

# JIFFYDOS

version 6.01

This document is a dump of JiffyDOS version 6.01. The parts concerning JiffyDOS have been commented by Magnus Nyman (magnus.q.nyman@telia.se), also known as Harlekin/FairLight. All new JiffyDOS routines are specified in the text. The comments to other parts of this document are partially rewritten from 'The Bible', Commodore Reference Manual. Some text has been added, to increment the knowledge of the original Commodore routines. Some errors have been corrected, and old routines has been removed. Also included are my own thoughts of what can be improved.

If you find any errors, feel free to contact me on the address above.

//Magnus

Document revision B. 1996-10-03

**New Zeropage addresses in the JiffyDOS system.**

\$26 ; allflag/rsize 2

\$27 ; comsav 2

\$9b ; AKTFLG, flags if jiffyDOS funktionkeys is  
; aktivated. If 0, funktionkeys are enabled.

\$9f ; CJLA, JiffyDOS default filenameumber.

\$A3 ; ldflg/qflag, Used in all disk access and LOAD  
routines.

\$a6 ; TFLAG, temp store of JiffyDOS command number.

\$b0/\$b1 ; KEYPTR, vector to table of JiffyDOS funktion keys.

\$B0 ; sprsav, Saved, then restored by LOAD routine (no  
other usage).

\$B1 ; rassav 1

\$B2 ; regsav 1

\$be ; DRVBYT, JiffyDOS default device number.

### DISK ACCESS

The 64'er access the serial devices through the CIA at \$DD00. The "bits" have the following connection! The JiffyDOS performs it's own timing and handshaking, allowing us to use both the data AND the clock lead to transfer data. This has two advantages. 1) We send/receive two bits at a time. 2) Because of the heavy timing that is done before the bits are sent, we don't need any timing for the next bytes, and we can send 4\*2 bits (ie. the entire byte) without any more timing!

adr \$DD00	56576
bit 7	Serial Bus Data Input
6	Serial Bus Clock Pulse Input
5	Serial Bus Data Output
4	Serial Bus Clock Pulse Output
3	Serial Bus ATN Signal Output

Bits used

; E000

```
    sta a56
    jsr ebc0f
    lda a61
    cmp #$88
    bcc ie00e
ie00b jsr ebad4
ie00e jsr ebccc
    lda a07
    clc
    adc #$81
    beq ie00b
    sec
    sbc #$01
    pha
    ldx #$05
ie01e lda f69,x
    ldy f61,x
    sta f61,x
    sty f69,x
    dex
    bpl ie01e
    lda a56
    sta a70
    jsr eb853
    jsr ebf4
    lda #$c4
    ldy #$bf
    jsr ie059
    lda #$00
    sta a6f
    pla
    jsr ebab9
    rts
ie043 sta a71
    sty a72
    jsr ebbca
    lda #$57
    jsr eba28
    jsr ie05d
    lda #$57
    ldy #$00
    jmp eba28
ie059 sta a71
    sty a72
ie05d jsr ebbc7
    lda (p71),y
    sta a67
    ldy a71
    iny
    tya
    bne ie06c
    inc a72
ie06c sta a71
    ldy a72
ie070 jsr eba28
    lda a71
    ldy a72
    clc
```

```

        adc #$05
        bcc ie07d
        iny
ie07d   sta a71
        sty a72
        jsr eb867
        lda #$5c
        ldy #$00
        dec a67
        bne ie070
        rts
        tya
        and f44,x
        ror $
        brk
        pla
        plp
        lda (p46),y
        brk
        jsr ebc2b
        bmi ie0d3
        bne ie0be
        jsr efff3
        stx a22
        sty a23
        ldy #$04
        lda (p22),y
        sta a62
        iny
        lda (p22),y
        sta a64
        ldy #$08
        lda (p22),y
        sta a63
        iny
        lda (p22),y
        sta a65
        jmp ie0e3
ie0be   lda #$8b
        ldy #$00
        jsr ebba2
        lda #$8d
        ldy #$e0
        jsr eba28
        lda #$92
        ldy #$e0
        jsr eb867
ie0d3   ldx a65
        lda a62
        sta a65
        stx a62
        ldx a63
        lda a64
        sta a63
        stx a64
ie0e3   lda #$00
        sta a66
        lda a61
        sta a70
        lda #$80
        sta a61

```

```

        jsr eb8d7
        ldx #$8b
        ldy #$00
ie0f6  jmp ebbd4

```

#### **E0F9 BIOERR: HANDLE I/O ERROR IN BASIC**

This routine is called whenever BASIC wishes to call one of the KERNAL I/O routines. It is also used to handle I/O errors in BASIC.

```

.e0f9  cmp #$f0          ; test error
        bne $e104
        sty $38        ; MEMSIZ, highest address in BASIC
        stx $37
        jmp $a663      ; do CLR without aborting I/O
.e104  tax             ; put error flag i (X)
        bne $e109      ; if error code $00, then set error code $1e
        ldx #$1e
.e109  jmp $a437      ; do error

```

#### **E10C BCHOUT: OUTPUT CHARACTER**

This routine uses the KERNAL routine CHROUT to output the character in (A) to an available output channel. A test is made for a possible I/O error.

```

.e10c  jsr $ffd2      ; output character in (A)
        bcs $e0f9      ; if carry set, handle I/O error
        rts           ; else return

```

#### **E112 BCHIN: INPUT CHARACTER**

This routine uses the KERNAL routine CHRIN to input a character to (A) from an available input channel. A test is made for a possible I/O error.

```

.e112  jsr $ffc6      ; input character from CHRIN
        bcs $e0f9      ; if carry set, handle I/O error
        rts           ; else return

```

#### **E118 BCKOUT: SET UP FOR OUTPUT**

This routine uses the KERNAL routine CHKOUT to open an output channel, and tests for possible I/O error. On entry (X) must hold the the logical file number as used in OPEN.

```

.e118  jsr $e4ad      ; open output channel via CHKOUT
        bcs $e0f9      ; if carry set, handle I/O error
        rts           ; else return

```

#### **E11E BCKIN: SET UP FOR INPUT**

This routine uses the KERNAL routine CHKIN to open an input channel. A test as made for possible I/O error.

```

.e11e  jsr $ffc6      ; open input channel via CHKIN
        bcs $e0f9      ; if carry set, handle I/O error
        rts           ; else return

```

#### **E124 BGETIN: GET ONT CHARACTER**

This routine uses the KERNAL routine GETIN to get a character from the keyboard buffer into (A). A test is made for possible I/O error.

```

.e124  jsr $ffe4      ; GETIN, get character from keyboard buffer
        bcs $e0f9      ; if carry set, handle I/O error
        rts           ; else return

```

#### **E12A SYS: PERFORM SYS**

This routine enables machine language routines to be executed from BASIC. The routine evaluates the address and confirms that it is a numeric number. The return address is set up, and the user routine is executed.

```
.e12a  jsr $ad8a          ; evaluate text & confirm numeric
      jsr $b7f7        ; convert fac#1 to integer in LINNUM
      lda #$e1         ; set return address on stack to $ea46
      pha
      lda #$46
      pha
      lda $030f        ; SPREG, user flag register
      pha
      lda $030c        ; SAREG, user (A) register
      ldx $030d        ; SXREG, user (X) register
      ldy $030e        ; SYREG, user (Y) register
      plp
      jmp ($14)        ; execute user routine, exit with rts
.e146  php
      sta $030c        ; store in SAREG, user (A) register
      stx $030d        ; store in SXREG, user (X) register
      sty $030e        ; store in SYREG, user (Y) register
      pla
      sta $030f        ; store in SPREG, user flag register
      rts              ; back
```

#### **E156 SAVET: PERFORM SAVE**

This routine is sets parameters for save, and calls the save routine. The start and end addresses are obtained from TXTTAB and VARTAB. Finally, a test is made if any errors ocured.

```
      jsr $e1d4        ; get SAVE paramerters from text
      ldx $2d          ; VARTAB, start of variables
      ldy $2e
      lda #$2b         ; <TXTTAB, start of BASIC text
      jsr $ffd8        ; execute SAVE
      bcs $e0f9        ; if carry is set, handle I/O errors
      rts
```

#### **E165 VERFYT: PERFORM LOAD/SAVE**

This routine is essentially the same for both LOAD and VERIFY. The entry point determins which is performed, by setting VERCK accordingly. The LOAD/VERIFY parameters, filename, device etc. are obtained from text before the KERNAL routine LOAD is called. A test is made for I/O errors. At this point, the two functios are distiguished. VERIFY reads the the status word and prints the message OK or ?VERIFY error depending on the result of the test. LOAD reads the I/O status word for a possible ?LOAD error, then updates the pointers to text and variables, exiting via CLR.

```
.e165  lda #$01          ; flag verify
      bit $00a9        ; mask, will execute lda #$01 if address $e168
      sta $0a          ; store in VRECK, LOAD/VERIFY flag
      jsr $e1d4        ; get LOAD/VERIFY parameters from text
      lda $0a          ; get VRECK
      ldx $2b          ; TXTTAB, start of BASIC
      ldy $2c
      jsr $ffd5        ; execute LOAD, KERNAL routine
      bcs $e1d1        ; if carry set, handle error
      lda $0a          ; test VRECK for LOAD or VERIFY
      beq $e195        ; do LOAD
      ldx #$1c         ; set error $1c, VERIFY error
      jsr $ffb7        ; do READST, get status I/O word
```

```

        and #$10           ; %00010000, test for mismatch
        bne $e19e         ; data mismatch, do error
        lda $7a           ; <TXTPTR
        cmp #$02
        beq $e194
        lda #$64           ; set address to text OK
        ldy #$a3           ; at $a364
        jmp $able         ; output string in (A/Y)
.e194   rts
.e195   jsr $ffb7         ; do READST, get status I/O for LOAD
        and #$bf           ; %10111111, test all but EOI
        beq $e1a1         ; nope, no errors
        ldx #$1d           ; set error $1d, LOAD error
.e19e   jmp $a437         ; do error
.e1a1   lda $7b           ; >TXTPTR
        cmp #$02
        bne $e1b5
        stx $2d           ; set VARTAB, start of variables
        sty $2e
        lda #$76           ; set address to text READY
        ldy #$a3           ; at $a376
        jsr $able         ; output string in (A/Y)
        jmp $a52a         ; do CLR and restart BASIC
.e1b5   jsr $a68e         ; reset TXTPTR
        jsr $a533         ; rechain BASIC lines
        jmp $a677         ; do RESTORE and reset OLDTXT

```

#### **E1BE OPENT: PERFORM OPEN**

This routine extracts parameters from text and performs the OPEN routine in KERNAL. A test is made for I/O errors.

```

.e1be   jsr $e219         ; get parameters from text
        jsr $ffc0         ; execute OPEN
        bcs $eld1         ; if carry set, handle error
        rts

```

#### **E1C7 CLOSET: PERFORM CLOSE**

The parameters for CLOSE are obtained from text, and the logical filename placed in (A), The KERNAL routine CLOSE is performed, and a test is made for I/O errors.

```

.e1c7   jsr $e219         ; get parameters from text
        lda $49           ; logical file number
        jsr $ffc3         ; perform CLOSE
        bcc $e194         ; if carry set, handle error, else return
.e1d1   jmp $e0f9         ; jump to error routine

```

#### **E1D4 SLPARA: GET PARAMETERS FOR LOAD/SAVE**

This routine gets the filename, devicenumber and secondary address for LOAD/VERIFY and SAVE operations. The KERNAL routines SETNAM and SETLFS are used to do this. Default parameters are set up, and a new JiffyDOS routine is called at \$eldd. It jumps to \$f73a where the original SETLFS is performed, but also makes a test to find the first serial device number, and pokes it into FA. Then tests are made if any of the parameters were given. If so, these are set up as wanted.

```

.e1d4   lda #$00           ; clear length of filename
        jsr $ffbd         ; SETNAM
        ldx #$01         ; default FA, device number is #01
        ldy #$00         ; default SA, secondary address is #00
.e1dd   jsr $f73a         ; SETLFS, and device number in new JiffyDOS routine

```

```

    jsr $e206      ; test if "end of line", if so end here
    jsr $e257      ; set up given filename and perform SETNAM
    jsr $e206      ; test if "end of line", if so end here
    jsr $e200      ; check for comma, and input one byte, FA, to (X)
    ldy #$00
    stx $49
    jsr $ffba      ; perform new SETLFS with device number
    jsr $e206      ; test if "end of line", if so end here
    jsr $e200      ; check for comma, and input one byte, SA, to (X)
    txa            ; transfer (X) to (Y)
    tay
    ldx $49        ; get FA
    jmp $ffba      ; perform SETLFS with both device number and
secondary        address. Then exit

```

#### **E200 COMBYT: GET NEXT ONE-BYTE PARAMETER**

This routine checks if the next character of text is a comma, and then inputs the parameter following into (X).

```

.e200  jsr $e20e      ; check for comma
        jmp $b79e     ; input one byte parameter to (X)

```

#### **E206 DEFLT: CHECK DEFAULT PARAMETERS**

This routine tests CHRGOT to see if a optional parameter was included in the text. If it was, a normal exit is performed via RTS. If not, the return address on the stack is discarded, and the routine exits both this and the calling routine.

```

.e206  jsr $79        ; get CHRGOT
        bne $e20d     ; if last character is a character, do normal exit
        pla           ; else, remove return address
        pla           ; to exit this AND the calling routine.
.e20d  rts           ; exit

```

#### **E20E CMMERR: CHECK FOR COMMA**

This routine confirms that the next character in the text is a comma. It also test that the comma is not immediately followed by a terminator. If so, exit and do SYNTAX error.

```

.e20e  jsr $aefd      ; confirm comma
.e211  jsr $79        ; get CHRGOT
        bne $e20d     ; else than null
        jmp $af08     ; execute SYNTAX error

```

#### **E219 OCPARA: GET PARAMETERS FOR OPEN/CLOSE**

This routine gets the logical file number, device number, secondary address and filename for OPEN/CLOSE. Initially the default filename is set to null, and the device number to #1. The logical filenumber is compulsory, and is obtained from text and placed in <FORPNT. The other parameters are optinal and are obtained if present. The device number is stored in >FORPNT. The parameters are set via the KERNAL routines SETNAM and SETLFS.

```

.e219  lda #$00       ; default filename is null
        jsr $ffbd     ; SETNAM
        jsr $e211     ; confirm TXTPNT is no terminator, if so - error
        jsr $b79e     ; input one byte character to (X)
        stx $49       ; store logical filenumber in <FORPNT
        txa           ; set default parameters to
        ldx #$01      ; device = #1
        ldy #$00      ; secondary address = #0
        jsr $ffba     ; SETLFS

```

```

        jsr $e206      ; test if "end of line", if so end here
        jsr $e200      ; check for comma, and input FA, device number
        stx $4a        ; store in >FORPNT
        ldy #$00       ; secondary address = #0
        lda $49        ; logical file number from temp store
        cpx #$03       ; test if serial device
        bcc $e23f      ; nope
        dey           ; if serial, set secondary address to $ff
.e23f   jsr $ffba      ; SETLFS
        jsr $e206      ; test if "end of line", if so end here
        jsr $e200      ; check for comma, and input SA, secondary address
        txa
        tay           ; SA to (Y)
        ldx $4a        ; FA
        lda $49        ; LA
        jsr $ffba      ; SETLFS
        jsr $e206      ; test if "end of line", if so end here
        jsr $e20e      ; check for comma only
.e257   jsr $ad9e      ; evaluate expression in text
        jsr $b6a3      ; do string housekeeping
        ldx $22        ; pointers to given filename
        ldy $23
        jmp $ffbd      ; SETNAM and exit

```

#### **E264 COS: PERFORM COS**

This routine manipulates the input COS to be calculated with SIN.  $\text{COS}(X) = \text{SIN}(X+\pi/2)$ , where  $X$  is in radians. We use it as  $\text{Fac}\#1 = \text{SIN}(\text{fac}\#1 + \pi/2)$ , ie  $\pi/2$  is added to  $\text{fac}\#1$  and the following SIN is performed.

```

        lda #$e0       ; set address to pi/2
        ldy #$e2       ; at $e2e0
        jsr $b867      ; add fltp at (A/Y) to fac#1

```

#### **E26B SIN: PERFORM SIN**

```

ie26b   jsr ebc0c
        lda #$e5
        ldy #$e2
        ldx a6e
        jsr ebb07
        jsr ebc0c
        jsr ebccc
        lda #$00
        sta a6f
        jsr eb853
        lda #$ea
        ldy #$e2
        jsr eb850
        lda a66
        pha
        bpl ie29d
        jsr eb849
        lda a66
        bmi ie2a0
        lda a12
        eor #$ff
        sta a12
ie29d   jsr efbf4
ie2a0   lda #$ea
        ldy #$e2
        jsr eb867

```

```

        pla
        bpl ie2ad
        jsr ebb4
ie2ad   lda #$ef
        ldy #$e2
        jmp ie043

```

#### **E2B4 TAN: PERFORM TAN**

```

.e2b4   jsr ebbca
        lda #$00
        sta a12
        jsr ie26b
        ldx #$4e
        ldy #$00
        jsr ie0f6
        lda #$57
        ldy #$00
        jsr ebba2
        lda #$00
        sta a66
        lda a12
        jsr ie2dc
        lda #$4e
        ldy #$00
        jmp ebb0f
ie2dc   pha
        jmp ie29d

```

#### **E2E0 PI2: TABLE OF TRIGONOMETRY CONSTANTS**

The following constants are held in 5 byte flpt for trigonometry evaluation.

```

.e2e0   81 49 0f da a2   ; 1.570796327 (pi/2)
.e2e5   83 49 0f da a2   ; 6.28318531 (pi*2)
.e2ea   7f 00 00 00 00   ; 0.25
.e2ef   05                ; 5 (one byte counter for SIN series)
.e2f0   84 e6 1a 2d 1b   ; -14.3813907 (SIN constant 1)
.e2f5   86 28 07 fb f8   ; 42.0077971 (SIN constant 2)
.e2fa   87 99 68 89 01   ; -76.7041703 (SIN constant 3)
.e2ff   87 23 35 df e1   ; 81.6052237 (SIN constant 4)
.e304   86 a5 5d e7 28   ; -41.3417021 (SIN constant 5)
.e309   83 49 0f ds a2   ; 6.28318531 (SIN constant 6, pi*2)

```

#### **E30E ATN: PERFORM ATN**

```

.e30e   lda $66
        pha
        bpl ie316
        jsr ebb4
ie316   lda a61
        pha
        cmp #$81
        bcc ie324
        lda #$bc
        ldy #$b9
        jsr ebb0f
ie324   lda #$3e
        ldy #$e3
        jsr ie043
        pla

```

```

        cmp #$81
        bcc ie337
        lda #$e0
        ldy #$e2
        jsr eb850
ie337   pla
        bpl ie33d
        jmp efb4
ie33d   rts

```

### **E33E ATNCON: TABLE OF ATN CONSTANTS**

The table holds a 1 byte counter and the following 5 byte flpt constants.

```

.e33e   0b                ; 13 (one byte counter for ATN series)
.e33f   76 b3 83 bd d3   ; -0.000684793912 (ATN constant 1)
.e344   79 1e f4 a6 f5   ; 0.00485094216 (ATN constant 2)
.e349   7b 83 fc b0 10   ; -0.161117018 (ATN constant 3)
.e34e   7c 0c 1f 67 ca   ; 0.034209638 (ATN constant 5)
.e353   7c de 53 cb c1   ; -0.0542791328 (ATN constant 6)
.e358   7d 14 64 70 4c   ; 0.0724571965 (ATN constant 7)
.e35d   7d b7 ea 51 7a   ; -0.0898023954 (ATN constant 8)
.e362   7d 63 30 88 7e   ; 0.110932413 (ATN constant 9)
.e367   7e 92 44 99 3a   ; -0.14283908 (ATN constant 10)
.e36c   7e 4c cc 91 c7   ; 0.19999912 (ATN constant 11)
.e371   7f aa aa aa 13   ; -0.333333316 (ATN constant 12)
.e376   81 00 00 00 00   ; 1 (ATN constant 13)

```

### **E37B BASSFT: BASIC WARM START**

This is the BASIC warm start routine that is vectored at the very start of the BASIC ROM. The routine is called by the 6510 BRK instruction, or STOP/RESTORE being pressed. It outputs the READY prompt via the IERROR vector at \$0300. The original IERROR vector points to \$e38b, but JiffyDOS uses the error routine as an input to check new commands. If the error code, in (X) is larger than \$80, then only the READY text will be displayed.

```

.e37b   jsr $ffcc        ; CLRCHN, close all I/O channels
        lda #$00
        sta $13         ; input prompt flag
        jsr $a67a       ; do CLR
        cli             ; enable IRQ
.e386   ldx #$80        ; error code #$80
        jmp ($0300)     ; perform error, JiffyDOS at $f763
.e38b   txa             ; error number
        bmi $e391       ; larger than $80
        jmp $a43a       ; nope, print error
.e391   jmp $a474       ; print READY

```

### **E394 INIT: BASIC COLD START**

This is the BASIC cold start routine that is vectored at the very start of the BASIC ROM. BASIC vectors and variables are set up, and power-up message is output, and BASIC is restarted.

```

.e394   jsr $e4b7       ; Init JiffyDOS commands & funktionkeys
        jsr $e3bf       ; Initialize BASIC
        jsr $e422       ; output power-up message
        ldx #$fb       ; reset stack
        txs
        bne $e386       ; output READY, and restart BASIC

```

### **E3A2 INITAT: CHRGET FOR ZEROPAGE**

This is the CHRGET routine which is transferred to RAM starting at \$0073 on power-up or reset.

```
.e3a2  inc $7a          ; .0073  inc $7a          ; increment <TXTPTR
      bne $e3a8      ;          bne $0079       ; skip high byte
      inc $7b        ;          inc $7b        ; increment >TXTPTR
.e3a8  lda $ea60      ; .0079  lda $ea60      ; CHRGET entry, read TXTPTR
      cmp #$3a      ;          cmp #$3a      ; colon (terminator), sets
(Z)
      bcs $e3b9      ;          bcs $008a
      cmp #$20      ;          cmp #$20      ; space, get next character
      beq $e3a2      ;          bne $0073
      sec           ;          sec
      sbc #$30      ;          sbc #$30      ; zero
      sec           ;          sec
      sbc #$d0      ;          sbc #$d0
.e3b9  rts           ; .008a  rts
```

#### **E3BA RNDSED: RANDOM SEED FOR ZEROPAGE**

This is the initial value of the seed for the random number function. It is copied into RAM from \$008b-\$008f. Its fltp value is 0.811635157.

```
.e3ba  80 4f c7 52 58
```

#### **E3BF INITCZ: INITIALISE BASIC RAM**

This routine sets the USR jump instruction to point to ?ILLIGAL QUANTITY error, sets ADRAY1 and ADRAY2, copies CHRGET and RNDSED to zeropage, sets up the start and end locations for BASIC text and sets the first text byte to zero.

```
.e3bf  lda #$4c      ; opcode for JMP
      sta $54      ; store in JMPER
      sta $0310    ; USRPOK, set USR JMP instruction
      lda #$48
      ldy #$b2     ; vector to $b248, ?ILLIGAL QUANTITY
      sta $0311
      sty $0312    ; store in USRADD
      lda #$91
      ldy #$b3     ; vector to $b391
      sta $05
      sty $06     ; store in ADRAY2
      lda #$aa
      ldy #$b1     ; vector to $b1aa
      sta $03
      sty $04     ; store in ADRAY1
      ldx #$1c    ; copy the CHRGET routine and RNDSED to RAM
.e3e2  lda $e3a2,x   ; source address
      sta $73,x   ; destination address
      dex        ; next byte
      bpl $e3e2   ; till ready
      lda #$03
      sta $53     ; store #3 in FOUR6, garbage collection
      lda #$00
      sta $68     ; init BITS, fac#1 overflow
      sta $13     ; init input prompt flag
      sta $18     ; init LASTPT
      ldx #$01
      stx $01fd
      stx $01fc
      ldx #$19
      stx $16     ; TEMPPT, pointer to descriptor stack
```

```

        sec                ; set carry to indicate read mode
        jsr $ff9c          ; read MEMBOT
        stx $2b            ; set TXTTAB, bottom of RAM
        sty $2c
        sec                ; set carry to indicate read mode
        jsr $ff99          ; read MEMTOP
        stx $37            ; set MEMSIZ, top of RAM
        sty $38
        stx $33            ; set FRETOP = MEMTOP
        sty $34
        ldy #$00
        tya
        sta ($2b),y        ; store zero at start of BASIC
        inc $2b            ; increment TXTTAB to next memory position
        bne $e421          ; skip msb
        inc $2c
.e421  rts                ; return

```

#### **E422 INITMS: OUTPUT POWER-UP MESSAGE**

This routine outputs the startup message. It then calculates the number of BASIC bytes free by subtracting the TXTTAB from MEMSIZ, and outputs this number. The routine exits via NEW.

```

.e422  lda $2b            ; read TXTTAB, start of BASIC
        ldy $2c
        jsr $a408          ; check for memory overlap
        lda #$73           ; $e473, startup message
        ldy #$e4
        jsr $able          ; output (A/Y)
        lda $37            ; MEMSIZ, highest address in BASIC
        sec                ; prepare for subtract
        sbc $2b            ; subtract TXTTAB
        tax                ; move to (X)
        lda $38            ; and highbyte
        sbc $2c
        jsr $bdcd          ; output number in (A/X)
        lda #$60           ; $e460
        ldy #$e4           ; pointer to 'BASIC BYTES FREE'
        jsr $able          ; output (A/Y)
        jmp $a644          ; perform NEW

```

#### **E447 JIFFYDOS VECTORS**

This table contains jump vectors that are transferred to \$0300-\$030b. Some vectors are standard Commodore, but some are modified for JiffyDOS.

```

.e447  63 f7                ; IERROR VEC, print basic error message ($f763)
        ; Original IERROR VEC points to $e38b
.e449  83 4a                ; IMAIN VECTOR, basic warm start ($a483)
.e44b  64 ea                ; ICRNCH VECTOR, tokenise basic text ($ea64)
        ; Original ICRNCH VECTOR points to $a57c
.e44d  a1 a7                ; IQPLOP VECTOR, list basic text ($a7a1)
.e44f  e4 a7                ; IGONE VEXTOR, basic character dispatch ($a7ea)
.e451  86 ea                ; IEVAL VECTOR, evaluate basic token ($ae86)

```

#### **E453 INIT JIFFYDOS COMMANDS**

This routine transfers the vectors \$0300-\$030b to set up the JiffyDOS commands.

```

.e453  ldx #$0b            ; 6 vectors to be copied
.e455  lda $e447,x
        sta $0300,x

```

```

dex                ; next byte
bpl $e455          ; ready
rts                ; return

```

#### **E45F WORDS: POWER UP MESSAGE**

This is the power up message displayed on the screen when the 'Commie' is switched on or reset. The strings are separated by a zero byte.

```

jiffydos v6.01 (c)1989 cmd
c-64 basic v2  xxxxx basic bytes free

```

```

.e45f 00 20 42 41 53 49 43 20 ; basic
.e467 42 59 54 45 53 20 46 52 ; bytes fr
.e46f 45 45 0d 00 93 0d 20 20 ; ee
.e477 20 20 20 20 20 4a 49 46 ; jif
.e47f 46 59 44 4f 53 20 56 36 ; fydos v6
.e487 2e 30 31 20 28 43 29 31 ; .01 (c)1
.e48f 39 38 39 20 43 4d 44 20 ; 989 cmd
.e497 20 0d 0d 20 43 2d 36 34 ; c-64
.e49f 20 42 41 53 49 43 20 56 ; basic v
.e4a7 32 20 20 20 00 81 ; 2

```

#### **E4AD PATCH FOR BASIC CHKOUT CALL**

This is a short patch added for the KERNAL ROM to preserv (A) when there was no error returned from BASIC calling the CHKOUT routine. This corrects a bug in the early versions of PRINT# and CMD.

```

.e4ad pha                ; temp store (A)
      jsr $ffc9          ; CHKOUT
      tax
      pla                ; retrieve (A)
      bcc $e4b6
      txa
.e4b6 rts

```

#### **E4B7 INIT JIFFYDOS COMMANDS AND FUNKTIONKEYS**

This routine initialises the JiffyDOS commands by jumping to \$e453 where the \$0300-vectors are set up. Then it sets up the vectors at \$b0 to point to the funktionkey table at \$f672. The entry at \$e4c2 disables the funktionkeys after a @f command.

```

.e4b7 jsr $e453          ; init JiffyDOS command vectors
      lda #$72           ; Set up JiffyDOS function key vector
      sta $b0           ; to $f672
      lda #$f6
      sta $b1
.e4c2 inx                ; (X)=0
      stx $9b           ; AKTFLT, aktivte/deaktivte funktion keys
      rts

.e4c6 lda #$6f           ; #$6f=command channel
      jsr $f0e4         ; prepare for input
      jsr $ffcf         ; input byte from command channel
      cmp #$35          ; equal to #$35 (#)
      rts

```

```
tax          ; free byte
tax          ; free byte
```

#### **E4D3 RS232 PATCH**

This patch has been added to the RS232 input routine in KERNAL v.3. It initialises the RS232 parity byte, RIPRTY, on reception of a start bit.

```
.e4d3  sta $a9          ; RINONE, check for start bit
        lda #$01
        sta $ab        ; RIPRTY, RS232 input parity
        rts
```

#### **E4DA RESET CHARACTER COLOUR**

This routine is a patch in KERNAL version 3 to fix a bug with the colour code. The routine is called by 'clear a screen line', and sets the character colour to COLOR.

```
.e4da  lda $0286       ; get COLOR
        sta ($f3),y    ; and store in current screen position
        rts
```

#### **E4E0 PAUSE AFTER FINDING TAPE FILE???????????????**

This routine would continue tape loading without pressing C= when a file was found. This could probably be removed, since JiffyDOS not uses tape junk.

```
.e4e0  adc #$02
.e4e2  ldy $91
        iny
        bne $e4eb
        cmp $a1
        bne $e4e2
.e4eb  rts
```

#### **E4EC RS232 TIMING TABLE - PAL**

Timingtable for RS232 NMI for use with PAL machines. This table contains the prescaler values for setting up the RS232 baudrates. The table contains 10 entries which corresponds to one of the fixed RS232 rates, starting with lowest (50 baud) and finishing with the highest (2400 baud). Since the clock frequency is different between NTSC and PAL systems, there is another table for NTSC machines at \$fec2.

```
.e4ec  19 29          ; 50 baud
.e4ee  44 19          ; 75 baud
.e4f0  1a 11          ; 110 baud
.e4f2  e8 0d          ; 134.5 baud
.e4f4  70 0c          ; 150 baud
.e4f6  06 06          ; 300 baud
.e4f8  d1 02          ; 600 baud
.e4fa  37 01          ; 1200 baud
.e4fc  ae 00          ; (1800) 2400 baud
.e4fe  69 00          ; 2400 baud
```

#### **E500 IOBASE: GET I/O ADDRESS**

The KERNAL routine IOBASE (\$fff3) jumps to this routine. It returns the base address \$dc00 in (X/Y)

```
.e500  ldx #$00        ; set (X/Y) to $dc00
        ldy #$dc
        rts
```

### **E505 SCREEN: GET SCREEN SIZE**

The KERNAL routine SCREEN (\$ffed) jumps to this routine. It returns the screen size; columns in (X) and rows in (Y).

```
.e505  ldx #$28          ; 40 columns
        ldy #$19          ; 25 rows
        rts
```

### **E50A PLOT: PUT/GET ROW AND COLUMN**

The KERNAL routine PLOT (\$fff0) jumps to this routine. The option taken depends on the state of carry on entry. If it is set, the column is placed in (Y) and the row placed in (X). If carry is clear, the cursor position is read from (X/Y) and the screen pointers are set.

```
.e50a  bcs $e513        ; if carry set, jump
        stx $d6          ; store TBLX, current row
        sty $d3          ; store PNTR, current column
        jsr $e56c        ; set screen pointers
.e513  ldx $d6          ; read TBLX
        ldy $d3          ; read PNTR
        rts
```

### **E518 CINT1: INITIALISE I/O**

This routine is part of the KERNAL CINT init routine. I/O default values are set, <shift+cbm> keys are disabled, and cursor is switched off. The vector to the keyboard table is set up, and the length of the keyboardbuffer is set to 10 characters. The cursor color is set to lightblue, and the key-repeat parameters are set up.

```
.e518  jsr $e5a0        ; set I/O defaults
        lda #$00
        sta $0291        ; disable <SHIFT + CBM> by writing zero into MODE
        sta $cf          ; the cursor blink flag, set BLNON on
        lda #$48
        sta $028f
        lda #$eb        ; set the KEYLOG vector to point at $eb48
        sta $0290
        lda #$0a        ; set max number of character is keyboard buffer to
10
        sta $0289        ; XMAX
        sta $028c        ; How many 1/60 of a second to wait before key is
repeated. Used together with $028b
        lda #$0e        ; set character colour to light blue
        sta $0286        ; COLOR
        lda #$04        ; How many $028c before a new entry is
        sta $028b        ; put in the keyboard buffer, KOUNT
        lda #$0c
        sta $cd          ; store in BLCNT, cursor toggle timer
        sta $cc          ; store in BLNSW, cursor enable
```

### **E544 CLEAR SCREEN**

This routine sets up the screen line link table (\$d9 - \$f2), LDTB1, which is used to point out the address to the screen. The later part of the routine performs the screen clear, line by line, starting at the bottom line. It continues to the next routine which is used to home the cursor.

```
.e544  lda $0288        ; get HIBASE, top of screen memory
        ora #$80        ; fool around
        tay
        lda #$00
        tax
```

```

.e54d  sty $d9,x      ; store in screen line link table, LDTB1
        clc
        adc #$28     ; add #40 to next line
        bcc $e555
        iny         ; inc page number
.e555  inx           ; next
        cpx #$1a    ; till all 26?? is done
        bne $e54d
        lda #$ff
        sta $d9,x   ; last pointer is $ff
        ldx #$18    ; start clear screen with line $18 (bottom line)
.e560  jsr $e9ff    ; erase line (X)
        dex         ; next
        bpl $e560   ; till screen is empty

```

#### **E566 HOME CURSOR**

This routine puts the cursor in the top left corner by writing its column and line to zero.

```

.e566  ldy #$00
        sty $d3     ; write to PNTR, cursor column
        sty $d6     ; write to TBLX, line number

```

#### **E56C SET SCREEN POINTRES**

This routine positions the cursor on the screen and sets up the screen pointers. On entry, TBLX must hold the line number, and PNTR the column number of the cursor position. A major bug has been removed from the original commodore KERNAL. It sometimes caused the computer to crash, when deleting characters from the bottom line.

```

.e56c  ldx $d6      ; read TBLX
        lda $d3     ; read PNTR
.e570  ldy $d9,x   ; read value from screen line link table, LDTB1
        bmi $e57c   ; heavy calculations??? jump when ready
        clc
        adc #$28
        sta $d3     ; PNTR
        dex
        bpl $e570
.e57c  jsr $e9f0    ; set start of line (X)
        lda #$27
        inx
.e582  ldy $d9,x   ; LDTB1
        bmi $e58c
        clc
        adc #$28
        inx
        bpl $e582
.e58c  sta $d5     ; store in LMNX, physical screen line length
        jmp $ea24   ; sync color pointer
.e591  cpx $c9     ; read LXSP, check cursor at start of input
        beq $e598
        jmp $e6ed   ; retreat cursor
.e598  rts

        nop        ; A free byte!!! (own serial number haha)

```

#### **E59A SET I/O DEFAULTS**

The default output device is set to 3 (screen), and the default input device is set to 0 (keyboard). The VIC chip registers are set from the video chip setup table. The cursor is then set to the home position.

```

.e59a jsr $e5a0      ; set I/O defaults
      jmp $e566      ; home cursor and exit routine
.e5a0 lda #$03
      sta $9a        ; DFLTO, default output device - screen
      lda #$00
      sta $99        ; DFLTN, default input device - keyboard
      ldx #$2f
.e5aa lda $ecb8,x    ; VIC chip setup table
      sta $cfff,x    ; VIC chip I/O registers
      dex            ; next
      bne $e5aa      ; till ready
      rts

```

#### **E5B4 LP2: GET CHARACTER FROM KEYBOARD BUFFER**

It is assumed that there is at least one character in the keyboard buffer. This character is obtained and the rest of the queue is moved up one by one to overwrite it. On exit, the character is in (A).

```

.e5b4 ldy $0277      ; read KEYD, first character in keyboard buffer
queue
      ldx #$00
.e5b9 lda $0278,x    ; overwrite with next in queue
      sta $0277,x
      inx
      cpx $c6        ; compare with NDX, number of characters in queue
      bne $e5b9      ; till all characters are moved
      dec $c6        ; decrement NDX
      tya            ; transfer read character to (A)
      cli            ; enable interrupt
      clc
      rts

```

#### **E5CA INPUT FROM KEYBOARD**

This routine uses the previous routine to get characters from the keyboard buffer. Each character is output to the screen, unless it is <shift/RUN>. If so, the contents of the keyboard buffer is replaced with LOAD <CR> RUN <CR>. The routine ends when a carriage routine is encountered. The JSR at \$e5e7 is a patch in JiffyDOS to test if the F-keys or other valid JiffyDOS keys are pressed. If not, this routine continues as normal.

```

.e5ca jsr $e716      ; output to screen
.e5cd lda $c6        ; read NDX, number of characters in keyboard queue
      sta $cc        ; BLNSW, cursor blink enable
      sta $0292      ; AUTODN, auto scroll down flag
      beq $e5cd      ; loop till key is pressed
      sei            ; disable interrupt
      lda $cf        ; BLNON, last cursor blink (on/off)
      beq $e5e7
      lda $ce        ; GDBLN, character under cursor
      ldx $0287      ; GDCOL, background color under cursor
      ldy #$00
      sty $cf        ; clear BLNON
      jsr $ea13      ; print to screen
.e5e7 jsr $f9e5      ; Get character from keyboard buffer. JiffyDOS fixx
      cmp #$83      ; test if <shift/RUN> is pressed
      bne $e5fe      ; nope
      ldx #$09      ; transfer 'LOAD <CR> RUN <CR>' to keyboard buffer
      sei
      stx $c6        ; store #9 in NDX, characters in buffer
.e5f3 lda $ece6,x    ; 'LOAD <CR> RUN <CR>' message in ROM

```

```

        sta $0276,x      ; store in keyboard buffer
        dex
        bne $e5f3       ; all nine characters
        beq $e5cd       ; allways jump
.e5fe   cmp # $0d        ; carriage return pressed?
        bne $e5ca       ; nope, go to start
        ldy $d5         ; get LNMX, screen line length
        sty $d0         ; CRSV, flag input/get from keyboard
.e606   lda ($d1),y     ; PNT, screen address
        cmp # $20       ; space?
        bne $e60f       ; nope
        dey
        bne $e606       ; next
.e60f   iny
        sty $c8         ; store in INDX, end of logical line for input
        ldy # $00
        sty $0292       ; AUTODN
        sty $d3         ; PNTR, cursor column
        sty $d4         ; QTSW, reset quote mode
        lda $c9         ; LXSP, corsor X/Y position
        bmi $e63a
        ldx $d6         ; TBLX, cursor line number
        jsr $e591       ; retreat cursor
        cpx $c9         ; LXSP
        bne $e63a
        lda $ca
        sta $d3         ; PNTR
        cmp $c8         ; INDX
        bcc $e63a
        bcs $e65d

```

### **E632 INPUT FROM SCREEN OR KEYBOARD**

This routine is used by INPUT to input data from devices not on the serial bus, ie. from screen or keyboard. On entry (X) and (Y) registers are preserved. A test is made to determine which device the input is to be from. If it is the screen, then quotes and <RVS> are tested for and the character is echoed on the screen. Keyboard inputs make use of the previous routine.

```

.e632   tya             ; preserve (X) and (Y) registers
        pha
        txa
        pha
        lda $d0        ; CRSW, INPUT/GET from keyboard or screen
        beq $e5cd       ; input from keyboard
.e63a   ldy $d3         ; PNTR, cursor column
        lda ($d1),y     ; read from current screen address
        sta $d7         ; temp store
        and # $3f
        asl $d7
        bit $d7
        bpl $e64a
        ora # $80
.e64a   bcc $e650
        ldx $d4         ; QTSW, editor in quotes mode
        bne $e654       ; yepp
.e650   bvs $e654
        ora # $40
.e654   inc $d3         ; PNTR
        jsr $e684       ; do quotes test
        cpy $c8         ; INDX, end of logical line for input

```

```

        bne $e674
.e65d  lda #$00
        sta $d0          ; CRSW
        lda #$0d
        ldx $99          ; DFLTN, default input device
        cpx #$03          ; screen
        beq $e66f          ; yes
        ldx $9a          ; DFLTO, default output device
        cpx #$03          ; screen
        beq $e672          ; yes
.e66f  jsr $e716          ; output to screen
.e672  lda #$0d
.e674  sta $d7
        pla
        tax              ; restore (X) and (Y) registers
        pla
        tay
        lda $d7
        cmp #$de
        bne $e682
        lda #$ff
.e682  clc
        rts

```

#### **E684 QUOTES TSET**

On entry, (A) holds the character to be tested. If (A) holds ASCII quotes, then the quotes flag is toggled.

```

.e684  cmp #$22          ; ASCII quotes (")
        bne $e690          ; nope, return
        lda $d4          ; QTSW, quotes mode flag
        eor #$01          ; toggle on/off
        sta $d4          ; store
        lda #$22          ; restore (A) to #$22
.e690  rts

```

#### **E691 SET UP SCREEN PRINT**

The RVS flag is tested to see if reversed characters are to be printed. If insert mode is on, the insert counter is decremented by one. When in insert mode, all characters will be displayed, ie. DEL RVS etc. The character colour is placed in (X) and the character is printed to the screen and the cursor advanced.

```

.e691  ora #$40
.e693  ldx $c7          ; test RVS, flag for reversed characters
        beq $e699          ; nope
.e697  ora #$80          ; set bit 7 to reverse character
.e699  ldx $d8          ; test INSRT, flag for insert mode
        beq $e69f          ; nope
        dec $d8          ; decrement number of characters left to insert
.e69f  ldx $0286          ; get COLOR, current character colour code
        jsr $ea13          ; print to screen
        jsr $e6b6          ; advance cursor
.e6a8  pla
        tay
        lda $d8          ; INSRT
        beq $e6b0
        lsr $d4
.e6b0  pla
        tax
        pla

```

```

    clc
    cli
    rts

```

#### **E6B6 ADVANCE CURSOR**

The cursor is advanced one position on the screen. If this puts it beyond the 40th column, then it is placed at the beginning of the next line. If the length of that line is less than 80, then this new line is linked to the previous one. A space is opened if data already exists on the new line. If the cursor has reached the bottom of the screen, then the screen is scrolled down.

```

.e6b6  jsr $e8b3          ; check line increment
        inc $d3          ; increment PNTR, cursor column on current line
        lda $d5          ; LNMX, physical screen line length
        cmp $d3          ; compare to PNTR
        bcs $e700        ; not beyond end of line, exit
        cmp #$4f         ; $4f = 79
        beq $e6f7        ; put cursor on new logical line
        lda $0292        ; AUTODN, auto scroll down flag
        beq $e6cd        ; auto scroll is on
        jmp $e967        ; open a space on the screen
.e6cd  ldx $d6           ; read TBLX, current line number
        cpx #$19         ; $19 = 25
        bcc $e6da        ; less than 25
        jsr $e8ea        ; scroll down
        dec $d6          ; place cursor on line 24
        ldx $d6
.e6da  asl $d9,x         ; clear bit7 in LDTB1 to indicate that it is line 2
        lsr $d9,x         ; in the logical line
        inx              ; next line
        lda $d9,x         ; set bit7 in LDTB1 to indicate that it is line 1
        ora #$80          ; in the logical line
        sta $d9,x
        dex
        lda $d5           ; add $28 (40) to LNMX to allow 80 characters
        clc              ; on the logical line
        adc #$28
        sta $d5

```

#### **E6ED RETREAT CURSOR**

The screen line link table is searched, and then the start of line is set. The rest of the routine sets the cursor onto the next line for the previous routine.

```

.e6ed  lda $d9,x         ; LDTB1, screen line link table
        bmi $e6f4        ; test bit7
        dex              ; next line
        bne $e6ed        ; till all are done
.e6f4  jmp $e9f0        ; set start of line
.e6f7  dec $d6          ; decrement TBLX, cursor line
        jsr $e87c        ; goto next line
        lda #$00
        sta $d3          ; set PNTR, the cursor column, to zero
.e700  rts

```

#### **E701 BACK ON TO PREVIOUS LINE**

This routine is called when using <DEL> and <cursor LEFT>. The line number is tested, and if the cursor is already on the top line, then no further action is taken. The screen pointers are set up and the cursor placed at the end of the previous line.

```

.e701  ldx $d6          ; test TBLX, physical line number
       bne $e70b      ; if not on top line, branch
       stx $d3        ; set PNTR to zero as well
       pla
       pla
       bne $e6a8      ; always jump
.e70b  dex             ; decrement TBLX
       stx $d6        ; and store
       jsr $e56c      ; set screen pointers
       ldy $d5        ; get LNMX
       sty $d3        ; and store in PNTR
       rts

```

### **E716 OUTPUT TO SCREEN**

This routine is part of the main KERNAL CHROUT routine. It prints CBM ASCII characters to the screen and takes care of all the screen editing characters. The cursor is automatically updated and scrolling occurs if necessary. On entry, (A) must hold the character to be output. On entry all registers are stored on the stack. For convinience, the routine is split into sections showing the processing of both shifted and unshifted character.

```

.e716  pha           ; store (A), (X) and (Y) on stack
       sta $d7       ; temp store
       txa
       pha
       tya
       pha
       lda #$00
       sta $d0       ; store in CRSW
       ldy $d3       ; PNTR, cursor positions on line
       lda $d7       ; retrieve from temp store
       bpl $e72a     ; do unshifted characters
       jmp $e7d4     ; do shifted characters

```

UNSHIFTED CHARACTERS. Ordinary unshifted ASCII characters and PET graphics are output directly to the screen. The following control codes are trapped and precessed: <RETURN>, <DEL>, <CRSR RIGHT>, <CRSR DOWN>. If either insert mode is on or quotes are open (except for <DEL>) then the control characters are not processed, but output as reversed ASCII literals.

```

.e72a  cmp #$0d      ; <RETURN>?
       bne $e731     ; nope
       jmp $e891     ; execute return
.e731  cmp #$20      ; <SPACE>?
       bcc $e745
       cmp #$60      ; #$60, first PET graphic character?
       bcc $e73d
       and #$df      ; %11011111
       bne $e73f
.e73d  and #$3f      ; %00111111
.e73f  jsr $e684     ; do quotes test
       jmp $e693     ; setup screen print
.e745  ldx $d8       ; INSRT, insert mode flag
       beq $e74c     ; mode not set
       jmp $e697     ; output reversed charcter
.e74c  cmp #$14      ; <DEL>?
       bne $e77e     ; nope
       tya           ; (Y) holds cursor column
       bne $e759     ; not start of line

```

```

        jsr $e701          ; back on previous line
        jmp $e773
.e759   jsr $e8a1          ; check line decrement
        dey              ; decrement cursor column
        sty $d3          ; and store in PNTR
        jsr $ea24        ; synchronise colour pointer
.e762   iny              ; copy character at cursor position (Y+1) to (Y)
        lda ($d1),y      ; read character
        dey
        sta ($d1),y      ; and store it one position back
        iny
        lda ($f3),y      ; read character colour
        dey
        sta ($f3),y      ; and store it one position back
        iny              ; more characters to move
        cpy $d5          ; compare with LNMX, length of physical screen line
        bne $e762        ; if not equal, move more characters
.e773   lda #$20
        sta ($d1),y      ; store <SPACE> at end of line
        lda $0286        ; COLOR, current character colour
        sta ($f3),y      ; store colour at end of line
        bpl $e7cb        ; always jump
.e77e   ldx $d4          ; QTSW, editor in quotes mode
        beq $e785        ; no
        jmp $e697        ; output reversed character
.e785   cmp #$12         ; <RVS>?
        bne $e78b        ; no
        sta $c7          ; RVS, reversed character output flag
.e78b   cmp #$13         ; <HOME>?
        bne $e792        ; no
        jsr $e566        ; home cursor
.e792   cmp #$1d         ; <CRSR RIGHT>?
        bne $e7ad        ; nope
        iny              ; increment (Y), internal counter for column
        jsr $e8b3        ; check line increment
        sty $d3          ; store (Y) in PNTR
        dey              ; decrement (Y)
        cpy $d5          ; and compare to LNMX
        bcc $e7aa        ; not exceeded line length
        dec $d6          ; TBLX, current physical line number
        jsr $e87c        ; goto next line
        ldy #$00
.e7a8   sty $d3          ; set PNTR to zero, cursor to the left
.e7aa   jmp $e6a8        ; finish screen print
.e7ad   cmp #$11         ; <CRSR DOWN>?
        bne $e7ce        ; no
        clc              ; prepare for add
        tya              ; (Y) holds cursor column
        adc #$28         ; add 40 to next line
        tay              ; to (Y)
        inc $d6          ; increment TBLX, physical line number
        cmp $d5          ; compare to LNMX
        bcc $e7a8        ; finish screen print
        beq $e7a8        ; finish screen print
        dec $d6          ; restore TBLX
.e7c0   sbc #$28
        bcc $e7c8
        sta $d3          ; store PNTR
        bne $e7c0
.e7c8   jsr $e87c        ; go to next line
.e7cb   jmp $e6a8        ; finish screen print

```

```
.e7ce jsr $e8cb      ; set colour code
      jmp $ec44      ; do graphics/text control
```

SHIFTED CHARACTERS. These are dealt with in the following order: Shifted ordinary ASCII and PET graphics characters, <shift RETURN>, <INST>, <CRSR UP>, <RVS OFF>, <CRSR LEFT>, <CLR>. If either insert mode is on, or quotes are open, then the control character is not processed but reversed ASCII literal is printed.

```
.e7d4 and #$7f        ; clear bit7
      cmp #$7f        ; compare to #$7f
      bne $e7dc        ; not equal
      lda #$5e        ; if #$7f, load #$5e
.e7dc cmp #$20        ; ASCII <SPACE>?
      bcc $e7e3
      jmp $e691        ; set up screen print
.e7e3 cmp #$0d        ; <RETURN>?
      bne $e7ea        ; nope
      jmp $e891        ; do return
.e7ea ldx $d4         ; read QTSW
      bne $e82d        ; if quotes mode, jump
      cmp #$14        ; <INST>?
      bne $e829        ; nope
      ldy $d5         ; LNMX
      lda ($d1),y     ; get screen character
      cmp #$20        ; space?
      bne $e7fe        ; nope
      cpy $d3         ; PNTR equal to LNMX
      bne $e805        ; nope
.e7fe cpy #$4f        ; #$4f=79, last character
      beq $e826        ; end of logical line, can not insert
      jsr $e965        ; open space on line
.e805 ldy $d5         ; LNMX
      jsr $ea24        ; synchronise colour pointer
.e80a dey            ; prepare for move
      lda ($d1),y     ; read character at pos (Y)
      iny
      sta ($d1),y     ; and move one step to the right
      dey
      lda ($f3),y     ; read character colour
      iny
      sta ($f3),y     ; move one step to the right
      dey            ; decrement counter
      cpy $d3         ; compare with PNTR
      bne $e80a        ; till all characters right of cursor are moved
      lda #$20        ; <SPACE>, ASCII #$20
      sta ($d1),y     ; store at new character position
      lda $0286        ; COLOR, current character colour
      sta ($f3),y     ; store at new colour position
      inc $d8         ; INSRT FLAG
.e826 jmp $e6a8        ; finish screen print
.e829 ldx $d8         ; INSRT FLAG
      beq $e832        ; insert mode is off
.e82d ora #$40
      jmp $e697        ; set up screen print
.e832 cmp #$11        ; <CRSR UP>?
      bne $e84c        ; nope
      ldx $d6         ; read TBLX
      beq $e871        ; at topline, do nothing
      dec $d6         ; else decrement TBLX
      lda $d3         ; PNTR
```

```

        sec                ; prepare for substract
        sbc #$28          ; back 40 columns for double line
        bcc $e847        ; skip
        sta $d3          ; store PNTR
        bpl $e871        ; finish screen print
.e847   jsr $e56c        ; set screen pointer
        bne $e871        ; finish screen print
.e84c   cmp #$12        ; <RVS OFF>?
        bne $e854        ; nope
        lda #$00
        sta $c7          ; RVS, disable reverse print
.e854   cmp #$1d        ; <CRSR LEFT>?
        bne $e86a        ; nope
        tya              ; (Y) holds cursor column
        beq $e864        ; at first position
        jsr $e8a1        ; check line decrement
        dey              ; one position left
        sty $d3          ; store in PNTR
        jmp $e6a8        ; finish screen print
.e864   jsr $e701        ; back to previous line
        jmp $e6a8        ; finish screen print
.e86a   cmp #$13        ; <CLR>?
        bne $e874        ; nope
        jsr $e544        ; clear screen
.e871   jmp $e6a8        ; finish screen print
.e874   ora #$80
        jsr $e8cb        ; set colour code
        jmp $ec4f        ; set graphics/text mode

```

#### **E87C GO TO NEXT LINE**

The cursor is placed at the start of the next logical screen line. This involves moving down two lines for a linked line. If this places the cursor below the bottom of the screen, then the screen is scrolled.

```

.e87c   lsr $c9          ; LXSP, cursor X-Y position
        ldx $d6          ; TBLX, current line number
.e880   inx              ; next line
        cpx #$19        ; 26th line
        bne $e888        ; nope, scroll is not needed
        jsr $e8ea        ; scroll down
.e888   lda $d9,x        ; test LTDB1, screen line link table if first of two
        bpl $e880        ; yes, jump down another line
        stx $d6          ; store in TBLX
        jmp $e56c        ; set screen pointers

```

#### **E891 OUTPUT <CARRIAGE RETURN>**

All editor modes are swithed off and the cursor placed at the start of the next line.

```

.e891   ldx #$00
        stx $d8          ; INSRT, disable insert mode
        stx $c7          ; RVS, disable reversed mode
        stx $d4          ; QTSW, disable quotes mode
        stx $d3          ; PNTR, put cursor at first column
        jsr $e87c        ; go to next line
        jmp $e6a8        ; finish screen print

```

#### **E8A1 CHECK LINE DECREMENT**

When the cursor is at the beginning of a screen line, if it is moved backwards, this routine places the cursor at the end of the line above. It tests both column 0 and column 40.

```

.e8a1  ldx #$02
       lda #$00
.e8a5  cmp $d3          ; test if PNTR is at the first column
       beq $e8b0      ; yepp
       clc           ; add $28 (40)
       adc #$28      ; to test if cursor is at line two in the logical
line
       dex
       bne $e8a5     ; test two lines
       rts
.e8b0  dec $d6        ; decrement line number
       rts

```

### **E8B3 CHECK LINE INCREMENT**

When the cursor is at the end of the screen, if it is moved forward, this routine places the cursor at the start of the line below.

```

.e8b3  ldx #$02
       lda #$27      ; start by testing position $27 (39)
.e8b7  cmp $d3        ; compare with PNTR
       beq $e8c2     ; brach if equal, and move cursor down
       clc           ; else, add $28 to test next physical line
       adc #$28
       dex           ; two lines to test
       bne $e8b7
       rts           ; return here without moving cursor down
.e8c2  ldx $d6        ; get TBLX
       cpx #$19     ; and test if at the 25th line
       beq $e8ca     ; yepp, return without moving down
       inc $d6      ; increment TBLX
.e8ca  rts

```

### **E8CB SET COLOUR CODE**

This routine is called by the output to screen routine. The Commodore ASCII code in (A) is compared with the ASCII colout code table. If a match is found, then the table offset (and hence the colour value) is stored in COLOR.

```

.e8cb  ldx #$0f      ; 16 values to be tested
.e8cd  cmp $e8da,x   ; compare with colour code table
       beq $e8d6     ; found, jump
       dex           ; next colour in table
       bpl $e8cd     ; till all 16 are tested
       rts           ; if not found, return
.e8d6  stx $0286     ; if found, store code in COLOR
       rts

```

### **E8DA COLOUR CODE TABLE**

This is a table containing 16 Commodore ASCII codes representing the 16 available colours. Thus red is represented as \$1c in the table, and would be obtained by PRINT CHR\$(28), or poke 646,2.

```

.e8da  90           ; color0, black
.e8db  05           ; color1, white
.e8dc  1c           ; color2, red
.e8dd  9f           ; color3, cyan
.e8de  9c           ; color4, purple
.e8df  1e           ; color5, green
.e8e0  1f           ; color6, blue
.e8e1  9e           ; color7, yellow

```

```

.e8e2  81          ; color8, orange
.e8e3  95          ; color9, brown
.e8e4  96          ; colorA, pink
.e8e5  97          ; colorB, grey1
.e8e6  98          ; colorC, grey2
.e8e7  99          ; colorD, light green
.e8e8  9a          ; colorE, light blue
.e8e9  9b          ; colorF, grey3

```

### **E8EA SCROLL SCREEN**

This routine scrolls the screen down by one line. If the top two lines are linked together, then the scroll down is repeated. The screen line link pointers are updated, each screen line is cleared and the line below is moved up. The keyboard is directly read from CIA#1, and the routine tests if <CTRL> is pressed. A JiffyDOS feature is the <CTRL S> option, which freezes the scroll till another key is pressed.

```

.e8ea  lda $ac          ; temp store SAL on stack
      pha
      lda $ad
      pha
      lda $ae          ; temp store EAL on stack
      pha
      lda $af
      pha
.e8f6  ldx #$ff
      dec $d6          ; decrement TBLX
      dec $c9          ; decrement LXSP
      dec $02a5        ; temp store for line index
.e8ff  inx
      jsr $e9f0        ; set start of line (X)
      cpx #$18
      bcs $e913
      lda $ecf1,x      ; read low-byte screen addresses
      sta $ac
      lda $da,x
      jsr $e9c8        ; move a screen line
      bmi $e8ff
.e913  jsr $e9ff        ; clear a screen line
      ldx #$00
.e918  lda $d9,x        ; calculate new screen line link table
      and #$7f        ; clear bit7
      ldy $da,x
      bpl $e922
      ora #$80        ; set bit7
.e922  sta $d9,x        ; store new value in table
      inx            ; next line
      cpx #$18        ; till all 25 are done
      bne $e918
      lda $f1          ; bottom line link
      ora #$80        ; unlink it
      sta $f1          ; and store back
      lda $d9          ; test top line link
      bpl $e8f6        ; line is linked, scroll again
      inc $d6          ; increment TBLX
      inc $02a5
.e938  jsr $eb42        ; lda #$7f, sta $dc00, rts
      lda $dc01        ; read keyboard decode column
      cmp #$fb        ; <CTRL> pressed
      bne $e956        ; nope, exit
      ldx $c6          ; NDX, number of characters in keyboard buffer

```

```

        beq $e938      ; freeze scroll as long as <CTRL> is pressed
        lda $0276,x   ; read character from keyboard buffer
        sbc #$13      ; subtract $13, "S"
        bne $e956     ; nope, did not press "S"
        sta $c6       ; clear NDX
.e94f   cli          ; allow interrupts
        cmp $c6       ; any new character in buffer
        beq $e94f     ; nope, still freeze
        sta $c6       ; clear NDX
.e956   ldx $d6       ; read TBLX
.e958   pla          ; retrieve EAL
        sta $af
        pla
        sta $ae
        pla          ; retrieve SAL
        sta $ad
        pla
        sta $ac
        rts          ; exit

```

### **E965 OPEN A SPACE ON THE SCREEN**

This routine opens a space on the screen for use with <INST>. If needed, the screen is then scrolled down, otherwise the screen line is moved and cleared. Finally the screen line link table is adjusted and updated.

```

.e965   ldx $d6       ; TBLX, current cursor line number
.e967   inx          ; test next
        lda $d9,x     ; LDTB1, screen line link table
        bpl $e967
        stx $02a5     ; temp line for index
        cpx #$18      ; bottom of screen
        beq $e981     ; yes
        bcc $e981     ; above bottom line
        jsr $e8ea     ; scroll screen down
        ldx $02a5     ; temp line for index
        dex
        dec $d6       ; TBLX
        jmp $e6da     ; adjust link table and end
.e981   lda $ac       ; push SAL, scrolling pointer
        pha
        lda $ad
        pha
        lda $ae       ; push EAL, end of program
        pha
        lda $af
        pha
.e98f   ldx #$19
        dex
        jsr $e9f0     ; set start of line
        cpx $02a5     ; temp line for index
        bcc $e9a6
        beq $e9a6
        lda $ecef,x   ; screen line address table
        sta $ac       ; SAL
        lda $d8,x     ; LDTB1
        jsr $e9c8     ; move screen line
        bmi $e98f
.e9a6   jsr $e9ff     ; clear screen line
        ldx #$17      ; fix screen line link table
.e9ab   cpx $02a5     ; temp line for index
        bcc $e9bf

```

```

        lda $da,x          ; LDTB1+1
        and #$7f
        ldy $d9,x          ; LDTB1
        bpl $e9ba
        ora #$80
.e9ba   sta $da,x
        dex                ; next line
        bne $e9ab          ; till line zero
.e9bf   ldx $02a5          ; temp line for index
        jsr $e6da          ; adjust link table
        jmp $e958          ; pull SAL and EAL

```

#### **E9C8 MOVE A SCREEN LINE**

This routine synchronises colour transfer, and then moves the screen line pointed to down, character by character. The colour codes for each character are also moved in the same way.

```

.e9c8   and #$03
        ora $0288          ; HIBASE, top of screen page
        sta $ad            ; store >SAL, screen scroll pointer
        jsr $e9e0          ; synchronise colour transfer
        ldy #$27           ; offset for character on screen line
.e9d4   lda ($ac),y        ; move screen character
        sta ($d1),y
        lda ($ae),y        ; move character colour
        sta ($f3),y
        dey                ; next character
        bpl $e9d4          ; till all 40 are done
        rts

```

#### **E9E0 SYNCHRONISE COLOUR TRANSFER**

This routine sets up a temporary pointer in EAL to the colour RAM address that corresponds to the temporary screen address held in EAL.

```

.e9e0   jsr $ea24          ; synchronise colour pointer
        lda $ac            ; SAL, pointer for screen scroll
        sta $ae            ; EAL
        lda $ad
        and #$03
        ora #$d8           ; setup colour ram to $d800
        sta $af
        rts

```

#### **E9F0 SET START OF LINE**

On entry, (X) holds the line number. The low byte of the address is set from the ROM table, and the highbyte derived from the screen link and HIBASE.

```

.e9f0   lda $ecf0,x        ; table of screen line to bytes
        sta $d1            ; <PNT, current screen line address
        lda $d9,x          ; LDTB1, screen line link table
        and #$03
        ora $0288          ; HIBASE, page of top screen
        sta $d2            ; >PNT
        rts

```

#### **E9FF CLEAR SCREEN LINE**

The start of line is set and the screen line is cleared by filling it with ASCII spaces. The corresponding line of colour RAM is also cleared to the value held in COLOR.

```

.e9ff  ldy #$27
        jsr $e9f0          ; set start of line
        jsr $ea24          ; synchronise colour pointer
.ea07  jsr $e4da          ; reset character colour, to COLOR
        lda #$20           ; ASCII space
        sta ($d1),y       ; store character on screen
        dey               ; next
        bpl $ea07         ; till hole line is done
        rts

        nop               ; free byte

```

### **EA13 PRINT TO SCREEN**

The colour pointer is synchronised, and the character in (A) directly stored in the screen RAM. The character colour in (X) is stored at the equivalent point in the colour RAM.

```

.ea13  tay               ; put print character in (Y)
        lda #$02
        sta $cd          ; store in BLNCT, timer to toggle cursor
        jsr $ea24          ; synchronise colour pointer
        tya              ; print character back to (A)
.ea1c  ldy $d3           ; PNTR, cursor column on line
        sta ($d1),y     ; store character on screen
        txa
        sta ($f3),y     ; stor character colour
        rts

```

### **EA24 SYNCHRONISE COLOUR POINTER**

The pointer to the colour RAM is set up according to the current screen line address. This is done by reading the current screen line address and mofedying it to colour RAM pointers and write it to USER at \$f3/\$f4

```

.ea24  lda $d1           ; copy screen line low byte
        sta $f3          ; to colour RAM low byte
        lda $d2          ; read'n modify the hi byte
        and #$03
        ora #$d8
        sta $f4          ; to suite the colour RAM
        rts

```

### **EA31 MAIN IRQ ENTRY POINT**

This routine services the normal IRQ that jumps through the hardware vector to \$ff48, and then continues to the CINV vector at \$0314. First it checks if the <STOP> key was pressed and updates the realtime clock. Next, the cursor is updated (if it is enabled, BLNSW). The blink counter, BLNCT, is decremented. When this reaches zero, the cursor is toggled (blink on/off). Finally it scans the keyboard. The processor registers are then restored on exit.

Area from \$ea64 to \$ea7b has been changed in the JiffyDOS system. Some routintes to handle the casetterecorder has been removed.

```

.ea31  jsr $ffea          ; update realtime clock, routine UDTIM
        lda $cc          ; read BLNSW to see if cursor is enabled
        bne $ea61        ; nope
        dec $cd          ; read BLNCT
        bne $ea61        ; if zero, toggle cursor - else jump
        lda #$14         ; blink speed
        sta $cd          ; restore BLCNT
        ldy $d3          ; get PNTR, cursor column
        lsr $cf          ; BLNON, flag last cursor blink on/off

```

```

        ldx $0287          ; get background colour under cursor, GDCOL
        lda ($d1),y        ; get screen character
        bcs $ea5c          ; ?
        inc $cf            ; increment BLNON
        sta $ce            ; temporary store character under cursor
        jsr $ea24          ; synchronise colour pointer
        lda ($f3),y        ; get colour under character
        sta $0287          ; store in GDCOL
        ldx $0286          ; get current COLOR
        lda $ce            ; retrieve character under cursor
.ea5c  eor #$80            ; toggle cursor by inverting character
        jsr $ea1c          ; print to screen by using part of 'print to screen'
.ea61  jmp $ea7b          ; skip

```

#### **EA64 JIFFYDOS CRNCH**

The ICRNCH VECTOR points to this routine after the JiffyDOS init.

```

.ea64  pla                ; get last stack entry
        pha                ; put back
        cmp #$98           ; equal to #$98
        beq $ea6d          ; yepp, do JiffyDOS CRNCH
.ea6a  jmp $a57c           ; jump to original CRNCH
.ea6d  jsr $f72c           ; test if key in buffer is a JiffyDOS command
        bne $ea6a          ; no command
        ldx $7a            ; position i keyboardbuffer
        ldy #$04           ; setup values for old routine
        tya
        jmp $a5e3          ; back into old CRNCH

byte .xxx                ; free byte?? serialnumber!!

```

#### **EA7B QUICK IRQ ENTRY POINT**

If you dont want the screenupdate, of if you take care of it yourself, you can use this quick exit, specially \$ea81.

```

.ea7b  jsr $ea87          ; scan keyboard
        lda $dc0d          ; clear CIA#1 I.C.R to enable next IRQ
.ea81  pla                ; restore (Y), (X), (A)
        tay
        pla
        tax
        pla
        rti                ; back to normal

```

#### **EA87 SCNKEY: SCAN KEYBOARD**

The KERNAL routine SCNKEY (\$ff9f) jumps to this routine. First, the shift-flag, SHFLAG, is cleared, and the keyboard tested for nokey. The keyboard is set up as a 8 \* 8 matrix, and is read one row at a time. \$ff indicates that no key has been pressed, and a zerobit, that one key has been pressed.

```

.ea87  lda #$00
        sta $028d          ; clear SHFLAG
        ldy #$40
        sty $cb
        sta $dc00          ; store in keyboard write register
        ldx $dc01          ; keyboard read register
        cpx #$ff           ; no key pressed
        beq $eafb          ; skip
        tay
        lda #$81           ; point KEYTAB vector to $eb81
        sta $f5

```

```

        lda #$eb
        sta $f6
        lda #$fe          ; bit0 = 0
        sta $dc00        ; will test first row in matrix
.eaa8   ldx #$08          ; scan 8 rows in matrix
        pha              ; temp store
.eaab   lda $dc01        ; read
        cmp $dc01        ; wait for value to settle (key bouncing)
        bne $eaab
.eab3   lsr a            ; test bit0
        bcs $eacc        ; no key pressed
        pha
.eab7   lda ($f5),y      ; get key from KEYTAB
        cmp #$05         ; value less than 5
        bcs $eac9        ; nope
        cmp #$03         ; value = 3
        beq $eac9        ; nope
        ora $028d
        sta $028d        ; store in SHFLAG
        bpl $eacb
.eac9   sty $cb          ; store keynumber we pressed in SFDX
.eacb   pla
.eacc   iny              ; key counter
        cpy #$41         ; all 64 keys (8*8)
        bcs $eadc        ; jump if ready
        dex              ; next key in row
        bne $eab3        ; row ready
        sec              ; prepare for rol
        pla
        rol a            ; next row
        sta $dc00        ; store bit
        bne $eaa8        ; always jump
.eadc   pla              ; clean up

```

#### **EADD PROCESS KEY IMAGE**

This routine decodes the pressed key, and calculates its ASCII value, by use of the four tables. If the pressed key is the same key as in the former interrupt, then the key-repeat-section is entered. The routine tests the RPTFLG if the key shall repeat. The new key is stored in the keyboard buffer, and all pointers are updated.

```

.eadd   jmp ($028f)      ; jump through KEYLOG vector, points to $eae0
.eae0   ldy $cb          ; SFDX, number of the key we pressed
        lda ($f5),y      ; get ASCII value from decode table
        tax              ; temp store
        cpy $c5         ; same key as former interrupt
        beq $eaf0        ; yepp
        ldy #$10         ; restore the repeat delay counter
        sty $028c        ; DELAY
        bne $eb26        ; always jump
.eaf0   and #$7f
        bit $028a        ; RPTFLG, test repeat mode
        bmi $eb0d        ; repeat all keys
        bvs $eb42        ; repeat none - exit routine
        cmp #$7f
.eafb   beq $eb26
        cmp #$14         ; <DEL> key pressed
        beq $eb0d        ; yepp...
        cmp #$20         ; <space> key pressed
        beq $eb0d        ; yepp...
        cmp #$1d         ; <CRSR LEFT/RIGHT>

```

```

        beq $eb0d          ; yepp..
        cmp #$11          ; <CRSRS DOWN/UP>
        bne $eb42        ; yepp..
.eb0d   ldy $028c         ; DELAY
        beq $eb17        ; skip
        dec $028c        ; decrement DELAY
        bne $eb42        ; end
.eb17   dec $028b        ; decremant KOUNT, repeat speed counter
        bne $eb42        ; end
        ldy #$04
        sty $028b        ; init KOUNT
        ldy $c6          ; read NDX, number of keys in keyboard queue
        dey
        bpl $eb42        ; end
.eb26   ldy $cb          ; read SFDX
        sty $c5          ; store in LSTX
        ldy $028d        ; read SHFLAG
        sty $028e        ; store in LSTSHF, last keyboard shift pattern
        cpx #$ff        ; no valid key pressed
        beq $eb42        ; end
        txa
        ldx $c6          ; NDX, number of keys in buffer
        cpx $0289        ; compare to XMAX, max numbers oc characters in
buffer   bcs $eb42        ; buffer is full, end
        sta $0277,x      ; store new character in keyboard buffer
        inx              ; increment counter
        stx $c6         ; and store in NDX
.eb42   lda #$7f
        sta $dc00        ; keyboard write register
        rts              ; exit

.eb48   lda $028d        ; SHFLAG
        cmp #$03        ; <SHIFT> and <CBM> at the same time
        bne $eb64        ; nope
        cmp $028e        ; same as LSTSHF
        beq $eb42        ; if so, end
        lda $0291        ; read MODE, shift key enable flag
        bmi $eb76        ; end
        lda $d018        ; VIC memory control register
        eor #$02        ; toggle character set, upper/lower case
        sta $d018        ; and store
        jmp $eb76        ; process key image
.eb64   asl a
        cmp #$08        ; test <CTRL>
        bcc $eb6b        ; nope
        lda #$06        ; set offset for ctrl
.eb6b   tax              ; to (X)
        lda $eb79,x      ; read keyboard select vectors, low byte
        sta $f5          ; store in KEYTAB, decode table vector
        lda $eb7a,x      ; read keyboard select vectors, high byte
        sta $f6          ; KEYTAB+1
.eb76   jmp $eae0        ; process key image

```

#### **EB79 KEYBOARD SELECT VECTORS**

This is a table of vectors pointing to the start of the four keyboard decode tables.

```

.eb79   81 eb          ; vector to unshifted keyboard, $eb81
.eb7b   c2 eb          ; vector to shifted keyboard, $ebc2
.eb7d   03 ec          ; vector to cbm keyboard, $ec03

```

.eb7f 78 ec ; vector to ctrl keyboard, \$ec78

**EB81 KEYBOARD 1 - UNSHIFTED**

This is the first of four keyboard decode tables. The ASCII code for the key pressed is at the intersection of the row (written to \$dc00) and the column (read from \$dc01). The matrix values are shown below. Note that left and right shift keys are seperated.

```
.eb81 14 0d 1d 88 85 86 87 11
.eb89 33 57 41 34 5a 53 45 01
.eb91 35 52 44 36 43 46 54 58
.eb99 37 59 47 38 42 48 55 56
.eba1 39 49 4a 30 4d 4b 4f 4e
.eba9 2b 50 4c 2d 2e 3a 40 2c
.ebb1 5c 2a 3b 13 01 3d 5e 2f
.ebb9 31 5f 04 32 20 02 51 03
.ebc1 ff ; free byte
```

DEL	RETURN	CRSR RI	F7	F1	F3	F5	CRSR DO
3	w	a	4	z	s	e	LE
							SHIFT
5	r	d	6	c	f	t	x
6	y	g	8	b	h	u	v
9	i	j	0	m	k	o	n
+	p	l	-	.	:	@	,
£	*	;	HOME	RI	=	^	/
				SHIFT			
1	<-	CTRL	2	SPACE	CBM	q	STOP

**EBC2 KEYBOARD 2 - SHIFTED**

This is the second of four keyboard decode tables. The ASCII code for the key pressed is at the intersection of the row (written to \$dc00) and the column (read from \$dc01). The matrix values are shown below.

```
.ebc2 94 8d 9d 8c 89 8a 8b 91
.ebca 23 d7 c1 24 da d3 c5 01
.ebd2 25 d2 c4 26 c3 c6 d4 d8
.ebda 27 d9 c7 28 c2 c8 d5 d6
.ebe2 29 c9 ca 30 cd cb cf ce
.ebea db d0 cc dd 3e 5b ba 3c
.ebf2 a9 c0 5d 93 01 3d de 3f
.ebfa 21 5f 04 22 a0 02 d1 83
.ec02 ff ; free byte
```

INST	RRETURN	CRSR LE	F8	F2	F4	F6	CRSR UP
#	W	A	\$	Z	S	E	LE
							SHIFT
%	R	D	&	C	F	T	X
'	Y	G	(	B	H	U	V
)	I	J	0	M	K	O	N
cbm gr	P	L	cbm gr	>	[	cbm gr	<
cbm gr	cbm gr	[	CLR	RI	=	pi	?
				SHIFT			
!	<-	CTRL	"	SPACE	CBM	Q	RUN

**EC03 KEYBOARD 3 - COMMODORE**

This is the third of four keyboard decode tables. The ASCII code for the key pressed is at the intersection of the row (written to \$dc00) and the column (read from \$dc01). The matrix values are shown below.

```
.ec03 94 8d 9d 8c 89 8a 8b 91
.ec0b 96 b3 b0 97 ad ae b1 01
.ec13 98 b2 ac 99 bc bb a3 bd
.ec1b 9a b7 a5 9b bf b4 b8 be
.ec23 29 a2 b5 30 a7 a1 b9 aa
.ec2b a6 af b6 dc 3e 5b a4 3c
.ec33 a8 df 5d 93 01 3d de 3f
.ec3b 81 5f 04 95 a0 02 ab 83
.ec43 ff ; free byte
```

INST	RETURN	CRSR LE	F8	F2	F4	F6	CRSR UP
pink	cbm gr	cbm gr	grey 1	cbm gr	cbm gr	cbm gr	LE
							SHIFT
grey 2	cbm gr	cbm gr	li	cbm gr	cbm gr	cbm gr	cbm gr
			green				
li blue	cbm gr	cbm gr	grey 3	cbm gr	cbm gr	cbm gr	cbm gr
)	cbm gr	cbm gr	0	cbm gr	cbm gr	cbm gr	cbm gr
cbm gr	cbm gr	cbm gr	cbm gr	>	[	cbm gr	<
cbm gr	cbm gr	]	CLR	RI	=	pi	?
				SHIFT			
orange	<-	CTRL	brown	SPACE	CBM	cbm gr	RUN

#### **EC44 GRAPHICS / TEXT CONTROL**

This routine is used to toggle between text and graphics character set, and to enable/disable the <shift-CBM> keys. The routine is called by the main 'output to screen' routine, and (A) holds a CBM ASCII code on entry.

```
.ec44 cmp #$0e ; <switch to lower case>
      bne $ec4f ; nope
      lda $d018 ; VIC memory control register
      ora #$02 ; set bit1
      bne $ec58 ; always branch
.ec4f cmp #$8e ; <switch to upper case>
      bne $ec5e ; nope
      lda $d018 ; VIC memory control register
      and #$fd ; clear bit1
.ec58 sta $d018 ; and store
.ec5b jmp $e6a8 ; finish screen print
.ec5e cmp #$08 ; <disable <shift-CBM>>
      bne $ec69 ; nope
      lda #$80
      ora $0291 ; disable MODE
      bmi $ec72 ; always jump
.ec69 cmp #$09 ; <enable <shift-CBM>>
      bne $ec5b ; nope, exit
      lda #$7f
      and $0291 ; enable MODE
.ec72 sta $0291 ; store MODE, enable/disable shift keys
      jmp $e6a8 ; finish screen print
```

#### **EC78 KEYBOARD 4 - CONTROL**

This is the last keyboard decode table. The ASCII code for the key pressed is at the intersection of the row (written to \$dc00) and the column (read from \$dc01). The matrix values are shown below.

A few special funktion are found in this table ie.

<ctrl H> - disables the upper/lower case switch

<ctrl I> - enables the upper/lower case switch

<ctrl S> - homes the cursor

<ctrl T> - deletes character

Note that the italic keys only represent a ASCII code, and not a CBM character.

Future implementations: Change some of the \$ff values which represents 'no key' to a valid ASCII code. ESC (\$1b) and why not use the F-keys for something useful.

```
.ec78 ff ff ff ff ff ff ff ff
.ec80 1c 17 01 9f 1a 13 05 ff
.ec88 9c 12 04 1e 03 06 14 18
.ec90 1f 19 07 9e 02 08 15 16
.ec98 12 09 0a 92 0d 0b 0f 0e
.eca0 ff 10 0c ff ff 1b 00 ff
.eca8 1c ff 1d ff ff 1f 1e ff
.ecb0 90 06 ff 05 ff ff 11 ff
.ecb8 ff ; free byte
```

red	<i>W</i>	<i>A</i>	cyan	<i>Z</i>	HOME	white	
purple	RVS ON	<i>D</i>	green	STOP	<i>F</i>	DEL	<i>X</i>
blue	<i>Y</i>	<i>G</i>	yellow	CBM	DISABLE	<i>U</i>	<i>V</i>
RVS ON	ENABLE	<i>J</i>	RVS OFF	RETURN	<i>K</i>	<i>O</i>	LOWER
	<i>P</i>	<i>L</i>			<i>J</i>	<i>@</i>	
red		CRSR RI			blue	green	
	<-		white			CRSR DO	

#### **ECB9 VIDEO CHIP SET UP TABLE**

This is a table of the initial values for the VIC chip registers at start up.

```
.ecb9 00 00 ; $d000/1, sprite0 - x,y cordinate
.ecbb 00 00 ; $d002/3, spritel - x,y cordinate
.ecbd 00 00 ; $d004/5, sprite2 - x,y cordinate
.ecbf 00 00 ; $d006/7, sprite3 - x,y cordinate
.ecc1 00 00 ; $d008/9, sprite4 - x,y cordinate
.ecc3 00 00 ; $d00a/b, sprite5 - x,y cordinate
.ecc5 00 00 ; $d00c/d, sprite6 - x,y cordinate
.ecc7 00 00 ; $d00e/f, sprite7 - x,y cordinate
.ecc9 00 ; $d010, sprite MSB
.ecca 9b ; $d011, VIC control register
.eccb 37 ; $d012,
.eccc 00 00 ; $d013/4, light pen x/y position
.ecce 00 ; $d015, sprite enable
.eccf 08 ; $d016, VIC control register 2
.ecd0 00 ; $d017, sprite y-expansion
.ecd1 14 ; $d018, VIC memory control register
.ecd2 0f ; $d019, VIC irq flag register
.ecd3 00 ; $d01a, VIC irq mask register
.ecd4 00 ; $d01b, sprite/background priority
.ecd5 00 ; $d01c, sprite multicolour mode
.ecd6 00 ; $d01d, sprite x-expansion
.ecd7 00 ; $d01e, sprite/sprite collision
.ecd8 00 ; $d01f, sprite/background collision
.ecd9 0e ; $d020, border colour (light blue)
.ecda 06 ; $d021, background colour 0 (blue)
.ecdb 01 ; $d022, background colour 1
.ecdc 02 ; $d023, background colour 2
```

```

.ecdd 03          ; $d024, background colour 3
.ecde 04          ; $d025, sprite multicolour register 0
.ecdf 00          ; $d026, sprite multicolour register 1
.ece0 01          ; $d027, sprite0 colour
.ecel 02          ; $d028, sprite1 colour
.ece2 03          ; $d029, sprite2 colour
.ece3 04          ; $d02a, sprite3 colour
.ece4 05          ; $d02b, sprite4 colour
.ece5 06          ; $d02c, sprite5 colour
.ece6 07          ; $d02d, sprite6 colour

```

#### **ECE7 SHIFT-RUN EQUIVALENT**

This is the message LOAD <CR> RUN <CR>, which is placed in the keyboard buffer when <shift-RUN> is pressed.

```

.ece7 4c 4f 41 44 0d ; LOAD <CR>
.ecec 52 55 4e 0d   ; RUN <CR>

```

#### **ECF0 LOW BYTE SCREEN LINE ADDRESSES**

This is a table of the low bytes of screen line addresses. The high byte of the addresses is obtained by derivation from the page on which the screen starts. There was an additional table of high byte addresses on the fixed screen PETs.

```

.ecf0 00 28 50 78 a0
.ecf5 c8 f0 18 40 68
.ecfa 90 b8 e0 08 30
.ecff 58 80 a8 d0 f8
.ed04 20 48 70 98 c0

```

#### **ED09 TALK: SEND 'TALK' / 'LISTEN'**

The KERNAL routine TALK (\$ffb4) and LISTEN (\$ffb1) are vectored here. The routine sends the command 'TALK' or 'LISTEN' on the serial bus. On entry (A) must hold the device number to which the command will be sent. The two entry points differ only in that to TALK, (A) is Ored with #\$40, and to LISTEN, (A) is Ored with #\$20. The UNTALK (#\$3f) and UNLISTEN (#\$5f) are also sent via this routine, but their values are set on entry. If there is a character waiting to go out on the bus, then this is output. Handshaking is performed, and ATN (attension) is set low so that the byte is interpreted as a command. The routine drops through to the next one to output the byte on the serial bus. Note that on conclusion, ATN must be set high.

```

.ed09 ora #$40          ; set TALK flag
      .byte $2c        ; bit $2009, mask ORA command
      ora #$20         ; set LISTEN flag
      jsr $f0a4        ; check serial bus idle
.ed11 pha
      bit $94          ; C3PO, character in serial buffer
      bpl $ed20        ; nope
      sec             ; prepare for ROR
      ror $a3         ; temp data area
      jsr $fbfe        ; JiffyDOS, send data to serial bus
      lsr $94         ; 3CPO
      lsr $a3
.ed20 pla
      sta $95          ; BSOUR, buffered character for bus
      sei
      jsr $f0ed        ; JiffyDOS, set data 1, and clear serial bit count
      cmp #$3f        ; UNTALK?
      bne $ed2e        ; nope

```

```

        jsr $ee85          ; set CLK 1
.ed2e  lda $dd00          ; serial bus I/O port
        ora #$08          ; clear ATN, prepare for command
        sta $dd00          ; store
.ed36  sei                ; disable interrupts
        jsr $ee8e          ; set CLK 1
        jsr $ee97          ; set data 1
        jsr $eeb3          ; delay 1 ms

```

#### **ED40 SEND DATA ON SERIAL BUS**

The byte of data to be output on the serial bus must have been previously stored in the serial buffer, BSOUR. An initial test is made for bus activity, and if none is detected then ST is set to #\$80, ie. ?DEVICE NOT PRESENT. The byte is output by rotating it right and sending the state of the carry flag. This is done eight times until the whole byte was sent. The CIA timer is set to 65 ms and the bus is checked for 'data accepted'. If timeout occurs before this happens then ST is set to #\$03, ie. write timeout. The routine is modified with a jump to \$f8ea where a test is done to see if this device is a JiffyDOS device. The result is stored in \$a3.

```

        sei                ; disable interrupts
        jsr $ee97          ; set data 1
        jsr $eea9          ; get serial in and clock
        bcs $edad          ; no activity, device not present.
        jsr $ee85          ; set CLK 1
        bit $a3            ; temp data area
        bpl $ed5a
.ed50  jsr $eea9          ; get serial in and clock
        bcc $ed50          ; wait for indata = 0
.ed55  jsr $eea9          ; get serial in and clock
.ed58  bcs $ed55          ; wait for indata = 1
.ed5a  jsr $eea9          ; get serial in and clock
        bcc $ed5a          ; wait for indata = 0
        jsr $ee8e          ; set CLK 0

        txa                ; transfer (X) to (A)
        pha                ; store (A) on stack
        ldx #$08           ; output 8 bits
.ed66  pha
        pla
        bit $dd00          ; serial bus I/O port
        bmi $ed72          ; no timeout
        pla                ; retrieve (A)
        tax                ; and restore (X)
        jmp $edb0          ; exit with flag write timeout
.ed72  jsr $ee97          ; serial output 1
        ror $95            ; BSOUR, buffered character for bus
        bcs $ed7c          ; prepare to output 1
        jsr $eea0          ; else, serial output 0
.ed7c  jsr $ee85          ; set CLK 1
        lda $dd00          ; serial bus I/O port
        and #$df           ; set data 1
        ora #$10           ; set CLK 0
        php
        pha
        jsr $f8ea          ; test if device on serial bus is a JiffyDOS device
        pla
        plp
        dex                ; decrement bit counter
        bne $ed66          ; next bit till all 8 are done
        pla

```

```

tax
lda #$04
sta $dc07      ; CIA timer B, high byte
lda #$19
sta $dc0f      ; set 1 shot, load and start CIA timer B
lda $dc0d      ; CIA ICR
.ed9f lda $dc0d
and #$02      ; timeout
bne $edb0      ; yep, flag write timeout
jsr $eea9      ; get serial in and clock
bcs $ed9f
cli           ; enable interrupts
rts

```

#### **EDAD FLAG ERRORS**

(A) is loaded with one of the two error flags, depending on the entry point. #\$80 signifies the device was not present, and #\$03 signifies a write timeout. The value is then set into the I/O status word, ST. The routine exits by clearing ATN and giving the final handshake.

```

.edad lda #$80      ; flag ?DEVICE NOT PRESENT
      .byte $2c      ; mask LDA #$03
.edb0 lda #$03      ; flag write timeout
.edb2 jsr $felc     ; set I/O status word
      cli
      clc
      bcc $ee03     ; always jump, do final handshake

```

#### **EDB9 SECOND: SEND LISTEN SA**

The KERNAL routine SECOND (\$ff93) is vectored here. On entry, (A) holds the secondary address. This is placed in the serial buffer and sent to the serial bus "under attension". Finally the routine drops through to the next routine to set ATN false.

```

.edb9 sta $95      ; store (A) in BSOUT, buffer for the serial bus
      jsr $ed36     ; handshake and send byte.

```

#### **EDBE CLEAR ATN**

The ATN, attension, line on the serial bus is set to 1, ie. ATN is now false and data sent on the serial bus will not be interpreted as a command.

```

.edbe lda $dd00     ; serial bus I/O port
      and #$f7      ; clear bit4, ie. ATN 1
      sta $dd00     ; store to port
      rts

```

#### **EDC7 TKSA: SEND TALK SA**

The KERNAL routine TKSA (\$ff96) is vectored here. On entry, (A) holds the secondary address. This is placed in the serial buffer and sent out to the serial bus "under attension". The routine drops through to the next routine to wait for CLK and clear ATN.

```

      sta $95      ; BSOUR, the serial bus buffer
      jsr $ed36     ; handshake and send byte to the bus

```

#### **EDCC WAIT FOR CLOCK**

This routine sets data = 0, ATN = 1 and CLK = 1. It then waits to recieve CLK = 0 from the serial bus.

```

      sei           ; disable interrupts
      jsr $eea0     ; set data 0

```

```

        jsr $edbe          ; set ATN 1
        jsr $ee85          ; set CLK 1
.edd6  bit $dd00          ; read serial bus I/O port
        bvs $edd6          ; test bit6, and wait for CLK = 0
        cli                ; enable interrupt
        rts

```

#### **EDDD CIOUT: SEND SERIAL DEFERRED**

The KERNAL routine CIOUT (\$ffa8) jumps to this routine. If there is a character awaiting output in the buffer, then it is sent on the bus to the new JiffyDOS send routine. The output flag, C3PO is set (ie. bit 7 = 1) and the contents of (A) is placed in the serial buffer.

```

.eddd  bit $94            ; C3PO flag, character in serial buffer
        bmi $ede6          ; yes
        sec                ; prepare for ROR
        ror $94            ; set C3PO
        bne $edeb          ; always jump
.ede6  pha                ; temp store
        jsr $fbfe          ; JiffyDOS send data to serial bus
        pla
.edeb  sta $95            ; store character in BSOUR
        clc                ; clear carry to indicate no errors
        rts

```

#### **EDEF UNTLK: SEND 'UNTALK'/'UNLISTEN'**

The KERNAL routine UNTALK (\$ffab) and UNLISTEN (\$ffae) are vectored here. ATN is set to 0, and CLK is set to 0. (A) is loaded with #\$5f for 'UNTALK' and #\$3f for 'UNLISTEN'. The command is sent to the serial bus via the 'send TALK/LISTEN' routine. Finally ATN is set to 1, and after a short delay, CLK and data are both set to 1.

```

.edef  sei                ; disable interrupts
        lda $dd00          ; serial bus I/O
        ora #$08           ; set bit4
        sta $dd00          ; and store, set ATN 0
        jsr $ee8e          ; set CLK 0
        lda #$5f           ; flag UNTALK
        .byte $2c          ; mask LDA #$3f with BIT $3fa9
        lda #$3f           ; flag UNLISTEN
        jsr $ed11          ; send command to serial bus
.ee03  jsr $edbe          ; clear ATN
.ee06  txa
        ldx #$0a           ; init delay
.ee09  dex                ; decrement counter
        bne $ee09          ; till ready
        tax
        jsr $ee85          ; set CLK 1
        jmp $ee97          ; set data 1

```

#### **EE13 ACPTR: RECIEVE FROM SERIAL BUS**

The KERNAL routine ACPTR (\$ffa5) points to this routine in the original Commodore KERNAL. JiffyDOS uses a routine at \$fbaa, which is the new ACPTR pointer. This routine is used when a device is not JiffyDOS equipped. A timing loop is entered using the CIA timer, and if a byte is not received in 65 ms, ST is set to #\$02, ie. a read timeout. A test is made for EOI and if this occurs, ST is set to #\$40, indicating end of file. The byte is then received from the serial bus and built up bit by bit in the temporary storage at #\$a4. This is transferred to (A) on exit, unless EOI has occurred.

```

.ee13 jmp $fbaa      ; Jump to JiffyDOS ACPTR, return if no JiffyDOS
device
      sta $a5      ; CNTDN, counter
      jsr $ee85    ; set CLK 1
.ee1b jsr $eea9    ; get serial in and clock
      bpl $ee1b    ; wait for CLK = 1
.ee20 lda #$01
      sta $dc07    ; setup CIA#1 timer B, high byte
      lda #$19
      sta $dc0f    ; set 1 shot, load and start CIA timer B
      jsr $ee97    ; set data 1
      lda $dc0d
.ee30 lda $dc0d    ; read CIA#1 ICR
      and #$02     ; test if timer B reaches zero
      bne $ee3e    ; timeout
      jsr $eea9    ; get serial in and clock
      bmi $ee30    ; CLK 1
      bpl $ee56    ; CLK 0
.ee3e lda $a5      ; CNTDN
      beq $ee47
      lda #$02     ; flag read timeout
      jmp $edb2    ; set I/O status word
.ee47 jsr $eea0    ; set data 1
      jsr $ee85    ; set CLK 1
      lda #$40     ; flag EOI
      jsr $felc    ; set I/O status word
      inc $a5      ; increment CNTDN, counter
      bne $ee20    ; again
.ee56 lda #$08     ; set up CNTDN to receive 8 bits
      sta $a5
.ee5a lda $dd00    ; serial bus I/O port
      cmp $dd00    ; compare
      bne $ee5a    ; wait for serial bus to settle
      asl
      bpl $ee5a    ; wait for data in =1
      ror $a4      ; roll in received bit in temp data area
.ee67 lda $dd00    ; serial bus I/O port
      cmp $dd00    ; compare
      bne $ee67    ; wait for bus to settle
      asl
      bmi $ee67    ; wait for data in =0
      dec $a5      ; one bit received
      bne $ee5a    ; repeat for all 8 bits
      jsr $eea0    ; set data 1
      bit $90      ; STATUS, I/O status word
      bvc $ee80    ; not EOI
      jsr $ee06    ; handshake and exit without byte
.ee80 lda $a4      ; read received byte
      cli          ; enable interrupts
      clc          ; clear carry, no errors
      rts

```

#### **EE85 SERIAL CLOCK ON**

This routine sets the clock outline on the serial bus to 1. This means writing a 0 to the port. This value is reversed by hardware on the bus.

```

.ee85 lda $dd00    ; serial port I/O register
      and #$ef     ; clear bit4, ie. CLK out =1
      sta $dd00    ; store
      rts

```

### **EE8E SERIAL CLOCK OFF**

This routine sets the clock outline on the serial bus to 0. This means writing a 1 to the port. This value is reversed by hardware on the bus.

```
.ee8e  lda $dd00          ; serial port I/O register
      ora #$10         ; set bit4, ie. CLK out =0
      sta $dd00       ; store
      rts
```

### **EE97 SERIAL OUTPUT 1**

This routine sets the data out line on the serial bus to 1. This means writing a 0 to the port. This value is reversed by hardware on the bus.

```
.ee97  lda $dd00          ; serial bus I/O register
      and #$df         ; clear bit5
      sta $dd00       ; store
      rts
```

### **EEA0 SERIAL OUTPUT 0**

This routine sets the data out line on the serial bus to 0. This means writing a 1 to the port. This value is reversed by hardware on the bus.

```
.eea0  lda $dd00          ; serial bus I/O resister
      ora #$20         ; set bit 5
      sta $dd00       ; store
      rts
```

### **EEA9 GET SERIAL DATA AND CLOCK IN**

The serial port I/O register is stabilised and read. The data is shifted into carry and CLK into bit 7. This way, both the data and clock can be determined by flags in the processor status register. Note that the values read are true, and do not need to be reversed in the same way as the output line do.

```
.eea9  lda $dd00          ; serial port I/O register
      cmp $dd00        ; compare
      bne $eea9        ; wait for bus to settle
      asl              ; shift data into carry, and CLK into bit 7
      rts
```

### **EEB3 DELAY 1 MS**

This routine is a software delay loop where (X) is used as counter, and are decremented for a period of 1 millisecond. The original (X) is stored on entry and (A) is messed up.

```
.eeb3  txa              ; move (X) to (A)
      ldx #$b8         ; start value
.eeb6  dex              ; decrement
      bne $eeb6        ; untill zero
      tax              ; (A) to (X)
      rts
```

### **EEBB RS232 SEND**

This routine is concerned with sending a byte on the RS232 port. The data is actually written to the port under NMI interrupt control. The CTS line generates an NMI when the port is ready for data. If all the bits in the byte have been sent, then a new RS232 byte is set up. Otherwise, this routine calculates parity and number of stop bits set up in the OPEN command. These bits are added to the end of the byte being sent.

```
.eebb  lda $b4          ; BITTS, RS232 out bit count
```

```

        beq $ef06          ; send new RS232 byte
        bmi $ef00
        lsr $b6           ; RODATA, RS232 out byte buffer
        ldx #$00
        bcc $eec8
        dex
.eec8   txa
        eor $bd           ; ROPRTY, RS232 out parity
        sta $bd
        dec $b4           ; BITTS
        beq $eed7
.eed1   txa
        and #$04
        sta $b5           ; NXTBIT, next RS232 bit to send
        rts
.eed7   lda #$20
        bit $0294         ; M51CDR, 6551 command register image
        beq $eef2         ; no patity
        bmi $eefc         ; mark/space transmit
        bvs $eef6         ; even parity
        lda $bd           ; ROPRTY, out parity
        bne $eee7
.eee6   dex
.eee7   dec $b4           ; BITTS, out bit count
        lda $0293         ; M51CTR, 6551 control register image
        bpl $eed1         ; one stop bit only
        dec $b4           ; BITTS
        bne $eed1
.eef2   inc $b4           ; BITTS
        bne $eee6
.eef6   lda $bd           ; ROPRTY
        beq $eee7
        bne $eee6
.eefc   bvs $eee7
        bvc $eee6
.ef00   inc $b4           ; BITTS
        ldx #$ff
        bne $eed1

```

#### **EF06 SEND NEW RS232 BYTE**

This routine sets up the system variables ready to send a new byte to the RS232 port. A test is made for 3-line or X-line modus. In X-line mode, DSR and CTS are checked.

```

.ef06   lda $0294         ; M51CDR, 6551 command register
        lsr a             ; test handshake mode
        bcc $ef13         ; 3-line mode (no handshake)
        bit $dd01         ; RS232 port
        bpl $ef2e         ; no DSR, error
        bvc $ef31         ; no CTS, error
.ef13   lda #$00
        sta $bd           ; ROPRTY, RS232 out parity
        sta $b5           ; NXTBIT, next bit to send
        ldx $0298         ; BITNUM, number of bits left to send
        stx $b4           ; BITTS, RS232 out bit count
        ldy $029d         ; RODBS, start page of out buffer
        cpy $029e         ; RODBE, index to end if out buffer
        beq $ef39         ; disable timer
        lda ($f9),y       ; RS232 out buffer
        sta $b6           ; RODATA, RS232 out byte buffer
        inc $029d         ; RODBS

```

rts

#### **EF2E NO DSR / CTS ERROR**

(A) is loaded with the error flag - \$40 for no DSR, and \$10 for no CTS. This is then ORed with 6551 status image and stored in RSSTAT.

```
.ef2e lda #$40          ; entrypoint for 'NO DSR'
      .byte $2c        ; mask next LDA-command
.ef31 lda #$10          ; entrypoint for 'NO CTS'
      ora $0297        ; RSSTAT, 6551 status register image
      sta $0297
```

#### **EF39 DISABLE TIMER**

This routine set the interrupt mask on CIA#2 timer B. It also clears the NMI flag.

```
.ef39 lda #$01
.ef3b sta $dd0d         ; CIA#2 interrupt control register
      eor $02a1         ; ENABL, RS232 enables
      ora #$80
      sta $02a1         ; ENABL
      sta $dd0d         ; CIA#2 interrupt control register
      rts
```

#### **EF4A COMPUTE BIT COUNT**

This routine computes the number of bits in the word to be sent. The word length information is held in bits 5 & 6 of M51CTR. Bit 7 of this register indicates the number of stop bits. On exit, the number of bits is held in (X).

```
.ef4a ldx #$09
      lda #$20
      bit $0293         ; M51CTR, 6551 control register image
      beq $ef54
      dex
.ef54 bvc $ef58
      dex
      dex
.ef58 rts
```

#### **EF59 RS232 RECEIVE**

This routine builds up the input byte from the RS232 port in RIDATA. Each bit is input from the port under NMI interrupt control. The bit is placed in INBIT before being passed to this routine, where it is shifted into the carry flag and then rotated into RIDATA. The bit count is decremented and parity updated.

```
.ef59 ldx $a9          ; RINONE, check for start bit?
      bne $ef90
      dec $a8           ; BITC1, RS232 in bit count
      beq $ef97         ; process received byte
      bmi $ef70
      lda $a7           ; INBIT, RS232 in bits
      eor $ab           ; RIPRTY, RS232 in parity
      sta $ab
      lsr $a7           ; INBIT, put input bit into carry
      ror $aa           ; RIDATA,
.ef6d rts
.ef6e dec $a8           ; BITC1
.ef70 lda $a7           ; INBIT
      beq $efdb
```

```

        lda $0293          ; M51CTR, 6551 control register image
        asl a
        lda #$01
        adc $a8            ; BITC1
        bne $ef6d          ; end

```

#### **EF7E SET UP TO RECEIVE**

This routine sets up the I.C.R. to wait for the receiver edge, and flags this into ENABL. It then flags the check for a start bit.

```

.ef7e  lda #$90
        sta $dd0d          ; CIA#2 I.C.R.
        ora $02a1          ; ENABL, RS232 enables
        sta $02a1
        sta $a9            ; RINONE, check for start bit
        lda #$02
        jmp $ef3b          ; disable timer and exit

```

#### **EF90 PROCESS RS232 BYTE**

The byte recieved from the RS232 port is checked against parity. This involvs checking the input parity options selected, and then verifying the parity bit calculated against that input. If the test is passed, then the byte is stored in the in-buffer. Otherwise an error is flagged into RSSTAT. A patch in KERNAL version 3, has been added to the input routine at \$ef94 to initialise the RS232 parity byte, RIPRTY, on reception of a start bit.

```

.ef90  lda $a7              ; INBIT, RS232 in bits
        bne $ef7e          ; set up to receive
        jmp $e4d3          ; patch, init parity byte
.ef97  ldy $029b           ; RIDBE, index to the end of in buffer
        iny
        cpy $029c          ; RIDBS, start page of in buffer
        beq $efca          ; receive overflow error
        sty $029b          ; RIDBE
        dey
        lda $aa            ; RIDATA, RS232 in byte buffer
        ldx $0298          ; BITNUM, number of bits left to send
.efa9  cpx #$09            ; full word to come?
        beq $efb1          ; yes
        lsr a
        inx
        bne $efa9
.efb1  sta ($f7),y         ; RIBUF, RS232 in buffer
        lda #$20
        bit $0294          ; M51CDR, 6551 command register image
        beq $ef6e          ; parity disabled
        bmi $ef6d          ; parity check disabled, TRS
        lda $a7            ; INBIT, parity check
        eor $ab            ; RIPRTY, RS232 in parity
        beq $efc5          ; receive parity error
        bvs $ef6d
        .byte $2c          ; mask
.efc5  bvc $ef6d
        lda #$01           ; receive parity error
        .byte $2c          ; mask
.efca  lda #$04            ; receive overflow
        .byte $2c          ; mask
.efcd  lda #$80            ; framing break
        .byte $2c          ; mask
.efd0  lda #$02            ; framing error
        ora $0297          ; RSSTAT, 6551 status register image

```

```

        sta $0297
        jmp $ef7e          ; set up to receive
.efdb  lda $aa            ; RIDATA
        bne $efd0        ; framing error
        beq $efcd        ; receive break

```

#### **EFE1 SUBMIT TO RS232**

This routine is called when data is required from the RS232 port. Its function is to perform the handshaking on the port needed to receive the data. If 3 line mode is used, then no handshaking is implemented and the routine exits.

```

.efel  sta $9a            ; DFLTO, default output device
        lda $0294        ; M51CDR, 6551 command register image
        lsr a
        bcc $f012        ; 3 line mode, no handshaking, exit
        lda #$02
        bit $dd01        ; RS232 I/O port
        bpl $f00d        ; no DRS, error
        bne $f012
.eff2  lda $02a1          ; ENABL, RS232 enables
        and #$02
        bne $eff2
.eff9  bit $dd01          ; RS232 I/O port
        bvs $eff9        ; wait for no CTS
        lda $dd01
        ora #$02
        sta $dd01        ; set RTS
.f006  bit $dd01
        bvs $f012        ; CTS set
        bmi $f006        ; wait for no DSR

```

#### **F00D NO DSR ERROR**

This routine sets the 6551 status register image to #40 when a no DSR error has occurred.

```

.f00d  lda #$40
        sta $0297        ; RSSTAT, 6551 status register image
.f012  clc
        rts

```

#### **F014 SEND TO RS232 BUFFER**

Note: The entry point to the routine is at

```

if014  jsr if028
if017  ldy a029e
        iny
        cpy a029d
        beq if014
        sty a029e
        dey
        lda a9e
        sta (pf9),y
if028  lda a02a1
        lsr a
        bcs if04c
        lda #$10
        sta add0e
        lda a0299
        sta add04
        lda a029a
        sta add05

```

```

        lda #$81
        jsr eef3b
        jsr eef06
        lda #$11
        sta add0e
if04c  rts

```

#### **F04D INPUT FROM RS232**

```

if04d  sta a99
        lda a0294
        lsr a
        bcc if07d
        and #$08
        beq if07d
        lda #$02
        bit add01
        bpl if00d
        beq if084
if062  lda a02a1
        lsr a
        bcs if062
        lda add01
        and #$fd
        sta add01
if070  lda add01
        and #$04
        beq if070
if077  lda #$90
        clc
        jmp eef3b
if07d  lda a02a1
        and #$12
        beq if077
if084  clc
        rts

```

#### **F086 GET FROM RS232**

```

if086  lda a0297
        ldy a029c
        cpy a029b
        beq if09c
        and #$f7
        sta a0297
        lda (pf7),y
        inc a029c
        rts
if09c  ora #$08
        sta a0297
        lda #$00
        rts

```

#### **F0A4 SERIAL BUS IDLE**

This routine checks the RS232 bus for data transmission/reception. The routine waits for any activity on the bus to end before setting I.C.R. The routine is called by serial bus routines, since these devices use IRQ generated timing, and conflicts may occur if they are all used at once.

```

.f0a4  pha                ; store (A)
        lda $02a1        ; ENABL, RS232 enables

```

```

        beq $f0bb          ; bus not in use
.f0aa  lda $02a1          ; ENABL
        and #$03          ; test RS232
        bne $f0aa        ; yes, wait for port to clear
        lda #$10
        sta $dd0d        ; set up CIA#2 I.C.R
        lda #$00          ; clear
        sta $02a1        ; ENABL
.f0bb  pla                ; retrieve (A)
        rts

```

#### **F0BD TABLE OF KERNAL I/O MESSAGES 1**

This is a table of messages used by the KERNAL in conjunction with its I/O routines. Bit 7 is set in the last character in each message as a terminator.

The table is split into two parts in the JiffyDOS kernal, since the tape messages have been removed, and being substituted by new routines.

```

.f0bd  0d 49 2f 4f 20 45 52 52 4f 52 20 a3      ; i/o error #
.f0c9  0d 53 45 41 52 43 48 49 4e 47 a0        ; searching
.f0d4  46 4f 52 a0                             ; for

```

#### **F0D8 JIFFYDOS CLEAR SPRITES**

This routine is called by JiffyDOS before executing timecritical routines that might be messed up by sprites on the screen. A loop is performed afterwards that lets sprites currently being displayd on the screen, to be finished.

```

.f0d8  lda #$00
        sta $d015        ; clear sprites
.f0dd  adc #$01          ; perform loop
        bne $f0dd        ; 256 times
        rts

```

#### **FOE2 JIFFYDOS SET CHKIN**

This routine is a new JiffyDOS routine which clears all I/O and sets up the current JiffyDOS filenameumber as default inputdevice by calling CHKIN.

```

.f0e2  lda $9f          ; JiffyDOS Logical Filenameumber
.f0e4  pha              ; store (A)
        jsr $ffcc      ; CLRCHN
        pla              ; retrieve (A)
        tax              ; (A) to (X)
        jmp $ffc6      ; CHKIN, open channel for input

```

#### **FOED JIFFYDOS SERIAL OUTPUT 1**

This is a patch to the original Commodore KERNAL, that clears the flag that indicates a JiffyDOS device, (\$a3), before setting the serial output to 1.

```

.f0ed  lda #$00          ; clear JiffyDOS device flag
        sta $a3
        jmp $ee97        ; serial output 1

```

#### **FOF4 JIFFYDOS SEND DRIVE COMMAND**

This routine uses the values in (X) and (Y) to send a command to the drive. (X) contains a offset to the command, and (Y) contains the length of the command.

```

.f0f4  txa              ; temp store (X)
        pha
        jsr $f7a2      ; open command channel for output

```

```

        pla
        tax                ; retrieve (X)
.f0fb  lda $f398,x        ; read command from table
        jsr $ffd2         ; output character to drive
        inx                ; next character
        dey                ; decrement counter
        bne $f0fb         ; till ready
        rts

```

#### **F106 TABLE OF KERNAL I/O MESSAGES 2**

This is the second part of the KERNAL I/O message table. Part 1 is to be found at address \$f0bd.

```

.f106  0d 4c 4f 41 44 49 4e c7                ; loading
.f10e  0d 53 41 56 49 4e 47 a0                ; saving
.f116  0d 56 45 52 49 46 59 49 4e c7         ; verifying
.f120  0d 46 4f 55 4e 44 a0                  ; found
.f127  0d 4f 4b 9d                            ; ok

```

#### **F12B PRINT MESSAGE IF DIRECT**

This is a routine to output a message from the I/O messages table at \$f0bd. On entry, (Y) holds the offset to control which message is printed. The routine tests if we are in program mode or direct mode. If in program mode, the routine exits. Else, the routine prints character after character untill it reaches a character with bit7 set.

```

.f12b  bit $9d                ; MSGFLG, test if direct or program mode
        bpl $f13c            ; program mode, don't print message
.f12f  lda $f0bd,y           ; get output character from table
        php                  ; store processor registers
        and #$7f             ; clear bit7
        jsr $ffd2           ; output character using CHROUT
        iny                  ; increment pointer to next character
        plp                  ; retrieve message
        bpl $f12f           ; untill bit7 was set
.f13c  clc                    ; clear carry to indicate no error
        rts

```

#### **F13E GETIN: GET a BYTE**

The KERNAL routine GETIN (\$ffe4) is vectored to this routine. It load a character into fac#1 from the external device indicated by DFLTIN. Thus, if device = 0, GET is from the keyboard buffer. If device = 2, GET is from the RS232 port. If niether of these devices then GET is further handled by the next routine, INPUT.

```

.f13e  lda $99                ; DFLTIN, default input device.
        bne $f14a            ; not keyboard
        lda $c6              ; NDX, number of keys in keyboard queue
        beq $f155           ; buffer empty, exit
        sei                  ; disable interrupts
        jmp $e5b4           ; get character from keyboard buffer, and exit
.f14a  cmp #$02              ; RS232
        bne $f166           ; nope, try next device
.f14e  sty $97                ; temp store
        jsr $f086           ; get character from RS232
        ldy $97              ; retrieve (Y)
.f155  clc
        rts

```

#### **F157 CHRIN: INPUT A BYTE**

The KERNAL routine CHRIN (\$ffcf) is vectored to this routine. It is similar in function to the GET routine above, and also provides a continuation to that routine. If the input device is 0 or 3, ie. keyboard or screen, then input takes place from the screen. INPUT/GET from other devices are performed by calls to the next routine. Two bytes are input from the device so that end of file can be set if necessary (ie. ST = #40)

```
.f157  lda $99          ; DFLTn, default input
      bne $f1a9       ; not keyboard, next device
      lda $d3         ; PNTR, cursor column on screen
      sta $ca        ; >LXSP, cursor position at start
      lda $d6         ; TBLX, cursor line number
      sta $c9        ; <LXSP
      jmp $e632       ; input from screen or keyboard
.f166  cmp #$03       ; screen
      bne $f173       ; nope, next device
      sta $d0         ; CRSW, flag INPUT/GET from keyboard
      lda $d5         ; LNMx, physical screen line length
      sta $c8        ; INDx, end of logical line for input
      jmp $e632       ; input from screen of keyboard
.f173  bcs $f1ad       ; RS232
      cmp #$02        ;
      beq $f1b8       ; yes, get data from RS232 port
.f179  jsr $fbaa       ; JiffyDOS ACPTR, get byte from serial bus
      pha            ; temp store on stack
      bit $a3         ; test bit6, if serial device is a JiffyDOS device
      bvc $f19c       ; no JiffyDOS device
      cpx #$00
      bne $f187
.f187  lda $c4         ; ??????
      cmp #$04
      bcc $f19c
      ldy #$00        ; clear offset
      lda ($bb),y     ; FNADR, pointer to current filename
      cmp #$24        ; first character is $, ie. directory
      beq $f19c       ; yes, exit
      inc $b9         ; increment SA
      jsr $f38b       ; execute TALK, and TKSA
      dec $b9         ; decrement SA
      asl $a3
.f19c  pla
      rts
.f19e  lda #$10        ; set bit4
      jmp $felc       ; write to STATUS
```

### **F1A3 VECTOR TABLE**

The following table contains three vectors that is copied to \$0300 when the @X command is executed.

```
.f1a3  eb e3          ; IERROR vector
.f1a5  83 a4          ; IMAIN vector
.f1a7  7c a5          ; ICRNCH vector

      lda
      sed a90
if1ad  =*+01
      lda f90a5,y
```

### **F1AD GET FROM SERIAL/RS232**

These routines, actually two different, is entered from the previous routine. The serial section checks the state of ST. If zero, then the data is recieved from the bus, otherwise carriage return (#0d) is returned in (A). In the second section, the recieved byte is read from the RS232 port.

```
.f1ad  lda $90          ; STATUS, I/O status word
      beq $f1b5       ; status OK
.f1b1  lda #$0d       ; else return <CR> and exit
.f1b3  clc
.f1b4  rts
.f1b5  jmp $fbaa      ; JiffyDOS ACPTR, get byte from serial bus
.f1b8  jsr $f14e      ; receive from RS232
      bcs $f1b4       ; end with carry set
      cmp #$00
      bne $f1b3       ; end with carry clear
      lda $0297       ; RSSTAT, 6551 status register
      and #$6         ; mask
      bne $f1b1       ; return with <CR>
      beq $f1b8       ; get from RS232
```

### **F1CA CHROUT: OUTPUT ONE CHARACTER**

The KERNAL routine CHROUT (\$ffd2) is vectored to this routine. On entry, (A) must hold the character to be output. The default output device number is examined, and output directed to relevant device. The screen, serial bus and RS232 all use previously described routines for their output. Some old taperoutines have been removed in the middle of this routine, and been changed to a JiffyDOS routine.

```
.f1ca  pha          ; temp store on stack
      lda $9a       ; DFLTO, default output device
      cmp #$03      ; screen?
      bne $f1d5     ; nope, test next device
      pla          ; retrieve (A)
      jmp $e716     ; output to screen
.f1d5  bcc $f1db     ; device <3
      pla          ; retrieve (A)
      jmp $eddd     ; send serial deferred
.f1db  lsr a
      pla
      sta $9e       ; PTR1, some tape junk left in the code
      txa
      pha
      tya
      pha
      bcc $f208     ; RS232
      jmp $f3f1     ; output device not present

.f1e8  jsr $f8bf
      jsr $e4c6
      cmp #$30
      rts
```

### **F1F1 JIFFYDOS DEFAULT DEVICE**

The following routine sets the default device number. It uses the GTBYTC procedure to read the specified device number.

```
.f1f1  jsr $b79b     ; GTBYTC, read device number from keyboardbuffer
      stx $ba       ; store in FA, current device number
      jsr $f75c     ; test if device FA is present.
      stx $be       ; If OK, store
```

rts

### **F1FC CHROUT: PART 2**

This is the second part of the CHROUT routine. It contains the last parts of the RS232 output routine.

```
.f1fc  clc
      pla
      tay
      pla
      tax
      lda $9e          ; PTR1
      bcc $f207
      lda #$00
.f207  rts
.f208  jsr $f017       ; send to RS232
      jmp $f1fc       ; end output
```

### **F20E CHKIN: SET INPUT DEVICE**

The KERNAL routine CHKIN (\$ffc6) is vectored to this routine. On entry, (X) must hold the logical file number. A test is made to see if the file is open, or ?FILE NOT OPEN. If the file is not an input file then ?NOT INPUT FILE. If the device is on the serial bus then it is commanded to TALK and secondary address is sent. ST is then checked, and if non-zero, ?DEVICE NOT PRESENT. Finally, the device number is stored in DLFTN.

```
.f20e  jsr $f30f       ; find file number
      beq $f216       ; ok, skip next command
      jmp $f701       ; I/O error #3, file not open
.f216  jsr $f31f       ; set file variables
      lda $ba         ; FA, current device number
      beq $f233       ; keyboard
      cmp #$03        ; screen
      beq $f233       ; yes
      bcs $f237       ; larger than 3, serial bus device
      cmp #$02        ; RS232
      bne $f22a       ; nope
      jmp $f04d       ; input from RS232
.f22a  ldx $b9         ; SA, current secondart address
      cpx #$60
      beq $f233
      jmp $f70a       ; I/O error #6, not output file
.f233  sta $99         ; DFLTN, default input device
      clc
      rts
.f237  tax
      jsr $ed09       ; send TALK to serial device
      lda $b9         ; SA
      bpl $f245       ; send SA
      jsr $edcc       ; wait for clock
      jmp $f248
.f245  jsr $edc7       ; send talk secondary address
.f248  txa
      bit $90         ; STATUS, I/O status word
      bpl $f233       ; store DFLTN, and exit
      jmp $f707       ; I/O error #5, device not present
```

### **F250 CHKOUT: SET OUTPUT DEVICE**

The KERNAL routine CHKOUT (\$ffc9) is vectored to this routinr. On entry (X) must hold the logical filenameumber. A test is made to see if the file is

open, or ?FILE NOT OPEN error. If the device is 0, ie. the keyboard, or the file is not an output file, then ?FILE OUTPUT FILE error is generated. If the device is on the serial bus, then it commanded to LISTEN and the secondary address is sent. ST is then checked and if non-zero, then ?DEVICE NOT PRESENT error. Finally, the device number is stored in DFLTO.

```
.f350 jsr $f30f      ; fine file number (X)
      beq $f258      ; OK
      jmp $f701      ; I/O error #3, file not open
.f258 jsr $f31f      ; set file values
      lda $ba        ; FA, current device number
      bne $f262      ; not keyboard
.f25f jmp $f70d      ; I/O error #7, not output file
.f262 cmp #$03       ; screen?
      beq $f275      ; yes
      bcs $f279      ; serial bus device
      cmp #$02       ; RS232
      bne $f26f      ; nope
      jmp $efef1     ; submit to RS232
.f26f ldx $b9        ; SA, current secondary address
      cpx #$60
      beq $f25f      ; not output file error
.f275 sta $9a        ; DFLTO, default output device
      clc           ; clear carry to indicate no errors
      rts
.f279 tax           ; file (X) to (A)
      jsr $ed0c      ; send LISTEN to serial device
      lda $b9        ; SA
      bpl $f286      ; send SA
      jsr $edbe      ; clear ATN
      bne $f289
.f286 jsr $edb9      ; send listen secondary address
.f289 txa
      bit $90        ; STATUS, I/O status word
      bpl $f275      ; OK, set output device
      jmp $f707      ; I/O error #5, device not present
```

#### **F291 CLOSE: CLOSE FILE, PART 1**

The KERNAL routine CLOSE (\$ff3c) is vectored here. The file parameters are fetched, and if not found, the routine exits without any action. It checks the device number associated with the file. If it is RS232, then the RS232 port is reset. If it is a serial device, the device is UNTALKed, or UNLISTENed. Finally the number of open logical files are decremented, and the table of active file numbers are updated. On entry (A) holds the file number to close. Old tape routines (\$f2cc-\$f2e1) has been removed for new JiffyDOS routines.

```
.f291 jsr $f314      ; find logical file, (X) holds location i table
      beq $f298      ; OK
      clc           ; file not found
      rts           ; and exit
.f298 jsr $f31f      ; get file values from table, position (X)
      txa
      pha           ; temp store
      lda $ba        ; FA, current device number
      beq $f2f1      ; keyboard?, update file table
      cmp #$03       ; screen
      beq $f2f1      ; yep, update file table
      bcs $f2ee      ; Serial bus
      cmp #$02       ; RS232
      bne $f2c8      ; nope, serial
```

```

        pla                ; retriev (A)
        jsr $f2f2         ; remove entry (A) from file table
        jsr $f483         ; init RS232 port by using part of RS232OPEN
        jsr $fe27         ; MEMTOP, read top of memory (X/Y)
        lda $f8           ; >RIBUF, RS232 input buffer
        beq $f2ba
        iny
.f2ba   lda $fa           ; >ROBUF, RS232 output buffer
        beq $f2bf
        iny
.f2bf   lda #$0          ; Clear RS232 input/output buffers
        sta $f8
        sta $fa
        jmp $f47d         ; Set new ROBOF values and set new MEMTOP
.f2c8   pla                ; retriev (A)
        jmp $f713

.f2cc   jsr $ffcc         ; CLRCHN, close all channels
.f2cf   lda #$6f
        jsr $f314         ; FIND FILE, test if file number #$6f is open.
        bne $f30e         ; file not open, return
        jmp $f2f3         ; close file #$6f

```

#### **F2D9 JIFFYDOS TEST DEVICE**

The following routine tests if a device is present. On entry (X) holds the device to be tested. Open to the device is performed, and afterwards the statusword can be read for result.

```

.f2d9   stx $ba          ; store (X) in FA
.f2db   tya
        pha
        jsr $f8b2         ; open 15,x,15
        jsr $f7a2         ; set command channel (15) as output
        php
        jsr $f2cc         ; close command channel
        plp
        pla
        tay
        ldx $ba
        rts

        sed $

```

#### **F2e CLOSE: CLOSE FILE, PART 2**

```

.f2ee   jsr $f642         ; UNTALK/UNLISTEN serial device
.f2f1   pla
.f2f2   tax
.f2f3   dec $98          ; decrement LDTND, number of open files
        cpx $98          ; compare LDTND to (X)
        beq $f30d         ; equal, closed file = last file in table
        ldy $98          ; else, move last entry to position of closed entry
        lda $0259,y       ; LAT, active filenames
        sta $0259,x
        lda $0263,y       ; FAT, active device numbers
        sta $0263,x
        lda $026d,y       ; SAT, active secondary addresses
        sta $026d,x
.f30d   clc
.f30e   rts                ; return

```

### **F30F FIND FILE**

This routine finds a logical file from it's file number. On entry, (X) must hold the logical file number to be found. LAT, the table of file numbers is searched, and if found (X) contains the offset to the position of the file in the table, and the Z flag is set. If not found, Z=0.

```
.f30f  lda #$00
      sta $90          ; clear STATUS
      txa              ; file number to search for
.f314  ldx $98         ; LDTND, number of open files
.f316  dex
      bmi $f32e        ; end of table, return
      cmp $0259,x      ; compare file number with LAT, table of open files
      bne $f316        ; not equal, try next
      rts              ; back with Z flag set
```

### **F31F SEET FILE VALUES**

This routine sets the current logical file number, device number and secondary address from the file parameter tables. On entry (X) must hold the offset to the position of the file in the table.

```
.f31f  lda $0259,x    ; LAT, table of active logical files
      sta $b8         ; store in LA
      lda $0263,x    ; FAT, table of active device numbers
      sta $ba         ; store in FA
      lda $026d,x    ; SAT, table of active secondary addresses
      sta $b9         ; store in SAT
.f32e  rts            ; return
```

### **F32F CLALL: ABORT ALL FILES**

The KERNAL routine CLALL (\$ffe7) is vectored here. The number of open files are set to zero, and the next routine is performed.

```
.f32f  lda #$00
      sta $98          ; clear LDTND, no open files
```

### **F333 CLRCHN: RESTORE TO DEFAULT I/O**

The KERNAL routine CLRCHN (\$ffcc) is vectored here. The default output device is UNLISTENed, if it is on the serial bus, and the default output is set to the screen. The default input device is UNTALKed, if it is on the serial bus, and the default input device is set to keyboard.

```
.f333  ldx #$03        ; check if device > 3 (serial bus is 4,5...)
      cpx $9a         ; test DFLTO, default output device
      bcs $f33c        ; nope, no serial device
      jsr $edfe        ; send UNLISTEN to serial bus
.f33c  cpx $99         ; test DFLTI, default input device
      bcs $f343        ; nope, no serial device
      jsr $edef        ; send UNTALK to serial bus
.f343  stx $9a         ; store screen as DFLTO
      lda #$00
      sta $99         ; store keyboard as DFLTI
      rts
```

### **F34A OPEN: OPEN FILE**

The KERNAL routine OPEN (\$ffc0) is vectored here. The file parameters must be set before entry. The routine reads the LAT, to see if file already exists, which will result in I/O error #2, ?FILE OPEN. A test is made to see if more than 10 files are open. If so, I/O error #1, ?TOO MANY FILES, will occur. The file parameters are set, and put in their respective tables. The device number is checked, and each kind of device jumps to

their own routine. Keyboard and screen will exit here with no further actions. RS232 is opened via a separate routine. SA, secondary address, and filename will be sent on the serial bus. Some tape routines are removed, and replaced with JiffyDOS code.

```
.f34a  ldx $b8          ; LA, current logical number
      bne $f351
      jmp $f70a       ; I/O error #6, not input file
.f351  jsr $f30f       ; find file (X)
      bne $f359
      jmp $f6fe       ; I/O error #2, file exists
.f359  ldx $98         ; LDTND, number of open files
      cpx #$0a        ; more than ten
      bcc $f362       ; nope
      jmp $f6fb       ; I/O error #1, too many files
.f362  inc $98         ; increment LDTND
      lda $b8         ; LA
      sta $0259,x     ; store in LAT, table of active file numbers
      lda $b9         ; SA
      ora #$60        ; fixx
      sta $b9         ; store in SA
      sta $026d,x     ; store in SAT, table of active secondary addresses
      lda $ba         ; FA
      sta $0263,x     ; store in FAT, table of active device numbers
      beq $f3d3       ; keyboard, end
      cmp #$03        ; screen
      beq $f3d3       ; yep, end
      bcc $f384       ; less than 3, not serial bus
      jsr $f3d5       ; send SA
      bcc $f3d3       ; end
.f384  cmp #$01        ; TAPE
      beq $f3f3       ; I/O error #5, device not present
      jmp $f409       ; open RS232 file
```

### **F38B JIFFYDOS TALK & TKSA**

This is a routine used by JiffyDOS to untalk device (A), then TALK and TKSA is executed to current device with current secondary address.

```
.f38b  jsr $ffab      ; UNTALK
      lda $ba         ; FA, current device number
      jsr $ffb4       ; TALK
      lda $b9         ; SA, current secondary address
      jmp $ff96       ; TKSA, send SA after TALK
```

### **F398 JIFFYDOS DIRECT DRIVE COMMANDS**

The following text/code is used to transfer, and is transferred to a selected drive. The first section is a \$22 byte long block used by the lock/unlock a file. The second section is code to execute a drive program at \$0600. The third section sets a byte in the drive memory to control the interleave. The fourth section sets a byte in the drive memory to control the 1541 head rattle.

```
.f398  4d 2d 57 00 06 1c ; M-W 00 06 1c, ie. write $1c bytes to $0600
      lda $0261         ; the following code is transferred to the drive
      sta $07           ; at $0600
      lda #$12
      sta $06
      ldx #$00
      stx $f9
      jsr $d586
      ldy $0267
```

```

        lda ($30),y
        eor #$40
        sta ($30),y
        jmp $d58a
.f3b0  4d 2d 45 00 06      ; M-E 00 06, ie. a memory execute at $0600
.f3b6  4d 2d 57 6a 00 01 ; M-W 6a 00 01, ie. memory write one byte at
$006a
.f3bc  4d 2d 57 69 00 01 ; M-W 69 00 01, ie. memory write one byte at
$0069

        ora (p50,x)
        ror a01
        sed $
        rol $
        sta aa6

if3d3  clc
        rts

```

### **F3D5 SEND SA**

This routine exits if there is no secondary address or filename specified. The I/O status word, ST, is reset, and the serial device is commanded to LISTEN. A check is made for a possible ?DEVICE NOT PRESENT error. Finally, the filename is sent to the device.

```

.f3d5  lda $b9           ; SA, current secondary address
        bmi $f3d3       ; exit
        ldy $b7         ; FNLEN, length of filename
        beq $f3d3       ; exit
        lda #$00
        sta $90         ; clear STATUS, I/O status word
        lda $ba         ; FA, current device number
        jsr $ed0c       ; send LISTEN to serial bus
        lda $b9         ; SA
        ora #$f0
        jsr $edb9       ; send LISTEN SA
        lda $90         ; STATUS
        bpl $f3f6       ; ok
.f3f1  pla              ; remove two stack entries for RTS command
        pla
.f3f3  jmp $f707       ; I/O error #5, device not present
.f3f6  lda $b7         ; FNLEN
        beq $f406       ; unlisten and exit
        ldy #$00         ; clear offset
.f3fc  lda ($bb),y     ; FNADR, pointer to filename
        jsr $eddd       ; send byte on serial bus
        iny             ; next character
        cpy $b7         ; until entire filename is sent
        bne $f3fc       ; again
.f406  jmp $f654       ; unlisten and exit

```

### **F409 OPEN RS232**

```

.f409  jsr $f483
        sty a0297
.f40f  cpy ab7
        beq if41d
        lda (pbb),y
        sta f0293,y
        iny
        cpy #$04

```

```

        bne if40f
if41d   jsr eef4a
        stx a0298
        lda a0293
        and #$0f
        beq if446
        asl a
        tax
        lda a02a6
        bne if43a
        ldy ffec1,x
        lda ffec0,x
        jmp if440
if43a   ldy fe4eb,x
        lda fe4ea,x
if440   sty a0296
        sta a0295
if446   lda a0295
        asl a
        jsr eff2e
        lda a0294
        lsr a
        bcc if45c
        lda add01
        asl a
        bcs if45c
        jsr if00d
if45c   lda a029b
        sta a029c
        lda a029e
        sta a029d
        jsr efe27
        lda af8
        bne if474
        dey
        sty af8
        stx af7
if474   lda afa
        bne if47d
        dey
        sty afa
        stx af9
if47d   sec
        lda #$f0
        jmp efe2d
if483   lda #$7f
        sta add0d
        lda #$06
        sta add03
        sta add01
        lda #$04
        ora add00
        sta add00
        ldy #$00
        sty a02a1
        rts

```

#### **F49E LOAD: LOAD RAM**

The kernal routine LOAD (\$ffd5) is vectoed here. If a relocated load is desired, then the start address is set in MEMUSS. The load/verify flag is

set, and the I/O status word is reset. A test is done on the device number, less than 3 results in illegal device number.

```
.f49e stx $c3          ; MEMUSS, relocated load address
      sty $c4
      jmp ($0330)     ; ILOAD vector. Points to $f4a5
.f4a5 sta $93          ; VRECK, load/verify flag
      lda #$00
      sta $90         ; clear STATUS, I/O status
      lda $ba         ; get FA, current device
      bne $f4b2       ; keyboard
.f4af jmp $f713       ; I/O error #9, illegal device
.f4b2 cmp #$03        ; screen?
      beq $f4af       ; yes, illegal device
```

#### **F4B8 LOAD FROM SERIAL BUS**

A filename is assumed by the routine, and if not present, a jump is made to a new JiffyDSO routine that sets filename to ':\*'. The message 'SEARCHING' is printed and the filename is sent with the TALK command and secondary address to the serial bus. If EOI occurs at this point, then ?FILE NOT FOUND is displayed. The message 'LOADING' or 'VERIFYING' is output and a loop is entered, which receives a byte from the serial bus, checks the <STOP> key and either stores the received byte, or compares it to the memory, depending on the state of VERCK. Finally the bus is UNTALKed.

```
.f4b8 bcc $f4af       ; device < 3, eg tape or RS232, illegal device
      ldy $b7        ; FNLEN, length of filename
      bne $f4bf       ; if length not is zero
      jmp $f659       ; fixx filename, JiffyDOS patch
.f4bf ldx $b9         ; SA, current secondary address
      jsr $f5af       ; print "SEARCHING"
      lda #$60
      sta $b9         ; set SA to $60
      jsr $f3d5       ; send SA and filename
      lda $ba         ; FA, current devicenumbr
      jsr $ed09       ; send TALK to serial bus
      lda $b9         ; SA
      jsr $edc7       ; send TALK SA
      jsr $ee13       ; receive from serial bus
      sta $ae         ; load address, <EAL
      lda $90         ; check STATUS
      lsr a
      lsr a
      bcs $f530       ; EOI set, file not found
      jsr $f179       ; receive from serial bus
      sta $af         ; load address, >EAL
      txa             ; retrieve SA and test relocated load
      bne $f4f0       ;
      lda $c3         ; use MEMUSS as load address
      sta $ae         ; store in <EAL
      lda $c4
      sta $af         ; store in >EAL
.f4f0 jmp $fac4       ; jump to JiffyDOS patch
.f4f3 jsr $ffe1       ; scan <STOP>
      bne $f4fb       ; not stopped
      jmp $f633
.f4fb jsr $fbaa       ; JiffyDOS ACPTR, receive from serial bus
      lda $90         ; read ST
      and #$fd        ; mask %11111101
      cmp $90
      sta $90
```

```

        bne $f4f3          ; EOI set
        ldy #$00
        ldx $a3
        lda $a4           ;
        cpy $93           ; VERIFY eller LOAD
        beq $f51a         ; jump to LOAD
        cmp ($ae),y       ; compare with memory
        beq $f51c         ; verified byte OK
        jsr $f19e         ;
        .byte $2c         ; mask next write command
.f51a  sta ($ae),y       ; store in memory
.f51c  stx $a3
        inc $ae           ; increment <EAL, next address
        bne $f524         ; skip MSB
        inc $af           ; increment >EAL
.f524  bit $90           ; test STATUS
        bvc $f4f3         ; get next byte
        jsr $edef         ; send UNTALK to serial bus
        jsr $f642
        bcc $f5a9         ; end routine
.f530  jmp $f704         ; I/O error #4, file not found

```

### **F533 JIFFYDOS @ COMMAND**

The following routine executes the @ command. First it tests if additional parameters are entered.

```

.f533  lda $b7           ; FNLEN, length of current filename
        beq $f546         ; no filename
        lda ($bb),y       ; test filename for
        cmp #$24          ; $, directory
        beq $f56c         ;
        jmp $fc9a         ; else goto

```

### **F540 JIFFYDOS LIST ASCII FROM DISK**

This routine lists an ascii file from disk. It reads one block of text from the disk (254 bytes) into the filename area. The text is then output using the 'print filename' routine.

```

.f540  tya               ; (Y) contains the command number
        pha               ; store on stack
        jsr $f8bf         ; open file with current parameters
        pla               ; retrieve
.f546  sta $a6           ; store
.f548  jsr $f911         ; input characters to buffer (filename area)
        bne $f568         ; exit if errors occurred
        lda $a6           ; get command number, should be $0f
        php
        beq $f557
        jsr $e4c6         ; input byte from command channel
        beq $f567         ; if byte =# then exit
.f557  jsr $f79a
        jsr $f5c1         ; print filename, ie. the input buffer
        bit $91           ; STKEY FLAG, test if <STOP> is pressed
        bpl $f567         ; exit
        plp
        bne $f548
        bvc $f548
        .byte $24         ; mask one byte, ie. PLP command
.f567  plp
.f568  rts

```

### **F569 JIFFYDOS BASIC DISC LIST**

The following routine reads the specified basic-file from disk and displays it to the screen. The entrypoint at \$f56c is used for showing the directory. First, the routine opens the file specified. IERROR vector is changed to \$f739, so a RTS command will be performed when a error occurs. Then the start address is read, and thrown away. A loop is performed that reads one block of bytes from the disk and is output through the basic LIST routine. On exit, the IERROR vector is restored.

```
.f569  ldx #$6c          ; get byte for SA, list basic program
      .byte $2c        ; mask next 2 bytes
.f56c  ldx #$60          ; get byte for SA, list directory
      jsr $f8c1        ; open file with current parameters
      lda #$39         ; setup JiffyDOS IERROR vector to point to
      sta $0300        ; $f739, a RTS-command
      ldy #$fc         ; set up (Y) pointer to 252
      jsr $fca6        ; read two garbage bytes (program start address)
.f57b  ldy #$00         ; set up (Y) pointer to 0
.f57d  jsr $fca6        ; read 254 bytes, store in input buffer
      bvs $f5a3        ; EOI, exit
      cpy #$02         ; if (Y) = 2
      beq $f5a3        ; exit
      cpy #$06
      bcc $f57d
      ldx $bb          ; read FNADR pointer, vector to input buffer
      stx $5f          ; store in temp vector
      ldx $bc
      stx $60
      ldy #$01
      sta ($5f),y
      jsr $a6c3        ; use part of LIST routine to output text
      jsr $f79a
      jsr $a6d4
      bit $91          ; STKEY FLAG, stop key
      bmi $f57b        ; not pressed, continue
.f5a3  lda #$63         ; restore JiffyDOS IERROR vector to $f763
      sta $0300        ; IERROR VEC
      rts
```

### **F5A9 LOAD END**

This is the last part of the loader routine which sets the (X/Y) register with the endaddress of the loaded program, clears carry and exit.

```
.f5a9  clc
      ldx $ae
      ldy $af
      rts
```

### **F5AF PRINT "SEARCHING"**

If MSGFLG indicates program mode then the message is not printed, otherwise the message "SEARCHING" is printed from the KERNAL I/O message table. If the length of filename > 0 then the message "FOR" is printed, and the routine drops through to print the filename.

```
.f5af  lda $9d          ; MSGFLG, direct or program mode?
      bpl $f5d1        ; program mode, don't print, exit
      ldy #$0c
      jsr $f12f        ; print "SEARCHING"
      lda $b7          ; FNLEN, length of current filename
      beq $f5d1        ; no name, exit
      ldy #$17
```

```
jsr $f12f      ; print "FOR"
```

### **F5C1 PRINT FILENAME**

Filename is pointed to by FNADR, and length in FNLEN. The KERNAL routine CHROUT is used to print filename.

```
.f5c1  ldy $b7          ; FNLEN, length of current filename
        beq $f5d1       ; exit
        ldy #$00
.f5c7  lda ($bb),y      ; get character in filename
        jsr $ffd2       ; output
        iny             ; next character
        cpy $b7         ; ready?
        bne $f5c7
.f5d1  rts             ; back
```

### **F5D2 PRINT "LOADING/VERIFYING"**

The load/verify flag is checked, and if the message to be output is flagged according to the result. This message is printed from the KERNAL I/O messages table.

```
.f5d2  ldy #$49        ; offset to verify message
        lda $93         ; VERCK, load/verify flag
        beq $f5da       ; verify
        ldy #$59        ; offset to load message
.f5da  jmp $f12b        ; output message flagged by (Y)
```

### **F5DD SAVE: SAVE RAM**

The KERNAL routine SAVE (\$ffd8) jumps to this routine. On entry, (X/Y) must hold the end address+1 of the area of memory to be saved. (A) holds the pointer to the start address of the block, held in zeropage. The current device number is checked to ensure that it is neither keyboard (0) or screen (3). Both of these result in ?ILLIGAL DEVICE NUMBER.

```
.f5dd  stx $ae         ; EAL , end address of block +1
        sty $af
        tax            ; move start pointer to (X)
        lda $00,x
        sta $c1        ; STAL, start address of block
        lda $01,x
        sta $c2
        jmp ($0332)    ; vector ISAVE, points to $f5ed
.f5ed  lda $ba         ; FA, current device number
        bne $f5f4      ; ok
.f5f1  jmp $f713       ; I/O error #9, illegal device number
.f5f4  cmp #$03        ; screen?
        beq $f5f1      ; yep, output error
        bcc $f5f1      ; less than 3, ie. tape, output error
```

### **F5FA SAVE TO SERIAL BUS**

A filename is assumed by the routine, or ?MISSING FILENAME error is called. The serial device is commanded to LISTEN, and the filename is sent along with the secondary address. The message 'SAVING' is printed, and a loop sends a byte to the serial bus and checks <STOP> key until the whole specified block of memory has been saved. Note that the first two bytes sent are the start address of the block. Finally the serial bus is UNLISTENed.

```
.f5fa  lda #$61
        sta $b9        ; set SA, secondary address, to #1
        ldy $b7        ; FNLEN, length of current filename
```

```

        bne $f605          ; ok
.f602  jmp $f710          ; I/O error #8, missing filename
.f605  jsr $f3d5          ; send SA & filename
        jsr $f68f          ; print 'SAVING' and filename
        lda $ba           ; FA, current device number
        jsr $ed0c          ; send LISTEN
        lda $b9           ; SA
        jsr $edb9          ; send LISTEN SA
        ldy #$00
        jsr $fb8e          ; reset pointer
        lda $ac           ; SAL, holds start address
        jsr $eddd          ; send low byte of start address
        lda $ad
        jsr $eddd          ; send high byte of start address
.f624  jsr $fcd1          ; check read/write pointer
        bcs $f63f
        lda ($ac),y       ; get character from memory
        jsr $eddd          ; send byte to serial device
        jsr $ffe1          ; test <STOP> key
        bne $f63a         ; not pressed
.f633  jsr $f642          ; exit and unlisten
        lda #$00          ; flag break
        sec
        rts
.f63a  jsr $fcdb          ; bump r/w pointer
        bne $f624         ; save next byte
.f63f  jsr $edfe          ; send UNLISTEN
.f642  bit $b9           ; SA
        bmi $f657
        lda $ba           ; FA
        jsr $ed0c          ; send LISTEN
        lda $b9
        and #$ef
        ora #$e0
        jsr $edb9          ; send UNLISTEN SA
.f654  jsr $edfe          ; send UNLISTEN
.f657  clc
        rts

```

#### **F659 JIFFYDOS DEFAULT FILENAME**

The following routine is executed when a missing filename is detected in the original loader routine. If so, the filename is set to ':\*', wildcard filename. On exit, a jump is made to the original loader with new filename parameters set.

```

.f659  lda $c6           ; NDX, number of characters in keyboard buffer
        beq $f602          ; if zero, output missing filename error
        lda #$02          ; store $02
        sta $b9           ; in SA, default secondary address
        ldx #$74          ; set up filename pointer to $f674
        ldy #$f6          ; ie. ':'
        jsr $ffbd          ; SETNAM
        jmp $f4bf          ; back to loader routine

        ldx #$33          ; offset
        ldy #$04          ; length
        jmp $f932          ; drive command

```

#### **F672 JIFFYDOS FUNKTION KEYS**

The following table contains the strings copied to the keyboard buffer when the funktionkeys are pressed. This table is pointed to by the FNKVEC at \$b0/\$b1. The strings are seperated by a zero-byte.

```
.f672 40 24 3a 2a 0d 00 ; F1 = '@$:*'
.f678 2f 00 ; F3 = '/'
.f67a 5e 00 ; F5 = arrow up
.f67c 25 00 ; F7 = '%'
.f67e 40 44 00 ; F2 = '@d'
.f681 40 54 00 ; F4 = '@t'
.f684 5f 00 ; F6 = arrow left
.f686 40 20 20 22 53 3a 00 ; F8 = '@ "S:'
```

```
clc
if68e rts
```

#### **F68F PRINT 'SAVING'**

MSGFLG is checked, and if direct mode is on, then the message 'SAVING' is flagged and printed from the KERNAL I/O message table.

```
.f68f lda $9d ; MSGFLG
      bpl $f68e ; not in direct mode, exit
      ldy #$51 ; offset to message in table
      jsr $f12f ; output 'SAVING'
      jmp $f5c1 ; output filename
```

#### **F69B UDTIM: BUMP CLOCK**

The KERNAL routine UDTIM (\$ffea) jumps to this routine. The three byte jiffy clock in RAM is incremented. If it has reached \$4f1a01, then it is reset to zero. this number represents 5184001 jiffies (each jiffy is 1/60 sec) or 24 hours. finally, the next routine is used to log the CIA key reading.

```
.f69b ldx #$00
      inc $a2 ; low byte of jiffy clock
      bne $f6a7
      inc $a1 ; mid byte of jiffy clock
      bne $f6a7
      inc $a0 ; high byte of jiffy clock
.f6a7 sec
      lda $a2 ; subtract $4f1a01
      sbc #$01
      lda $a1
      sbc #$1a
      lda $a0
      sbc #$4f
      bcc $f6bc ; and test carry if 24 hours
      stx $a0 ; yepp, reset jiffy clock
      stx $a1
      stx $a2
```

#### **F6BC LOG CIA KEY READING**

This routine tests the keyboard for either <STOP> or <RVS> pressed. If so, the keypress is stored in STKEY.

```
.f6bc lda $dc01 ; keyboard read register
      cmp $dc01
      bne $f6bc ; wait for value to settle
      tax
      bmi $f6da
```

```

        ldx #$bd
        stx $dc00          ; keyboard write register
.f6cc  ldx $dc01          ; keyboard read register
        cpx $dc01
        bne $f6cc          ; wait for value to settle
        sta $dc00
        inx
        bne $f6dc
.f6da  sta $91            ; STKEY, flag STOP/RVS
.f6dc  rts

```

#### **F6DD RDTIM: GET TIME**

The KERNAL routine RDTIM (\$ffde) jumps to this routine. The three byte jiffy clock is read into (A/X/Y) in the format high/mid/low. The routine exits, setting the time to its existing value in the next routine. The clock resolution is 1/60 second. SEI is included since part of the IRQ routine is to update the clock.

```

.f6dd  sei                ; disable interrupt
        lda $a2            ; read TIME
        ldx $a1
        ldy $a0

```

#### **F6E4 SETTIM: SET TIME**

The KERNAL routine SETTIM (\$ffdb) jumps to this routine. On entry, (A/X/Y) must hold the value to be stored in the clock. The format is high/mid/low, and clock resolution is 1/60 second. SEI is included since part of the IRQ routine is to update the clock.

```

.f6e4  sei                ; disable interrupt
        sta $a2            ; write TIME
        stx $a1
        sty $a0
        cli                ; enable interrupts
        rts

```

#### **F6ED STOP: CHECK <STOP> KEY**

The KERNAL routine STOP (\$ffe1) is vectored here. If STKEY =#7f, then <STOP> was pressed and logged whilst the jiffy clock was being updated, so all I/O channels are closed and the keyboard buffer reset.

```

.f6ed  lda $91            ; STKEY
        cmp #$7f          ; <STOP> ?
        bne $f6fa          ; nope
        php
        jsr $ffcc          ; CLRCHN, close all I/O channels
        sta $c6            ; NDX, number of characters in keyboard buffer
        plp
.f6fa  rts

```

#### **F6FB OUTPUT KERNAL ERROR MESSAGES**

The error message to be output is flagged into (A) depending on the entry point. I/O channels are closed, and then if KERNAL messages are enabled, "I/O ERROR #" is printed along with the error number.

```

.f6fb  lda #$01            ; error #1, too many files
        .byte $2c
.f6fe  lda #$02            ; error #2, file open
        .byte $2c
.f701  lda #$03            ; error #3, file not open
        .byte $2c

```

```

.f704  lda #$04          ; error #4, file not found
      .byte $2c
.f707  lda #$05          ; error #5, device not found
      .byte $2c
.f70a  lda #$06          ; error #6, not input file
      .byte $2c
.f70d  lda #$07          ; error #7, not output file
      .byte $2c
.f710  lda #$08          ; error #8, missing filename
      .byte $2c
.f713  lda #$09          ; error #9, illegal device number
      pha
      jsr $ffcc          ; CLRCHN, close all I/O channels
      ldy #$00
      bit $9d            ; test MSGFLAG, KERNAL messages enabled
      bvc $f729          ; no
      jsr $f12f          ; print "I/O ERROR #"
      pla
      pha
      ora #$30           ; convert (A) to ASCII number
      jsr $ffd2          ; use CHROUT to print number in (A)
.f729  pla
      sec
      rts

```

#### **F72C TEST JIFFY COMMAND**

This routine test the character in the current key in the buffer if it is a JiffyDOS command character. Output from this routine is (Y) which contains the value of the selected command. (Y)=\$ff if no command was found.

```

.f72c  ldy #$0c          ; number of characters to test
      jsr $79            ; CHARGOT, read current character in buffer again
.f731  cmp $f7dd,y       ; equal to byte in JiffyDOS command tab
      beq $f739          ; yepp, return
      dey               ; test next
      bpl $f731          ; till (Y)=$ff
.f739  rts               ; back

```

#### **F73A JIFFYDOS SLPARA**

This routine is executed from the original SLPARA. It executes SETLFS to set logical file parameters, as normal. But it also continues through the next routine to find a present device number.

```

.f73a  jsr $ffba          ; SETLFS

```

#### **.F73D JIFFYDOS TEST SERIAL DEVICE**

This routine tests a serial disk device number to see if it is present. The routine uses \$be as a internal counter for device number. A test is performed to make sure that the device number is within its limits, \$08-\$1f. If a device is not present, the routine continues searching for a present device. The second time we reset the counter to \$08 (after reaching \$1f) without finding a device, the routine exits with error #5, device not present.

```

.f73d  clc               ; clear carry
      php                ; store carry
      ldx $be            ; internal counter for devicenumbr
      cpx #$08           ; device $8
      bcc $f749          ; less than $8
.f745  cpx #$1f          ; serial device must be less than $1f (31)

```

```

        bcc $f750          ; less than $1f
.f749  plp                ; if carry set, this is second time
        bcs $f761          ; do error
        sec                ; set carry to indicate first reset
        php                ; store carry
        ldx #$08           ; start at $08 again
.f750  stx $be            ; store
        jsr $f2d9          ; test devicenummer (X)
        bcc $f75a          ; OK, device (X) is next present device
        inx                ; next devicenummer
        bne $f745          ; test next
.f75a  pla                ; clean ut stack
.f75b  rts                ; exit

.f75c  jsr $f2db          ; test devicenummer in FA
        bcc $f75b          ; ok
.f761  ldx #$05           ; ERROR, device not present

```

### **F763 IERROR: JIFFYDOS ERROR ROUTINE**

The ERROR vector IERROR (\$0300) points to this routine. On entry (X) holds the error number. A test is done to see if this is a SYNTAX error (\$0b). If not, it jumps to the original IERROR at \$e38b, where errors are taken care of as usual. Else, the routine continues by checking if the error was caused by a JiffyDOS command.

```

.f763  cpx #$0b           ; SYNTAX ERROR
        beq $f76a          ; yes, jump to command test
.f767  jmp $e38b          ; nope, normal error handler

```

### **F76A COMMAND: TEST FOR EXTRA JIFFYDOS COMMANDS**

The following routine tests if a JiffyDOS command has been entered. A subroutine is called to test this, and it leavs the JiffyDOS command number in (Y), if any found. It tests for a present serial device,

```

.f76a  jsr $f72c          ; test JiffyDOS command. On exit, (Y)=command number
        bne $f767          ; no JiffyDOS command
        sty $27            ; temp store
        tax
        bmi $f776
        pla
        pla
.f776  jsr $f73d          ; test serial device, if any present
        jsr $f838          ; command after '@'? Setname and open specified file
        lda $27            ; retrieve temp, command number.
        ldy #$00
        asl a              ; times 2
        tax                ; to (X)
        lda $f7f5,x        ; get low commandvector
        sta $55            ; store
        lda $f7f6,x        ; get high commandvector
        sta $56            ; store
.f78c  jsr $54            ; execute JiffyDOS command
        jsr $a8f8          ; ignore next statement, sort of REM
        jsr $f2cc          ; close all channels, and file 15 if open
        lda $9f            ; JiffyDOS default filenameumber
        jsr $ffc3          ; CLOSE
.f79a  jsr $ffcc          ; CLRCHN, close all I/O channels
        ldx $13            ; CHANNL
        beq $f75b          ; screen and keyboard are current I/O device, exit
        .byte $2c          ; else mask next LDX-command, and perform CHKOUT

```

```
.f7a2  ldx #$6f          ; command channel
       jmp $ffc9        ; CHKOUT, open channel for output.
```

#### **F7A7 JIFFYDOS ML-LOAD**

This is the entrypoint for £ and %, which loads machine language

```
.f7a7  tya
       iny
       .byte $2c        ; bit $xxxx, trick to skip 2 commands
```

#### **F7AA JIFFYDOS VERIFY**

This is the entrypoint for ` , which verifies a file

```
.f7aa  tya
```

#### **F7AB JIFFYDOS BASIC-LOAD**

This is the entrypoint for / and 'arrow up' which loads a basic program. The LOAD/VERIFY is performed. Depending on what command is executed, various end routines are performed.

```
.f7ab  iny
       sty $b9          ; SA, current secondary address
       ldx $2b          ; TXTTAB, start of basic
       ldy $2c
       jsr $ffd5        ; LOAD
       bcc $f7c0        ; load OK
       jmp $e0f9        ; handle I/O error
.f7ba  jmp $e195        ; load OK?
.f7bd  jmp $e17e        ; verify OK?
.f7c0  lda $27          ; test command number
       cmp #$0b         ; verify command (`)
       beq $f7bd        ; output verify OK
       bcs $f78c        ; command number larger than $0b
       cmp #$08         ; load ml (%)
       beq $f75b        ; if so exit
       bcc $f7ba        ; if command number less than 8, test if OK and exit
       stx $2d          ; VARTAB, set start of Basic variables
       sty $2e
       pla              ; remove RTS return address
       pla
       jsr $aad7        ; output CR/LF
       jsr $a533        ; rechain basic lines
       jmp $a871        ; perform RUN
```

#### **F7DD JIFFYDOS COMMAND TAB**

The tab contains the additional JiffyDOS commands. The \$0c first commands can be entered at the prompt, and are tested at \$f72c. The remaining commands must be entered after the @-character. The DOS 5.1 Wedge Commands are not checked here.

```
.f7dd  40              ; @
.f7de  5f              ; <-
.f7df  2a              ; *
.f7e0  ac              ; dot in lower right corner. (Same as *. Possible
                       ; future expansion)
.f7e1  22              ; "
.f7e2  12              ; . (Same as ". Possible future expansion)
.f7e3  2f              ; /
.f7e4  ad              ; right angle in top right corner. (Same as /.
Possible future expansion)
.f7e5  25              ; %
```

```
.f7e6 5e          ; arrow up
.f7e7 ae          ; right angle in lower left corner. (Same as 'arrow
up'. Possible future expansion)
.f7e8 27          ; ´
.f7e9 5c          ; £
```

The following command characters must be entered after the @-character.

```
.f7ea 44          ; D
.f7eb 4c          ; L
.f7ec 54          ; T
.f7ed 23          ; #
.f7ee 42          ; B
.f7ef 46          ; F
.f7f0 4f          ; O
.f7f1 50          ; P
.f7f2 51          ; Q
.f7f3 58          ; X
.f7f4 47          ; G
```

#### **F7F5 JIFFYDOS COMMAND VECTORS**

The following table contains the JiffyDOS command vectors. The vectors are in the same order as the command characters above.

```
.f7f5 33 f5          ; execute @ at $f533
.f7f7 59 e1          ; execute <- at $e159
.f7f9 39 fa          ; execute * at $fa39
.f7fb 39 fa          ; execute XX at $fa39
.f7fd 2b f7          ; execute " at $f72b
.f7ff 2b f7          ; execute . at $f72b
.f801 ab f7          ; execute / at $f7ab
.f803 ab f7          ; execute XX at $f7ab
.f805 a7 f7          ; execute % at $f7a7
.f807 ab f7          ; execute 'arrow up' at $f7ab
.f809 ab f7          ; execute XX at $f7ab
.f80b aa f7          ; execute ´ at $f7aa
.f80d a7 f7          ; execute £ at $f7a7
```

The following commands are extra JiffyDOS commands and to come after the @-character

```
.f80f 69 f5          ; execute D at $f569
.f811 d4 f8          ; execute L at $f8d4
.f813 40 f5          ; execute T at $f540
.f815 f1 f1          ; execute # at $f1f1
.f817 2c f9          ; execute B at $f92c
.f819 c2 e4          ; execute F at $e4c2
.f81b 25 f8          ; execute O at $f825
.f81d 97 fa          ; execute P at $fa97
.f81f bc fc          ; execute Q at $fc2c
.f821 a0 fc          ; execute X at $fca0
.f823 24 f9          ; execute G at $f924
```

#### **F825 JIFFYDOS OLD**

The following routine performs a basic old after a new or reset. The routine performs a rechain to set up correct pointers etc.

```
.f825 iny
      tya
      sta ($2b),y      ; store XX in $08XX to reinit basic
      jsr $a533        ; LINKPRG, rechain basic lines
```

```

txa
adc #$02
tax
lda $23
adc #$00          ; (X) and (Y) contains start of variables
tay
jmp $e1a7        ; set start of variables, and restart basic.

```

### **F838 JIFFYDOS COMMAND PART 2**

This routine is called from the JiffyDOS COMMAND routine and make a test for additional command characters after the '@' character. Only the command number \$0d-\$17 is tested. If text after '@' is not a JiffyDOS command (ie. a normal DOS command', or JiffyDOS command number less than \$10, a filename is expected. Tests are made for colon and quotes, the filename is evaluated, and parts of the OPEN/CLOSE routine is used to SETNAM. A test is made for additional device number after a comma. A free line on the screen is found, and some string-house keeping is done. Finally, the routine continues through to the next routine to open the command channel.

```

.f838  tya
       bne $f853
       sta $b7
.f83d  jsr $73          ; CHRGET, get character from buffer
       beq $f887        ; terminator found, exit
       ldy #$17         ; set pointer for start of command
       jsr $f731        ; test if character is JiffyDOS command
       bne $f858        ; nope, no command
       cpy #$0d         ; only test value $17 to $0d
       bcc $f858        ; if less than $0d, exit
       sty $27          ; temp store
       cpy #$10         ; read command value
       bcs $f887        ; if value larger than $10, filename is not expected
.f853  lda #$01
       jsr $a8fc        ; add TXTPTR by one
.f858  ldy #$ff        ; init pointer
.f85a  iny
       lda ($7a),y     ; read character from keyboard buffer
       beq $f867        ; terminator found
       cmp #$22         ; quotes?
       beq $f872        ; yes
       cmp #$3a         ; colon?
       bne $f85a        ; nope, next character
.f867  bit $9d          ; test MSGFLG, if direct mode
       bpl $f875
       clc
       jsr $aebd
       jmp $f878
.f872  jsr $a8fb        ; add value in (Y) to TXTPTR
.f875  jsr $ad9e        ; evaluate expression in text
.f878  jsr $e25a        ; use part of OPEN/CLOSE to SETNAM
       jsr $79          ; CHRGET
       cmp #$2c         ; test for comma ','
       bne $f887        ; nope
       jsr $b79b        ; use GTBYTC to read character after comma
.f885  stx $ba         ; store in FA, device number
.f887  ldy #$00
       bit $9d          ; test MSGFLG, if direct mode
       bpl $f89a
.f88d  lda ($d1),y     ; current screen line address, read from screen
       cmp #$20         ; space

```

```

        beq $f89a      ; yepp
        lda #$0d      ; carriage return
        jsr $e716     ; output to screen
        bne $f88d
.f89a  jsr $f75c      ; test if device FA is present
        lda #$ff
        jsr $b475
        lda $b7      ; FNLEN
        ldx $bb     ; FNADR, pointer to current filename
        ldy $bc
        jsr $b4c7
        jsr $b6a3    ; do string housekeeping
        stx $bb     ; store in FNADR, pointer to current filename
        sty $bc

```

#### **F8B2 OPEN COMMAND CHANNEL**

The following routine open the command channel. A test is done to see if it is allready open. If so, the command channel is closed before opened.

```

.f8b2  jsr $f2cf      ; close command channel if open
        lda $b7      ; read FNLEN, length of current filename, temp store
        ldx #$00     ; store 0
        stx $b7     ; in FNLEN
        ldx #$6f
        bne $f8c3    ; allways jump
        ldx #$6e
        lda $b7
.f8c3  stx $b9      ; store in SA, current secondary address
        stx $9f     ; store in JiffyDOS default filename
.f8c7  pha
        stx $b8     ; store in LA, current logical file number
        jsr $ffcc    ; CLRCHN, close all I/O channels
        jsr $ffc0    ; OPEN
        pla
        sta $b7     ; restore FNLEN, length of current filename
.f8d3  rts          ; return

```

#### **F8D4 JIFFYDOS LOCK/UNLOCK FILE**

This routine locks/unlocks speciyed file. The file is opened, and tests are made to check that everything is OK. If so a bunch of code are transfered to the drive, and executed. The code to be transfered is found at \$f398, after the memory-write command.

```

.f8d4  jsr $f1e8     ; open file and test if all is OK
        bne $f8d3    ; not ok
        ldx #$00     ; setup drivecommand at $f398+0.
        ldy #$22     ; length of string
        jsr $f8e4    ; execute
        ldy #$05     ; setup drivecommand at $f398+$22, length 5 bytes
        ldx #$22
.f8e4  jsr $f0f4     ; execute direct drivecommand
        jmp $ffcc    ; CLRCHN, close all I/O channels

```

#### **F8EA JIFFYDOS PATCH, SERIAL SEND**

This is a patch to the original Commodore KERNAL to send data on the serial bus.

```

.f8ea  sta $dd00     ; store in serial bus I/O port
        and #$08     ; test ATN, attension
        beq $f910    ; ATN = 1
        lda $95     ; BSOUR

```

```

        ror a
        ror a
        cpx #$02          ; bit counter =2
        bne $f910        ; if not, exit
        ldx #$1e
.f8fb  bit $dd00
        bpl $f905
        dex
        bne $f8fb
        beq $f90e
.f905  bit $dd00
        bpl $f905
        ora #$40
        sta $a3
.f90e  ldx #$02
.f910  rts

```

#### **F911 JIFFYDOS DISPLAY ASCII FILE**

The following routine is called by the LIST ASCII from disk. It clears the command channel and calls a routine that reads maximum 254 character from the file. This is repeated until the entire file is displayed.

```

.f911  ldy #$00
        jsr $f0e2        ; CLRCHN and perform CHKIN on (A)
.f916  jsr $fca9        ; read text into buffer
        bvs $f91d        ; finish
        bcc $f916        ; next
.f91d  sty $b7          ; FNLEN, length of current filename
        lda $90          ; STATUS
        and #$82
        rts

```

#### **F924 JIFFYDOS INTERLEAVE**

The following routine sets the interleave gapsize by writing the selected value to drive memory at position \$0069.

```

.f924  jsr $b79b        ; GEBYTC, getbyte from keyboard buffer
        txa              ; transfer gapsize to (A)
        ldx #$2d        ; setup drive command at $f398+2d, M-W 69 00 01
        bne $f930        ; jump always

```

#### **F92C JIFFYDOS BUMP DISABLE**

The following routine disables the 1541 head rattle. This is done by writing the value \$85 to drivememory at position \$006a.

```

.f92c  lda #$85
        ldx #$27        ; setup drive command at $f398+$27, M-W 6a 00 01
.f930  ldy #$06
        pha
        jsr $f0f4        ; execute drive command
        pla
        jmp $ffd2        ; write byte in (A) to drive, and return

```

#### **F93A JIFFYDOS MARK FILE FOR COPY**

This routine toggles the copy flag for one file, or for all selected files depending on the entry point. If entry at \$f93a, the copy flags for all files will be toggled, and if entry at \$f93d only one will be affected.

```

.f93a  ldx #$00        ; toggle flag for all files
        .byte $2c      ; mask LDX-command
.f93d  ldx #$06        ; toggle flag for current file

```

```

jsr $a68e      ; STXPT, reset TXTPTR to start of BASIC
ldy #$05
lda ($7a),y   ; test 5:th character
cmp #$12      ; <RVS ON>?
bne $f9b0     ; if not, directory not loadad, exit
pla
txa           ; store (X), the toggle flag, on stack
pha
ldy #$23      ; set offset to $23
.f94f ldx #$22      ; search for a quote marks (")
jsr $a917     ; use part of DATAN, to search for character
dey
jsr $a8fb     ; add offset in (Y) to TXTPTR
pla           ; read flag, set at start
pha
beq $f96c     ; 'toggle all files' are set
sta $d3
ldy #$01
.f960 iny
jsr $f16a     ; use part of 'input from screen'
cmp ($7a),y
bne $f977
sbc #$22
bne $f960
.f96c tay
lda ($7a),y   ; get character
eor #$0a      ; toggle between $20 (space) and $2a (*)
sta ($7a),y   ; store character
ldy #$04
sta ($d1),y
.f977 jsr $a8f8     ; DATA, perform data, skip line like REM
ldy #$05
sec
lda ($7a),y
sbc #$42
bne $f94f     ; next line
ldy #$02
sta ($7a),y
pla           ; set flag, read from stack
beq $f98d     ; if zero, all files were marked/unmarked, do LIST
lda #$8d
rts
.f98d jmp $a6a4 ; perform LIST

```

#### **F990 JIFFYDOS TOGGLE DRIVE COMMANDS**

This routine is continued from JiffyDOS get character. It tests if the keys <CTRL D> are pressed. If so, it increments the internal device counter and tests if it is present. The routine will return the new device number in (X), which will be printed, and the routine exits. If <CTRL D> were not pressed, it continues to test <CTRL A> and <CTRL W>. If not, the routine continues to the funktion key test.

```

.f990 bit $9d      ; test MSGFLG
bpl $f9b0        ; exit
tsx
ldy $0107,x
cpy #$e1
bne $f9b0        ; exit
cmp #$04         ; test code #$04, <CTRL D>, toggle drive
bne $f9b2        ; if not, jump to next test

```

```

        inc $be          ; increment JiffyDOS device counter
        jsr $f73d       ; test device number in $be, output (X)
        lda #$00
        jsr $bdcd       ; print numeric value in (A/X)
        jsr $aad7       ; output CR/LF
        jsr $f79a       ;
.f9b0   pla            ; retrieve (A)
        rts            ; and exit
.f9b2   cmp #$01       ; test code #$01, <CTRL A>, toggle all files for
copy    beq $f93a       ; toggle all files copy
        cmp #$17       ; test code #$01, <CTRL W>, toggle one file for copy
        beq $f93d       ; toggle single file copy

```

### **F9BA JIFFY DOS FUNKTION KEYS**

This routine test if a shifted, or unshifted funktion key were pressed. If so, it sends a string containing the command to the keyboard buffer. The vector in \$b0 points to the command sting table. The strings are in numerical order, and seperated by a null byte. To find the right string, the routine counts through them all till it reaches the X:th string.

```

.f9ba   ldy $9b        ; must be zero. Some internal JiffyDOS flag
        bne $f9b0       ; exit
        cmp #$8d        ; test keys F1 to F8
        bcs $f9b0       ; larger than F8, exit
        cmp #$85
        bcc $f9b0       ; less than F1, exit
        pla
        sbc #$85        ; substract #$85
        tax            ; transfer key number to (X)
        beq $f9d5       ; if F1, do right away
.f9cc   iny            ; increment pointer
        lda ($b0),y    ; read and skip string in function key table
        bne $f9cc       ; repeat till last byte in string
        dex            ; next string
        bne $f9cc       ; till (X) strings are skipped
.f9d4   iny
.f9d5   lda ($b0),y    ; read command from corresponding key
        beq $f9e2       ; if final character, exit
        cmp #$0d        ; <return>
        beq $f9e4       ;
        jsr $e716       ; output to screen
        bne $f9d4       ; next character
.f9e2   sta $d4
.f9e4   rts

```

### **F9E5 JIFFYDOS GET CHARACTER**

This routine is a new JiffyDOS routine to handle extended functions. It is called from \$e5ec, and starts with the original jump. The routine test the F-keys, and if a valid combination of <CTRL xx> is pressed. If quote mode or insert mode is activated, then this routine will exit.

```

.f9e5   jsr $e5b4       ; get character from keyboard buffer
        pha            ; temp store
        ldx $d4        ; test QTSV, if quote mode is activated
        bne $fa37       ; if not zero, quote mode is on - exit
        ldx $d8        ; test INSRT, inseret mode
        bne $fa37       ; if not zero, insert mode is on - exit
        cmp #$10       ; test code #$10, <CTRL P>, screen dump
        bne $f990       ; if not pressed, jump and test other keys

```

### **F9F5 JIFFYDOS SCREEN DUMP**

This routine performs a screen dump when the keys <CTRL P> are pressed. It reads \$d018 to determine if upper or lower character set is used, and sends the proper SA after LISTEN. The routine stores the cursor positions on the stack, and retrieves them, and replaces the cursor on exit. To print a character to the serial bus, the routine uses part of the KERNAL CIOUT routine.

```
.f9f5  lda #$04          ; printer device #4
      jsr $ffb1       ; send LISTEN to device #4
      lda $d018       ; test upper/lower character set
      and #$02
      beq $fa03
.f9f5  lda #$07          ; set SA=#$67
.f9f5  ora #$60          ; set SA=#$60
.f9f5  jsr $ff93       ; send SA after LISTEN
.f9f5  lda $d3         ; PNTR, cursor column
.f9f5  pha            ; temp store
.f9f5  lda $d6         ; TBLX, cursor line
.f9f5  pha            ; temp store
.f9f5  ldy #$00        ; column counter
.f9f5  sty $d4         ; clear quotes mode, by writing zero into QTSW
.f9f5  jsr $e50c       ; PLOT, put row and column
.f9f5  inc $d5         ; increment LNMX, maximum screen line length
.f9f5  jsr $f16a       ; input from screen
.f9f5  jsr $eddd       ; CIOUT, send data to serial bus (printer)
.f9f5  cmp #$0d        ; carriage return
.f9f5  bne $fa17       ; next character
.f9f5  inx            ; increment (X), line number
.f9f5  cpx #$19        ; till all 25 are done
.f9f5  bcs $fa2d       ; exit
.f9f5  asl $d5
.f9f5  bpl $fa0e       ; next line
.f9f5  inx
.f9f5  bne $fa0e       ; next line
.f9f5  jsr $ffae       ; UNLISTEN
.f9f5  pla            ; retrieve (X) and (Y)
.f9f5  tax
.f9f5  pla
.f9f5  tay
.f9f5  jsr $e50c       ; PLOT, put cursor on same position as on entry
.f9f5  pla            ; return to original 'get character' routine with
key
.f9f5  rts            ; code in (A)
```

### **FA39 JIFFYDOS COPY COMMAND**

The following routine is executed to copy files.

```
.fa39  sty $26          ; (Y) =0, temp store
.f9f5  jsr $f1e8       ; open command channel and read status
.f9f5  bne $fa38       ; not OK, exit
.f9f5  jsr $79         ; CHARGET
.f9f5  cmp #$52        ; R
.f9f5  bne $fa5a
.f9f5  dec $26
.f9f5  lda $26
.f9f5  jsr $f66b
.f9f5  jsr $e4c6       ; input byte from command channel, and compare to 5
.f9f5  beq $fa47       ; yepp
.f9f5  lda #$00
.f9f5  jsr $f66b
```

```

        lda #$4c          ; L
.fa5a   pha
        ldx $bf
        cpx $ba          ; compare to FA, current device number
        beq $fa37        ; exit
        jsr $f885
        ldx #$37          ; setup drive command at $f398+$37
        ldy #$02          ; 2 bytes long
        jsr $f0f4        ; send S:
        jsr $f5c1        ; print filename
        lda #$2c          ; ,
        sta ($bb),y      ; store in filename buffer
        iny
        pla              ; retrieve command
        sta ($bb),y      ; store in filename buffer
        iny
        lda #$2c
        sta ($bb),y      ; ,
        iny
        lda $26
        pha
        bne $fa83
        lda #$57
.fa83   sta ($bb),y      ; W
        iny
        sty $b7          ; update FNLEN, length of current filename
        ldy #$0c
.fa8a   jsr $fab2        ; set SA to (Y) and more
        jsr $f73d        ; test for present device
        jsr $f8b2        ; open command channel
        pla
        jsr $f541        ; use list ASCII from disk to perform copy

```

#### **FA97 TOGGLE PRINTER**

The following routine toggles the printer output funktion. It reads the CHANNL to determine if printmode is to be turned on or off.

```

.fa97   lda $13          ; CHANNL, contains 00 if current output is screen
        beq $faa7        ; toggle printer on
        cmp #$7f         ; CHANNL, contains 7f if current output is printer
        bne $fa38        ; jump to RTS
        jsr $abb7        ; CLRCHN, clear all channels, and set CHANNL=0
        lda #$7f
        jmp $ffc3        ; close file $7f

.faa7   ldx #$04          ; devicenummer #4 = printer
        jsr $73          ; CHRGET??
        jsr $e229        ; use part of OPEN routine to open dev#4
        jsr $f75c        ; test device number in FA
.fab2   sty $b9          ; SA, current secondary address
        ldx #$7f
        stx $13          ; CHANNL, current I/O channel
        lda $b7          ; FNLEN, length of current filename
        jmp $f8c7        ; perform CLRCHN and OPEN file (X)

        tax
        bne $fa8a
        lda $b5
        beq $face

```

#### FAC4 PATCH TO ORIGINAL "LOAD" ROUTINE

This routine is a patch to the original load routine and tests if the current device is a JiffyDOS device. If not, the routine jumps back to the original loader at \$f4f3. The routine disables the sprites and calculates the timing parameters to \$b1. Some handshaking is done

```
.fac4  jsr $f5d2      ; print "LOADING/VERIFYING"
      tsx          ; test if some return pointer on the stack is $f7
      lda $0102,x
      cmp #$f7
      bne $fad7    ; if not, don't store the $ae/$af parameters
      lda $ae
      sta $55
      lda $af
      sta $56
.fad7  bit $a3      ; ldflg, are we talking to a JiffyDOS device?
      bmi $fade    ; yes
      jmp $f4f3    ; nope, return to the original loadroutine.
.fade  sei         ; no interrupts
      ldy #$03
.fae1  lda $af,y    ; save $b0,$b1,$b2 on the stack
      pha
      dey
      bne $fae1
      lda $d015    ; any sprites enabled?
      sta $b0      ; store
      jsr $f0d8    ; clear all sprites not to mess up the timing!
.faf0  jsr $f6bc    ; <STOP> key pressed?
      bpl $fb27    ; yes - exit
      lda $d011    ; read finscroll
      and #$07     ; mask bits
      clc
      adc #$2f     ; add $2f - start of the visible screen
      sta $b1      ; store
      lda $dd00    ; read and store the lower three bits in $dd00
      and #$07     ; they contain PA2 and gfxbank pointers
      sta $b2
      sta $dd00    ; clear
      ora #$20     ; %00100000
      tax
.fb0c  bit $dd00    ; bit test
      bvc $fb0c    ; loop if input clk=0
      bpl $fb3e    ; goto LOADER if input data=0
      ldx #$64
.fb15  bit $dd00    ; bit test
      bvc $fb20    ; EOI if input clk=0
      dex
      bne $fb15    ; repeat $64 times
      lda #$42     ; status code $42, EOI & READ TIMEOUT
      .byte $2c
.fb20  lda #$40     ; status code $40, EOI
      jsr $felc    ; set STATUS
      clc         ; clear carry
      .byte $24    ; mask
.fb27  sec         ; set carry
      lda $b0      ; enable sprites
      sta $d015
      pla         ; restore zero page addresses
      sta $b0
      pla
      sta $b1
```

```

        pla
        sta $b2
        bcs $fb3b      ; if carry set, exit the normal way
        jmp $f528      ;
.fb3b   jmp $f633      ; exit and unlisten

```

### **FB3E THE JIFFYDOS XFER ROUTINE FOR LOAD**

```

.fb3e   bit $dd00      ; drive timing
        bpl $fb3e      ; loop if input data=0
        sec
.fb44   lda $d012      ; raster timing
        sbc $b1
        bcc $fb4f
        and #$07
        beq $fb44
.fb4f   lda $b2        ; lower three bits of $dd00
        stx $dd00      ; store %00100xxx in $dd00
        bit $dd00      ; bit test
        bvc $faf0     ; branch is input clk=0
        nop            ; timing
        sta $dd00      ; store %00000xxx in $dd00
        ora $dd00      ; read two first bits
        lsr a          ; move right two times
        lsr a
        nop            ; timing
        ora $dd00      ; read next two bits
        lsr a          ; move right two times
        lsr a
        eor $b2        ; "trixx" to handle the lower three bits of $dd00
        eor $dd00
        lsr a          ; move right two times
        lsr a
        eor $b2        ; "trixx" to handle the lower three bits of $dd00
        eor $dd00
        cpy $93        ; load/verify flag
        bne $fb83      ; branch if verify
        sta ($ae),y    ; store loaded byte in memory
.fb7a   inc $ae        ; next low-byte
        bne $fb44      ; fetch next byte
        inc $af        ; next high byte
        jmp $fb44      ; fetch next byte
.fb83   cmp ($ae),y    ; verify byte
        beq $fb7a      ; equal
        sec
        lda #$10       ; verify error
        sta $90        ; store in STATUS
        bne $fb7a      ; continue

        lda $c2
        sta $ad
        lda $c1
        sta $ac
        rts

```

### **FB97 JIFFYDOS DISABLE SPRITES BEFORE ACPTR**

This routine disables all the sprites on the screen, and continues the loading procedure. Afterwards the sprites are enabled again.

```

.fb97   pha          ; store the $d015 value on the stack
        jsr $f0d8     ; disable all the sprites

```

```

jsr $fbb4      ; continue the loader below
pla           ; restore
sta $d015     ; and enable sprites again
lda $a4       ;
rts

```

#### **FBA5 JIFFYDOS ACPTR**

This is the JiffyDOS ACPTR routine which fetches a byte from the serial bus. Entry point is \$fbaa where a test is done by checking \$a3 to see if the current device is a JiffyDOS device. Visible sprites are disabled, and raster-timing is done so that no serial access is done when there is a "bad rasterline"

```

.fba5  lda #$00          ; jump back to the normal load routine
       jmp $ee16
.fbaa  sei
       bit $a3          ; test $a3 to see if the device is a JiffyDOS drive
       bvc $fba5       ; nope, back to normal load routine
       lda $d015       ; sprites on screen that can mess up the critical
                       ; timing
       bne $fb97       ; yes, clear sprites before loading
.fbb4  lda $dd00        ; serial bus
       cmp #$40        ; test bit 6
       bcc $fbb4       ; loop
       and #$07        ; mask lower three bits
       pha            ; store
.fbbe  lda $d012        ; current raster line
       sbc $d011       ; subtract fine scroll register
       and #$07        ; mask upper bits
       cmp #$07
       bcs $fbbe       ; wait a little longer
       pla            ; restore
       sta $dd00       ; write output clk=0 and output data=0
       sta $a4
       ora #$20        ; set bit 5=1
       pha            ; store on stack
       nop            ; timing
       nop
       ora $dd00       ; first two bits
       lsr a           ; rotate right
       lsr a
       nop            ; timing
       ora $dd00       ; next bits
       lsr a           ; rotate right
       lsr a
       eor $a4         ; take care of the lower three bits in $dd00
       eor $dd00       ; next bits
       lsr a           ; rotate right
       lsr a
       eor $a4         ; take care of the lower three bits in $dd00
       eor $dd00       ; next bits
       sta $a4
       pla            ; restore from stack
       bit $dd00       ; bit test
       sta $dd00       ; store
       bvc $fc22
       bpl $fc1d
       lda #$42
       jmp $edb2

```

#### **FBFE PATCH SEND DATA ON SERIAL LINE**

The following routine is used to send a byte to a device on the serial bus. The routine checks if the device is a JiffyDOS device by reading \$a3. If not a JiffyDOS device, the routine jumps back to the original load routine at \$ed40.

```
.fbfe sei                ; disable interrupts
      bit $a3            ; ldflg
      bvc $fc14         ; test some more
.fc03 lda $d015         ; any sprites enabled
      beq $fc27         ; nope, continue the send byte routine
      pha                ; store number of sprites on the stack
      jsr $f0d8         ; disable all sprites
      jsr $fc27         ; send-byte routine
      pla                ; read stack
      sta $d015         ; enable sprites
      rts                ; return
.fc14 lda $a3            ; ldflg
      cmp #$a0
      bcs $fc03         ; go and test sprites
      jmp $ed40         ; original send data on serial bus

.fc1d lda #$40           ; %01000000 (EOI)
      jsr $felc         ; set I/O status
.fc22 lda $a4
.fc24 cli
      clc
      rts
```

#### FC27 JIFFYDOS PATCH SEND DATA ON SERIAL LINE

The bits in BSOUR are sent in the following order %22114334.

```
.fc27 txa                ; store (X) on the stack
      pha
      lda $95           ; BSOUR, the byte to send
      and #$f0          ; upper four bits
      pha                ; on stack
      lda $95           ; BSOUR, the byte to send
      and #$0f          ; lower four bits
      tax                ; to (X)
.fc33 lda $dd00         ; serial bus
      bpl $fc33         ; loop as long as data input=0
      and #$07          ; %00000111, mask lower three bits, PA2 and gfx bank
      sta $95           ; store
      sec
.fc3d lda $d012         ; time the send routine with the raster
      sbc $d011         ; badlines are not allowed during xfer
      and #$07
      cmp #$06
      bcs $fc3d         ; wait
      lda $95           ; 00000xxx
      sta $dd00         ; clear serial bus to "clock" the drive
      pla                ; upper four bits to send
      ora $95           ; set PA2 and gfx bank
      sta $dd00         ; send to drive over serial bus
      lsr a              ; next two bits
      lsr a
      and #$f0          ; clear low nybble
      ora $95           ; and set PA2 and gfx bank
      sta $dd00         ; send to drive over serial bus
```

```

        lda $fc8a,x      ; use (X) as offset for lownybble-table
        ora $95          ; set PA2 and gfx bank
        sta $dd00        ; send to drive over serial bus
        lsr a            ; next two bits
        lsr a
        and #$f0         ; clear low nybbles
        ora $95          ; set PA2 and gfx bank
        sta $dd00        ; send to drive over serial bus
        and #$0f
        bit $a3
        bmi $fc76
        ora #$10
.fc76  sta $dd00
        pla              ; restore (X)
        tax
        lda $95          ; PA2 and gfx bank
        ora #$10         ; set send clk=1
        sta $dd00        ; store
        bit $dd00        ; read serial bus
        bpl $fc24        ; branch if input data=0
        jmp $edb0        ; back

```

#### **FCA8 JIFFYDOS SENDTABLE**

A table of bit combinations for the lower nybble of the byte to send to a JiffyDOS device.

```

.fc8a  .byte %00000000 ; $00
        .byte %10000000 ; $80
        .byte %00100000 ; $20
        .byte %10100000 ; $a0
        .byte %01000000 ; $40
        .byte %11000000 ; $c0
        .byte %01100000 ; $60
        .byte %11100000 ; $e0
        .byte %00010000 ; $10
        .byte %10010000 ; $90
        .byte %00110000 ; $30
        .byte %10110000 ; $b0
        .byte %01010000 ; $50
        .byte %11010000 ; $d0
        .byte %01110000 ; $70
        .byte %11110000 ; $f0

        beq $fcbb
        ldx #$f7
        jmp $f5c1

```

#### **FCA0 JIFFYDOS X COMMAND**

The following routine sets the destination devicenummer when using the JiffyDOS copyroutine.

```

.fca0  jsr $b79b        ; GTBYTC, get destination device
        stx $bf          ; store in $bf
        rts             ; back

```

#### **FCA6 READ INTO BUFFER**

The following routine is used by the LIST ASCII and LIST BASIC directly from disk. It reads a number of bytes into the filename buffer area.

```

.fca6  jsr $f0e2        ; CLRCHN, and perform CHKIN on (A)

```

```

.fca9  jsr $ffcf          ; CHRIN
        sta ($bb),y      ; FNADR POINTER, store in buffer for current
filename
        iny              ; next character
        bit $90          ; test STATUS
        bvs $fcbb       ; exit
        cpy #$fe        ; max length
        bcs $fcbb       ; yepp
        cmp #$01        ; larger than 1
        bcs $fca9       ; yepp, repeat
.fcbb  rts

```

#### **FCBC DISABLE JIFFYDOS COMMANDS**

The following routine is called by the @X command and restores the IERROR, IMAIN and ICRNCH vector.

```

.fcbc  ldx #$05
.fcbe  lda $f1a3,x       ; table with original vectors
        sta $0300,x     ; store in vector table
        dex
.fcc5  bpl $fcbe
        stx $9b         ; (X)=255, JiffyDOS not activated.
        rts

        lda aa5
        ora (p29,x)
        sbc f0185,x
        sec
        lda aac
        sbc aae
        lda aad
        sbc aaf
        rts
        inc aac
        bne ifcel1
        inc aad
ifcel1 rts

```

#### **FCE2 POWER RESET ENTRY POINT**

The system hardware reset vector (\$FFFC) points here. This is the first routine executed when the computer is switched on. The routine firstly sets the stackpointer to #ff, disables interrupts and clears the decimal flag. It jumps to a routine at \$fd02 which checks for autostart-cartridges. If so, an indirectjump is performed to the cartridge coldstart vector at \$8000. I/O chips are initiated, and system constants are set up. Finally the IRQ is enabled, and an indirect jump is performed to \$a000, the basic cold start vector.

Future implementaions? - A patch to disable the \$8000 autostart if a special key is pressed.

```

.fce2  ldx #$ff
        sei
        txs             ; Set stackpointer to #ff
        cld
.fce7  jsr $fd02        ; Check ROM at $8000
        bne $fcef
        jmp ($8000)    ; Jump to autostartvector
.fcef  stx $d016
        jsr $fda3      ; Init I/O
        jsr $fd50      ; Init system constants

```

```

jsr $fd15      ; KERNAL reset
jsr $ff5b      ; Setup PAL/NTSC
cli
jmp ($a000)    ; Basic coldstart

```

#### **FD02 CHECK FOR 8-ROM**

Checks for the ROM autostartparametrar at \$8004-\$8008. It compares data with \$fd10, and if equal, set Z=1.

```

.fd02 ldx #$05      ; 5 bytes to check
.fd04 lda $fd0f,x   ; Identifier at $fd10
      cmp $8003,x   ; Compare with $8004
      bne $fd0f     ; NOT equal!
      dex
      bne $fd04     ; until Z=1
.fd0f rts

```

#### **FD10 8-ROM IDENTIFYER**

The following 5 bytes contains the 8-ROM identifier, reading "CBM80" with CBM ASCII. It is used with autostartcartridges. See \$fd02.

```

.fd10 c3 c2 cd 38 30 ; CBM80

```

#### **FD15 RESTOR: KERNAL RESET**

The KERNAL routine RESTOR (\$ff8a) jumps to this routine. It restores (copies) the KERNAL vectors at \$fd30 to \$0314-\$0333. Continues through VECTOR.

```

.fd15 ldx #$30
      ldy #$fd      ; $fd30 - table of KERNAL vectors
      clc          ; Clear carry to SET values.

```

#### **FD1A VECTOR: KERNAL MOVE**

The KERNAL routine VECTOR (\$ff8d) jumps to this routine. It reads or sets the vectors at \$0314-\$0333 depending on state of carry. X/Y contains the address to read/write area, normally \$fd30. See \$fd15.

A problem is that the RAM under the ROM at \$fd30 always gets a copy of the contents in the ROM then you perform the copy.

```

.fd1a stx $c3      ; MEMUSS - c3/c4 temporary used for address
      sty $c4
      ldy #$1f     ; Number of bytes to transfer
.fd20 lda $0314,y
      bcs $fd27     ; Read or Write the vectors
      lda ($c3),y
.fd27 sta ($c3),y
      sta $0314,y
      dey
      bpl $fd20     ; Again...
      rts

```

#### **FD30 KERNAL RESET VECTORS**

These are the vectors that is copied to \$0314-\$0333 when RESTOR is called. These vectors are the same in JiffyDOS, as in stock Commodore KERNAL.

```

.fd30 31 ea      ; CINV VECTOR: hardware interrupt ($ea31)
.fd32 66 fe      ; CBINV VECTOR: software interrupt ($fe66)
.fd34 47 fe      ; NMINV VECTOR: hardware nmi interrupt ($fe47)
.fd36 4a f3      ; IOPEN VECTOR: KERNAL open routine ($f3a4)
.fd38 91 f2      ; ICLOSE VECTOR: KERNAL close routine ($f291)
.fd3a 0e f2      ; ICHKIN VECTOR: KERNAL chkin routine ($f20e)

```

```

.fd3c 50 f2 ; ICKOUT VECTOR: KERNAL chkout routine ($f250)
.fd3e 33 f3 ; ICLRCH VECTOR: KERNAL clrchn routine ($f333)
.fd40 57 f1 ; IBASIN VECTOR: KERNAL chrin routine ($f157)
.fd42 ca f1 ; IBSOUT VECTOR: KERNAL chrout routine ($f1ca)
.fd44 ed f6 ; ISTOP VECTOR: KERNAL stop routine ($f6ed)
.fd46 3e f1 ; IGETIN VECTOR: KERNAL getin routine ($f13e)
.fd48 2f f3 ; ICLALL VECTOR: KERNAL clall routine ($f32f)
.fd4a 66 fe ; USRCMD VECTOR: user defined ($fe66)
.fd4c a5 f4 ; ILOAD VECTOR: KERNAL load routine ($f4a5)
.fd4e ed f5 ; ISAVE VECTOR: KERNAL save routine ($f5ed)

```

#### **FD50 RAMTAS: INIT SYSTEM CONSTANTS**

The KERNAL routine RAMTAS(\$ff87) jumps to this routine. It clears the pages 0,2 and 3 by writing 00 into them. It also sets the start of the cassette buffer - \$033c, and determines how much free RAM-memory there is. (The tapebuffer could probably be removed, since JiffyDOS doesn't use tapes at all.) The memorycheck is performed by writing two different bytes into all memory positions, starting at \$0400, till it reaches the ROM (the byte read is not the same as the one you wrote.) Note that the contents of the memory is restored afterwards. Finally, bottom of the memory, and top of screen-pointers are set.

Future implementations? - Make a faster RAMcheck routine which not reads all bytes from \$0400 and upwards. There can only be ROM at \$8000 to \$a000, so why bother to check elsewhere. Save a few bytes ad lots of time!!

```

.fd50 lda #$00
      tay
.fd53 sta $02,y ; Fill pages 0,2,3 with zeros
      sta $0200,y
      sta $0300,y
      iny
      bne $fd53 ; all 256 bytes
      ldx #$3c
      ldy #$03 ; Set tapebuffer to $033c
      stx $b2 ; Variables TAPE1 is used.
      sty $b3
      tay
      lda #$03
      sta $c2
.fd6c inc $c2
.fd6e lda ($c1),y ; Perform memorytest. Starting at $0400 and upwards.
      tax ; Store temporary in X-reg
      lda #$55
      sta ($c1),y ; Write #$55 into memory
      cmp ($c1),y ; and compare.
      bne $fd88 ; if not equal... ROM
      rol a
      sta ($c1),y ; Write #$AA into same memory
      cmp ($c1),y ; and compare again.
      bne $fd88 ; if not equal... ROM
      txa
      sta ($c1),y ; Restore stored value
      iny
      bne $fd6e ; Next memorypos
      beq $fd6c ; New page in memory
.fd88 tya ; The memorytest always exits when reaching a ROM
      tax
      ldy $c2
      clc
      jsr $fe2d ; Set top of memory. X and Y holds address.

```

```

lda #$08
sta $0282      ; Set pointer to bottom of memory ($0800)
lda #$04
sta $0288      ; Set pointer to bottom of screen ($0400)
rts

```

### **FD96 TAPE IRQ VECTORS**

This table contains the vectors to the four tape-IRQ routines. The vectors are: \$fc6a - tape write I, \$fcbd - tape write II, \$ea31 - normal IRQ, \$f92c - tape read. This table could probably be removed, to gain another 8 bytes of free ROM for own code.

```
.fd96 6a fc bd fc 31 ea 2c f9
```

### **FDA3 IOINIT: INIT I/O**

The KERNAL routine IOINIT (\$ff84) jumps to this routine. It sets the init-values for the CIAs (IRQ, DDRA, DRA etc.), the SID-volume, and the processor onboard I/O port.

```

.fda3 lda #$7f
      sta $dc0d      ; CIA#1 IRQ control register
      sta $dd0d      ; CIA#2 IRQ control register
      sta $dc00      ; CIA#1 data port $ (keyboard)
      lda #$08
      sta $dc0e      ; CIA#1 control register timer A
      sta $dd0e      ; CIA#2 control register timer A
      sta $dc0f      ; CIA#1 control register timer B
      sta $dd0f      ; CIA#2 control register timer B
      ldx #$00
      stx $dc03      ; CIA#1 DDRB. Port B is input
      stx $dd03      ; CIA#2 DDRB. Port B is input
      stx $d418      ; No sound from SID
      dex
      stx $dc02      ; CIA#1 DDRA. Port A is output
      lda #$07      ; %00000111
      sta $dd00      ; CIA#2 dataport A. Set Videobank to $0000-$3fff
      lda #$3f      ; %00111111
      sta $dd02      ; CIA#2 DDRA. Serial bus and videobank
      lda #$e7      ; 6510 I/O port - %XX100111
      sta $01
      lda #$2f      ; 6510 I/O DDR - %00101111
      sta $00

```

### **FDDD ENABLE TIMER**

This routine inits and starts the CIA#1 timer A according to the PAL/NTSC flag. Different system clocks rates are used in PAL/NTSC systems.

```

.fddd lda $02a6      ; PAL/NTSC flag
      beq $fdec      ; NTSC setup
      lda #$25
      sta $dc04      ; CIA#1 timer A - lowbyte
      lda #$40      ; PAL-setup #4025
      jmp $fdf3
.fdec lda #$95
      sta $dc04      ; CIA#1 timer A - lowbyte
      lda #$42      ; NTSC-setup #4295
.fdf3 sta $dc05      ; CIA#1 timer A - highbyte
      jmp $ff6e      ; start timer

```

### **FDF9 SETNAM: SAVE FILENAME DATA**

The KERNAL routine SETNAM (\$ffbd) jumps to this routine. On entry, A-reg holds the length of the filename, and X/Y the address in mem to the filename.

```
.fdf9  sta $b7          ; store length of filename in FNLEN
       stx $bb          ; store pointer to filename in FNADDR
       sty $bc
       rts
```

#### **FE00 SETLFS: SAVE FILE DETAILS**

The KERNAL routine SETLFS (\$ffba) jumps to this routine. On entry A-reg holds the logical filename, X the device number, and Y the secondary address.

```
.fe00  sta $b8          ; store logical filename in LA
       stx $ba          ; store devicenum in FA
       sty $b9          ; store secondary address in SA
       rts
```

#### **FE07 READST: READ STATUS**

The KERNAL routine READST (\$ffb7) jumps to this routine. The routine checks if the current devicenum is 2, (ie RS232) then the value of RSSTAT (the ACIA 6551 status) is returned in (A), and RSSTAT is cleared. Else it reads and returns the value of STATUS.

```
.fe07  lda $ba          ; read current device number from FA
       cmp #$02          ; device = RS232?
       bne $fela          ; nope, read STATUS
       lda $0297          ; RSSTAT
       pha              ; temp store
       lda #$00
       sta $0297          ; clear RSSTAT
       pla
       rts
```

#### **FE18 SETMSG: FLAG STATUS**

The KERNAL routine SETMSG (\$ff90) jumps to this routine. On entry, the value in (A) is stored in MSGFLG, then the I/O status is placed in (A). If routine is entered at \$felc the contents in (A) will be stored in STATUS.

```
.fe18  sta $9d          ; store MSGFLG
.fe1a  lda $90          ; read STATUS
.fe1c  ora $90
       sta $90
       rts
```

#### **FE21 SETTMO: SET TIMEOUT**

The KERNAL routine SETTMO (\$ffa2) jumps to this routine. On entry the value in (A) is stored in the IEEE timeout flag. (Who uses IEEE nowadays?)

```
.fe21  sta $0285          ; store in TIMOUT
       rts
```

#### **FE25 MEMTOP: READ/SET TOP OF MEMORY**

The KERNAL routine MEMTOP (\$ffa9) jumps to this routine. If carry is set on entry, the top of memory address will be loaded into (X/Y). If carry is clear on entry, the top of memory will be set according to the contents in (X/Y)

```
.fe25  bcc $fe2d          ; carry clear?
       ldx $0283          ; read memtop from MEMSIZ
```

```

        ldy $0284
.fe2d  stx $0283      ; store memtop in MEMSIZ
        sty $0284
        rts

```

#### **FE34 MEMBOT: READ/SET BOTTOM OF MEMORY**

The KERNAL routine MEMBOT (\$ff9c) jumps to this routine. If carry is set on entry, the bottom of memory address will be loaded into (X/Y). If carry is clear on entry, the bottom of memory will set according to the contents in (X/Y)

```

.fe34  bcc $fe3c      ; carry clear?
        ldx $0281      ; read membot from MEMSTR
        ldy $0282
.fe3c  stx $0281      ; store membot in MEMSTR
        sty $0282
        rts

```

#### **FE43 NMI ENTRY POINT**

The processor jumps to this routine every time a NMI occurs (see jump vector at \$ffffa). On entry all processor registers will be put on the stack. The routine will check the presents of a ROM cartridge at \$8000 with autostart, and warm start it. Otherwise, the following warm start routine is called.

```

.fe43  sei              ; disable interrupts
        jmp ($0318)     ; jump to NMINV, points normally to $fe47
.fe47  pha              ; store (A), (X), (Y) on the stack
        txa
        pha
        tya
        pha
        lda #$7f        ; CIA#2 interrupt control register
        sta $dd0d
        ldy $dd0d
        bmi $fe72      ; NMI caused by RS232? If so - jump
        jsr $fd02      ; check for autostart at $8000
        bne $fe5e
        jmp ($8002)     ; Jump to warm start vector
.fe5e  jsr $f6bc        ; Scan one row in the keymatrix and store value in
$91
        jsr $ffe1      ; Check $91 to see if <STOP> was pressed
        bne $fe72      ; <STOP> not pressed, skip part of following routine

```

#### **FE66 WARM START BASIC**

This routine is called from the NMI routine above. If <STOP> was pressed, then KERNAL vectors are restored to default values, I/O vectors initialised and a jump to (\$a002), the Basic warm start vector. The NMI routine continues at \$fe72 by checking the RS232, if there is anything to send.

```

.fe66  jsr $fd15      ; KERNAL reset
        jsr $fda3      ; init I/O
        jsr ee518      ; init I/O
        jmp ($a002)     ; jump to Basic warm start vector

.fe72  tya              ; Read CIA#2 interrupt control register
        and $02a1      ; mask with ENABL, RS232 enable
        tax              ; temp store in (X)
        and #$01        ; test if sending (%00000001)
        beq $fea3      ; nope, jump to recieve test

```

```

        lda $dd00          ; load CIA#1 DRA
        and #$fb          ; mask bit2 (RS232 send)
        ora $b5           ; NXTBIT, next bit to send
        sta $dd00        ; and write to port
        lda $02a1
        sta $dd0d        ; write ENABL to CIA#2 I.C.R
        txa               ; get temp
        and #$12         ; test if recieving (bit1), or waiting for reciever
edge (bit4) ($12 = %00010010)
        beq $fe9d        ; nope, skip reciever routine
        and #$02         ; test if recieving
        beq $fe9a        ; nope
        jsr $fed6        ; jump to NMI RS232 in
        jmp $fe9d
.fe9a   jsr $ff07        ; jump to NMI RS232 out
.fe9d   jsr $eebb        ; RS232 send byte
        jmp $feb6        ; goto exit
.fea3   txa             ; get temp
        and #$02         ; test bit1
        beq $feae        ; nope
        jsr $fed6        ; NMI RS232 in???
        jmp $feb6        ; goto exit
.feae   txa             ; set temp
        and #$10         ; test bit4
        beq $feb6        ; nope, exit
        jsr $ff07        ; NMI RS232 out
.feb6   lda $02a1        ; ENABL
        sta $dd0d        ; CIA#2 interrupt control register
        pla              ; restore registers (Y), (X), (A)
        tay
        pla
        tax
        pla
        rti              ; back from NMI

```

#### **FEC2 RS232 TIMING TABLE - NTSC**

Timingtable for RS232 NMI for use with NTSC machines. The table containe 10 entries which corresponds to one of the fixed RS232 rates, starting with lowest (50 baud) and finishing with the highest (2400 baud). Since the clock frequency is different between NTSC and PAL systems, there is another table for PAL machines at \$e4ec.

Future implementations? Remove the table if you run a PAL machine, and put some own code here.

```

        cmp ($27,x)
        rol fc51a,x
        ora ($74),y
        $sl $0ced
        eor $06
        beq $fed2
        lsr $01
.fed2   clv
        brk
        $dc ($00),y

```

#### **FED6 NMI RS232 IN**

This routine inputs a bit from the RS232 port and sets the baudrate timing for the next bit. Continues to the RS232 recieve routine.

```

.fed6   lda $dd01          ; RS232 I/O port

```

```

and #$01          ; test bit0, received data
sta $a7           ; store in INBIT
lda $dd06         ; lowbyte of timer B
sbc #$1c
adc $0299         ; <BAUDOF
sta $dd06         ; store timer B
lda $dd07         ; highbyte of timer B
adc $029a         ; >BAUDOF
sta $dd07         ; store timer B
lda #$11
sta $dd0f         ; CIA#2 control register B
lda $02a1         ; ENABL
sta $dd0d         ; CIA#2 interrupt control register
lda #$ff
sta $dd06
sta $dd07
jmp $ef59         ; jump to RS232 receive routine

```

#### **FF07 NMI RS232 OUT**

This routine sets up the baudrate for sending the bits out, and adjusts the number of bits remaining to send.

```

.ff07  lda $0295         ; M51AJB - non standard BPS time
      sta $dd06         ; timer B low
      lda $0296
      sta $dd07         ; timer B high
      lda #$11
      sta $dd0f         ; CIA#2 control register B
      lda #$12
      eor $02a1
      sta $02a1         ; ENABL, RS232 enables
      lda #$ff
      sta $dd06
      sta $dd07         ; timer B
      ldx $0298         ; BITNUM, number of bits still to send in this byte
      stx $a8          ; BITC1, RS232 bitcount
      rts
.ff2f  tax
      lda $0296
      rol
      tay
      txa
      adc #$c8
      sta $0299
      tya
      adc #$00
      sta $029a
      rts

```

#### **FF41 FAKE IRQ**

Fake IRQ entry that clears bit4 which is later tested for HW or SW interrupt. This entry will always create a hardware interrupt.

```

nop              ; don't ask me??
nop
php              ; store processor reg.
pla              ; get reg
and #$ef        ; clear bit4
pha              ; store reg

```

#### **FF48 IRQ ENTRY**

This routine is pointed to by the hardware IRQ vector at \$fffe. This routine is able to distinguish between a hardware IRQ, and a software BRK. The two types of interrupts are processed by its own routine.

```
.ff48 pha                ; Store Acc
      txa
      pha                ; Store X-reg
      tya
      pha                ; Store Y-reg
      tsx
      lda $0104,x        ; Read byte on stack written by processor?
      and #$10           ; check bit 4 to determine HW or SW interrupt
      beq $ff58
      jmp ($0316)        ; jump to CBINV. Points to FE66, basic warm start
.ff58 jmp ($0314)        ; jump to CINV. Points to EA31, main IRQ entry point
```

#### **FF5B CINT: INIT SCREEN EDITOR**

The KERNAL routine CINT (\$FF81) jumps to this routine. It sets up VIC for operation. The original CINT is at \$e518, and this patch checks out if this is a PAL or NTSC machine. This is done by setting the raster compare register to 311, which is the number of scanlines in a PAL machine. If no interrupt occurs, then it's a NTSC machine.

```
.ff5b jsr $e518         ; original I/O init
.ff5e lda $d012         ; wait for top of screen
      bne $ff5e         ; at line zero
      lda $d019        ; Check IRQ flag register if interrupt occurred
      and #$01         ; only first bit
      sta $02a6        ; store in PAL/NTSC flag
      jmp $fddd        ; jump to ENABLE TIMER
```

#### **FF6E START TIMER**

This routine starts the CIA#1 timer and jumps into a routine that handles the serial clock.

```
.ff6e lda #$81         ; Enable IRQ when timer B reaches zero
      sta $dc0d        ; CIA#1 interrupt controll register
      lda $dc0e        ; CIA#1 control register A
      and #$80
      ora #$11         ; Force load of timer A values -bit4, and start -
bit0  sta $dc0e        ; Action!
      jmp $ee8e        ; Continue to 'serial clock off'
```

#### **FF80 KERNAL VERSION ID**

This byte contains the version number of the KERNAL.

```
.ff80 sed $
```

#### **FF81 KERNAL JUMP TABLE**

This table contains jump vectors to the I/O routines. This is a Commodore standard, so no matter what system you are using (VIC20, C64, C128, Plus4 etc) the jump vectors are always located at this position.

```
.ff81 jmp $ff5b        ; CINT, init screen editor
.ff84 jmp $fda3        ; IOINT, init input/output
.ff87 jmp $fd50        ; RAMTAS, init RAM, tape screen
.ff8a jmp $fd15        ; RESTOR, restore default I/O vector
.ff8d jmp $fd1a        ; VECTOR, read/set I/O vector
```

```

.ff90 jmp $fe18 ; SETMSG, control KERNAL messages
.ff93 jmp $edb9 ; SECOND, send SA after LISTEN
.ff96 jmp $edc7 ; TKSA, send SA after TALK
.ff99 jmp $fe25 ; MEMTOP, read/set top of memory
.ff9c jmp $fe34 ; MEMBOT, read/set bottom of memory
.ff9f jmp $ea87 ; SCNKEY, scan keyboard
.ffa2 jmp $fe21 ; SETTMO, set IEEE timeout
.ffa5 jmp $fbaa ; ACPTR, input byte from serial bus. JiffyDOS points
; to $fbaa, the original to $ee13.

.ffa8 jmp $eddd ; CIOUT, output byte to serial bus
.ffab jmp $edef ; UNTALK, command serial bus UNTALK
.ffaе jmp $edfe ; UNLSN, command serial bus UNLSN
.ffb1 jmp $ed0c ; LISTEN, command serial bus LISTEN
.ffb4 jmp $ed09 ; TALK, command serial bus TALK
.ffb7 jmp $fe07 ; READST, read I/O status word
.ffba jmp $fe00 ; SETLFS, set logical file parameters
.ffbd jmp $fdf9 ; SETNAM, set filename
.ffc0 jmp ($031a) ; OPEN, open file
.ffc3 jmp ($031c) ; CLOSE, close file
.ffc6 jmp ($031e) ; CHKIN, prepare channel for input
.ffc9 jmp ($0320) ; CHKOUT, prepare channel for output
.ffcb jmp ($0322) ; CLRCHN, close all I/O
.ffcf jmp ($0324) ; CHRIN, input byte from channel
.ffd2 jmp ($0326) ; CHROUT, output byte to channel
.ffd5 jmp $f49e ; LOAD, load from serial device
.ffd8 jmp $f5dd ; SAVE, save to serial device
.ffdb jmp $f6e4 ; SETTIM, set realtime clock
.ffde jmp $f6dd ; RDTIM, read realtime clock
.ffe1 jmp ($0328) ; STOP, check <STOP> key
.ffe4 jmp ($032a) ; GETIN, get input from keyboard
.ffe7 jmp ($032c) ; CLALL, close all files and channels
.ffeа jmp $f69b ; UDTIM, increment realtime clock
.ffed jmp $e505 ; SCREEN, return screen organisation
.fff0 jmp $e50a ; PLOT, read/set cursor X/Y position
.fff3 jmp $e500 ; IOBASE, return IOBASE address

```

#### **FFF4 SYSTEM HARDWARE VECTORS**

This table contains jumpvectors for system reset, IRQ, and NMI. The IRQ and NMI vectors points to addresses which contains an indirect jump to RAM, to provide user defined routines.

```

.ffe4

.ffffa 43 fe ; NMI hardware vector
.ffffc e2 fc ; System reset vector
.ffffe 48 ff ; IRQ hardware vector

```