9

OPERATING SYSTEM ROUTINES AND SPECIAL EFFECTS

A whole book would be needed to describe all the features of the operating system and the routines it contains. This chapter briefly introduces two operating system calls, 'OSBYTE' and 'OSWORD'. Between them, they perform a wide variet of tasks. It then shows how operating system routines can be intercepted and replaced by user defined ones. Finally it takes a look at some of the special effects which can be obtained using either the operating system commands or specialised hardware provided by the BBC Microcomputer and Acorn Electron.

9.1 OSBYTE and OSWORD

OSBYTE - &FFF4

OSBYTE calls can be used to access several operating system routines. The particular routine is selected by the number passed in the accumulator. The X and Y registers are used to pass any parameters needed to the routine and to pass back any results which may be produced as a result of the call, e.g.

```
LDA #12
LDX #10
JSR osbyte
```

will call OSBYTE 12 which sets the keyboard auto repeat rate, in this case to 10 centiseconds.

```
LDA #129
LDX #&9D
LDY #&FF
JSR osbyte
```

will call OSBYTE 129 which performs the INKEY function, in this case it is being called with a negative value (&9D = -99) and performs a keyboard scan to see if the key with this value, the Space bar, is being pressed.

On exit, X and Y contain &FF if the key being scanned was pressed and 0 otherwise.

*FX calls are used to access OSBYTE calls from BASIC. In this case the values of the registers are passed in the following way:

```
*FX 12,10
```

This will have the same effect as the first example.

However *FX calls do not return results so it is not appropriate to replace the second example given by a *FX call.

OSWORD - &FFF1

OSWORD routines are similar to OSBYTE routines, the difference being that parameters are not passed in the X and Y registers, instead they are passed in a parameter block, and the X and Y registers are used to contain the address of this block, e.g.

```
LDA #2
LDX #&80
LDY #&00
JSR osword
```

This calls OSWORD 2 which sets the value of the system clock to the five byte value which is stored in memory starting at the address &0080.

```
LDA #1
LDX #&80
LDY #&00
JSR osword
```

This calls OSWORD 1 which reads the system clock, the five byte value being returned in memory starting at the address &0080.

9.2 Revectoring operating system routines

Many of the operating system routines are not entered directly by jumping to their position in the ROM, instead they are entered via addresses stored in the RAM.

For example, to use the operating system write character routine (OSWRCH) (the instruction which has been used in previous examples is JSR &FFEE). Location &FFEE, however, contains not the start of the routine, but the instruction JMP (&20E), since the address of the code for OSWRCH is stored in locations &20E and &20F in RAM. These locations are known as the ' vector' for this routine.

Accessing routines indirectly, via vectors in the RAM has several advantages. In different operating systems the entry position of the routine may alter, but this will not affect the user since the instructions JSR &FFEE or JMF (&20E) will still access it. The difference will be dealt with by the operating system which will store the correct addresses in locations &20E and &20F.

In addition the user can intercept any of the routines by ' revectoringthem. For example he could change the contents of &20E and &20F so that they contained the address of a user defined routine. One use of this is shown below.

Pretty Printer -- prettyprint

When printing text to the screen it is often difficult to ensure that words will not be broken at the end of the line. The following routine achieves this. When linked in, it will buffer characters up to a space or carriage retum character, and then only output the characters on the same line if there is room without splitting them. Note that the routine does not deal with control characters (codes less than 32) that have trailing characters. The routine should be linked into the OSWRCH vector at &20E - &20F, by typing

```
!&20E =!&20E AND &FFFF0000 OR prettyprint
```

Notice that the variable ' linelengthis set to the length

of the line (19, 39 or 79,depending on the screen mode selected).

On entry A holds the character to be printed. X and Y are irrelevant.

A typical call would be any call to OSWRCH.

On exit all registers have been preserved.

An example of the output of this is: (40 column screen)

```
This text is Prettily Printed This text
is Prettily Printed This text is
Prettily Printed This text is Prettily
Printed This text is Prettily Printed
```

```
.prettyprint
   PHA
   STX savedx          Save X register
   LDX pointer         Get line pointer
   CMP #ASC" "         Is it a space ?
   BEQ isspace
   STA buffer,X        Store character
   INX                 Increment pointer
   CPX # linelength    Is buffer full ?
   BNE exit            If not, get next character
   STX pointer         Update pointer
   BEQ newline         Branch always
.isspace
   CPX #0
   BEQ exit
   JSR getpos
   LDX #0              Set X to zero for printbuffer
   LDA pos             Get cursor position( x-coord)
   CLC
   ADC pointer         Get cursor x+pointer
   CMP #£linelength    If >= linelength
   BCS newline         print buffer
   LDA pos             If cursor is at beginning
   BEQ printbuffer     of a line print the buffer
   LDA #ASC " "        Else print a space
   JSR printchar
   JMP printbuffer     and print the buffer
```

```
.newline
   LDA #13
         JSR printchar
         LDA #10
         JSR printchar
.printbuffer
   LDA buffer,X        Get characters
   JSR printchar       print characters
   INX                 increment pointer
   CPX pointer         if line pointer<> line end
   BNE printbuffer     then get next character
   LDX #0
.exit
   STX pointer         Save pointer
   LDX savedx          Restore X register
   PLA                 Restore A
   RTS                 And Return
.getpos
   TYA
   PHA                 Save Y
   LDA #&86            Osbyte 86 is read cursor position
   JSR osbyte          Returns pos and vpos in X and Y
   STX pos             X holds X coordinate of cursor
   PLA                 Restore Y
   TAY
   RTS
.printchar
   JMP (oldoswrch)     oldoswrch holds original contents
                       of &20E and &20F
```

9.3 Screen Scrolling

On both the BBC Microcomputer and the Acorn Electron, there are two screen-scrolling methods known as software scrolling and hardware scrolling. Software scrolling is often slow. If you define a text window to cover the whole screen (VDU 28,0,24,39,0 in MODES 6 or 7 only) and then scroll and screen (by moving the cursor off the bottom of the screen), you will notice the scrolling slowing down as it attempts to move all the screen memory up a line. An

alternative and faster method of scrolling has been incorporated in the hardware.

A section of each computer incorporates a register designed to hold the start of screen memory.. In the BBC machine it is the 6845 CRTC (Cathode Ray Tube Controller), and in the Electron it is a section of the ULA (Uncommitted Logic Array). To employ this screen-scrolling method, it is only necessarv to change the number in the register. On the next vertical sync, the new screen will be displayed starting at that number. To scroll the screen up on the BBC machine, simply type:

```
MODE 6
VDU 23; 12, &0C; 0; 0; 0; VDU 23; 13, &28; 0; 0; 0;
```

and on the Electron

```
MODE 6
?&FE02 = &A0 : ?&FE03 = &30
```

To explain: on the BBC machine there are in fact two registers which control hardware scroll. These are registers 12 and 13. Register 12 contains the high byte of the start address, and register 13 contains the low byte. Things are not quite this simple however, as the start address held in the two registers is only to the nearest 8 bytes (1 character cell in the MODEs 0 to 6), and so the number put into the registers is the start address, DIV 8. In the above example, the new address is &6000 + 40 * 8 (=&6140) DIV 8, which is &C28, and so we put &C in register 12 and &28 in register 13.

On the Electron things are not quite the same. The address held in the hardware is not to the nearest 8 bytes, but to the nearest 64 bytes.

The value to put into the Electron' sULA is the address of the top of the screen, divided by 2. The two registers are at &FE02 and &FE03 (low byte and high byte). The address, &6140, is written there by working out &6140 DIV 2 (=&30A0), and then writing the low and high bytes of the new value into the registers.

You will have noticed that the hardware scroll

operation is not exactly the same as that of the operating system scroll, in that the top line is filled not with spaces but with what was on the bottom line when the process began. This is because the memory map will ' wrap round' , as in the following diagram;

	0	1	2	3		37	38	39
0	&6140	&6148	&6150	...	...	&6268	&6270	&6278
1	&6141	&6149	&6151	...	...	&6269	&6271	&6279
2	&6142	&614A	&6152	...	...	&626A	&6273	&627B
3	&6143	&614B	&6153	...	...	&626B	&6273	&627C
4	&6144	&614C	&6154	...	...	&626C	&6274	&627C
5	&6145	&614D	&6155	...	...	&626D	&6275	&627D
6	&6146	&614E	&6156	...	...	&626E	&6277	&6?7E
7	&6147	&614F	&6157	...	...	&626F	&6277	&627F
8	&6280	&6288	&6290	...	...	&64E8	&64F0	&64F8
	. .	. .	. .	. .	. .	. .	. .	. .
24*8+6	&7F46	&7F4E	&7F56	...	...	&606E	&6076	&607E
24*8+7	&7F47	&7F4F	&7F57	...	...	&606F	&6077	&607F

9.4 Palette handling

Both the BBC Microcomputer and the Acorn Electron provide a palette facility in the ' soft'screen modes. On the Electron all modes are ' soft'screen modes, but on the BBC machine Teletext (MODE 7) has no palette facility. The idea is that each mode can display a certain number of colours at any one time (16 in MODE 2, 4 in MODE 5 and so on).

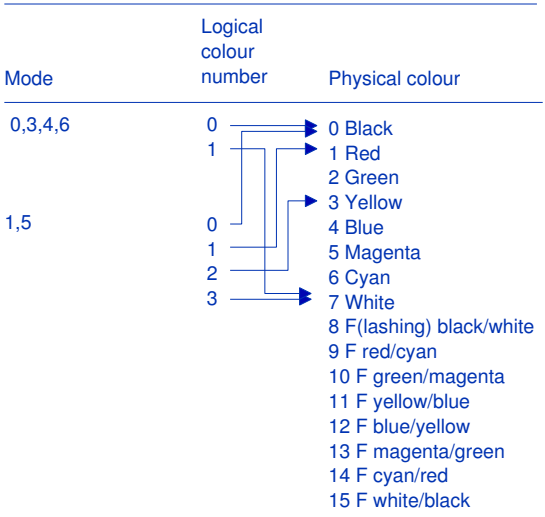
Essentially the palette provides a mapping between the screen memory and what appears on the screen. The screen memory contains logical colours, and these are represented by the palette as physical colours which you see displayed on the screen. Thus in MODE 1, where there are four logical colours, each can be represented as any of the sixteen physical colours which these computers are capable of producing. Use the VDU19 statement to tell the computer how to represent a particular logical colour as a particular physical colour, i.e.

```
VDU 19, logical colour, physical colour;0;
```

For example

```
VDU 19,1,3;0;
```

This tells the computer to display all occurences of logical colour 1 in its memory as physical colour 3 (Yellow). Think of the palette as a mapping of physical colours onto logical colours, where all that the VDU 19 statement does is to simply change the mapping. The illustration below should make this clear;



Another way of changing the palette is to call OSWORD with A set to 12. In this you simply set up a block of 5 bytes to this format:

paletteblock	logical colour
paletteblock+1	Physical colour
paletteblock+2	0
paletteblock+3	0
paletteblock+4	0

Then OSWORD is called in the normal manner (with X and Y pointing to the parameter block, ' paletteblockin this case). This has precisely the same effect as VDU 19, except that it is faster and also may be called from an interrupt or Event routine (See Interrupts below).

OSWORD 12 is not available on BBC machines with OS 0.10.

9.5 Interrupts, events and BREAKs

Interrupts

Both the BBC Microcomputer, and the Acorn Electron run under interrupt' Interrupts allow the machine to update its own internal variables, without the user even realising that their program is not in complete control.

On the 6502 there is an interrupt request pin (IRQ) which, when a signal hits it, tells the processor that an interrupt request has occurred. The 6502 then has the option of ignoring the interrupt. This decision is made by the state of an interrupt flag. If the flag is set, then the interrupt will be ignored, otherwise the operating system will deal with it.

The interrupt flag can be altered with the two assembler instructions:

Mnemonic	Description
SEI	set interrupt disable flag
CLI	clear interrupt disable flag (Default state)

Note that the interrupt flag should not really be altered, as then all interrupt driven devices (keyboard, flashing colours, sound, etc.) would stop working.

There are two interrupt vectors provided by the

operating system. These are IRQ1V (at &204), through which all interrupt requests are passed, and IRQ2V (at &206), through which any unrecognised interrupts are passed. Normally, IRQ2V would be used, but if you want to update your own device before the operating system can act, then you should use IRQ1V. The routine must first handle the interrupt, then disable the device that caused the interrupt, and finally, it must perform a ' JMF(oldIRQIV)' (where ' oldIRQ1Vis the old contents of IRQI V). This should only be used if there is no other way of achieving the desired effect.

One way to set up interrupts on the BBC Microcomputer is by the User 6522 VIA (Versatile Interface Adapter). This has two timers, which can be set to count down from any particular 16-bit value, and to cause an interrupt request on reaching zero. Also, it will be necessary to write a routine to handle this, and to put the address of the entry point of the routine in the correct vector (in this case IRQ2V). Note that the routine must perform an ' RTI'in order to transfer control back to the operating system. RTI stands for return from interrupt.

All this might seem a bit messy, and so a second kind of interrupt peculiar to the BBC microcomputer and Electron has been implemented. This second kind of interrupt is called an Event. (It is not implemented on BBC Microcomputers with OS 0.1).

Events

These operate in a similar way to interrupts in that they are totally transparent (undetectable by the user program) and are indirect, via a vector (at &220).

Certain occurrences within the machine have events associated with them, and these events can be trapped by the user. These are:

0 — Buffer empty, where X gives buffer identitv
1 — Buffer full, where X gives buffer identity and Y holds character that could not be stored.
2 — Keyboard interrupt
3 — ADC conversion complete
4 — Start of TV field pulse (vertical sync)
5 — Interval timer crossing zero
6 — Escape condition detected
7 — R5423 receive error

8 — Remote procedure call detected (on Econet)

Events can be selectively disabled and enabled with OSBYTEs 13 and 14, where X specifies the event. Note that the default state is all events disabled.

Example event handler:

```
 10 REM Event handler
 20
 30 vsynccounter=&70
 40 DIM code 100
 50 FOR pass = 0 TO 2 STEP 2
 60 P%=code
 70[ OPT pass
 80.event
 90   PHP                Preserve status
100   CMP #4             Is event for us?
110   BNE notvsync       If not, then return
120   INC vsynccounter   Count vsyncs
130.notvsync
140   PLP                Restore status
150]  RTS                Return
160]
170 NEXT pass
180
190 eventvec=&220
200 ?eventvec=FNlo(event)
210 eventvec?1=FNhi(event)
220 *FX 14 4
230 END
240
250 DEF FNlo(value) = value AND &FF
260
270 DEF FNhi(value) = (value AND&FF00) DIV 256
```

BRKs

The 6502 supports a BRK instruction. This generates a software interrupt, which is similar to the interrupt request described earlier, except that it cannot be disabled. BASIC and the operating system use BRKs for flagging errors. This means that the BRK handler

will print an error message. The standard format of the BRK error message is;

BRK instruction (op-code is &00)
Fault number (one byte)
Fault message (string of characters terminated by a zero byte)

Thus it is possible to put error messages in a program, and have them printed out by the BRK handler (which, incidentally, is normally handled by the language). This can be useful for debugging purposes. A useful macro for this is:

```
 DEF FNerror(err, error$)
[OPT pass
 BRK              Cause BREAK
 EQUB err         Fault number
 EQUS error$      Error message
 EQUB 0           Message terminator
]
=pass
```

A typical call to this would be:

```
OPT FNerror(60, "Hello")
```

where ' 60'is the fault number, and ' Hello'is the message to be printed when that BRK is activated.

It is, of course, possible to write your own BRK handler, by simply putting the start address of a suitable routine in the BRK vector (&202). For example, the following routine prints out all registers at a BRK:

```
  0 REM BREAK Handler
 10
 20 oswrch = &FFEE
 30 osnewl = &FFE7
 40 stringptr=&70
 50 exit=stringptr
 60 temp=stringptr+2
 70 exit = !&202
 80 DIM code 200
 90 FOR pass = 0 TO 2 STEP 2
```

```
100 P% = code
110[ OPT pass
120.header
130   EQUS " A  X  Y  PC  N V U B D I Z C"
                              + CHR$10 + CHR$ 13
140.break
150   TYA                  X and Y
160   PHA                  Push all registers so
170   TXA                  they may be printed out
180   PHA
190   LDA &FC              Get accumulator
200   PHA
210   JSR osnewl           go onto a new line
220   LDX #FNlo(header)    Print "A X Y PC..
230   LDY #FNhi(header)
240   JSR atomstring
250   PLA                  Get accumulator
260   JSR hexandspace      Print it and a space
270   PLA                  Get X register
280   JSR hexandspace      Print it and a space
290   PLA                  Get Y register
300   JSR hexandspace      Print it and a space
310   LDA &FE              FE an FD hold the
320   JSR printhex         program counter where
330   LDA &FD              the BREAK occured
340   JSR hexandspace
350   PLA                  Status register
360   JSR printbinary      Print P in binary
370   JMP exit             Return to old error
380                        handler
390.hexandspace
400   JSR printhex         Print A in hexadecimal
410   LDA #ASC" "          Print a space,
420   JMP oswrch           then return
430
450   PHA                  Save accumulator
460   LSR A                Get top nibble
470   LSR A
480   LSR A
490   LSR A
500   JSR print            Print top hex digit
510   PLA                  Get bottom nibble
520   AND #&OF
530.print                  Print bottom hex digit
540   CMP #&0A
550   BCC notalpha         If not 0..9 get char
560   ADC #6               Carry is set here (add 7)
570.notalpha
580   ADC #&30             Convert to ASCII
```

```
 590   JMP oswrch           Print and return
 600
 610.printbinary
 620   LDX #8               Eight bits per byte
 630.binaryloop
 640   ASL A                Get a bit
 650   STA temp             Print either 0 or 1
 660   LDA #ASC"0"
 670   BCC printzero
 680   LDA #ASC"1"
 690.printzero
 700   JSR oswrch           Print 0 or 1
 710   LDA #ASC" "          followed by a space
 720   JSR oswrch
 730   LDA temp
 740   DEX
 750   BNE binaryloop       Get next bit
 760   RTS                  Return
 770
 780.atomstring
 790   STX stringptr        Address of string
 800   STY stringptr+1      is given in X and Y
 810   LDY #&I00            Y is pointer along string
 820.atomstringloop
 830   LDA (stringptr),Y    Get next character
 840   JSR oswrch           Print it
 850   INY                  Increment pointer
 860   CMP #13              Is it a RETURN
 870   BNE atomstringloop   If not, repeat loop
 880   RTS                  Else return
 890
 900]
 910 NEXT pass
 920 !&202=!&202 AND &FFFF000O OR break
 930[OPT 2
 940.test
 950   LDA #&01
 960   LDX #&23
 970   LDY #&45
 980   SED
 990   CLC
1000   BRK
1010   EQUB 75
1020   EQUS "HELLO"
1030   EQUB 0
1040]
1050 CALL test
1060
1070 DEF FNIo(value)=value AND &FF
```

```
1080
1090 DEF FNhi(value)=(value AND &FF00)DIV &100


>RUN

 A  X  Y  PC  N V U B D I Z C
01 23 45 1BB6 0 0 1 1 1 0 0 0
HELLO at line 1050
>
```