&86	CLEAR	C132
&87	CLG	C485
&88	CLOSEIN	D298
&89	CLOSEOUT	D2A1
&8A	CLS	C25A
&8B	CONT	CBCØ
&8C	DATA	E8EF
&8D	DEF	D117
&8E	DEFINT	D618
&8F	DEFREAL	D61C
&9Ø	DEFSTR	D614
&91	DEG	D4E7
&92	DELETE	E728
&93	DIM	D67D
&94	DRAW	C4C6
&95	DRAWR	C4CB
&96	EDIT	CØ52
&97	ELSE	E8F3
&98	END	CB65
&99	ENT	D385
&9A	ENV	D84E
&9B	ERASE	D9ØØ
&9C	ERROR	CA8F
&9D	EVERY	C979
&9E	FOR	C529
&9F	GOSUB	C6ED
&AØ	GOTO	C6E8
&A1	IF	C6C7
&A2	INC	C22A
&A3	INPUT	DB2B
&A4	KEY	D439
&A5	LET	D654
&A6	LINE	DAF8
&A7	LIST	EØF7
&A8	LOAD	E9F6
&A9	LOCATE	C2D2
&AA	MEMORY	F4EF
&AB	MERGE	EAA6
&AC	MID$	F993
&AD	MODE	C24F
&AE	MOVE	C5Ø5
&AF	MOVER	C5ØA
&BØ	NEXT	C5FB
&B1	NEW	C12B
&B2	ON	C7E3
&B3	ON BREAK	C8CB
&B4	ON ERROR GOTO	CBF8
&B5	ON SQ	C94Ø
&B6	OPENIN	D25F
&B7	OPENOUT	D256
&B8	ORIGIN	C48C
&B9	OUT	F177
&BA	PAPER	C2ØA
&BB	PEN	C212
&BC	PLOT	C4DØ
&BD	PLOTR	C4D5
&BE	POKE	F15F
&BF	PRINT	F1FD
&CØ	–	E8F3
&C1	RAD	D4EB
&C2	RANDOMISE	D55E
&C3	READ	DCEB
&C4	RELEASE	D31E
&C5	REM	E8F3
&C6	RENUM	E7DF
&C7	RESTORE	DCD9

Token	Word	Addr	Token	Word	Addr
&C8	RESUME	CC∅3	&E4	FN	-
&C9	RETURN	C7∅F	&E5	SPC	-
&CA	RUN	E9BD	&E6	STEP	-
&CB	SAVE	EC∅9	&E7	SWAP	-
&CC	SOUND	D2C∅	&EA	TAB	-
&CD	SPEED	D494	&EB	THEN	-
&CE	STOP	CB5A	&EC	TO	-
&CF	SYMBOL	F69D	&ED	USING	-
&D∅	TAG	C319	&EF	=	-
&D1	TAGOFF	C32∅	&F1	<	-
&D2	TROFF	DDE6	&F4	+	FCCC
&D3	TRON	DDE2	&F5	MINUS	FCE1
&D4	WAIT	F17D	&F6	*	FCF5
&D5	WEND	C776	&F7	/	FD12
&D6	WHILE	C747	&F8	-	D4F4
&D7	WIDTH	C3E3	&F9	DIV	FD37
&D8	WINDOW	C2E1	&FA	AND	FD58
&D9	WRITE	F47B	&FB	MOD	FD49
&DA	ZONE	F1F6	&FC	OR	FD63
&DB	DI	C8E1	&FD	XOR	FD6D
&DC	EI	C8E7	&FE	NOT	-
&E3	ERL	-			

Not all the tokens are associated with addresses. STEP is recognised by the FOR routine, while SPC and USING are picked up by PRINT.

Similarly, some tokens have special meanings that are not associated with words. &FF warns that the next byte is a function token, for example. Some bytes which look like tokens mean something quite different. &∅2 introduces an integer variable, &∅D introduces a real variable and &1D introduces an address. &1E introduces a two-byte integer constant, and &1F a real constant.

It is interesting and instructive to examine the stored program, which will be found from ∅17∅ upwards, and compare the codes with a listing. This will tell you more than could be conveyed in a hundred pages of explanation.

Appendix 1
SUPPORT PROGRAMS

Two programs given here will assist you in exploring the operating system and BASIC interpreter. The first is a disassembler which will operate on ROM-borne code. It will also dump in hexadecimal or alphabetic form. The display can be sent to screen or printer.

It should be noted that a blank line is inserted at the end of a code block, which helps to identify data insertions. Setting the program up may be tedious, but the results make that well worth while.

```
100    MEMORY &A4FF
110    GOSUB 4560
120    PF=0 : PC=0
130    CLS
140    PRINT TAB(10);"1. Display Mode."
150    PRINT TAB(10);"2. Print Mode."
160    PRINT TAB(10);"3. Disassemble."
170    PRINT TAB(10);"4. Hex dump
180    PRINT TAB(10);"5. Text dump."
190    PRINT TAB(15);
200    INPUT "Select Option.";K
210    IF K<1 OR k>5 THEN 130
220    ON K GOTO 230,240,250,260,270
230    PF=0 : GOTO 130
240    PF=8 : GOTO 130
250    DF=0 : GOTO 280
260    DF=1 : GOTO 280
270    DF=2
280    CLS
290    INPUT "Start Address ",C
300    INPUT "End Address ",E
310    C=C-65536*(C<0)
320    E=E-65536*(E<0)
330    IF E<C THEN 130
340    C=C-1
350    IF DF=0 THEN 450
360    PRINT #PF, : GOSUB 4620 : PRINT #PF,HEX$(C+1,4)

370    GOSUB 4490
380    IF B<&20 OR B>&7F THEN BB=&3F ELSE BB=B
```

```
390   P=INT(C/(8*DF)) : Q=C-(8*DF*P)
400   IF Q=0 THEN PRINT#PF : GOSUB 4620 : PRINT#PF,HE
      X$(C,4);
410   IF DF=1 THEN X$=HEX$(B,2) ELSE X$=CHR$(BB)
420   PRINT #PF,TAB(6+(4-DF)*Q);X$;
430   IF C>=E THEN : PRINT#PF : PRINT#PF : PC=PC+1 :
      GOSUB 4620 : INPUT C : GOTO 130
440   GOTO 370
450   GOSUB 4490
460   NA=B
470   OS$=HEX$(C,4)+"  "+HEX$(B,2)+" "
480   AS$=""
490   MA=NA\64 : MD=NA MOD 64
500   ON (MA+1) GOTO 510,1300,1370,1480
510   MA=NA\8 : MB=NA MOD 8
520   ON (MA+1) GOTO 530,620,710,800,890,980,1070,116
      0
530   ON (MB+1) GOTO 540,550,560,570,580,590,600,610
540   AS$="NOP" : GOTO 1250
550   AS$="LD   BC," : GOSUB 4210 : GOTO 1250
560   AS$="LD   (BC),A" : GOTO 1250
570   AS$="INC  BC" : GOTO 1250
580   AS$="INC  B" : GOTO 1250
590   AS$="DEC  B" : GOTO 1250
600   AS$="LD   B," : GOSUB 4160 : GOTO 1250
610   AS$="RLC  A" : GOTO 1250
620   ON (MB+1) GOTO 630,640,650,660,670,680,690,700
630   AS$="EX   AF/AF'" : GOTO 1250
640   AS$="ADD  HL,BC" : GOTO 1250
650   AS$="LD   A,(BC)" : GOTO 1250
660   AS$="DEC  BC" : GOTO 1250
670   AS$="INC  C" : GOTO 1250
680   AS$="DEC  C" : GOTO 1250
690   AS$="LD   C," : GOSUB 4160 : GOTO 1250
700   AS$="RRC  A" : GOTO 1250
710   ON (MB+1) GOTO 720,730,740,750,760,770,780,790
720   AS$="DJNZ " : GOSUB 4280 : GOTO 1250
730   AS$="LD   DE," : GOSUB 4210 : GOTO 1250
740   AS$="LD   (DE),A" : GOTO 1250
750   AS$="INC  DE" : GOTO 1250
760   AS$="INC  D" : GOTO 1250
770   AS$="DEC  D" : GOTO 1250
780   AS$="LD   D," : GOSUB 4160 : GOTO 1250
790   AS$="RL   A" : GOTO 1250
800   ON (MB+1) GOTO 810,820,830,840,850,860,870,880
810   AS$="JR   " : GOSUB 4280 : FF=1 : GOTO 1250
820   AS$="ADD  HL,DE" : GOTO 1250
830   AS$="LD   A,(DE)" : GOTO 1250
840   AS$="DEC  DE" : GOTO 1250
850   AS$="INC  E" : GOTO 1250
860   AS$="DEC  E" : GOTO 1250
870   AS$="LD   E," : GOSUB 4160 : GOTO 1250
880   AS$="RR   A" : GOTO 1250
890   ON (MB+1) GOTO 900,910,920,930,940,950,960,970
900   AS$="JR   NZ," : GOSUB 4280 : GOTO 1250
```

```
910   AS$="LD    HL," : GOSUB 4210 : GOTO 1250
920   AS$="LD    (" : GOSUB 4210 : AS$=AS$+"),HL" : GO
      TO 1250
930   AS$="INC   HL" : GOTO 1250
940   AS$="INC   H" : GOTO 1250
950   AS$="DEC   H" : GOTO 1250
960   AS$="LD    H," : GOSUB 4160 : GOTO 1250
970   AS$="DAA   " : GOTO 1250
980   ON (MB+1) GOTO 990,1000,1010,1020,1030,1040,105
      0,1060
990   AS$="JR    Z," : GOSUB 4280 : GOTO 1250
1000  AS$="ADD   HL,HL" : GOTO 1250
1010  AS$="LD    HL,(" : GOSUB 4210 : AS$=AS$+")" : GO
      TO 1250
1020  AS$="DEC   HL" : GOTO 1250
1030  AS$="INC   L" : GOTO 1250
1040  AS$="DEC   L" : GOTO 1250
1050  AS$="LD    L," : GOSUB 4160 : GOTO 1250
1060  AS$="CPL   " : GOTO 1250
1070  ON (MB+1) GOTO 1080,1090,1100,1110,1120,1130,11
      40,1150
1080  AS$="JR    NC," : GOSUB 4280 : GOTO 1250
1090  AS$="LD    SP," : GOSUB 4210 : GOTO 1250
1100  AS$="LD    (" : GOSUB 4210 : AS$=AS$+"),A" : GOT
      O 1250
1110  AS$="INC   SP" : GOTO 1250
1120  AS$="INC   (HL)" : GOTO 1250
1130  AS$="DEC   (HL)" : GOTO 1250
1140  AS$="LD    (HL)," : GOSUB 4160 : GOTO 1250
1150  AS$="SCF   " : GOTO 1250
1160  ON (MB+1) GOTO 1170,1180,1190,1200,1210,1220,12
      30,1240
1170  AS$="JR    C," : GOSUB 4280 : GOTO 1250
1180  AS$="ADD   HL,SP" : GOTO 1250
1190  AS$="LD    A,(" : GOSUB 4210 : AS$=AS$+")" : GOT
      O 1250
1200  AS$="DEC   SP" : GOTO 1250
1210  AS$="INC   A" : GOTO 1250
1220  AS$="DEC   A" : GOTO 1250
1230  AS$="LD    A," : GOSUB 4160 : GOTO 1250
1240  AS$="CCF   " : GOTO 1250
1250  PRINT #PF,OS$;TAB(19):AS$
1260  IF FF<>0 THEN PRINT #PF : FF=0 : GOSUB 4620
1270  GOSUB 4620
1280  IF C>=E THEN INPUT F : GOTO 130
1290  GOTO 450
1300  IF NA<> 118 THEN 1320
1310  AS$="HALT" : GOTO 1360
1320  AS$="LD    "
1330  MA= MD\8 : MB=MD MOD 8
1340  ME=MA
1350  GOSUB 4400 : AS$=AS$+"," : ME=MB : GOSUB 4400 :
       GOTO 1360
1360  GOTO 1250
1370  MA=MD\8 : MB=MD MOD 8
```

```
1380  ON (MA+1) GOTO 1390,1400,1410,1420,1430,1440,14
      50,1460
1390  AS$="ADD  A," : GOTO 1470
1400  AS$="ADC  " : GOTO 1470
1410  AS$="SUB  " : GOTO 1470
1420  AS$="SBC  " : GOTO 1470
1430  AS$="AND  " : GOTO 1470
1440  AS$="XOR  " : GOTO 1470
1450  AS$="OR   " : GOTO 1470
1460  AS$="CP   " : GOTO 1470
1470  ME=MB : GOSUB 4400 : GOTO 1250
1480  MA=MD\8 : MB=MD MOD 8
1490  ON (MA+1) GOTO 1500,1590,1680,1770,1860,1950,20
      40,2130
1500  ON (MB+1) GOTO 1510,1520,1530,1540,1550,1560,15
      70,1580
1510  AS$="RET  NZ" : GOTO 2220
1520  AS$="POP  BC" : GOTO 2220
1530  AS$="JP   NZ," : GOSUB 4210 : GOTO 2220
1540  AS$="JP   " : GOSUB 4210 : FF=1 : GOTO 2220
1550  AS$="CALL NZ," : GOSUB 4210 : GOTO 2220
1560  AS$="PUSH BC" : GOTO 2220
1570  AS$="ADD  A," : GOSUB 4160 : GOTO 2220
1580  AS$="RESET " : FF=1 : GOTO 2220
1590  ON (MB+1) GOTO 1600,1610,1620,1630,1640,1650,16
      60,1670
1600  AS$="RET  Z" : GOTO 2220
1610  AS$="RET  " : FF=1 : GOTO 2220
1620  AS$="JP   Z," : GOSUB 4210 : GOTO 2220
1630  GOTO 2230
1640  AS$="CALL Z," : GOSUB 4210 : GOTO 2220
1650  AS$="CALL " : GOSUB 4210 : GOTO 2220
1660  AS$="ADC  " : GOSUB 4160 : GOTO 2220
1670  AS$="LOW JUMP " : FF=1 : GOSUB 4210 : GOTO 2220

1680  ON (MB+1) GOTO 1690,1700,1710,1720,1730,1740,17
      50,1760
1690  AS$="RET  NC" : GOTO 2220
1700  AS$="POP  DE" : GOTO 2220
1710  AS$="JP   NC," : GOSUB 4210 : GOTO 2220
1720  AS$="OUT  (" : GOSUB 4160 : AS$=AS$+",A" : GOTO
       2220
1730  AS$="CALL NC," : GOSUB 4210 : GOTO 2220
1740  AS$="PUSH DE" : GOTO 2220
1750  AS$="SUB  " : GOSUB 4160 : GOTO 2220
1760  AS$="SIDE CALL " : GOSUB4210 : GOTO 2220
1770  ON (MB+1) GOTO 1780,1790,1800,1810,1820,1830,18
      40,1850
1780  AS$="RET  C" : GOTO 2220
1790  AS$="EXX  " : GOTO 2220
1800  AS$="JP   C," : GOSUB 4210 : GOTO 2220
1810  AS$="IN   A,(" : GOSUB 4160 : AS$=AS$+")" : GOT
      O 2220
1820  AS$="CALL C," : GOSUB 4210 : GOTO 2220
1830  AX$="IX" : GOTO 2420
```

```
1840  AS$="SBC  " : GOSUB 4160 : GOTO 2220
1850  AS$="FAR CALL " : GOSUB 4210 : GOTO 2220
1860  ON (MB+1) GOTO 1870,1880,1890,1900,1910,1920,19
      30,1940
1870  AS$="RET  PO" : GOTO 2220
1880  AS$="POP  HL" : GOTO 2220
1890  AS$="JP   PO," : GOSUB 4210 : GOTO 2220
1900  AS$="EX   (SP)/HL" : GOTO 2220
1910  AS$="CALL PO," : GOSUB 4210 : GOTO 2220
1920  AS$="PUSH HL" : GOTO 2220
1930  AS$="AND  " : GOSUB 4160 : GOTO 2220
1940  AS$="RAM LAM " : GOTO 2220
1950  ON (MB+1) GOTO 1960,1970,1980,1990,2000,2010,20
      20,2030
1960  AS$="RET  PE" : GOTO 2220
1970  AS$="JP   (HL)" : FF=1 : GOTO 2220
1980  AS$="JP   PE" : GOTO 2220
1990  AS$="EX   DE,HL" : GOTO 2220
2000  AS$="CALL PE," : GOSUB 4210 : GOTO 2220
2010  GOTO 3260
2020  AS$="XOR  " : GOSUB 4160 : GOTO 2220
2030  AS$="FIRM JUMP " : FF=1 : GOSUB 4210 : GOTO 222
      0
2040  ON (MB+1) GOTO 2050,2060,2070,2080,2090,2100,21
      10,2120
2050  AS$="RET  P" : GOTO 2220
2060  AS$="POP  AF" : GOTO 2220
2070  AS$="JP   P," : GOSUB 4210 : GOTO 2220
2080  AS$="DI   " : GOTO 2220
2090  AS$="CALL P," : GOSUB 4210 : GOTO 2220
2100  AS$="PUSH AF" : GOTO 2220
2110  AS$="OR   " : GOSUB 4160 : GOTO 2220
2120  AS$="USER RESTART " : FF=1 : GOTO 2220
2130  ON (MB+1) GOTO 2140,2150,2160,2170,2180,2190,22
      00,2210
2140  AS$="RET  M" : GOTO 2220
2150  AS$="LD   SP,HL" : GOTO 2220
2160  AS$="JP   M," : GOSUB 4210 : GOTO 2220
2170  AS$="EI   " : GOTO 2220
2180  AS$="CALL M," : GOSUB 4210 : GOTO 2220
2190  AX$="IY" : GOTO 2420
2200  AS$="CP   " : GOSUB 4160 : GOTO 2220
2210  AS$="INTERRUPT " : GOTO 2220
2220  GOTO 1250
2230  GOSUB 4490 : RE=B : OS$=OS$+HEX$(B,2)+" "
2240  MA=RE\64 : MD=RE MOD 64
2250  ON (MA+1) GOTO 2260,2360,2380,2400
2260  MC=MD\8 : MB=MD MOD 8
2270  ON (MC+1) GOTO 2280,2290,2300,2310,2320,2330,23
      50,2350
2280  AS$="RLC  " : ME=MB : GOSUB 4400 : GOTO 2220
2290  AS$="RRC  " : ME=MB : GOSUB 4400 : GOTO 2220
2300  AS$="RL   " : ME=MB : GOSUB 4400 : GOTO 2220
2310  AS$="RR   " : ME=MB : GOSUB 4400 : GOTO 2220
2320  AS$="SLA  " : ME=MB : GOSUB 4400 : GOTO 2220
```

```
2330  AS$="SRA   " : ME=MB : GOSUB 4400 : GOTO 2220
2340  GOTO 4380
2350  AS$="SRL   " : ME=MB : GOSUB 4400 : GOTO 2220
2360  MC=MD\8 : MB=MD MOD 8 : ME=MB
2370  AS$="BIT "+STR$(MC)+"," : GOSUB 4400 : GOTO 222
      0
2380  MC=MD\8 : MB=MD MOD 8 : ME=MB
2390  AS$="RES "+STR$(MC)+"," : GOSUB 4400 : GOTO 222
      0
2400  MC=MD\8 : MB=MD MOD 8 : ME=MB
2410  AS$="SET "+STR$(MC)+"," : GOSUB 4400 : GOTO 222
      0
2420  GOSUB 4490 : RE=B : OS$=OS$+HEX$(RE,2)+" "
2430  MA= RE\64 : MD=RE MOD 64
2440  ON (MA+1) GOTO 2450,2700,2880,3060
2450  MC=MD\8 : MB=MD MOD 8
2460  ON (MC+1) GOTO 2470,2480,2500,2510,2530,2580,26
      30,2680
2470  GOTO 4380
2480  IF MB<>1 THEN 4380
2490  AS$="ADD  "+AX$+",BC" : GOTO 2220
2500  GOTO 4380
2510  IF MB<>1 THEN 4380
2520  AS$="ADD  "+AX$+",DE" : GOTO 2220
2530  IF MB>3 THEN 4380
2540  ON (MB+1) GOTO 4380,2550,2560,2570
2550  AS$="LD   "+AX$+"," : GOSUB 4210 : GOTO 2220
2560  AS$="LD   (" : GOSUB 4210 : AS$=AS$+"),"+AX$ :
      GOTO 2220
2570  AS$="INC  "+AX$ : GOTO 2220
2580  IF MB=0 OR MB>3 THEN 4380
2590  ON MB GOTO 2600,2610,2620
2600  AS$="ADD  "+AX$+","+AX$ : GOTO 2220
2610  AS$="LD   "+AX$+",(" : GOSUB 4210 : AS$=AS$+")"
       : GOTO 2220
2620  AS$="DEC  "+AX$ : GOTO 2220
2630  IF MB<4 OR MB=7 THEN 4380
2640  ON (MB-3) GOTO 2650,2660,2670
2650  AS$="INC  " : GOSUB 4390 : GOTO 2220
2660  AS$="DEC  " : GOSUB 4390 : GOTO 2220
2670  AS$="LD   " : GOSUB 4390 : AS$=AS$+"," : GOSUB
      4160 : GOTO 2220
2680  IF MB<>1 THEN 4380
2690  AS$="ADD  "+AX$+",SP" : GOTO 2220
2700  MC=MD\8 : MB=MD MOD 8
2710  ON (MC+1) GOTO 2720,2740,2760,2780,2800,2820,28
      40,2860
2720  IF MB<>6 THEN 4380
2730  AS$="LD   B," : GOSUB 4390 : GOTO 2220
2740  IF MB<>6 THEN 4380
2750  AS$="LD   C," : GOSUB 4390 : GOTO 2220
2760  IF MB<>6 THEN 4380
2770  AS$="LD   D," : GOSUB 4390 : GOTO 2220
2780  IF MB<>6 THEN 4380
2790  AS$="LD   E," : GOSUB 4390 : GOTO 2220
```

```
2800  IF MB<>6 THEN 4380
2810  AS$="LD   H," : GOSUB 4390 : GOTO 2220
2820  IF MB<>6 THEN 4380
2830  AS$="LD   L," : GOSUB 4390 : GOTO 2220
2840  IF MB=6  THEN 4380
2850  AS$="LD   " : GOSUB 4390 : AS$=AS$+"," : ME=MB
      : GOSUB 4400 : GOTO 2220
2860  IF MB<>6 THEN 4380
2870  AS$="LD   A," : GOSUB 4390 : GOTO 2220 2840 IF
      MB=6  THEN 4380
2880  MC=MD\8 : MB=MD MOD 8
2890  ON (MC+1) GOTO 2900,2920,2940,2960,2980,3000,30
      20,3040
2900  IF MB<>6 THEN 4380
2910  AS$="ADD  A," : GOSUB 4390 : GOTO 2220
2920  IF MB<>6 THEN 4380
2930  AS$="ADC  A," : GOSUB 4390 : GOTO 2220
2940  IF MB<>6 THEN 4380
2950  AS$="SUB  A," : GOSUB 4390 : GOTO 2220
2960  IF MB<>6 THEN 4380
2970  AS$="SBC  A," : GOSUB 4390 : GOTO 2220
2980  IF MB<>6 THEN 4380
2990  AS$="AND  A," : GOSUB 4390 : GOTO 2220
3000  IF MB<>6 THEN 4380
3010  AS$="XOR  A," : GOSUB 4390 : GOTO 2220
3020  IF MB<>6 THEN 4380
3030  AS$="0R   A," : GOSUB 4390 : GOTO 2220
3040  IF MB<>6 THEN 4380
3050  AS$="CP   A," : GOSUB 4390 : GOTO 2220
3060  MC=MD\8 : MB=MD MOD 8
3070  ON (MC+1) GOTO 4380,3080,4380,4380,4040,4080,43
      80,4120
3080  IF MB<>3 THEN 4380
3090  GOSUB 4490 : NC=B : OS$=OS$+HEX$(B,2)+"  "
3100  GOSUB 4490 : RH=B : OS$=OS$+HEX$(B,2)+"  "
3110  IF (RH-6) MOD 8 <> 0 THEN 4380
3120  MC=RH\64 : MB=RH \ 8
3130  ON (MC+1) GOTO 3140,3220,3230,3240
3140  ON (MB+1) GOTO 3150,3160,3170,3180,3190,3200,43
      80,3210
3150  AS$="RLC  " : GOSUB 3250 : GOTO 1250
3160  AS$="RRC  " : GOSUB 3250 : GOTO 1250
3170  AS$="RL   " : GOSUB 3250 : GOTO 1250
3180  AS$="RR   " : GOSUB 3250 : GOTO 1250
3190  AS$="SLA  " : GOSUB 3250 : GOTO 1250
3200  AS$="SRA  " : GOSUB 3250 : GOTO 1250
3210  AS$="SRL  " : GOSUB 3250 : GOTO 1250
3220  AS$="BIT  " : GOSUB 3255 : GOTO 1250
3230  AS$="RES  " : GOSUB 3255 : GOTO 1250
3240  AS$="SET  " : GOSUB 3255 : GOTO 1250
3250  AS$=AS$+"("+AX$+"+"+HEX$(NC)+")" : RETURN
3255  AS$=AS$+HEX$((RH AND &38)/8)+",("+AX$+"+"+HEX$(
      NC)+")" : RETURN
3260  GOSUB 4490 : RE=B : OS$=OS$+HEX$(B,2)+"  "
3270  MA=RE\64 : MB=RE MOD 64
```

```
3280  ON (MA+1) GOTO 4380,3290,3810,4380
3290  MD=MB\8 : MB=MB MOD 8
3300  ON (MD+1) GOTO 3310,3400,3470,3540,3610,3670,37
      30,3760
3310  ON (MB+1) GOTO 3320,3330,3340,3350,3360,3380,33
      90,3390
3320  AS$="IN   B,(C)" : GOTO 1250
3330  AS$="OUT  (C),B" : GOTO 1250
3340  AS$="SBC  HL,BC" : GOTO 1250
3350  AS$="LD   (" : GOSUB 4210 : AS$=AS$+"),BC" : GO
      TO 1250
3360  AS$="NEG  " : GOTO 1250
3370  AS$="RET  N" : GOTO 1250
3380  AS$="IM0  " : GOTO 1250
3390  AS$="LD   I,A" : GOTO 1250
3400  ON (MB+1) GOTO 3410,3420,3430,3440,4380,3450,43
      80,3460
3410  AS$="IN   C,(C)" : GOTO 1250
3420  AS$="OUT  (C),C" : GOTO 1250
3430  AS$="ADC  HL,BC" : GOTO 1250
3440  AS$="LD   BC,(" : GOSUB 4210 : AS$=AS$+")" : GO
      TO 1250
3450  AS$="RET  I" : GOTO 1250
3460  AS$="LD   R,A" : GOTO 1250
3470  ON (MB+1) GOTO 3480,3490,3500,3510,4380,4380,35
      20,3530
3480  AS$="IN   D,(C)" : GOTO 1250
3490  AS$="OUT  (C),D" : GOTO 1250
3500  AS$="SBC  HL,DE" : GOTO 1250
3510  AS$="LD   (" : GOSUB 4210 : AS$=AS$+"),DE" : GO
      TO 1250
3520  AS$="IM1  " : GOTO 1250
3530  AS$="LD   A,I" : GOTO 1250
3540  ON (MB+1) GOTO 3550,3560,3570,3580,4380,4380,35
      90,3600
3550  AS$="IN   E,(C)" : GOTO 1250
3560  AS$="OUT  (C),E" : GOTO 1250
3570  AS$="ADC  HL,DE" : GOTO 1250
3580  AS$="LD   DE,(" : GOSUB 4210 : AS$=AS$+")" : GO
      TO 1250
3590  AS$="IM2  " : GOTO 1250
3600  AS$="LD   A,R" : GOTO 1250
3610  ON (MB+1) GOTO 3620,3630,3640,3650,4380,4380,43
      80,3660
3620  AS$="IN   H,(C)" : GOTO 1250
3630  AS$="OUT  (C),H" : GOTO 1250
3640  AS$="SBC  HL,HL" : GOTO 1250
3650  AS$="LD   (" : GOSUB 4210 : AS$=AS$+"),HL" : GO
      TO 1250
3660  AS$="RRD  " : GOTO 1250
3670  ON (MB+1) GOTO 3680,3690,3700,3710,4380,4380,43
      80,3720
3680  AS$="IN   L,(C)" : GOTO 1250
3690  AS$="OUT  (C),L" : GOTO 1250
3700  AS$="ADC  HL,HL" : GOTO 1250
```

```
3710  AS$="LD   HL,(" : GOSUB 4210 : AS$=AS$+")" : GO
      TO 1250
3720  AS$="RLD  " : GOTO 1250
3730  ON (MB+1) GOTO 4380,4380,3740,3750,4380,4380,43
      80,4380
3740  AS$="SBC  HL,SP" : GOTO 1250
3750  AS$="LD   (" : GOSUB 4210 : AS$=AS$+"),SP" : GO
      TO 1250
3760  ON (MB+1) GOTO 3770,3780,3790,3800,4380,4380,43
      80,4380
3770  AS$="IN   A,(C)" : GOTO 1250
3780  AS$="OUT  (C),A" : GOTO 1250
3790  AS$="ADC  HL,SP" : GOTO 1250
3800  AS$="LD   SP,(" : GOSUB 4210 : AS$=AS$+")" : GO
      TO 1250
3810  MD= MB\8 : MB=MB MOD 8
3820  IF MD<4 OR MB>3 THEN 4380
3830  ON (MD-3) GOTO 3840,3890,3940,3990
3840  ON (MB+1) GOTO 3850,3860,3870,3880
3850  AS$="LDI  " : GOTO 1250
3860  AS$="CPI  " : GOTO 1250
3870  AS$="INI  " : GOTO 1250
3880  AS$="OUTI " : GOTO 1250
3890  ON (MB+1) GOTO 3900,3910,3920,3930
3900  AS$="LDD  " : GOTO 1250
3910  AS$="CPD  " : GOTO 1250
3920  AS$="IND  " : GOTO 1250
3930  AS$="OUTD " : GOTO 1250
3940  ON (MB+1) GOTO 3950,3960,3970,3980
3950  AS$="LDIR " : GOTO 1250
3960  AS$="CPIR " : GOTO 1250
3970  AS$="INIR " : GOTO 1250
3980  AS$="OTIR " : GOTO 1250
3990  ON (MB+1) GOTO 4000,4010,4020,4030
4000  AS$="LDDR " : GOTO 1250
4010  AS$="CPDR " : GOTO 1250
4020  AS$="INDR " : GOTO 1250
4030  AS$="OTDR " : GOTO 1250
4040  ON (MB+1) GOTO 4380,4050,4380,4060,4380,4070,43
      80,4380
4050  AS$="POP  "+AX$ : GOTO 1250
4060  AS$="EX   (SP),"+AX$ : GOTO 1250
4070  AS$="PUSH "+AX$ : GOTO 1250
4080  IF MB>1 THEN 4380
4090  ON (MB+1) GOTO 4100,4110
4100  AS$="ADC  A," : GOSUB 4390 : GOTO 1250
4110  AS$="JP   ("+AX$+")" : GOTO 1250
4120  IF MB<>1 THEN 4380
4130  AS$="LD   SP,"+AX$ : GOTO 1250
4140  IF MB<>1 THEN 4380
4150  AS$="LD   SP,"+AX$ : GOTO 1250
4160  AS$=AS$+"#" : GOSUB 4490
4170  NC=B
4180  AS$=AS$+HEX$(NC,2)
4190  OS$=OS$+HEX$(NC,2)+" "
```

```
4200  RETURN
4210  GOSUB 4490
4220  NC=B
4230  GOSUB 4490
4240  NE=B
4250  AS$=AS$+HEX$(NE,2)+HEX$(NC,2)
4260  OS$=OS$+HEX$(NC,2)+" "+HEX$(NE,2)+" "
4270  RETURN
4280  GOSUB 4490
4290  ND=B : OS$=OS$+HEX$(ND,2)+" "
4300  IF ND>127 THEN 4340
4310  ND=ND+C+1
4320  AS$=AS$+HEX$(ND,4)
4330  RETURN
4340  ND=ND+C-255
4350  GOTO 4320
4360  RETURN
4370  RETURN
4380  AS$="Invalid Code" : GOTO 2220
4390  AS$=AS$+"("+AX$+"+" : GOSUB 4160 : AS$=AS$+")"
      : RETURN
4400  ON (ME+1) GOTO 4410,4420,4430,4440,4450,4460,44
      70,4480
4410  AS$=AS$+"B" : RETURN
4420  AS$=AS$+"C" : RETURN
4430  AS$=AS$+"D" : RETURN
4440  AS$=AS$+"E" : RETURN
4450  AS$=AS$+"H" : RETURN
4460  AS$=AS$+"L" : RETURN
4470  AS$=AS$+"(HL)" : RETURN
4480  AS$=AS$+"A" : RETURN
4490  C=C+1
4500  Q=INT(C/256)
4510  POKE &A614,Q
4520  POKE &A613,(C-256*Q)
4530  CALL &A600
4540  B= PEEK(&A615)
4550  RETURN
4560  FOR X=&A600 TO &A612
4570  READ Y : POKE X,Y : NEXT
4580  RETURN
4590  DATA &2A,&13,&A6,&CD,&00,&B9,&F5
4600  DATA &CD,&06,&B9,&7E,&32,&15,&A6
4610  DATA &F1,&CD,&0C,&B9,&C9
4620  IF PF<>8 THEN RETURN
4630  PC=PC+1
4640  IF PC<60 THEN RETURN
4650  FOR Y=1 TO 6
4660  PRINT #PF
4670  NEXT
4680  PC=PC-60
4690  RETURN
```

The second program allows you to call particular routines with known register contents. The register contents on exit are reported, and any floating point numbers pointed to be DE and HL are evaluated. The program should be used with care, since some calls can upset the applecart somewhat.

```
100    GOSUB 500
110    CLS
120    INPUT "Set BC ",BC
130    BC=BC-65536*(BC<0)
140    B=INT(BC/256) : C=BC-B*256
150    POKE &4001,C : POKE &4002,B
160    INPUT "Set DE ",DE
170    DE=DE-65536*(DE<0)
180    D=INT(DE/256) : E=DE-D*256
190    POKE &4004,E : POKE &4005,D
200    INPUT "Set HL ",HL
210    HL=HL-65536*(HL<0)
220    H=INT(HL/256) : L=HL-H*256
230    POKE &4007,L : POKE &4008,H
240    INPUT "Set A ",A
250    POKE &400A,A
260    INPUT "Set CALL ",NM
270    NM=NM-65536*(NM<0)
280    N=INT(NM/256) : M=NM-N*256
290    POKE &400C,M : POKE &400D,N
300    CALL &4000
310    BC=PEEK(&4020)+256*PEEK(&4021)
320    DE=PEEK(&4022)+256*PEEK(&4023)
330    HL=PEEK(&4024)+256*PEEK(&4025)
340    A=PEEK(&4026)
350    PRINT "BC=";HEX$(BC,4)
360    PRINT "DE=";HEX$(DE,4)
370    PRINT "HL=";HEX$(HL,4)
380    PRINT "A= ";HEX$(A,2)
390    R=DE
400    N$="DE"
410    GOSUB 600
420    R=HL
430    N$="HL"
440    GOSUB 600
450    PRINT
460    INPUT Y
490    GOTO 110
500    FOR P=&4000 TO &401F
510    READ Q : POKE P,Q
520    NEXT
530    RETURN
540    DATA 1,0,0,&11,0,0,&21,0
550    DATA 0,&3E,0,&CD,0,0,&ED,&43
560    DATA &20,&40,&ED,&53,&22,&40,&22,&24
570    DATA &40,&E5,&21,&26,&40,&77,&E1,&C9
600    IF ABS(R)<16384 THEN RETURN
610    PRINT FP(";N$;")=":
```

```
620    FOR X=4 TO 0 STEP -1
630    PRINT HEX$(PEEK(R+X),2):
640    NEXT
650    PRINT " = ";
660    Z=0
670    FOR X=0 TO 3
680    Z=(Z+PEEK(R+X))/256
690    NEXT
700    IF Z<0.5 THEN Z=Z+0.5 : PRINT"+";ELSE PRINT "-"
       ;
710    Z=Z*2^(PEEK(R+4)-&80)
720    PRINT Z
730    RETURN
```

Appendix 2
INDEX BY LOCATION

This is an index of the Labels used to identify the locations, subroutines and vectors on the Amstrad CPC 464. The references are to memory locations and not page numbers. The numbers are given in Hexadecimal.

The Index below is set out in numeric order.

Location	Label
B9ØØ,BA5E	U ROM ENABLE
B9Ø3,BA68	U ROM DISABLE
B9Ø6,BA4A	L ROM ENABLE
B9Ø9,BA54	L ROM DISABLE
B9ØC,BA72	ROM RESTORE
B9ØF,BA7E	ROM SELECT
B912,BAA2	CURR SELECTION
B915,BAA2	PROBE ROM
B918,BA8C	ROM DESELECT
B91B,BAA6	LDIR
B91E,BAAC	LDDR
B921	KL POLL SYNCHRONOUS
ØØØB,B97C	LOW PCHL
ØØØ8,B982	LOW JUMP
ØØ1B,B9B1	FAR PCHL
ØØ23,B9B9	FAR ICALL
ØØ18,B9BF	FAR CALL
ØØ13,BA1Ø	SIDE PCHL
ØØ1Ø,BA16	SIDE CALL
ØØ28,BA2E	FIRM JUMP
ØØ2Ø,BACB	RAM LAM
BBØØ,19EØ	KM INITIALISE
BBØ3,1A1E	KM RESET
BBØ9,1A42	KM READ CHAR
BBØ6,1A3C	KM WAIT CHAR
BBØC,1A77	KM CHAR RETURN
BBØF,1ABD	KM SET EXPAND
BB12,1B2E	KM GET EXPAND
BB15,1A7B	KM EXP BUFFER
BB18,1B56	KM WAIT KEY
BB1B,1B5C	KM READ KEY
BB1E,1CBD	KM TEST KEY

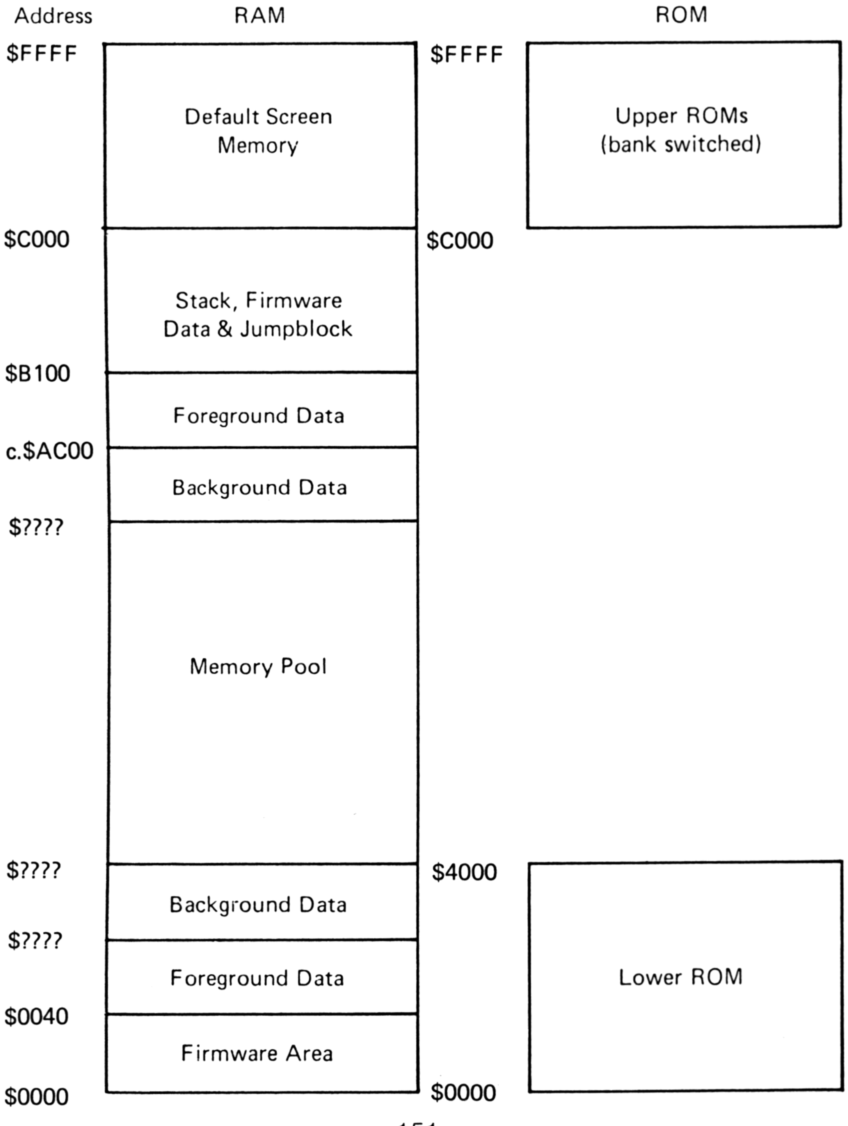
Location	Label
BB21,1BB3	KM GET STATE
BB24,1C5C	KM GET JOYSTICK
BB27,1D52	KM SET TRANSLATE
BB2A,1D3E	KM GET TRANSLATE
BB2D,1D57	KM SET SHIFT
BB3Ø,1D43	KM GET SHIFT
BB33,1D5C	KM SET CONTROL
BB36,1D48	KM GET CONTROL
BB39,1CAB	KM SET REPEAT
BB3C,1CA6	KM GET REPEAT
BB3F,1C6D	KM SET DELAY
BB42,1C69	KM GET DELAY
BB45,1C71	KM ARM BREAK
BB48,1C82	KM DISARM BREAK
BB4B,1C9Ø	KM BREAK EVENT
BB4E,1Ø78	TXT INITIALISE
BB51,1Ø88	TXT RESET
BB54,1415	TXT VDU ENABLE
BB57,144B	TXT VDU DISABLE
BB5A,14ØØ	TXT OUTPUT
BB5D,1334	TXT WR CHAR
BB6Ø,13AB	TXT READ CHAR
BB63,137A	TXT SET GRAPHIC
BB66,12ØC	TXT WIN ENABLE
BB69,1256	TXT GET WINDOW
BB6C,154Ø	TXT CLEAR WINDOW
BB6F,115E	TXT SET COLUMN
BB72,1174	TXT SET ROW
BB75,1174	TXT SET CURSOR
BB78,118Ø	TXT GET CURSOR
BB7B,1289	TXT CUR ENABLE
BB7E,129A	TXT CUR DISABLE
BB81,1279	TXT CUR ON
BB84,1281	TXT CUR OFF
BB87,11CE	TXT VALIDATE
BB8A,1268	TXT PLACE CURSOR
BB8D,1268	TXT REMOVE CURSOR
BB9Ø,12A9	TXT SET PEN
BB93,12BD	TXT GET PEN
BB96,12AE	TXT SET PAPER
BB99,12C3	TXT GET PAPER
BB9C,12C9	TXT INVERSE
BB9F,137A	TXT SET BACK
BBA2,1387	TXT GET BACK
BBA5,12D3	TXT GET MATRIX
BBA8,12F1	TXT SET MATRIX
BBAB,12FD	TXT SET M TABLE
BBAE,132A	TXT GET M TABLE
BBB1,14CB	TXT GET CONTROLS
BBB4,1ØE8	TXT STREAM SELECT

Location	Label
BBB7,11Ø7	TXT SWAP STREAMS
BBBA,15BØ	GRA INITIALISE
BBBD,15DF	GRA RESET
BBCØ,15F4	GRA MOVE ABSOLUTE
BBC3,15F1	GRA MOVE RELATIVE
BBC6,15FC	GRA ASK CURSOR
BBC9,16Ø4	GRA SET ORIGIN
BBCC,1612	GRA GET ORIGIN
BBCF,1734	GRA WIN WIDTH
BBD2,1779	GRA WINDOW HEIGHT
BBD5,17A6	GRA GET W WIDTH
BBDB,17C5	GRA CLEAR WINDOW
BBDE,17F6	GRA SET PEN
BBE1,18Ø4	GRA GET PEN
BBE4,17FD	GRA SET PAPER
BBEA,1813	GRA PLOT ABSOLUTE
BBED,181Ø	GRA PLOT RELATIVE
BBFØ,1827	GRA TEST ABSOLUTE
BBF3,1824	GRA TEST RELATIVE
BBF6,1839	GRA LINE ABSOLUTE
BBF9,1836	GRA LINE RELATIVE
BBFC,1945	GRA WR CHAR
BBFF,ØAAØ	SCR INITIALISE
BCØ2,ØAB1	SCR RESET
BCØ5,ØB3C	SCR SET OFFSET
BCØ8,ØB45	SCR SET BASE
BCØB,ØB5Ø	SCR GET LOCATION
BCØE,ØACA	SCR SET MODE
BC11,ØAEC	SCR GET MODE
BC14,ØAF2	SCR CLEAR
BC17,ØB57	SCR CHAR LIMITS
BC1A,ØB64	SCR CHAR POSITION
BC1D,ØB95	SCR DOT POSITION
BC2Ø,ØBF9	SCR NEXT BYTE
BC23,ØCØ5	SCR PREV BYTE
BC26,ØC13	SCR NEXT LINE
BC29,ØC2D	SCR PREV LINE
BC2C,ØC86	SCR INK ENCODE
BC2F,ØCAØ	SCR INK DECODE
BC32,ØCEC	SCR SET INK
BC35,ØD14	SCR GET INK
BC38,ØCF1	SCR SET BORDER
BC3B,ØD19	SCR GET BORDER
BC3E,ØCE4	SCR SET FLASHING
BC41,ØCE8	SCR GET FLASHING
BC44,ØDB3	SCR FILL BOX
BC47,ØDB7	SCR FLOOD BOX
BC4A,ØDDF	SCR CHAR INVERT
BC4D,ØDFA	SCR HW ROLL
BC5Ø,ØE3E	SCR SW ROLL

Location	Label
BC53,ØEF3	SCR UNPACK
BC56,ØF49	SCR REPACK
BC59,ØC49	SCR ACCESS
BC5C,ØC6B	SCR PIXELS
BC5F,ØFC4	SCR HORIZONTAL
BC62,1Ø2F	SCR VERTICAL
BC65,237Ø	CAS INITIALISE
BC68,237F	CAS SET SPEED
BC6B,238E	CAS NOISY
BC6E,2A4B	CAS START MOTOR
BC71,2A4F	CAS STOP MOTOR
BC74,2A51	CAS RESTORE MOTOR
BC77,2392	CAS IN OPEN
BC7A,23FC	CAS IN CLOSE
BC7D,24Ø1	CAS IN ABANDON
BC8Ø,2435	CAS IN CHAR
BC83,24AB	CAS IN DIRECT
BC86,249A	CAS RETURN
BC89,2496	CAS TEST EOF
BC8C,23AB	CAS OUT OPEN
BC8F,2415	CAS OUT CLOSE
BC92,242E	CAS OUT ABANDON
BC95,245B	CAS OUT CHAR
BC98,24EA	CAS OUT DIRECT
BC9B,2528	CAS CATALOG
BC9E,283F	CAS WRITE
BCA1,2836	CAS READ
BCA4,2851	CAS CHECK
BCA7,1E68	SOUND RESET
BCAA,1F9F	SOUND QUEUE
BCAD,2Ø6C	SOUND CHECK
BCBØ,2Ø89	SOUND ARM EVENT
BCB3,2Ø4A	SOUND RELEASE
BCB6,1ECB	SOUND HOLD
BCB9,1EE6	SOUND CONTINUE
BCBC,2338	SOUND AMPL ENVELOPE
BCBF,233D	SOUND TONE ENVELOPE
BCC2,2349	SOUND A ADDRESS
BCC5,234E	SOUND T ADDRESS
BCC8,ØØ5C	KL CHOKE OFF
BCCB,Ø329	KL ROM WALK
BCCE,Ø332	KL INIT BACK
BCD1,Ø2A1	KL LOG EXT
BCD4,Ø2B2	KL FIND COMMAND
BCD7,Ø163	KL NEW FRAME FLY
BCDA,Ø16A	KL ADD FRAME FLY
BCDD,Ø17Ø	KL DEL FRAME FLY
BCEØ,Ø176	KL NEW FAST TICKER
BCE3,Ø17D	KL ADD FAST TICKER
BCE6,Ø183	KL DEL FAST TICKER

Location	Label
BCE9,Ø1B3	KL ADD TICKER
BCEC,Ø15C	LK DEL TICKER
BCEF,Ø1D2	KL INIT EVENT
BCF2,Ø1E2	KL EVENT
BCF5,Ø228	KL SYNC RESET
BCF8,Ø285	KL DEL SYNCHRONOUS
BCFB,Ø256	KL NEXT SYNC
BCFE,Ø21A	KL DO SYNC
BDØ1,Ø277	KL DONE SYNC
BDØ4,Ø295	KL EVENT DISABLE
BDØ7,Ø29B	KL EVENT ENABLE
BDØA,Ø28E	KL DISARM EVENT
BDØD,ØØ99	KL TIME PLEASE
BD1Ø,ØØA3	KL TIME SET
BD13,Ø5DC	MC BOOT PROGRAM
BD16,Ø6ØB	MC START PROGRAM
BD19,Ø7BA	MC WAIT FLYBACK
BD1C,Ø776	MC SET MODE
BD1F,Ø7C6	MC SCREEN OFFSET
BD22,Ø786	MC CLEAR INKS
BD25,Ø799	MC SET INKS
BD28,Ø7E6	MC RESET PRINTER
BD2B,Ø7F2	MC PRINT CHAR
BD2E,Ø81B	MC BUSY PRINTER
BD31,Ø8Ø7	MC SEND PRINTER
BD34,Ø826	MC SOUND REGISTER
BD37,Ø888	JUMP RESTORE
BDCD,1263	TXT DRAW CURSOR
BDDØ,1263	TXT UNDRAW CURSOR
BDD3,134A	TXT WRITE CHAR
BDD6,13CØ	TXT UNWRITE
BDD9,14ØC	TXT OUT ACTION
BDDC,1816	GRA PLOT
BDDF,182A	GRA TEXT
BDE2,183C	GRA LINE
BDE5,ØC82	SCR READ
BDE8,ØC68	SCR WRITE
BDEB,ØAF7	SCR MODE CLEAR
BDEE,1C9Ø	KM TEST BREAK
BDF1,Ø7F8	MC WAIT PRINTER

Appendix 3
MEMORY MAP



Index

This book is the definitive guide for all serious programmers on the Amstrad CPC464.

Don Thomasson has examined every aspect of the Amstrad — its peripherals, the ROM and the RAM routines. This book contains a breakdown and explanation of all of the following:

Memory Map
Input/Output Map
Outer Peripherals
Jumpblock Entries
RAM routines
Main Reset
Printer routines
Interrupt Handler
Event System
Screen RAM
Streams
Parameters
Mode control
Addresses
Inks
Flash system
General routines
Colour

Windows
Matrix data
Text output
Graphics VDU
Keyboard routines
Input routines
Key/code table
Break functions
Cassette messages
Cassette routines
Cassette calls
File types
Sound calls
External ROM command words
External ROM routines
BASIC routines
BASIC interpreter

All of the routines available in the Amstrad are detailed with explanations and tables, as well as information on how to use the routines.

The book also contains a guide to all possible ROM configurations. The appendices include two programs that will allow you to examine the routines in the Amstrad and test various parameters.

If you are involved in programming the Amstrad CPC464 then you must have this book.

£7.95

Melbourne House Publishers

ISBN 0-86161-199-3

9 780861 611997

Amstrad Whole Memory Guide