# 6 Operating System Vectors

Many of the operating system routines are indirected through addresses stored in RAM. This enables other software to intercept these calls as they are made.

During a reset the operating system stores the addresses of its internal routines for such things as reading and writing characters in locations in page two. The official entry point of these routines point to instructions like JMP (vector). If another piece of software replaces the address stored in the vector then each subsequent call is passed to the intercepting software.

Consider the following example:

This program assembles a routine which intercepts ‘$’ and ‘£’ characters passed to the OSWRCH routine and exchanges them.

```
 10 DIM code% 100
 20 WRCHV=&20E
 30 FOR opt%=0 TO 3 STEP3
 40 P%=code%
 50 [
 60 OPT opt%
 70 .init     LDA WRCHV             \ A=lo byte of vector
 80           STA ret_vec           \ make a copy
 90           LDA WRCHV+1           \ A=hi byte of vector
100           STA ret_vec+1         \ make a copy
110           LDX #intrcpt AND &FF  \ X=lo byte of new routine
120           LDY #intrcpt DIV &100 \ Y=hi byte of new routine
130           SEI                   \ disable interrupts
140           STX WRCHV             \ store new routine address
150           STY WRCHV+1           \ in WRCH Vector
160           CLI                   \ enable interrupts
170           RTS                   \ finished initialisation
180 .intrcpt  CMP #ASC"£"           \ trying to print a £ ?
190           BEQ pound             \ if so branch
200           CMP #ASC"$"           \ trying to print a $ ?
210           BEQ dollar            \ if so branch
220           JMP (ret_vec)         \ neither goto old routine
230 .pound    LDA #ASC"$"           \ replace £ with $
240           JMP (ret_vec)         \ goto old routine
250 .dollar   LDA #ASC"£"            \ replace $ with £
260           JMP (ret_vec)          \ goto old routine
270 .ret_vec  EQUW 0                 \ space for return vector
280 ]
290 NEXT
300 CALL init
```

This program, although not very long, illustrates a few points regarding the way in which vectors should be intercepted.

One of the most important aspects concerning the interception of calls through vectors is to make sure that the call is passed on to the previous owner of the vector. There are occasions when a routine is intended to be the sole replacement of a vector but as a rule it is good programming practice to copy the old vector contents to a returning vector. By returning via the old vector contents any number of intercepting routines can be daisy chained into the operating system call.

While the initialising routine is changing the vector contents to point at the new routine it is wise to disable interrupts, It would obviously be quite catastrophic if the OSWRCH routine were to be called when the vector was only half changed. An interrupt handling routine is unlikely to use the WRCHV but there is no reason why it should not.

The intention in this section has been to make programmers aware of the problems which may occur when intercepting these vectors. They have been implemented so that they may be used to insert extra code into some of the operating system routines and individuals should not be afraid of using them to this end. However, careful thought is required; take full account of the ramifications of altering the operating systems usual response to calls. If in doubt try out a routine. Play about with trivial examples such as the one given above. There is nothing to be lost and much to be learnt. Os and filing system calls indirection vectors

# OS and filing system calls indirection vectors

The vector addresses associated with those operating system calls which are indirected are given in the detailed description of each call in chapter 2. The entry conditions with which the routine whose address is containedwithin these vectors will be unchanged from the initial OS call.

# Other page 2 vectors

The other vectors reserved for containing the addresses of other operating system and miscellaneous routines are described below. These are :

| **Name** | **addr.** | **description** |
|---|---|---|
| USERV | &200 | The user vector |
| BRKV | &202 | The BRK vector |
| IRO1V | &204 | Primary interrupt vector |
| IRO2V | &206 | Unrecognised IRQ vector |
| FSCV | &21E | File system control entry |
| EVNTV | &220 | Event vector |
| UPTV | &222 | User print routine |
| NETV | &224 | Econet vector |
| VDUV | &226 | Unrecognised VDU commands |
| KEYV | &228 | Keyboard vector |
| INSV | &22A | Insert into buffer vector |
| REMV | &22C | Remove from buffer vector |
| CNPV | &22E | Count/purge buffer vector |
| IND1V | &230 | unused/reserved for future expansion |
| IND2V | &232 | unused/reserved for future expansion |
| IND3V | &234 | unused/reserved for future expansion |

## 6.1 The User Vector &200

The user vector is called by the operating system in three circumstances.

(a) When *CODE is passed to the command line interpreter

The *CODE command takes two parameters which are placed in the X and Y registers. The user vector is then called with an accumulator value of zero. OSBYTE &88 may also be used to generate a *CODE command.

(b) When *LINE is passed to the command line interpreter

The *LINE command takes a line of text as a parameter. The user vector is entered with the X and Y registers containing the address of this text and A=1.

(c) When an OS WORD call &E0 to &FF has been made The user vector is entered with the register values they were when the original OS WORD call was made.

The default address stored in this vector points to a routine which generates an error with the message ‘Bad command’ and error number &FE.

This vector is fully implemented on the BBC microcomputer and the Electron. On a Tube machine only the vector on the I/O processor is offered these calls.

Listed below is a program which assembles a routine to intercept calls made to the user vector. It may be noticed that this routine does not offer the calls back to the original vector routine, this is because the default routine generates an error. There should only be one user vector handling routine active at any one time.

```
  0 REM User vector handling routine

 10 DIM code% &100
 20 OSASCI=&FFE3
 30 USERV=&200
 40 FOR opt%=0 TO 3 STEP 3
 50 P%=code%
 60 [
 70 OPT opt%

 80 .init    LDX #userrt AND &FF  \ X=lo byte of routine addr.
 90          LDY #userrt DIV &100 \ Y=hi byte of routine addr.
100          SEI                  \ disable interrupts
110          STX USERV            \ set up vector with addr.
120          STY USERV+1
130          CLI                  \ enable interrupts
140          RTS                  \ and return
150 .userrt  CMP #1               \ compare contents of A with1
160          BCC code             \ A<1 then must be *CODE
170          BNE osword           \ now if A<>1 must be OSWORD

180          STX &70              \ *LINE routine
190          STY &71              \ store text address in page0
200          LDY #&FF             \ set Y as loop counter
210 .loop    INY                  \ beginining of loop Y=Y+1
220          LDA (&70),Y          \ load first byte of string
230          JSR OSASCI           \ print it
240          CMP #&D              \ was character a cr?
250          BNE loop             \ if not get the next char.
260          RTS                  \ if it was return

290 .code    TXA                  \ A=X
300          JSR prntbt           \ print value of X
310          JSR space            \ print a space
320          TYA                  \ A=Y
330          JSR prntbt           \ print value of Y
340          JMP new_ln           \ print newline and return

350 .osword  PHA                  \ save contents of A
360          LDX #&FF             \ set X as loop counter
370 .loop1   INX                  \ beginning of loop, X=X+1
380          LDA string,X         \ load character from string
390          JSR OSASCI           \ print it
400          CMP #ASC"&"          \ & char. is end of string
410          BNE loop1            \ loop if not end of string
420          PLA                  \ reload the value of A
430          JSR prntbt           \ print it out in hex
440          JMP new_ln           \ print cr and return
450 .space   LDA #&20             \ A=space character
460          JMP OSASCI           \ print and return

470 .new_ln  LDA #&D              \ A=carriage return character
480          JMP OSASCI           \ print cr and return

490 .string  EQUS "OSWORD &"      \ string for OSWORD routine

499\*** This routine prints hex number given in A

500 .prntbt  PHA                  \ save copy of accumulator
510          LSR A
520          LSR A
530          LSR A
540          LSR A                \ shift nibble hi to lo
550          JSR nibble           \ print hi nibble hex digit
560          PLA                  \ reload accumulator
570 .nibble  AND #&0F             \ mask out high nibble
580          CMP #&0A             \ digit or letter?
590          BCC number           \ A<10print number
600          ADC #&06             \ otherwise add 7 (C=1)
610 .number  ADC #&30             \ add &30 to convert to ASCII
620          JMP OSASCI           \ print character and return

630 ]
640 NEXT
650 CALL init
```

Once assembled this routine will respond to *CODE by printing out the parameters passed with the command. A *LINE command will result in the parameter string being repeated on the screen and an OS WORD in the region &E0 to &FF will print out the number of the call.

e.g.

```
>*CODE 1,2
01 02
>*LINE SOME TEXT
SOME TEXT
>A%=&E0:CALL &FFF1
OSWORD &E0
>
```

## 6.2 The BRK Vector &202

When a BRK instruction (op code value 0) is executed an interrupt is generated. The operating system stores the address of the byte following the BRK instruction in &FD and &FE, offers the BRK to paged ROMs with service call &06, stores the ROM number of the currently active paged ROM for recovery using OSBYTE &BA (ROM active at last BRK), restores registers, selects the current language ROM and then passes the call to the BRKV code.

The BRK instruction is normally used on Acorn machines to represent an error condition and the BRK vector routine is an error handling routine. In BASIC this error handling routine starts off by putting its house in order and then prints out an error message.

In addition to the use of BRKs for the generation of errors it is often useful in machine code programming to include BRKs (break-points) as a debugging aid.

If a BRK instruction is executed on the Electron, the BRK vector is entered with the following conditions:

(a) The A, X and Y registers are unchanged from when the BRK instruction was executed.

(b) An RTI instruction will return execution to the address two bytes after the BRK instruction (i.e. jumps over the byte following the BRK). The RTL instruction also restores the status register value from the stack.

(c) The address of the byte following the BRK instruction is stored in zero page locations &FD and &FE, This address can then be used for indexed addressing.

Error handling BRK routines should not return to the code which executed the BRK but should reset the stack (using a TXS instruction) and JMP into a suitable reset entry point. In fact the convention used by Acorn is to follow the BRK instruction by:

a single byte error number
an error message
a zero byte to terminate the message

and the BRK routine prints out the error name. The BRK handling routine should normally be implemented by the current language. Service paged ROMs should copy a BRK instruction followed by the error number and message down into RAM when wishing to generate an error. This has to be done because otherwise the current language ROM is paged in and the BRK handling routine tries to print out the error message from the wrong ROM. The bottom of page 1 is often used and is quite safe as long as the BRK handling routine resets the stack pointer.

The use of BRKs as break-points in machine code programming can be of great use to the machine code programmer. The example below shows how a BRK handling routine may be used to print out the register values. This routine could be further enhanced by printing out the value of the byte following the BRK instruction which would then give the programmer 256 individually identifiable break-points.

```
 10 REM Primitive BRK handling routine
 20 DIM code% &100
 30 OSASCI=&FFE3
 40 OSRDCH=&FFE0
 50 BRKV=&202
 60 FOR opt%=0 TO 3 STEP 3
 70 P%=code%
 80 [
 90 OPT opt%

100 .init   LDX #brkrt AND &FF   \ load registers with address
110         LDY #brkrt DIV &100
120         SEI                  \ disable interrupts
130         STX BRKV             \ set up BRK vector
140         STY BRKV+1
150         CLI                  \ enable interrupts and return
160         RTS

170 .brkrt  PHA                  \ save A (X and Y not used)
180         STA byte             \ store A in workspace
190         LDA #ASC"A"          \ register id
200         JSR prntrg           \ print register value
210         STX byte             \ store X in workspace
220         LDA #ASC"X"          \ register id
230         JSR prntrg          \ print register value
240         STY byte            \ store Y in workspace
250         LDA #ASC"Y"         \ register id
260         JSR prntrg          \ print register value
270         JSR new_ln          \ print carriage return
280         JSR OSRDCH          \ wait for key press
290         PLA                 \ restore A
300         RTI                 \ return

310 .prntrg JSR OSASCI          \ print register id
320         LDA #ASC":"
330         JSR OSASCI          \ print colon
340         JSR space           \ print space
350         LDA #ASC"&"
360         JSR OSASCI          \ print ampersand
370         LDA byte            \ get register value
380         JSR prntbt          \ print hex number
390         JSR space
400         JSR space           \ print two spaces
410         RTS

420 .space  LDA #&20
430         JMP OSASCI          \ print space

440 .new_ln LDA #&D
450         JMP OSASCI          \ print carriage return

460 .prntbt PHA                 \ for comments refer to
470         LSR A               \ previous example
480         LSR A
490         LSR A
500         LSR A
510         JSR nibble
520         PLA
530 .nibble AND #&0F
540         CMP #&0A
550         BCC number
560         ADC #&06
570 .number ADC #&30
580         JMP OSASCI
590 .byte   EQUB 0              \ workspace byte

600 .test   BRK                 \ cause an error
610         EQUB 0              \ RTI returns to next byte
620         DEX                 \ Loop X times
630         BNE test            \ if X=0 loop again
640         RTS
650 ]
660 NEXT
670 CALL init
680 A%=1:X%=8:Y%=&FF:CALL test
```

## 6.3 The interrupt vectors, IRQ1V &204 and IRQ2V &206

The interrupt system on the Electron is described in chapter 7. The function of the two interrupt vectors are described there.

## 6.4 The event vector, EVNTV &220

This vector is called by the operating system during its interrupt routine to provide users with an easy to use interrupt, A number of ‘events’ may cause the event handling routine to be called via this vector but unlike an interrupt the reason for the call is passed to the routine. The value in the accumulator indicates the type of event:

| **event no.** | **cause of event** |
|---|---|
| 0 | output buffer becomes empty |
| 1 | input buffer becomes full |
| 2 | character entering input buffer |
| 3 | ADC conversion complete |
| 4 | start of VSYNC |
| 5 | interval timer crossing zero |
| 6 | ESCAPE condition detected |
| 7 | RS423 error detected |
| 8 | Econet event |
| 9 | user event |

To avoid unecessary and time consuming calls to the event vector two OSBYTE calls are used to enable and disable these event calls being made. These are &D (13) for disabling and &E (14) for enabling events.

The event handling routine should not enable interrupts and not last for more than about 2 milliseconds. So that event handling routines may be daisy chained they should preserve registers and return using the old vector contents.

### Output buffer empty 0

This event enters the event handling routine with the buffer number (see OSBYTE &15/*FX21) in X. It is generated when a buffer becomes empty (i.e. just after the last character is removed).

### Input buffer full 1

This event enters the event handling routine with the buffer number (see OSBYTE &15/*FX 21) in X. It is generated when the operating system fails to enter a character into a buffer because it is full. Y contains the character value which could not be inserted.

### Character entering input buffer 2

This event is normally generated by a key press and the ASCII value of the key is placed in Y, It is generated independently of the input stream selected.

### ADC conversion complete 3

When an ADC conversion is completed on a channel this event is generated. The event handling routine is entered with the channel number on which the conversion was made in Y. This event is generated by the Plus 1 expansion software.

### Start of vertical sync 4

This event is generated 50 times per second coincident with vertical sync. One use of this event is to time the change to a video ULA register so that the change to the screen occurs during fly back and not while the screen is being refreshed. This avoids flickering on the screen.

| | |
|---|---|
| **Interval timer crossing zero** | **5** |

This event uses the interval timer (see OSWORD calls &3 and &4, in chapter 4). This timer is a 5 byte value incremented 100 times per second. The event is generated when the timer reaches zero.

| | |
|---|---|
| **ESCAPE condition detected** | **6** |

When the ESCAPE key is pressed or an ESCAPE is received from the RS423 (if RS423 ESCAPEs are enabled) this event is generated.

| | |
|---|---|
| **RS423 error event** | **7** |

This event should be generated by software servicing expansion RS423 hardware.

| | |
|---|---|
| **Network error event** | **8** |

This event is generated when a network event is detected. If the net expansion is not present then this could be used for user events.

| | |
|---|---|
| **User event** | **9** |

This event number has been set aside for the user event, This is most usefully generated from a user interrupt handling routine to enable other user software to trap an interrupt easily (e.g. an event generated from an interrupt driven utility in paged ROM). An event may be generated using OSEVEN, see section 2.10

## 6.5 User print vector, UPTV &222

A user print routine can be implemented by intercepting this vector, Whenever a change in printer type is made using OSBYTE &05 the print vector is called. A user print routine should respond when printer type 3 is called.

The operating system will activate the user printer routine and there after call it regularly at intervals of 10 milliseconds. Characters will be placed in the printer buffer and it is up to the user printer routine to remove characters and send them to the printer hardware. When the printer routine finds that the buffer is empty it should then declare itself inactive. The operating system will then re-activate the routine when characters start entering the buffer again.

The user printer driver should preserve all registers and return via the old UPTV value.

On entry:

X contains the buffer number to be used
Y contains the printer number (i.e. the *FX 5 value)

N. B. The routine should only respond if it recognises the printer number as its own.

The accumulator contains a reason code for the call:

A=0

When the printer driver is active the operating system makes this call every 10 ins. The printer driver should examine its hardware and if it is ready for another character should remove a character from the assigned buffer and send it to the printer. A call to the REMV vector should be made to obtain the character (see section 6.9.2) or use OSBYTE &91, When the printer driver has emptied the printer buffer it should then declare itself inactive by making an OSBYTE call &7B. This will allow the user to select a new printer driver using OSBYTE &5, will stop further calls with A=0 and thereafter when the printer buffer is used again will cause a call with A=1 to be made (see below).

A=1

When a printer driver is inactive this call is made to tell the routine that the printer buffer is no longer empty and the printer driver should now become active. If the printer driver is able to become active it should remove a character from the assigned buffer and if the buffer is still not empty it should return with the carry flag clear to indicate that it is now active. Having thus signalled itself as active the printer driver will receive the 10 ms calls with A=0.

A=2

When the VDU drivers receive a VDU2 this call is made. Characters may be printed even when this control character has not been received if certain *FX3 options are selected.

A=3

This call is made when a VDU3 is received.

A=5

The selection of a new printer driver will cause this call to be made to the printer vector. Any OSBYTE &5 call causes this call to be made.

## 6.6 Econet vector, NETV &224

The Econet vector allows the Network filing system to intercept a wide range of operating system functions. This vector is called with a reason code in the accumulator. The conditions under which this vector is called are:

A=0,1,2,3 and 5

These codes are used to control the net printer. These calls are made under identical circumstances as for the user print vector described above. The net printer is assigned the printer number 4.

A=4

OSWRCH call made. This call is indirected through the net vector after OSBYTE &D0 has been used. The Y register contains the value originally passed in the accumulator. If, on exit, the carry flag is set then the output call is not performed.

A=6

OSRDCH call made. This call is indirected through the net vector after OSBYTE &CF has been used. The ASCII value for a key read should be returned in the accumulator.

A=7

OSBYTE call made. This indirection is performed after OSBYTE &CE has been used. The OSBYTE parameters are stored in locations &EF, &F0 and &F1. If the overflow flag is set on return from this call then the OSBYTE call is not performed.

A=8

OSWORD call made. Circumstances as for call with A=7.

A=&0D

After completion of a line of input using OSWORD &01 this call is made. This is implemented so that the Network filing system doesn’t takeover the RDCH routine in the middle of line input.

## 6.7 VDU extension vector, VDUV &226

This vector is called when the VDU drivers are presented with an unknown command or a known command in a non-graphics MODE.

A VDU 23,n command with a value of n in the range 2 to 31 will cause a call to be made to this vector with the carry flag set. The accumulator will contain the value n.

An unrecognised PLOT command or the use of a PLOT command in a non-graphics MODE will result in this call being made with the carry flag clear. The accumulator will contain the PLOT number used.

## 6.8 The keyboard vector, KEYV &228

This vector is used whenever the keyboard is being looked at. There are four different calls made through this vector on the Electron.

(a) Test SHIFT and CTRL keys On entry: C=0, V=0

Should exit with the N (negative) flag set if the CTRL key is pressed and with the V (overflow) flag set if the SHIFT key is pressed.

(b) Scan keyboard as for OSBYTE &79

On entry: C=1 , V=0 other parameters identical to OSBYTE &79

Should exit with the appropriate register values (see OSBYTE details) but with A=X.

(c) Timer interrupt service with keys active

On entry: C=1, V=1

This entry is actually used for the bulk of all keyboard processing. After an interrupt the actual keyboard scan is carried out during this call. On the Electron which doesn't use an interrupt driven keyboard, intercepting this call to the KEYV routine and returning it speeds up the machine enormously.

(d) Timer interrupt service with no keys active

On entry: C=0, V=1

## 6.9 The buffer maintenance vectors

This vector and the two following vectors enable the user to intercept or use the operating system buffer maintenance routines.

The operating system uses buffers for keyboard input, RS423 input and output, the printer, the sound system (4 buffers) and the speech system. These buffers contain data which should be processed by the various routines. Even though the servicing routine may not be able to respond to the request immediately the calling routine returns (unless the buffer is full) and is able to get on with its foreground task. While a buffer contains a queue of data for processing, the interrupt routine (the background task) sees to it that the relevant routines service this data.

In this way the user is able to type ahead when the machine is unable to respond immediately and may initiate sounds which then continue while he issues further commands.

Buffers operate on a first in first out (FIFO) basis for obvious reasons.

The Acorn BBC range of machines use the following numbers as buffer IDs:

| **title** | **number** |
|---|---|
| keyboard buffer | 0 |
| RS423 input buffer | 1 |
| RS423 output buffer | 2 |
| printer buffer | 3 |
| SOUND channel 0 buffer | 4 |
| SOUND channel 1 buffer | 5 |
| SOUND channel 2 buffer | 6 |
| SOUND channel 3 buffer | 7 |
| speech buffer | 8 |

On the BBC microcomputer and the Electron memory is reserved for each of these buffers even though the software/hardware using the buffer may not be present. The buffer maintenance calls still service these buffers but the contents will not be processed by the relevant service routine. The expansion software/hardware will use the appropriate buffer when installed. Thus when the speech expansion is fitted on a BBC microcomputer the speech buffer is used and on an Electron with a Plus 1 the printer buffer is used.

The following OSBYTE calls may also be of interest when considering the buffer facilities:

| **description** | **OSBYTE number** |
|---|---|
| flush selected buffer class | &0F (15) |
| flush particular buffer | &15 (21) |
| get buffer status | &80 (128) |
| insert value into buffer | &8A (138) |
| get character from buffer | &91 (145) |
| examine buffer status | &98 (152) |
| insert value into i/p buffer | &99 (153) |

### 6.9.1 Insert value into buffer vector, INSV &22A

This vector contains the address of a routine which inserts a value into a selected buffer.

Entry parameters:
A=value to be inserted
X=buffer id

On exit:
A and X are preserved
Y is undefined

C flag is set if insertion failed (i.e.buffer full)

### 6.9.2 Remove value from buffer vector, REMV &22C

This vector contains the address of a routine which removes a value from the selected buffer. This routine may also be used to examine the next character to be removed from a buffer without actually removing it.

Entry parameters:
X=buffer ID
V= 1 (overflow flag set) if only examination requested

On exit:
A contains next byte to be removed (examination call)
(A undefined for removal call)
X is preserved
Y contains the value of the byte removed from the buffer
(Y undefined for examination call)
C flag is set if buffer empty when call made

### 6.9.3 Count/purge buffer vector, CNPV &22E

This vector contains the address of a routine which may be used to clear the contents of a buffer or to return information about the free space or contents of a buffer.

Entry parameters:
X=buffer ID
V=1 (overflow flag set) to purge buffer
V=0 (overflow flag clear) for count operation
C=1 count operation returns amount of free space
C=0 count operation returns length of buffer contents

On exit:
X and Y contain value of count (low byte, high byte)
X and Y are preserved for a purge operation
A is undefined
V and C are preserved

### 6.9.4 Using the buffer vectors

It should be noted that none of the buffer maintenance routines check for valid buffer IDs. Using a buffer ID outside the assigned range will have undefined effects unless specifically intercepted.

None of these vectors are implemented on second processors and so none of the buffer maintenance calls are sent across the Tube. Calls using the buffer vectors should always be made by code resident in the I/O processor. It should be noted that considerable manipulation of the buffers may be carried out using OS routines such as OSBYTE, OSWRCH, OSWORD etc. which may affect buffer contents either directly or indirectly. Routines intercepting these vectors must always be resident on the I/O processor, ideally in service type paged ROMs.

The program below illustrates how the buffer vectors can be intercepted to implement a much larger printer buffer. The standard printer buffer is less than &100 bytes long and since printers as a rule tend to be quite sluggish peripherals this buffer rapidly fills up. A buffer is required which will hold a reasonable sized listing, or a document before filling up and refusing to accept further input. Having placed the item for printing in an enlarged buffer the user may return to word processing or programming leaving the operating system to get on with the printing.

The routine used below creates a buffer of variable size as defined by the variable ‘size’. The usefulness of this program is limited. For the reasons given above it will only work when run on a non-Tube mahine. It will only work as long as its code is not corrupted; this means that renumbering the program after it has been run will crash the machine as BASIC tramples all over the area originally reserved for the assembled code. Similarly another language ROM is unlikely to allow the routine to run in peace. If this routine becomes corrupted the machine is totally disabled because each time a key is pressed this routine is called. Experimenting with this example will provide valuable experience in the use of critical operating system routines. One note of warning however, be sure to save a copy of the program before trying to run it; it is quite possible for the program to corrupt itself or even crash the machine irrevocably so that a power on reset is required (that is, the machine will have to be turned off, then on again).

This program consists of three main routines which intercept the buffer maintenance calls for the printer buffer. Calls for any of the other buffers are carefully handed on to the original routines pointed to by the contents of the buffer vectors. An area of RAM is reserved for use as a buffer by using a DIM statement. Four bytes of zero page memory are used to house two 16 bit pointers.

One pointer is used as an index for the insertion of values into the buffer and the other pointer is used as an index for the removal of bytes. When a pointer reaches the end of the buffer it is pointed to the beginning again, In this way the two pointers cycle through the buffer space. A full buffer is detected by incrementing the input pointer and comparing it to the output pointer. If the two pointers are equal the buffer is full, the character cannot be inserted; the input pointer is restored. If after the removal of a character the output pointer becomes equal to the input pointer then the buffer is now empty. By using this system the full size of the buffer is always available to contain data.

```
 10 REM user printer buffer routine
 20 MODE7
 30 size=&2000
 40 DIM buffer size
 50 DIM code% &400
 60 INSV=&22A
 70 RMV=&22C
 80 CNPV=&22E
 90 ptrblk=&80: !ptrblk=buffer+buffer*&10000
100 ip_ptr=ptrblk:op_ptr=ptrblk+2
110 FOR I=0 TO 3 STEP 3
120 P%=code%
130 [
140 OPT I
150 .init    LDA INSV             \ make copies of old vector
160          STA ret1             \ contents to pass on calls
170          LDA INSV+1
180          STA ret1+1
190          LDA RMV
200          STA ret2
210          LDA RMV+1
220          STA ret2+1
230          LDA CNPV
240          STA ret3
250          LDA CNPV+1
260          STA ret3+1
270          LDX #ins AND &FF     \ store address of new
280          LDY #ins DIV &100    \ routines in vectors
290          SEI                  \ disable interrupts
300          STX INSV
310          STY INSV+1
320          LDX #rem AND &FF
330          LDY #rem DIV &100
340          STX RMV
350          STY RMV+1
360          LDX #cnp AND &FF
```

```
370          LDY #cnp DIV &100
380          STX CNPV
390          STY CNPV+1
400          CLI                   \ enable interrupts
410          RTS                   \ finished
420 .wrkbt   EQUB 0                \ byte of RAM workspace
430 .ret1    EQUW 0                \ reserve space for vectors
440 .ret2    EQUW 0
450 .ret3    EQUW 0
460 .wrngbf1 PLP:PLA:JMP (ret1)    \restore S & A, call OS
470 \New insert char. into buffer routine
480 .ins     PHA:PHP               \ save A and status register
490          CPX #3                \ is buffer id 3 ?
500          BNE wrngbf1           \ if not pass to old routine
510          PLP                   \ not passing on, tidy stack
520          LDA ip_ptr            \ A=lo byte of input pointer
530          PHA                   \ store on stack
540          LDA ip_ptr+1          \ A=hi byte of input pointer
550          PHA                   \ store on stack
560          LDY #0                \ Y=0 so ip_ptr incremented
570          JSR inc_ptr           \  by the inc_ptr routine
580          JSR compare           \ compare the two pointers
590          BEQ insfail           \ if ptrs equal, buffer full
600          PLA:PLA:PLA           \ don't need ip_ptr copy now
610          STA (ip_ptr),Y        \ A off stack, insrt in bufr
620          CLC                   \ insertion success, C=0
630          RTS                   \ finished
640 .insfail  PLA                  \ buffer was full so must
650          STA ip_ptr+1          \  restore ip_ptr which was
660          PLA                   \  stored on the stack
670          STA ip_ptr
680          PLA
690          SEC                   \ insertion failes so C=a
700          RTS                   \ finished
710 .wrngbf2 PLP:JMP (ret2)        \ restore 5, call OS
720 \New remove char. from buffer routine
730 .rem     PHP                   \ save status register
740          CPX #3                \ is buffer id 3 ?
750          BNE wrngbf2           \ if not use OS routine
760          PLP                   \ restore status register
770          BVS examine           \ V=1, examine not remove
780 .remsr    JSR compare          \ compare i/p and o/p ptrs
790          BEQ empty             \ if the same, buffer empty
800          LDY #2                \ Y=2 so that increment ptr
810          JSR inc_ptr           \ routine inc's op_ptr
820          LDY #0                \ Y=0, for next instruction
830          LDA (op_ptr),Y        \ fetch character from bufr
840          TAY                   \ return it in Y
850          CLC                   \ buffer not empty, C=0
860          RTS                   \ return
```

```
 870 .empty   SEC                  \ buffer empty, C=a
 880          RTS                  \ return
 890 .examine LDA op_ptr           \ examine only, so store a
 900          PHA                  \ copy of the oip pointer
 910          LDA op_ptr+1         \ on the stack to restore
 920          PHA                  \ ptr after fetch
 930          JSR remsr            \ fetch byte from buffer
 940          PLA                  \ restore ptr from stack
 950          STA op_ptr+1         \ (if buffer was empty
 960          PLA                  \ C=1 from fetch call)
 970          STA op_ptr
 980          TYA                  \ examine requires ch, in A
 990          RTS                  \ finished
1000 .wrngbf3 PLP:JMP (ret3)       \ restore 5, call OS
1010 \ New count/purge buffer routine
1020 .cnp     PHP                  \ save status reg. on stack
1030          CPX #3               \ is buffer id 3 ?
1040          BNE wrngbf3          \ if not pass toold subr
1050          PLP                  \ restore status register
1060          PHP                  \ save again
1070          BVS purge            \ if V=1, purge required
1080          BCC len              \ if C=0, amount in buffer
1090          LDA ip_ptr           \ o/w free space request
1100          PHA
1110          LDA ip_ptr+1         \ store ip_ptr on stack
1120          PHA
1130          LDX #0               \ X=0 for use as counter
1140          STX wrkbt            \ wrkbt=0 for hi counter
1150          LDY #0               \ Y=0, so ip_ptr incr'd
1160 .loop1   JSR inc_ptr          \ increment ip_ptr
1170          JSR compare          \ does it equalop_ptr
1180          BEQ finshdl          \ if so count~free space
1190          INX                  \ X=X+1
1200          BNE no_inc           \ if X=0 don't inc wrkbt
1210          INC wrkbt            \ hi byte of count inc'd
1220 .no_inc  JMP loop1            \ loop round again
1230 .finshdl PLA                  \ restore ip_ptr off stack
1240          STA ip_ptr+1
1250          PLA
1260          STA ip_ptr
1270          LDY wrkbt            \ Y=hi byte of free space
1280          PLP                  \ restore status register
1290          RTS                  \ finished
1300 .len      LDA op_ptr          \ store op_ptr on stack
1310          PHA
1320          LDA op_ptr+1
1330          PHA
1340          LDX #0               \ X=0 for use as counter
1350          STX wrkbt            \ wrkbt=0 hi byte of count
1360          LDY #2               \ Y=2 so op_ptr incremented
1370 .loop2   JSR compare          \ are ptrs equal ?
```

```
1380          BEQ finshd2          \ if so buffer empty
1390          JSR inc_ptr          \ increment op_ptr
1400          INX                  \ increment count
1410          BNE no_inc2          \ if X=0 then increment hi
1420          INC wrkbt            \ byte of count
1430 .no_inc2 JMP loop2            \ loop round again
1440 .finshd2 PLA                  \ restore op_ptr off stack
1450          STA op_ptr+1
1460          PLA
1470          STA op_ptr
1480          LDY wrkbt            \ Y=hi byte of length
1490          PLP                  \ restore status register
1500          RTS                  \ finished
1510 .purge   LDA #buffer AND &FF  \ to purge buffer reset
1520          STA ip_ptr           \ oip and i/p ptrs to
1530          STA op_ptr           \ start of buffer
1540          LDA #buffer DIV &100
1550          STA ip_ptr+1
1560          STA op_ptr+1
1570          PLP                  \ restore status register
1580          RTS                  \ return
1590 \ Increment pointer routine. Y=0 op_ptr, Y=2 ip_ptr
1600 .inc_ptr CLC                  \ C=0
1610          LDA ptrblk,Y         \ A=?(ptrblk+Y)
1620          ADC #1               \ A=A+1+C
1630          STA ptrblk,Y         \ ?(ptrblk+Y)=A
1640          LDA ptrblk+1,Y       \ A=?(ptrblk+1+Y)
1650          ADC #0               \ A=A+0+C
1660          STA ptrblk+1,Y       \ ?(ptrblk+1+Y)=A
1670          CMP #(buffer+size) DIV &100 \ hi byte end of bufr
1680          BNE home             \ not end of buffer
1690          LDA ptrblk,Y         \ A=low byte of pointer
1700          CMP #(buffer+size) AND &FF \ end of buffer ?
1710          BNE home
1720          LDA #buffer AND &FF  \ if the end of buffer has
1730          STA ptrblk,Y         \ been reached set pointer
1740          LDA #buffer DIV &100 \ to the beginning again
1750          STA ptrblk+1,Y
1760 .home    RTS                  \ return
1770 \ Compare pointers, if equal Z=1 don't care otherwise
1780 .compare LDA ip_ptr+1
1790          CMP op_ptr+1         \ compare ptr high bytes
1800          BNE return           \ if not equal return
1810          LDA ip_ptr
1820          CMP op_ptr           \ compare pointr low bytes
1830 .return  RTS                  \ return
1840 ]
1850 NEXT
1860 CALL init
```

This program requires the presence of the Plus 1 expansion to be of any use. It could however be modified to replace any of the operating system' s buffers. A paged ROM version of this program can be found in chapter 10.

## 6.10 Unused vectors, IND1V IND2V & IND3V &230

These vectors are reserved by Acorn for future expansion. Software which uses these vectors cannot be guaranteed to be compatible with any future versions of operating system software or other Acorn products.

## 6.11 The default vector table

The BBC microcomputer operating system (version 1.2 onwards) and the Electron operating system contain a table of default values in a block of data. This may be accessed using the following addresses:

&FFB6 – contains the length of the data in bytes
&FFB7 – contains the low byte of the data’s address
&FFB8 – contains the high byte of the data’s address