```
; *******************************************
; The Commodore VIC-20 / C64 operating system
; *******************************************

C64 = 1        ; set to 1 for assembling the C64 ROM images
VIC = 1 - C64  ;     or 0 for assembling the VIC ROM images

JIFFY = 0      ; set to 1 for JIFFY DOS
PAL   = 1      ; set to 0 for NTSC

; Based on following sources
; ------------------------------------------------------------
; Disassembly done with the "Black Smurf Disassembler"
; The complete Commodore inner space anthology - Karl Hildon
; Vic 20 Programmers reference - Commodore
; Many comments merged in from Lee Davison's disassembly
; Der MOS 6567/6569 Videocontroller (VIC-II) - Christian Bauer

; Recommended assembler
; --------------------------------------------------------------
; Use the opensource cross assembler BSA (Black Smurf Assembler)
; for creating ROM images and assembly listings.
; BSA runs on MAC OSX, Linux, Unix and Windows

MACRO PUSHW(Word)
   LDA Word+1
   PHA
   LDA Word
   PHA
ENDMAC

MACRO PULLW(Word)
   PLA
   STA Word
   PLA
   STA Word+1
ENDMAC

MACRO LDAX(Word)
   LDA Word
   LDX Word+1
ENDMAC

MACRO STAX(Ptr)
   STA Ptr
   STX Ptr+1
ENDMAC

MACRO LAYI(Word)
   LDA #<Word
   LDY #>Word
ENDMAC

MACRO LDAY(Word)
   LDA Word
   LDY Word+1
ENDMAC

MACRO STAY(Ptr)
   STA Ptr
   STY Ptr+1
ENDMAC
```

```
MACRO LDXY(Word)
   LDX Word
   LDY Word+1
ENDMAC

MACRO STXY(Ptr)
   STX Ptr
   STY Ptr+1
ENDMAC

MACRO Print_Msg(Msg)
   LDA #<Msg
   LDY #>Msg
   JSR Print_String
ENDMAC

#if VIC

VIC_BASE = $9000
VIC_REGS = $10

; **************************************
; VIC-20 Video Interface Chip (MOS 6560)
; **************************************

;  #| Adr. |Bit7|Bit6|Bit5|Bit4|Bit3|Bit2|Bit1|Bit0| Function
; --+------+----+----+----+----+----+----+----+----+--------------------
;  0| 9000 |INTL|             HSA                  | horiz. adjustment
; --+------+---------------------------------------+--------------------
;  1| 9001 |                  VSA                  | vertical adjustment
; --+------+---------------------------------------+--------------------
;  2| 9002 | SA9|             COLS                 | screen columns
; --+------+---------------------------------------+--------------------
;  3| 9003 | RA8|             ROWS                 | screen rows
; --+------+---------------------------------------+--------------------
;  4| 9004 |                  RAST                 | raster value
; --+------+---------------------------------------+--------------------
;  5| 9005 |  1 |�   SMA        |�     CMA          | screen/char memory
; --+------+---------------------------------------+--------------------
;  6| 9006 |                  LPH                  | light pen horiz.
; --+------+---------------------------------------+--------------------
;  7| 9007 |                  LPV                  | light pen vertical
; --+------+---------------------------------------+--------------------
;  8| 9008 |                  PAD1                 | paddle 1
; --+------+---------------------------------------+--------------------
;  9| 9009 |                  PAD2                 | paddle 2
; --+------+---------------------------------------+--------------------
; 10| 900a | BSW|             BASS                 | bass sound
; --+------+---------------------------------------+--------------------
; 11| 900b | ASW|             ALTO                 | alto sound
; --+------+---------------------------------------+--------------------
; 12| 900c | SSW|             SOPR                 | soprano sound
; --+------+---------------------------------------+--------------------
; 13| 900d | NSW|             NOIS                 | noise sound
; --+------+---------------------------------------+--------------------
; 14| 900e |      AUXC         |�     VOL          | aux. color / volume
; --+------+---------------------------------------+--------------------
; 15| 900f |      SCOL         | REV|     BCOL     | screen/border color
; --+------+----+----+----+----+----+----+----+----+--------------------

VIC_R0 = $9000       ; interlace / screen origin - horizontal
```

```
                     ; bit 7   : 1 = interlace on
                     ; bit 6-0 : horizontal screen adjustment (5)

VIC_R1 = $9001       ; bit 7-0 : vertical screen adjustment (25)

VIC_R2 = $9002       ; screen address and number of colums
                     ; bit 7   : screen memory address 9
                     ; bit 6-0 : number of columns (22)

VIC_R3 = $9003       ; raster / # of rows / character size
                     ; bit 7   : bit 8 of raster value
                     ; bit 6-1 : number of rows (23)
                     ; bit 0   : 1 = 8 x 16 character size

VIC_R4 = $9004       ; raster value
                     ; bit 7-0 : raster value (bit 8 in R3)

VIC_R5 = $9005       ; screen memory / character memory
                     ; bit 7   : must be 1
                     ; bit 6-4 : screen memory address 12-10
                     ; bit 3-0 : character memory
                     ; -------------------------
                     ; 0000 ROM  $8000 upper case normal
                     ; 0001 ROM  $8400 upper case reversed
                     ; 0010 ROM  $8800 lower case normal
                     ; 0011 ROM  $8C00 lower case reversed
                     ; 1100 RAM  $1000 user defined
                     ; 1101 RAM  $1400 user defined
                     ; 1110 RAM  $1800 user defined
                     ; 1111 RAM  $1C00 user defined

VIC_R6 = $9006       ; light pen horizontal

VIC_R7 = $9007       ; light pen vertical

VIC_R8 = $9008       ; paddle 1

VIC_R9 = $9009       ; paddle 2

VIC_RA = $900A       ; bit 7   : bass sound switch (1 = enable)
                     ; bit 6-0 : bass frequency (clock / (127 - X))

VIC_RB = $900B       ; bit 7   : alto sound switch (1 = enable)
                     ; bit 6-0 : alto frequency

VIC_RC = $900C       ; bit 7   : soprano sound switch (1 = enable)
                     ; bit 6-0 : soprano frequency

VIC_RD = $900D       ; bit 7   : noise sound switch (1 = enable)
                     ; bit 6-0 : noise frequency

VIC_RE = $900E       ; auxiliary color / loudness
                     ; bit 7-4 : auxiliary color
                     ; bit 3-0 : loudness (volume)

VIC_RF = $900F       ; screen color / reverse mode / border color
                     ; bit 7-4 : screen color
                     ; bit 3   : reverse mode (1 = on)
                     ; bit 2-0 : border color

; **********************
; VIC-20 VIA 1 (MOS 6522)
; ***********************

VIA1_DATB = $9110    ; VIA 1 data register B (I/O)
                     ; ---------------------------
                     ; bit 7   DSR  in
                     ; bit 6   CTS  in
                     ; bit 5
                     ; bit 4   DCD  in
                     ; bit 3   RI   in
                     ; bit 2   DTR  out
                     ; bit 1   RTS  out
                     ; bit 0   data in

VIA1_DATA = $9111    ; VIA 1 data register A (I/O)
                     ; ---------------------------
                     ; bit 7   IEC ATN out
                     ; bit 6   cassette switch
                     ; bit 5   light pen
                     ; bit 4   joy 2
                     ; bit 3   joy 1
                     ; bit 2   joy 0
                     ; bit 1   IEC DATA in
                     ; bit 0   IEC CLK  in

VIA1_DDRB = $9112    ; VIA 1 data direction register B
VIA1_DDRA = $9113    ; VIA 1 data direction register A
VIA1_T1CL = $9114    ; VIA 1 timer 1 low  order counter/latch
VIA1_T1CH = $9115    ; VIA 1 timer 1 high order counter/latch
VIA1_T1LL = $9116    ; VIA 1 timer 1 low  order latch
VIA1_T1LH = $9117    ; VIA 1 timer 1 high order latch
VIA1_T2CL = $9118    ; VIA 1 timer 2 low  order counter/latch
VIA1_T2CH = $9119    ; VIA 1 timer 2 high order counter/latch
VIA1_SR   = $911A    ; VIA 1 shift register

VIA1_ACR  = $911B    ; VIA 1 auxiliary control register
                     ; --------------------------------
                     ; bit 7   : T1 PB7 enabled/disabled
                     ; bit 6   : T1 free run/one shot
                     ; bit 5   : T2 clock PB6/�2
                     ; bit 432 : function
                     ; ------------------
                     ; 000   shift register disabled
                     ; 001   shift in , rate controlled by T2
                     ; 010   shift in , rate controlled by �2
                     ; 011   shift in , rate controlled by external clock
                     ; 100   shift out, rate controlled by T2, free run
                     ; 101   shift out, rate controlled by T2
                     ; 110   shift out, rate controlled by �2
                     ; 111   shift out, rate controlled by external clock
                     ; bit 1   : PB latch (1 = enabled)
                     ; bit 0   : PA latch (1 = enabled)

VIA1_PCR   = $911C   ; VIA 1 peripheral control register (PCR)
                     ; bit   function
                     ; ---   --------
                     ; 765   CB2 control
                     ; ---   -----------
                     ; 000   Interrupt Input Mode
                     ; 001   Independent Interrupt Input Mode
                     ; 010   Input Mode
                     ; 011   Independent Input Mode
                     ; 100   Handshake Output Mode
                     ; 101   Pulse Output Mode
                     ; 110   Manual Output Mode, CB2 low
                     ; 111   Manual Output Mode, CB2 high
                     ;  4   CB1 edge positive/negative
                     ; 321   CA2 control
                     ; ---   -----------
                     ; 000   Interrupt Input Mode
                     ; 001   Independent Interrupt Input Mode
                     ; 010   Input Mode
                     ; 011   Independent Input Mode
                     ; 100   Handshake Output Mode
                     ; 101   Pulse Output Mode
                     ; 110   Manual Output Mode, CA2 low
                     ; 111   Manual Output Mode, CA2 high
                     ;  0   CA1 edge positive/negative

; The status bit is a not normal flag. It goes high if both an interrupt
; flag in the IFR and the corresponding enable bit in the IER are set.
; It can be cleared only by clearing all the active flags in the IFR or
; disabling all active interrupts in the IER.

VIA1_IFR   = $911D   ; VIA 1 interrupt flag register) (IFR)
                     ; bit   function      cleared by
                     ; ---   --------      ----------
                     ;  7   interrupt status   clearing all enabled interrupts
                     ;  6   T1 interrupt   read T1C_l, write T1C_h
                     ;  5   T2 interrupt   read T2C_l, write T2C_h
                     ;  4   CB1 transition   read or write port B
                     ;  3   CB2 transition   read or write port B
                     ;  2   8 shifts done   read or write the shift register
                     ;  1   CA1 transition   read or write port A
                     ;  0   CA2 transition   read or write port A

; If enable/disable bit is a zero during a write to this register, each
; 1 in bits 0-6 clears the corresponding bit in the IER. if this bit is
; a one during a write to this register, each 1 in bits 0-6 will set the
; corresponding IER bit

VIA1_IER   = $911E   ; VIA 1 interrupt enable register (IER)
                     ; bit   function
                     ; ---   --------
                     ;  7   enable/disable
                     ;  6   T1 interrupt
                     ;  5   T2 interrupt
                     ;  4   CB1 transition
                     ;  3   CB2 transition
                     ;  2   8 shifts done
                     ;  1   CA1 transition
                     ;  0   CA2 transition

VIA1_DATN    = $911F ; VIA 1 DRA, no handshake
                     ; bit   function
                     ; ---   --------
                     ;  7   ATN out
                     ;  6   cassette switch
                     ;  5   joystick fire, light pen
                     ;  4   joystick left
                     ;  3   joystick down
                     ;  2   joystick up
                     ;  1   serial dat in
                     ;  0   serial clk in
; **********************
; VIC-20 VIA2 (MOS 6522)
; **********************

VIA2_DATB  = $9120   ; VIA 2 DRB, keyboard column drive
VIA2_DATA  = $9121   ; VIA 2 DRA, keyboard row port
                     ; Vic 20 keyboard matrix layout
                     ;       c7   c6   c5   c4   c3   c2   c1   c0
                     ;   +------------------------------------------------
                     ; r7|   F7   F5   F3   F1   DN  RGT  RET  DEL
                     ; r6|    /   UP    =  RSH HOME    ;    *    �
                     ; r5|    ,    @    :    .    -    L    P    +
                     ; r4|    0    O    K    M    N    J    I    9
                     ; r3|    8    U    H    B    V    G    Y    7
                     ; r2|    6    T    F    C    X    D    R    5
                     ; r1|    4    E    S    Z  LSH    A    W    3
                     ; r0|    2    Q  CBM   SP  RUN  CTL  LFT    1

VIA2_DDRB  = $9122   ; VIA 2 data direction register B
VIA2_DDRA  = $9123   ; VIA 2 data direction register A
VIA2_T1CL  = $9124   ; VIA 2 T1 low  order counter/latch
VIA2_T1CH  = $9125   ; VIA 2 T1 high order counter/latch
VIA2_T1LL  = $9126   ; VIA 2 T1 low  order latch
VIA2_T1LH  = $9127   ; VIA 2 T1 high order latch
VIA2_T2CL  = $9128   ; VIA 2 T2 low  order counter/latch
VIA2_T2CH  = $9129   ; VIA 2 T2 high order counter/latch
VIA2_SR    = $912A   ; VIA 2 shift register              (SR)
VIA2_ACR   = $912B   ; VIA 2 auxiliary control register  (ACR)
VIA2_PCR   = $912C   ; VIA 2 peripheral control register (PCR)

; the status bit is a not normal flag. it goes high if both an interrupt
; flag in the IFR and the corresponding enable bit in the IER are set.
; it can be cleared only by clearing all the active flags in the IFR or
; disabling all active interrupts in the IER.

VIA2_IFR   = $912D   ; VIA 1 Interrupt Flag Register (IFR)
                     ; bit   function        cleared by
                     ; ---  ---------        ----------
                     ;  7   interrupt status clearing all enabled interrupts
                     ;  6   T1 interrupt     read T1C_l, write T1C_h
                     ;  5   T2 interrupt     read T2C_l, write T2C_h
                     ;  4   CB1 transition   read or write port B
                     ;  3   CB2 transition   read or write port B
                     ;  2   8 shifts done    read or write the shift register
                     ;  1   CA1 transition   read or write port A
                     ;  0   CA2 transition   read or write port A

; If enable/disable bit is a zero during a write to this register, each
; 1 in bits 0-6 clears the corresponding bit in the IER. if this bit is
; a one during a write to this register, each 1 in bits 0-6 will set the
; corresponding IER bit

VIA2_IER   = $912E   ; VIA 1 Interrupt Enable Register (IER)
                     ; bit   function
                     ; ---   --------
                     ;  7   enable/disable
                     ;  6   T1 interrupt
                     ;  5   T2 interrupt
                     ;  4   CB1 transition
                     ;  3   CB2 transition
                     ;  2   8 shifts done
                     ;  1   CA1 transition
                     ;  0   CA2 transition

VIA2_DATN  = $912F   ; VIA 2 DRA, keyboard row, no handshake

; register names for keyboard driver

KEYB_COL   = VIA2_DATB
KEYB_ROW   = VIA2_DATA
KEYB_ROWN  = VIA2_DATN

; key coordinates

CTRL_COL = %11111011 ; $fb = col 2
CTRL_ROW = %11111110 ; $fe = row 0

STND_COL = %11110111 ; $7f = col 3

; constants for screen editor

COLS          =  22 ; screen columns
ROWS          =  23 ; screen rows
COLINK        =   4 ; possible physical lines per logical line
COLMAX        =  88 ; maximum line length of a logical line
COLRAM_PAGE   = $94 ; default page of color RAM
Default_Color =   6 ; blue

IEC_PCR       = $912C; VIA 2 peripheral control register (PCR)
IEC_DRAN      = $911F
IEC_TIM_H     = $9129; VIA 2 T2H, timer high
IEC_IFR       = $912D; VIA 2 IFR, interrupt flag register
IEC_ATN_BIT   = $80
IEC_IFR_BIT   = $20
IEC_CLK_BIT   = %00000010; $02
IEC_DAT_BIT   = %00100000; $20
RS232_C_BIT   = %00100000; $20
RS2_IRQ_REG   = $911E
RS2_DSR_CTS   = $9110
RS2_TIM_LOW   = $9114
RS2_TIM_HIG   = $9115
MEM_CONTROL   = $9005

OPTION_ROM    = $A000
BASIC_ROM     = $C000
#endif

#if C64

VIC_BASE      = $D000
VIC_CONTROL_1 = $D011
VIC_RASTER    = $D012
VIC_SPR_ENA   = $D015
VIC_CONTROL_2 = $D016

VIC_REGS = $2F

; ******************************************
; C-64 VIC-II (MOS (6566/7) Video Controller
; ******************************************

;  #| Adr.  |Bit7|Bit6|Bit5|Bit4|Bit3|Bit2|Bit1|Bit0| Function
; --+-------+----+----+----+----+----+----+----+----+------------------------
;  0| $d000 |                  M0X                  | X coordinate sprite 0
; --+-------+---------------------------------------+------------------------
;  1| $d001 |                  M0Y                  | Y coordinate sprite 0
; --+-------+---------------------------------------+------------------------
;  2| $d002 |                  M1X                  | X coordinate sprite 1
; --+-------+---------------------------------------+------------------------
;  3| $d003 |                  M1Y                  | Y coordinate sprite 1
; --+-------+---------------------------------------+------------------------
;  4| $d004 |                  M2X                  | X coordinate sprite 2
; --+-------+---------------------------------------+------------------------
;  5| $d005 |                  M2Y                  | Y coordinate sprite 2
; --+-------+---------------------------------------+------------------------
;  6| $d006 |                  M3X                  | X coordinate sprite 3
; --+-------+---------------------------------------+------------------------
;  7| $d007 |                  M3Y                  | Y coordinate sprite 3
; --+-------+---------------------------------------+------------------------
;  8| $d008 |                  M4X                  | X coordinate sprite 4
; --+-------+---------------------------------------+------------------------
;  9| $d009 |                  M4Y                  | Y coordinate sprite 4
; --+-------+---------------------------------------+------------------------
; 10| $d00a |                  M5X                  | X coordinate sprite 5
; --+-------+---------------------------------------+------------------------
; 11| $d00b |                  M5Y                  | Y coordinate sprite 5
; --+-------+---------------------------------------+------------------------
; 12| $d00c |                  M6X                  | X coordinate sprite 6
; --+-------+---------------------------------------+------------------------
; 13| $d00d |                  M6Y                  | Y coordinate sprite 6
; --+-------+---------------------------------------+------------------------
; 14| $d00e |                  M7X                  | X coordinate sprite 7
; --+-------+---------------------------------------+------------------------
; 15| $d00f |                  M7Y                  | Y coordinate sprite 7
; --+-------+----+----+----+----+----+----+----+----+------------------------
; 16| $d010 |M7X8|M6X8|M5X8|M4X8|M3X8|M2X8|M1X8|M0X8| MSBs of X coordinates
; --+-------+----+----+----+----+----+----+----+----+------------------------
; 17| $d011 |RST8| ECM| BMM| DEN|RSEL|    YSCROLL   | Control register 1
; --+-------+----+----+----+----+----+--------------+------------------------
; 18| $d012 |                 RASTER                | VIC_RASTER counter
; --+-------+---------------------------------------+------------------------
; 19| $d013 |                  LPX                  | Light pen X
; --+-------+---------------------------------------+------------------------
; 20| $d014 |                  LPY                  | Light pen Y
; --+-------+----+----+----+----+----+----+----+----+------------------------
; 21| $d015 | M7E| M6E| M5E| M4E| M3E| M2E| M1E| M0E| Sprite enabled
; --+-------+----+----+----+----+----+----+----+----+------------------------
; 22| $d016 |  - |  - | RES| MCM|CSEL|    XSCROLL   | Control register 2
; --+-------+----+----+----+----+----+----+----+----+------------------------
; 23| $d017 |M7YE|M6YE|M5YE|M4YE|M3YE|M2YE|M1YE|M0YE| Sprite Y expansion
; --+-------+----+----+----+----+----+----+----+----+------------------------
; 24| $d018 |VM13|VM12|VM11|VM10|CB13|CB12|CB11|  - | Memory pointers
; --+-------+----+----+----+----+----+----+----+----+------------------------
; 25| $d019 | IRQ|  - |  - |  - | ILP|IMMC|IMBC|IRST| Interrupt register
; --+-------+----+----+----+----+----+----+----+----+------------------------
; 26| $d01a |  - |  - |  - |  - | ELP|EMMC|EMBC|ERST| Interrupt enabled
; --+-------+----+----+----+----+----+----+----+----+------------------------
; 27| $d01b |M7DP|M6DP|M5DP|M4DP|M3DP|M2DP|M1DP|M0DP| Sprite data priority
; --+-------+----+----+----+----+----+----+----+----+------------------------
; 28| $d01c |M7MC|M6MC|M5MC|M4MC|M3MC|M2MC|M1MC|M0MC| Sprite multicolor
; --+-------+----+----+----+----+----+----+----+----+------------------------
; 29| $d01d |M7XE|M6XE|M5XE|M4XE|M3XE|M2XE|M1XE|M0XE| Sprite X expansion
; --+-------+----+----+----+----+----+----+----+----+------------------------
; 30| $d01e | M7M| M6M| M5M| M4M| M3M| M2M| M1M| M0M| Sprite-sprite collision
; --+-------+----+----+----+----+----+----+----+----+------------------------
; 31| $d01f | M7D| M6D| M5D| M4D| M3D| M2D| M1D| M0D| Sprite-data collision
; --+-------+----+----+----+----+----+----+----+----+-------------------------
; 32| $d020 |  - |  - |  - |  - |         EC        | Border color
; --+-------+----+----+----+----+-------------------+-------------------------
; 33| $d021 |  - |  - |  - |  - |        B0C        | Background color 0
; --+-------+----+----+----+----+-------------------+-------------------------
; 34| $d022 |  - |  - |  - |  - |        B1C        | Background color 1
; --+-------+----+----+----+----+-------------------+-------------------------
; 35| $d023 |  - |  - |  - |  - |        B2C        | Background color 2
; --+-------+----+----+----+----+-------------------+-------------------------
; 36| $d024 |  - |  - |  - |  - |        B3C        | Background color 3
; --+-------+----+----+----+----+-------------------+-------------------------
; 37| $d025 |  - |  - |  - |  - |        MM0        | Sprite multicolor 0
; --+-------+----+----+----+----+-------------------+-------------------------
; 38| $d026 |  - |  - |  - |  - |        MM1        | Sprite multicolor 1
; --+-------+----+----+----+----+-------------------+-------------------------
; 39| $d027 |  - |  - |  - |  - |        M0C        | Color sprite 0
; --+-------+----+----+----+----+-------------------+-------------------------
; 40| $d028 |  - |  - |  - |  - |        M1C        | Color sprite 1
; --+-------+----+----+----+----+-------------------+-------------------------
; 41| $d029 |  - |  - |  - |  - |        M2C        | Color sprite 2
; --+-------+----+----+----+----+-------------------+-------------------------
; 42| $d02a |  - |  - |  - |  - |        M3C        | Color sprite 3
; --+-------+----+----+----+----+-------------------+-------------------------
; 43| $d02b |  - |  - |  - |  - |        M4C        | Color sprite 4
; --+-------+----+----+----+----+-------------------+-------------------------
; 44| $d02c |  - |  - |  - |  - |        M5C        | Color sprite 5
; --+-------+----+----+----+----+-------------------+-------------------------
; 45| $d02d |  - |  - |  - |  - |        M6C        | Color sprite 6
; --+-------+----+----+----+----+-------------------+-------------------------
; 46| $d02e |  - |  - |  - |  - |        M7C        | Color sprite 7
; --+-------+----+----+----+----+-------------------+-------------------------

; **********************************************
; C-64 CIA1 (MOS 6526) Complex Interface Adapter
; **********************************************

;  #| Adr.  |Bit7|Bit6|Bit5|Bit4|Bit3|Bit2|Bit1|Bit0| Function
; --+-------+----+----+----+----+----+----+----+----+-------------------------
;  0| $dc00 |                  PRA                  | Data port A
; --+-------+---------------------------------------+-------------------------
;  1| $dc01 |                  PRB                  | Data port B
; --+-------+---------------------------------------+-------------------------
;  2| $dc02 |                 DDRA                  | Data direction A
; --+-------+---------------------------------------+-------------------------
;  3| $dc03 |                 DDRB                  | Data direction B
; --+-------+---------------------------------------+-------------------------
;  4| $dc04 |                 TALO                  | Timer A low
; --+-------+---------------------------------------+-------------------------
;  5| $dc05 |                 TAHI                  | Timer A high
; --+-------+---------------------------------------+-------------------------
;  6| $dc06 |                 TBLO                  | Timer B low
; --+-------+---------------------------------------+-------------------------
;  7| $dc07 |                 TBHI                  | Timer B high
; --+-------+----+----+----+----+----+----+----+----+-------------------------
;  8| $dc08 | S/A|  0 |  0 |  0 |       TODS        | Time Of Day [1/10 sec]
; --+-------+----+----+----+----+----+----+----+----+-------------------------
;  9| $dc09 |  0 |  TODS [10s]  |�    TODS [1s]     | Time Of Day [sec]
; --+-------+----+--------------+-------------------+-------------------------
; 10| $dc0a |  0 |  TODM [10m]  |     TODM [1m]     | Time Of Day [min]
; --+-------+----+--------------+-------------------+-------------------------
; 11| $dc0b |AMPM|  TODH [10h]  |     TODH [1h]     | Time Of Day [hour]
; --+-------+----+--------------+-------------------+-------------------------
; 12| $dc0c |                  SDR                  | Serial Data Register
; --+-------+----+----+----+----+----+----+----+----+-------------------------
; 13| $dc0d |MODE|  0 |  0 |IFLG|ISDR|IARM|ITBU|ITBA| Interrupt Control Reg.
; --+-------+----+----+----+----+----+----+----+----+-------------------------
; 14| $dc0e | Hz |DSDR|CNTP| ONE|CONT|TBUN|TBUN|STOP| Control Timer A
; --+-------+----+----+----+----+----+----+----+----+-------------------------
; 15| $dc0f |MODE|   TACT  | ONE|CONT|TAUN|TAUN|STOP| Control Timer B
; --+-------+----+----+----+----+----+----+----+----+-------------------------

CIA1_PRA   = $DC00   ; CIA1 Peripheral data Register A
                     ; keyboard column

CIA1_PRB   = $DC01   ; CIA1 Peripheral data Register B
                     ; keyboard row
                     ; C64  keyboard matrix layout
                     ;       c7   c6   c5   c4   c3   c2   c1   c0
                     ;   +-------------------------------------------------
                     ; r7|  RUN    /    ,    N    V    X  LSH   DN
                     ; r6|    Q   UP    @    O    U    T    E   F5
                     ; r5|  CBM    =    :    K    H    F    S   F3
                     ; r4|   SP  RSH    .    M    B    C    Z   F1
                     ; r3|    2  HOM    -    0    8    6    4   F7
                     ; r2|  CTL    ;    L    J    G    D    A  RGT
                     ; r1|  LFT    *    P    I    Y    R    W  RET
                     ; r0|    1    �    +    9    7    5    3  DEL

CIA1_DDRA  = $DC02   ; CIA1 Data Direction  Register A
CIA1_DDRB  = $DC03   ; CIA1 Data Direction  Register B
CIA1_TALO  = $DC04   ; CIA1 Timer A Low  register
CIA1_TAHI  = $DC05   ; CIA1 Timer A High register
CIA1_TBLO  = $DC06   ; CIA1 Timer B Low  register
CIA1_TBHI  = $DC07   ; CIA1 Timer B High register
CIA1_TODT  = $DC08   ; CIA1 Time Of Day 1/10 seconds
CIA1_TODS  = $DC09   ; CIA1 Time Of Day seconds
CIA1_TODM  = $DC0A   ; CIA1 Time Of Day minutes
CIA1_TODH  = $DC0B   ; CIA1 Time Of Day hours
CIA1_SDR   = $DC0C   ; CIA1 Serial Data Register
CIA1_ICR   = $DC0D   ; CIA1 Interrupt Control Register
CIA1_CRA   = $DC0E   ; CIA1 Control Register A
CIA1_CRB   = $DC0F   ; CIA1 Control Register B

; **********************************************
; C-64 CIA2 (MOS 6526) Complex Interface Adapter
; **********************************************

CIA2_PRA   = $DD00   ; CIA2 Peripheral data Register A
CIA2_PRB   = $DD01   ; CIA2 Peripheral data Register B
CIA2_DDRA  = $DD02   ; CIA2 Data Direction  Register A
CIA2_DDRB  = $DD03   ; CIA2 Data Direction  Register B
CIA2_TALO  = $DD04   ; CIA2 Timer A Low  register
CIA2_TAHI  = $DD05   ; CIA2 Timer A High register
CIA2_TBLO  = $DD06   ; CIA2 Timer B Low  register
CIA2_TBHI  = $DD07   ; CIA2 Timer B High register
CIA2_TODT  = $DD08   ; CIA2 Time Of Day 1/10 seconds
CIA2_TODS  = $DD09   ; CIA2 Time Of Day seconds
CIA2_TODM  = $DD0A   ; CIA2 Time Of Day minutes
CIA2_TODH  = $DD0B   ; CIA2 Time Of Day hours
CIA2_SDR   = $DD0C   ; CIA2 Serial Data Register
CIA2_ICR   = $DD0D   ; CIA2 Interrupt Control Register
CIA2_CRA   = $DD0E   ; CIA2 Control Register A
CIA2_CRB   = $DD0F   ; CIA2 Control Register B
```

```
; register names for keyboard driver

KEYB_COL   = CIA1_PRA
KEYB_ROW   = CIA1_PRB
KEYB_ROWN  = CIA1_PRB

; key coordinates

CTRL_COL = %01111111 ; $7f = col 7
CTRL_ROW = %11111011 ; $fb = row 2

STND_COL = %01111111 ; $7f = col 7

D6510       =     0
R6510       =     1

VIA2_IER = CIA1_ICR    ; CIA1 Interrupt Control Register
VIA2_T2CH= $DC07
VIA2_T2CL= $DC06
VIA2_T1CL= $DC04
VIA2_T1CH= $DC05

COLS           =  40
ROWS           =  25
COLINK         =   2
COLMAX         =  80
COLRAM_PAGE    = $D8
Default_Color = 14   ; Light blue

MEM_CONTROL    = $D018
IEC_TIM_H      = $DC07; CIA 1 TIH, timer high
IEC_IFR        = CIA1_ICR    ; CIA 1 CRB, interrupt flag register
IEC_PCR        = $DD00; VIA 2 peripheral control register (PCR)
IEC_DRAN       = $DD00; CIA 2 DRA, IEC bus

; bit 7 IEC Bus Data  Input
;     6 IEC Bus Clock Input
;     5 IEC Bus Data  Output
;     4 IEC Bus Clock Output
;     3 IEC Bus ATN Signal Output
;     2 User port PA2
;   0-1 VIC memory address

RS2_DSR_CTS    = $DD01
RS2_IRQ_REG    = CIA2_ICR
RS2_TIM_LOW    = $DD04
RS2_TIM_HIG    = $DD05
IEC_ATN_BIT    = $08
IEC_IFR_BIT    = $02
IEC_CLK_BIT    = %00010000; $10
IEC_DAT_BIT    = %00100000; $20
RS232_C_BIT    = %00000100; $04

BASIC_SCREEN    = $0400
BASIC_RAM_START = $0800
OPTION_ROM      = $8000
BASIC_ROM       = $A000
#endif

; Some names for special characters

CR    = $0D ; Carriage Return
```

```
LF    = $0A ; Line Feed
QUOTE = $22 ; Quote
SEMIC = $3B ; Semicolon

; These locations contain the JMP instruction and traget address of the
; USR command. They are initialised so that if you try to execute a USR
; call without changing them you will receive an ILLEGAL QUANTITY error
; message.

#if VIC
Basic_USR = $00      ; USR() JMP instruction ($4C)
USRVEC    = $01      ; USR() vector (Illegal_Quantity)
#endif

; This vector points to the address of the BASIC routine which converts
; a floating point number to an integer, however BASIC does not use this
; vector. It may be of assistance to the programmer who wishes to use
; data that is stored in floating point format. The parameter passed by
; the USR command is available only in that format for example.

ADRAY1 = $03         ; float to fixed vector (Float_To_Integer)

; This vector points to the address of the BASIC routine which converts
; an integer to a floating point number, however BASIC does not use this
; vector. It may be used by the programmer who needs to make such a
; conversion for a machine language program that interacts with BASIC.
; To return an integer value with the USR command for example.

ADRAY2 = $05         ; fixed to float vector (Integer_To_Float)

; The cursor column position prior to the TAB or SPC is moved here from
; $D3, and is used to calculate where the cursor ends up after one of
; these functions is invoked. Note that the value contained here shows
; the position of the cursor on a logical line. Since one logical line
; can be up to four VIC or two C64 physical lines long, the value stored
; here can range from 0 to 87 on VIC and 0 to 79 on C64.

CHARAC = $07         ; search character
ENDCHR = $08         ; scan quotes flag
TRMPOS = $09         ; TAB column save

; The routine that converts the text in the input buffer into lines of
; executable program tokens, and the routines that link these program
; lines together, use this location as an index into the input buffer
; area. After the job of converting text to tokens is done, the value
; in this location is equal to the length of the tokenized line.

; The routines which build an array or locate an element in an array use
; this location to calculate the number of DIMensions called for and the
;  amount of storage required for a newly created array, or the number
; of subscripts when referencing an array element.

VERCK  = $0A         ; load/verify flag, 0 = load, 1 = verify
COUNT  = $0B         ; line crunch/array access/logic operators

; This is used as a flag by the routines that build an array or
; reference an existing array. It is used to determine whether a
; variable is in an array, whether the array has already been
; DIMensioned, and whether a new array should assume default size.

DIMFLG = $0C         ; DIM flag
```

```
; This flag is used to indicate whether data being operated upon is
; string or numeric. A value of $FF in this location indicates string
; data while a $00 indicates numeric data.

VALTYP = $0D         ; data type flag, $FF = string, $00 = numeric

; If the above flag indicates numeric then a $80 in this location
; identifies the number as an integer, and a $00 indicates a floating
; point number.

INTFLG = $0E         ; data type flag, $80 = integer, $00 = floating pt.

; The garbage collection routine uses this location as a flag to
; indicate that garbage collection has already been tried before adding
; a new string. If there is still not enough memory, an OUT OF MEMORY
; error message will result.

; LIST uses this byte as a flag to let it know when it has come to a
; character string in quotes. It will then print the string,rather than
; search it for BASIC keyword tokens.

; This location is also used during the process of converting a line of
; text in the BASIC input buffer into a linked program line of BASIC
; keyword tokens to flag a DATA line is being processed.

GARBFL = $0F         ; garbage collected/open quote/DATA flag

; If an opening parenthesis is found, this flag is set to indicate that
; the variable in question is either an array variable or a user-defined
; function.

SUBFLG = $10         ; subscript/FNx flag

; This location is used to determine whether the sign of the value
; returned by the functions SIN, COS, ATN or TAN is positive or negative

; Also the comparison routines use this location to indicate the outcome
; of the compare. For A <=> B the value here will be $01 if A > B,
; $02 if A = B, and $04 if A < B. If more than one comparison operator
; was used to compare the two variables then the value here will be a
; combination of the above values.

INPFLG = $11         ; input mode, $00 = INPUT, $40 = GET, $98 = READ
TANSGN = $12         ; ATN sign/comparison evaluation flag

; When the default input or output device is used the value here will
; be a zero, and the format of prompting and output will be the standard
; screen output format. The location $B8 is used to decide what device
; actually to put input from or output to.

; The print CR/LF code at Print_CR suggests that b7 of this byte is
; an AutoLF flag bit but if it is used as such it would break lots of
; other parts of the code

IOPMPT = $13         ; current I/O channel

; Used whenever a 16 bit integer is used e.g. the target line number for
; GOTO, LIST, ON, and GOSUB also the number of a BASIC line that is to
; be added or replaced. additionally PEEK, POKE, WAIT, and SYS use this
; location as a pointer to the address which is the subject of the
; command.
```

```
LINNUM = $14         ; temporary integer

; This location points to the next available slot in the temporary
; string descriptor stack located at $19-$21.

TEMPPT = $16         ; descriptor stack pointer, next free

; This contains information about temporary strings which hve not yet
; been assigned to a string variable.

LASTPT = $17         ; current descriptor stack item pointer
TEMPST = $19         ; to $21, descriptor stack

; These locations are used by BASIC multiplication and division
; routines. They are also used by the routines which compute the size of
; the area required to store an array which is being created.

INDEXA = $22         ; misc temp word
INDEXB = $24         ; misc temp word

FAC3   = $25         ; auxiliary Floating Point Accumulator

#if JIFFY
COMSAV = $27
#endif

; Two byte pointer to where the BASIC program text is stored.

TXTTAB = $2B         ; start of memory

; Two byte pointer to the start of the BASIC variable storage area.

VARTAB = $2D         ; start of variables

; Two byte pointer to the start of the BASIC array storage area.

ARYTAB = $2F         ; end of variables

; Two byte pointer to end of the start of free RAM.

STREND = $31         ; end of arrays

; Two byte pointer to the bottom of the string text storage area.

FRESPC = $33         ; bottom of string space

; Used as a temporary pointer to the most current string added by the
; routines which build strings or move them in memory.

UTLSTP = $35         ; string utility ptr

; Two byte pointer to the highest address used by BASIC +1.

MEMSIZ = $37         ; end of memory

; These locations contain the line number of the BASIC statement which
; is currently being executed. A value of $FF in location $3A means that
; BASIC is in immediate mode.

CURLIN = $39         ; current line number

; When program execution ends or stops the last line number executed is
; stored here.

OLDLIN = $3B         ; break line number

; These locations contain the address of the start of the text of the
; BASIC statement that is being executed.  The value of the pointer to
; the address of the BASIC text character currently being scanned is
; stored here each time a new BASIC statement begins execution.

OLDTXT = $3D         ; continue pointer

; These locations hold the line number of the current DATA statement
; being READ. If an error concerning the DATA occurs this number will
; be moved to $39/$3A so that the error message will show the line that
; contains the DATA statement rather than in the line that contains the
; READ statement.

DATLIN = $3F         ; current DATA line number

; These locations point to the address where the next DATA will be READ
; from. RESTORE sets this pointer back to the address indicated by the
; start of BASIC pointer.

DATPTR = $41         ; DATA pointer

; READ, INPUT and GET all use this as a pointer to the address of the
; source of incoming data, such as DATA statements, or the text input
; buffer.

INPPTR = $43         ; READ pointer

VARNAM = $45         ; current variable name

; These locations point to the value of the current BASIC variable.
; Specifically they point to the byte just after the two-character
; variable name.

VARPNT = $47         ; current variable address

; The address of the BASIC variable which is the subject of a FOR/NEXT
; loop is first stored here before being pushed onto the stack.

FORPNT = $49         ; FOR/NEXT variable pointer

; The expression evaluation routine creates this to let it know whether
; the current comparison operation is a < $01, = $02 or > $04 comparison
; or combination.

YSAVE  = $4B         ; BASIC execute pointer temporary/precedence flag

ACCSYM = $4D         ; comparrison evaluation flag

; These locations are used as a pointer to the function that is created
; during function definition. During function execution it points to
; where the evaluation results should be saved.

FUNCPT = $4E         ; FAC temp store/function/variable/garbage pointer

; Temporary Pointer to the current string descriptor.

DESCPT = $50         ; FAC temp store/descriptor pointer
```

```
GARBSS = $53         ; garbage collection step size

; The first byte is the 6502 JMP instruction $4C, followed by the
; address of the required function taken from the table at $C052.

JUMPER = $54         ; JMP opcode for functions

FUNJMP = $55         ; functions jump vector

; this address is sometimes used as high btye for the FUNJMP
; and as rounding byte (5th. byte of mantissa) for FAC1

FAC2M5 = $56         ; FAC2 mantissa 5 = rounding byte

; Temporary storage for floating points values (5 bytes)
; and temporary pointer (block pointer, array pointer)

FACTPA = $57         ; FAC temp store ($57 - $5B)
TMPPTA = $58         ; temp pointer A
TMPPTB = $5A         ; temp pointer B

; Temporary storage for floating points values (5 bytes)
; and temporary variables

FACTPB = $5C         ; FAC temp store ($5C - $60)
TMPVA1 = $5D         ; temp variable (counter)
TMPVA2 = $5E         ; temp variable (counter)
TMPPTC = $5F         ; temp pointer C

; Floating point accumulator 1

FAC1EX = $61         ; FAC1 exponent
FAC1M1 = $62         ; FAC1 mantissa 1
FAC1M2 = $63         ; FAC1 mantissa 2
FAC1M3 = $64         ; FAC1 mantissa 3
FAC1M4 = $65         ; FAC1 mantissa 4
FAC1SI = $66         ; FAC1 sign

SGNFLG = $67         ; constant count/negative flag
FAC1OV = $68         ; FAC1 overflow

; Floating point accumulator 2

FAC2EX = $69         ; FAC2 exponent
FAC2M1 = $6A         ; FAC2 mantissa 1
FAC2M2 = $6B         ; FAC2 mantissa 2
FAC2M3 = $6C         ; FAC2 mantissa 3
FAC2M4 = $6D         ; FAC2 mantissa 4
FAC2SI = $6E         ; FAC2 sign

; String pointer and FAC sign comparison and FAC rounding

STRPTR = $6F         ; string pointer & FAC variables

; this address is sometimes used as high btye for the STRPTR
; and as rounding byte (5th. byte of mantissa) for FAC1

FAC1M5 = $70         ; FAC1 mantissa 5 = rounding byte

TMPPTD = $71         ; temp BASIC execute/array pointer low byte/index

; Basic CHRGET (with increment) and CHRGOT (no increment) routine
```

```
CHRGET = $73          ; next program byte, BASIC byte get
CHRGOT = $79          ; scan memory, BASIC byte get
TXTPTR = $7A          ; BASIC execute pointer
ISNUM  = $80          ; numeric test entry

RNDX   = $8B          ; RND() seed, five bytes

STATUS = $90          ; serial status byte
                      ;   function
                      ; bit   casette      serial bus
                      ; ---   --------      ----------
                      ;  7   end of tape      device not present
                      ;  6   end of file      EOI
                      ;  5   checksum error
                      ;  4   read error
                      ;  3   long block
                      ;  2   short block
                      ;  1            time out read
                      ;  0            time out write

STKEY  = $91          ; keyboard row, bx = 0 = key down
                      ; bit   key
                      ; ---   ------
                      ;  7   [DOWN]
                      ;  6   /
                      ;  5   ,
                      ;  4   N
                      ;  3   V
                      ;  2   X
                      ;  1   [L SHIFT]
                      ;  0   [STOP]

SVXT   = $92          ; timing constant for tape read
VERCKK = $93          ; load/verify flag, load = $00, verify = $01
C3PO   = $94          ; serial output: deferred character flag
                      ; $00 = no character waiting, $xx = character waiting
BSOUR  = $95          ; serial output: deferred character
                      ; $FF = no character waiting, $xx = waiting character
SYNO   = $96          ; cassette block synchronization number
TEMPX  = $97          ; register save

; The number of currently open I/O files is stored here. The maximum
; number that can be open at one time is ten. The number stored here is
; used as the index to the end of the tables that hold the file numbers,
; device numbers, and secondary addresses.

LDTND  = $98          ; open file count

; The default value of this location is 0.

DFLTN  = $99          ; input device number

; The default value of this location is 3.

DFLTO  = $9A          ; output device number
                      ;
                      ; number   device
                      ; ------   ------
                      ;  0      keyboard
                      ;  1      cassette
                      ;  2      RS-232C
```

```
                     ;  3      screen
                     ;  4-31   serial bus

PRTY   = $9B         ; tape character parity
DPSW   = $9C         ; byte received flag
MSGFLG = $9D         ; message mode flag,
                     ; $C0 = both control and kernal messages,
                     ; $80 = control messages only,
                     ; $40 = kernal messages only,
                     ; $00 = neither control or kernal messages
PTR1   = $9E         ; index to cassette file name/header ID
PTR2   = $9F         ; tape Pass 2 error log corrected

; These three locations form a counter which is updated 60 times a
; second, and serves as a software clock which counts the number of
; jiffies that have elapsed since the computer was turned on. After 24
; hours and one jiffy these locations are set back to $000000.

JIFFYH = $A0         ; jiffy clock high byte
JIFFYM = $A1         ; jiffy clock mid byte
JIFFYL = $A2         ; jiffy clock low byte
TSFCNT = $A3         ; EOI flag byte/tape bit count or Jiffy device flag
TBTCNT = $A4         ; tape bit cycle phase
CNTDN  = $A5         ; tape synch byte count/serial bus bit count
BUFPNT = $A6         ; tape buffer index
INBIT  = $A7         ; receiver input bit temp storage
BITCI  = $A8         ; receiver bit count in
RINONE = $A9         ; receiver start bit check flag, $90 = no start bit
RIDATA = $AA         ; receiver byte buffer/assembly location
RIPRTY = $AB         ; receiver parity bit storage
SAL    = $AC         ; tape buffer start pointer; scroll screen
EAL    = $AE         ; tape buffer end   pointer; scroll screen
CMPO   = $B0         ; tape timing constant (word)

; These two locations point to the address of the cassette buffer. This
; pointer must be greater than or equal to $0200 or an ILLEGAL DEVICE
; NUMBER error will be sent when tape I/O is tried. This pointer must
; also be less that $8000 or the routine will terminate early.

TAPE1  = $B2         ; tape buffer start pointer

; RS232 routines use this to count the number of bits transmitted and
; for parity and stop bit manipulation. Tape load routines use this
; location to flag when they are ready to receive data bytes.

BITTS  = $B4         ; transmitter bit count out

; This location is used by the RS232 routines to hold the next bit to
; be sent and by the tape routines to indicate what part of a block the
; read routine is currently reading.

NXTBIT = $B5         ; transmitter next bit to be sent

; RS232 routines use this area to disassemble each byte to be sent from
; the transmission buffer pointed to by $F9.

RODATA = $B6         ; transmitter byte buffer/disassembly location

; Disk filenames may be up to 16 characters in length while tape
; filenames be up to 187 characters in length.
; If a tape name is longer than 16 characters the excess will be
; truncated by the SEARCHING and FOUND messages, but will still be
; present on the tape.
; A disk file is always referred to by a name. This location will always
; be greater than zero if the current file is a disk file.

; An RS232 OPEN command may specify a filename of up to four characters.
; These characters are copied to locations $293 to $296 and determine
; baud rate, word length, and parity, or they would do if the feature
; was fully implemented.

FNLEN  = $B7         ; file name length

LA     = $B8         ; logical file
SA     = $B9         ; secondary address
FA     = $BA         ; current device number
                     ; number   device
                     ; ------   ------
                     ;  0      keyboard
                     ;  1      cassette
                     ;  2      RS-232C
                     ;  3      screen
                     ;  4-31   serial bus
FNADR  = $BB         ; file name pointer
ROPRTY = $BD         ; tape write byte/RS232 parity byte

; Used by the tape routines to count the number of copies of a data
; block remaining to be read or written.

#if JIFFY
Jiffy_Device = $BE
#endif
FSBLK  = $BE         ; tape copies count
MYCH   = $BF         ; parity count ??
CAS1   = $C0         ; tape motor interlock
STAL   = $C1         ; I/O start addresses
MEMUSS = $C3         ; kernal setup pointer
LSTX   = $C5         ; current key pressed
                     ;
                     ;  # key    # key    # key    # key
                     ; -- ---   -- ---   -- ---   -- ---
                     ; 00 1   10 none   20 [SPACE]   30 Q
                     ; 01 3   11 A      21 Z      31 E
                     ; 02 5   12 D      22 C      32 T
                     ; 03 7   13 G      23 B      33 U
                     ; 04 9   14 J      24 M      34 O
                     ; 05 +   15 L      25 .      35 @
                     ; 06 [UKP]   16 ;      26 none   36 ^
                     ; 07 [DEL]   17 [CSR R]   27 [F1]   37 [F5]
                     ; 08 [<-]   18 [STOP]   28 none   38 2
                     ; 09 W   19 none   29 S      39 4
                     ; 0A R   1A X      2A F      3A 6
                     ; 0B Y   1B V      2B H      3B 8
                     ; 0C I   1C N      2C K      3C 0
                     ; 0D P   1D ,      2D :      3D -
                     ; 0E *   1E /      2E =      3E [HOME]
                     ; 0F [RET]   1F [CSR D]   2F [F3]   3F [F7]

NDX    = $C6         ; keyboard buffer length/index

; When the [CTRL][RVS-ON] characters are printed this flag is set to
; $12, and the print routines will add $80 to the screen code of each
; character which is printed, so that the caracter will appear on the
; screen with its colours reversed.
; Note that the contents of this location are cleared not only upon
; entry of a [CTRL][RVS-OFF] character but also at every carriage return

RVS    = $C7         ; reverse flag $12 = reverse, $00 = normal

; This pointer indicates the column number of the last nonblank
; character on the logical line that is to be input. Since a logical
; line can be up to 88 characters long this number can range from 0-87.

INDX   = $C8         ; input [EOL] pointer

; These locations keep track of the logical line that the cursor is on
; and its column position on that logical line.

; Each logical line may contain up to four 22 column physical lines. So
; there may be as many as 23 logical lines, or as few as 6 at any one
; time. Therefore, the logical line number might be anywhere from 1-23.
; Depending on the length of the logical line, the cursor column may be
; from 1-22, 1-44, 1-66 or 1-88.

; For a more on logical lines, see the description of the screen line
; link table, $D9.

ICRROW = $C9         ; input cursor row
ICRCOL = $CA         ; input cursor column

; The keyscan interrupt routine uses this location to indicate which key
; is currently being pressed. The value here is then used as an index
; into the appropriate keyboard table to determine which character to
; print when a key is struck.

; The correspondence between the key pressed and the number stored here
; is as follows:

; $00   1      $10   unused $20   [SPC]  $30   Q
; $01   3      $11   A      $21   Z      $31   E
; $02   5      $12   D      $22   C      $32   T
; $03   7      $13   G      $23   B      $33   U
; $04   9      $14   J      $24   M      $34   O
; $05   +      $15   L      $25   .      $35   @
; $06   [PND]  $16   ;      $26   unused $36   [U ARROW]
; $07   [DEL]  $17   [RIGHT]$27   [F1]   $37   [F5]
; $08   [<-]   $18   [STOP] $28   unused $38   2
; $09   W      $19   unused $29   S      $39   4
; $0A   R      $1A   X      $2A   F      $3A   6
; $0B   Y      $1B   V      $2B   H      $3B   8
; $0C   I      $1C   N      $2C   K      $3C   0
; $0D   P      $1D   ,      $2D   :      $3D   -
; $0E   *      $1E   /      $2E   =      $3E   [HOME]
; $0F   [RET]  $1F   [DOWN] $2F   [F3]   $3F   [F7]

SFDX   = $CB         ; which key

; When this flag is set to a nonzero value, it indicates to the routine
; that normally flashes the cursor not to do so. The cursor blink is
; turned off when there are characters in the keyboard buffer, or when
; the program is running.

BLNSW  = $CC         ; cursor enable, $00 = flash cursor

; The routine that blinks the cursor uses this location to tell when
; it's time for a blink. The number 20 is put here and decremented every
; jiffy until it reaches zero. Then the cursor state is changed, the
; number 20 is put back here, and the cycle starts all over again.

BLNCT  = $CD         ; cursor timing countdown

; The cursor is formed by printing the inverse of the character that
; occupies the cursor position. If that characters is the letter A, for
; example, the flashing cursor merely alternates between printing an A
; and a reverse-A. This location keeps track of the normal screen code
; of the character that is located at the cursor position, so that it
; may be restored when the cursor moves on.

GDBLN  = $CE         ; character under cursor

; This location keeps track of whether, during the current cursor blink,
; the character under the cursor was reversed, or was restored to
; normal. This location will contain 0 if the character is reversed, and
; 1 if the character is not reversed.

BLNON  = $CF         ; cursor blink phase

; input from keyboard or screen, $xx = input is available from the
; screen, $00 = input should be obtained from the keyboard

INSRC  = $D0         ; input from keyboard or screen

; These locations point to the address in screen RAM of the first column
; of the logical line upon which the cursor is currently positioned.

LINPTR = $D1         ; current screen line pointer

; This holds the cursor column position within the logical line pointed
; to by LINPTR. Since a logical line can comprise up to four physical
; lines, this value may be from $00 to $57.

CSRIDX = $D3         ; cursor column

; A nonzero value in this location indicates that the editor is in quote
; mode. Quote mode is toggled every time that you type in a quotation
; mark on a given line, the first quote mark turns it on, the second
; turns it off, the third turns it on, etc.

; If the editor is in this mode when a cursor control character or other
; nonprinting character is entered, a printed equivalent will appear on
; the screen instead of the cursor movement or other control operation
; taking place. Instead, that action is deferred until the string is
; sent to the string by a PRINT statement, at which time the cursor
; movement or other control operation will take place.

; The exception to this rule is the DELETE key, which will function
; normally within quote mode. The only way to print a character which is
; equivalent to the DELETE key is by entering insert mode. Quote mode
; may be exited by printing a closing quote or by hitting the RETURN or
; SHIFT-RETURN keys.

CSRMOD = $D4         ; cursor quote flag

; The line editor uses this location when the end of a line has been
; reached to determine whether another physical line can be added to the
; current logical line or if a new logical line must be started.
```

```
LINLEN = $D5         ; current screen line length

; This location contains the current physical screen line position of
; the cursor, 0 to 22 for VIC or 0 to 24 for C64.

TBLX   = $D6         ; cursor row

; The ASCII value of the last character printed to the screen is held
; here temporarily.

LASTKY = $D7         ; checksum byte/temporary last character

; When the INST key is pressed, the screen editor shifts the line to the
; right, allocates another physical line to the logical line if
; necessary (and possible), updates the screen line length in $D5, and
; adjusts the screen line link table at $D9. This location is used to
; keep track of the number of spaces that has been opened up in this way.

; Until the spaces that have been opened up are filled, the editor acts
; as if in quote mode. See location $D4, the quote mode flag. This means
; that cursor control characters that are normally nonprinting will
; leave a printed equivalent on the screen when entered, instead of
; having their normal effect on cursor movement, etc. The only
; difference between insert and quote mode is that the DELETE key will
; leave a printed equivalent in insert mode, while the INSERT key will
; insert spaces as normal.

INSRTO = $D8         ; insert count

; This table contains 23/25 entries, one for each row of the screen
; display. Each entry has two functions. Bits 0-3 indicate on which of
; the four pages of screen memory the first byte of memory for that row
; is located. This is used in calculating the pointer to the starting
; address of a screen line at $D1.
;
; The high byte is calculated by adding the value of the starting page
; of screen memory held in $288 to the displacement page held here.
;
; The other function of this table is to establish the makeup of logical
; lines on the screen. While each screen line is only 22/40 characters
; long, BASIC allows the entry of program lines that contain up to 88
; characters. Therefore, some method must be used to determine which
; physical lines are linked into a longer logical line, so that this
; longer logical line may be edited as a unit.
;
; The high bit of each byte here is used as a flag by the screen editor.
; That bit is set when a line is the first or only physical line in a
; logical line. The high bit is reset to 0 only when a line is an
; extension to this logical line.

SLLTBL = $D9         ; to SLLTBL + $18 inclusive, screen line link table

; This pointer is synchronized with the pointer to the address of the
; first byte of screen RAM for the current line kept in location $D1. It
; holds the address of the first byte of colour RAM for the
; corresponding screen line.

#if VIC
SCROWM = $F2         ; screen row marker
#endif

USER   = $F3         ; colour RAM pointer
```

```
; This pointer points to the address of the keyboard matrix lookup table
; currently being used. Although there are only 64 keys on the keyboard
; matrix, each key can be used to print up to four different characters,
; depending on whether it is struck by itself or in combination with the
; SHIFT, CTRL, or C= keys.

; These tables hold the ASCII value of each of the 64 keys for one of
; these possible combinations of keypresses. When it comes time to print
; the character, the table that is used determines which character is
; printed.

; The addresses of the tables are:

;   KBD_NORMAL          ; unshifted
;   KBD_SHIFTED         ; shifted
;   KBD_CBMKEY          ; commodore
;   KBD_CONTROL         ; control

KBDPTR = $F5         ; keyboard pointer

; When device the RS232 channel is opened two buffers of 256 bytes each
; are created at the top of memory. These locations point to the address
; of the one which is used to store characters as they are received.

RXPTR  = $F7         ; RS232 Rx pointer

; These locations point to the address of the 256 byte output buffer
; that is used for transmitting data to RS232 devices.

TXPTR  = $F9         ; RS232 Tx pointer

BASSTO = $FF         ; FAC1 to string output base

STACK  = $0100       ; bottom of the stack page

BUF    = $0200       ;   input buffer. for some routines the byte before
                     ;   the input buffer needs to be set to a specific
                     ;   value for the routine to work correctly

FILTBL = $0259       ;   .. to $0262 logical file table
DEVTBL = $0263       ;   .. to $026C device number table
SECATB = $026D       ;   .. to $0276 secondary address table
KBUFFR = $0277       ;   .. to $0280 keyboard buffer
OSSTAR = $0281       ;   OS start of memory low byte
OSTOP  = $0283       ;   OS top of memory low byte
STIMOT = $0285       ;   serial bus timeout flag
COLOR  = $0286       ;   current colour code
CSRCLR = $0287       ;   colour under cursor
SCNMPG = $0288       ;   screen memory page
KBMAXL = $0289       ;   maximum keyboard buffer size
KEYRPT = $028A       ;   key repeat. $80 = repeat all, $40 = repeat none,
                     ;   $00 = repeat cursor movement keys, insert/delete
                     ;   key and the space bar
KRPTSP = $028B       ;   repeat speed counter
KRPTDL = $028C       ;   repeat delay counter

; This flag signals which of the SHIFT, CTRL, or C= keys are currently
; being pressed.

; A value of $01 signifies that one of the SHIFT keys is being pressed,
; a $02 shows that the C= key is down, and $04 means that the CTRL key
; is being pressed. If more than one key is held down, these values will
; be added e.g $03 indicates that SHIFT and C= are both held down.

; Pressing the SHIFT and C= keys at the same time will toggle the
; character set that is presently being used between the uppercase/
; graphics set, and the lowercase/uppercase set.

; While this changes the appearance of all of the characters on the
; screen at once it has nothing whatever to do with the keyboard shift
; tables and should not be confused with the printing of SHIFTed
; characters, which affects only one character at a time.

SHFLAG   = $028D     ; keyboard shift/control flag
                     ; bit   key(s) 1 = down
                     ; ---   ---------------
                     ; 7-3   unused
                     ;  2   CTRL
                     ;  1   C=
                     ;  0   SHIFT

; This location, in combination with the one above, is used to debounce
; the special SHIFT keys. This will keep the SHIFT/C= combination from
; changing character sets back and forth during a single pressing of
; both keys.

LSTSHF   = $028E     ; SHIFT/CTRL/C= keypress last pattern

; This location points to the address of the Operating System routine
; which actually determines which keyboard matrix lookup table will be
; used.

; The routine looks at the value of the SHIFT flag at $28D, and based on
; what value it finds there, stores the address of the correct table to
; use at location $F5.

KEYLOG   = $028F     ; keyboard decode logic pointer

; This flag is used to enable or disable the feature which lets you
; switch between the uppercase/graphics and upper/lowercase character
; sets by pressing the SHIFT and Commodore logo keys simultaneously.

MODE     = $0291     ; shift mode switch, $00 = enabled, $80 = locked

; This location is used to determine whether moving the cursor past the
; xx  column of a logical line will cause another physical line to be
; added to the logical line.

; A value of 0 enables the screen to scroll the following lines down in
; order to add that line; any nonzero value will disable the scroll.

; This flag is set to disable the scroll temporarily when there are
; characters waiting in the keyboard buffer, these may include cursor
; movement characters that would eliminate the need for a scroll.

AUTODN   = $0292     ; screen scrolling flag, $00 = enabled

M51CTR   = $0293     ; pseudo 6551 control register. the first character of
                     ; the OPEN RS232 filename will be stored here
                     ; bit   function
                     ; ---   --------
                     ;  7   2 stop bits/1 stop bit
                     ; 65   word length
```

```
                        ; ---   -----------
                        ; 00   8 bits
                        ; 01   7 bits
                        ; 10   6 bits
                        ; 11   5 bits
                        ;  4   unused
                        ; 3210   baud rate
                        ; ----   ---------
                        ; 0000   user rate *
                        ; 0001      50
                        ; 0010      75
                        ; 0011     110
                        ; 0100     134.5
                        ; 0101     150
                        ; 0110     300
                        ; 0111     600
                        ; 1000    1200
                        ; 1001    1800
                        ; 1010    2400
                        ; 1011    3600
                        ; 1100    4800 *
                        ; 1101    7200 *
                        ; 1110    9600 *
                        ; 1111   19200 *   * = not implemented
M51CDR   = $0294        ; pseudo 6551 command register. the second character of
                        ; the OPEN RS232 filename will be stored here
                        ; bit   function
                        ; ---   --------
                        ; 765   parity
                        ; ---   ------
                        ; xx0   disabled
                        ; 001   odd
                        ; 011   even
                        ; 101   mark
                        ; 111   space
                        ;  4   duplex half/full
                        ;  3   unused
                        ;  2   unused
                        ;  1   unused
                        ;  0   handshake - X line/3 line
M51AJB   = $0295        ; Nonstandard Bit Timing. the third and fourth character
                        ; of the OPEN RS232 filename will be stored here
RSSTAT   = $0297        ; RS-232 status register
                        ; bit   function
                        ; ---   --------
                        ;  7   break
                        ;  6   no DSR detected
                        ;  5   unused
                        ;  4   no CTS detected
                        ;  3   unused
                        ;  2   Rx buffer overrun
                        ;  1   framing error
                        ;  0   parity error
BITNUM   = $0298        ; number of bits to be sent/received
BAUDOF   = $0299        ; time of one bit cell
RIDBE    = $029B        ; index to Rx buffer end
RIDBS    = $029C        ; index to Rx buffer start
RODBS    = $029D        ; index to Tx buffer start
RODBE    = $029E        ; index to Tx buffer end
IRQTMP   = $029F        ; saved IRQ

#if C64
```

```
ENABL    = $02A1
TODSNS   = $02A2
TRDTMP   = $02A3
TD1IRQ   = $02A4
SCROWM   = $02A5     ; screen row marker
TVSFLG   = $02A6     ; PAL / NTSC flag
#endif

IERROR   = $0300     ; vector to the print BASIC error message routine
IMAIN    = $0302     ; Vector to the main BASIC program Loop
ICRNCH   = $0304     ; Vector to the the ASCII text to keywords routine
IQPLOP   = $0306     ; Vector to the list BASIC program as ASCII routine
IGONE    = $0308     ; Vector to the execute next BASIC command routine
IEVAL    = $030A     ; Vector to the get value from BASIC line routine

; Before every SYS command each of the registers is loaded with the
; value found in the corresponding storage address. Upon returning to
; BASIC with an RTS instruction, the new value of each register is
; stored in the appropriate storage address.

; This feature allows you to place the necessary values into the
; registers from BASIC before you SYS to a Kernal or BASIC ML routine.
; It also enables you to examine the resulting effect of the routine on
; the registers, and to preserve the condition of the registers on exit
; for subsequent SYS calls.

SAREG    = $030C     ; A for SYS command
SXREG    = $030D     ; X for SYS command
SYREG    = $030E     ; Y for SYS command
SPREG    = $030F     ; P for SYS command

#if C64
Basic_USR= $0310     ; USR() JMP instruction ($4C)
USRVEC   = $0311     ; USR() vector
#endif

CINV     = $0314     ; IRQ vector
CBINV    = $0316     ; BRK vector
NMINV    = $0318     ; NMI vector

IOPEN    = $031A     ; kernal vector - open a logical file
ICLOSE   = $031C     ; kernal vector - close a logical file
ICHKIN   = $031E     ; kernal vector - open channel for input
ICKOUT   = $0320     ; kernal vector - open channel for output
ICLRCH   = $0322     ; kernal vector - close input and output channels
IBASIN   = $0324     ; kernal vector - input character from channel
IBSOUT   = $0326     ; kernal vector - output character to channel
ISTOP    = $0328     ; kernal vector - check if stop key is pressed
IGETIN   = $032A     ; kernal vector - get character from keyboard queue
ICLALL   = $032C     ; kernal vector - close all channels and files
USRCMD   = $032E     ; kernal vector - user IRQ
ILOAD    = $0330     ; kernal vector - load
ISAVE    = $0332     ; kernal vector - save

TBUFFR   = $033C     ; to $03FB - cassette buffer

; hardware equates

#if C64
   IO_Base_Address = $DC00
#endif
```

```
#if VIC
   IO_Base_Address = $9110
#endif

; command tokens

TK_END     = $80   ; END token
TK_FOR     = $81   ; FOR token
TK_NEXT    = $82   ; NEXT token
TK_DATA    = $83   ; DATA token
TK_INFL    = $84   ; INPUT# token
TK_INPUT   = $85   ; INPUT token
TK_DIM     = $86   ; DIM token
TK_READ    = $87   ; READ token

TK_LET     = $88   ; LET token
TK_GOTO    = $89   ; GOTO token
TK_RUN     = $8A   ; RUN token
TK_IF      = $8B   ; IF token
TK_RESTORE = $8C   ; RESTORE token
TK_GOSUB   = $8D   ; GOSUB token
TK_RETURN  = $8E   ; RETURN token
TK_REM     = $8F   ; REM token

TK_STOP    = $90   ; STOP token
TK_ON      = $91   ; ON token
TK_WAIT    = $92   ; WAIT token
TK_LOAD    = $93   ; LOAD token
TK_SAVE    = $94   ; SAVE token
TK_VERIFY  = $95   ; VERIFY token
TK_DEF     = $96   ; DEF token
TK_POKE    = $97   ; POKE token

TK_PRINFL  = $98   ; PRINT# token
TK_PRINT   = $99   ; PRINT token
TK_CONT    = $9A   ; CONT token
TK_LIST    = $9B   ; LIST token
TK_CLR     = $9C   ; CLR token
TK_CMD     = $9D   ; CMD token
TK_SYS     = $9E   ; SYS token
TK_OPEN    = $9F   ; OPEN token

TK_CLOSE   = $A0   ; CLOSE token
TK_GET     = $A1   ; GET token
TK_NEW     = $A2   ; NEW token

; secondary keyword tokens

TK_TAB     = $A3   ; TAB( token
TK_TO      = $A4   ; TO token
TK_FN      = $A5   ; FN token
TK_SPC     = $A6   ; SPC( token
TK_THEN    = $A7   ; THEN token

TK_NOT     = $A8   ; NOT token
TK_STEP    = $A9   ; STEP token

; operator tokens

TK_PLUS    = $AA   ; + token
TK_MINUS   = $AB   ; - token
TK_MUL     = $AC   ; * token
```

```
TK_DIV     = $AD   ; / token
TK_POWER   = $AE   ; ^ token
TK_AND     = $AF   ; AND token

TK_OR      = $B0   ; OR token
TK_GT      = $B1   ; > token
TK_EQUAL   = $B2   ; = token
TK_LT      = $B3   ; < token

; function tokens

TK_SGN     = $B4   ; SGN token
TK_INT     = $B5   ; INT token
TK_ABS     = $B6   ; ABS token
TK_USR     = $B7   ; USR token

TK_FRE     = $B8   ; FRE token
TK_POS     = $B9   ; POS token
TK_SQR     = $BA   ; SQR token
TK_RND     = $BB   ; RND token
TK_LOG     = $BC   ; LOG token
TK_EXP     = $BD   ; EXP token
TK_COS     = $BE   ; COS token
TK_SIN     = $BF   ; SIN token

TK_TAN     = $C0   ; TAN token
TK_ATN     = $C1   ; ATN token
TK_PEEK    = $C2   ; PEEK token
TK_LEN     = $C3   ; LEN token
TK_STRS    = $C4   ; STR$ token
TK_VAL     = $C5   ; VAL token
TK_ASC     = $C6   ; ASC token
TK_CHRS    = $C7   ; CHR$ token

TK_LEFTS   = $C8   ; LEFT$ token
TK_RIGHTS  = $C9   ; RIGHT$ token
TK_MIDS    = $CA   ; MID$ token
TK_GO      = $CB   ; GO token
TK_PI      = $FF   ; PI token

; KERNAL Jump Table
; ----------------------------------------------------------------------
; ACPTR       $FFA5 65445    Input byte from serial port
; CHKIN       $FFC6 65478    Open channel for input
; CHKOUT      $FFC9 65481    Open a channel for output
; CHRIN       $FFCF 65487    Get a character from the input channel
; CHROUT      $FFD2 65490    Output a character
; CIOUT       $FFA8 65448    Output byte to serial port
; CLALL       $FFE7 65511    Close all channels and files
; CLOSE       $FFC3 65475    Close a specified logical file
; CLRCHN      $FFCC 65484    Clear I/O channels
; GETIN       $FFE4 65512    Get character from keyboard buffer
; IOBASE      $FFF3 65523    Define I/O memory page
; LISTEN      $FFB1 65457    Command devices on the serial bus to LISTEN
; LOAD        $FFD5 65493    Load RAM from a device
; MEMBOT      $FF9C 65436    Read/set the bottom of memory
; MEMTOP      $FF99 65433    Read/set the top of memory
; OPEN        $FFC0 65472    Open a logical file
; PLOT        $FFF0 65520    Read or set cursor location
; RDTIM       $FFDE 65502    Read system clock
; READST      $FFB7 65463    Read I/O status word
; RESTOR      $FF8A 65415    Restore default I/O vectors
; SAVE        $FFD8 65496    Save RAM to device
; SCNKEY      $FF9F 65439    Scan keyboard
; SCREEN      $FFED 65517    Return screen format
; SECOND      $FF93 65427    Send secondary address after LISTEN
; SETLFS      $FFBA 65466    Set logical, first and second addresses
; SETMSG      $FF90 65424    Control KERNAL messages
; SETNAM      $FFBD 65469    Set filename
; SETTIM      $FFDB 65499    Set the system clock
; SETTMO      $FFA2 65442    Set timeout on serial bus
; STOP        $FFE1 65505    Scan stop key
; TALK        $FFB4 65460    Command serial bus device to TALK
; TKSA        $FF96 65430    Send secondary address after TALK
; UDTIM       $FFEA 65514    Update the system clock
; UNLSN       $FFAE 65454    Command serial bus to UNLISTEN
; UNTLK       $FFAB 65451    Command serial bus to UNTALK
; VECTOR      $FF8D 65421    Read/set vectored I/O


; **********************
; BASIC scalar variables
; **********************

; ---------+-----+-----+-----+-----+-----+-----+-----+-----
; Type     | Exa.|  0  |  1  |  2  |  3  |  4  |  5  |  6
; ---------+-----+-----+-----+-----+-----+-----+-----+-----
; Float    | AB  |  A  |  B  | EXP | MSB |�MAN | MAN |�LSB
; ---------+-----+-----+-----+-----+-----+-----+-----+-----
; Integer  | AB% |  A^ |� B^ | MSB | LSB |  0  |  0  |  0
; ---------+-----+-----+-----+-----+-----+-----+-----+-----
; Function | AB( |  A^ |� B  | LFP | MFP | LBP | MBP |  0
; ---------+-----+-----+-----+-----+-----+-----+-----+-----
; String   | AB$ |  A  |� B^ |�LEN | LSP | MSP |  0  |  0
; ---------+-----+-----+-----+-----+-----+-----+-----+-----

; ************
; BASIC arrays
; ************

; ---------+-----+-----+-----+---------------+
; Type     | Exa.|  0  |  1  |  Element Size |
; ---------+-----+-----+-----+---------------+
; Float    | AB  |  A  |  B  |       5       |
; ---------+-----+-----+-----+---------------+
; Integer  | AB% |  A^ |� B^ |       2       |
; ---------+-----+-----+-----+---------------+
; String   | AB$ |  A  |� B^ |       3       |
; ---------+-----+-----+-----+---------------+

; The circumflex ^ indicates characters OR'ed with $80

; Array header:

; Byte  0   : 1st. character of name
; Byte  1   : 2nd, character of name
; Byte  2   : dimension count (1, 2, or 3)
; Byte  3-4 : Hi/Lo lements of 1st. dimension, (e.g. 11 for dim a(10)
; Byte  5-6 : elements of 2nd. dimension if dimension count > 1
; Byte  7-8 : elements of 3rd. dimension if dimension count > 2

          .org    BASIC_ROM
#if C64
          .store  BASIC_ROM,$2000,"basic_64.rom"
```

```
#endif
#if VIC
          .store  BASIC_ROM,$2000,"basic_20.rom"
#endif
          .word   Basic_Cold_Start
BASIC_BRK .word   Basic_Warm_Start
BASIC_ID  .byte   "CBMBASIC"

; *********************
  Basic_Statement_Table
; *********************

   .word Basic_END      -1; $80
   .word Basic_FOR      -1; $81
   .word Basic_NEXT     -1; $82
   .word Basic_DATA     -1; $83
   .word Basic_INPUTN   -1; $84
   .word Basic_INPUT    -1; $85
   .word Basic_DIM      -1; $86
   .word Basic_READ     -1; $87
   .word Basic_LET      -1; $88
   .word Basic_GOTO     -1; $89
   .word Basic_RUN      -1; $8A
   .word Basic_IF       -1; $8B
   .word Basic_RESTORE  -1; $8C
   .word Basic_GOSUB    -1; $8D
   .word Basic_RETURN   -1; $8E
   .word Basic_REM      -1; $8F
   .word Basic_STOP     -1; $90
   .word Basic_ON       -1; $91
   .word Basic_WAIT     -1; $92
   .word Basic_LOAD     -1; $93
   .word Basic_SAVE     -1; $94
   .word Basic_VERIFY   -1; $95
   .word Basic_DEF      -1; $96
   .word Basic_POKE     -1; $97
   .word Basic_PRINTN   -1; $98
   .word Basic_PRINT    -1; $99
   .word Basic_CONT     -1; $9A
   .word Basic_LIST     -1; $9B
   .word Basic_CLR      -1; $9C
   .word Basic_CMD      -1; $9D
   .word Basic_SYS      -1; $9E
   .word Basic_OPEN     -1; $9F
   .word Basic_CLOSE    -1; $A0
   .word Basic_GET      -1; $A1
   .word Basic_NEW      -1; $A2

; ********************
  Basic_Function_Table
; ********************

   .word Basic_SGN   ; $B4
   .word Basic_INT   ; $B5
   .word Basic_ABS   ; $B6
   .word Basic_USR   ; $B7
   .word Basic_FRE   ; $B8
   .word Basic_POS   ; $B9
   .word Basic_SQR   ; $BA
   .word Basic_RND   ; $BB
   .word Basic_LOG   ; $BC
   .word Basic_EXP   ; $BD
   .word Basic_COS   ; $BE
   .word Basic_SIN   ; $BF
   .word Basic_TAN   ; $C0
   .word Basic_ATN   ; $C1
   .word Basic_PEEK  ; $C2
   .word Basic_LEN   ; $C3
   .word Basic_STR   ; $C4
   .word Basic_VAL   ; $C5
   .word Basic_ASC   ; $C6
   .word Basic_CHR   ; $C7
   .word Basic_LEFT  ; $C8
   .word Basic_RIGHT ; $C9
   .word Basic_MID   ; $CA

; ********************
  Basic_Operator_Table
; ********************

   .byte $79,Basic_PLUS     -1 ; $AA
   .byte $79,Basic_MINUS    -1 ; $AB
   .byte $7B,Basic_MULTIPLY -1 ; $AC
   .byte $7B,Basic_DIVIDE   -1 ; $AD
   .byte $7F,Basic_POWER    -1 ; $AE
   .byte $50,Basic_AND      -1 ; $AF
   .byte $46,Basic_OR       -1 ; $B0
   .byte $7D,Basic_GREATER  -1 ; $B1
   .byte $5A,Basic_EQUAL    -1 ; $B2
   .byte $64,Basic_LESS     -1 ; $B3

; *******************
  Basic_Keyword_Table
; *******************

   .byte   "END"^    ; END
   .byte   "FOR"^    ; FOR
   .byte   "NEXT"^   ; NEXT
   .byte   "DATA"^   ; DATA
   .byte   "INPUT#"^ ; INPUT#
   .byte   "INPUT"^  ; INPUT
   .byte   "DIM"^    ; DIM
   .byte   "READ"^   ; READ
   .byte   "LET"^    ; LET
   .byte   "GOTO"^   ; GOTO
   .byte   "RUN"^    ; RUN
   .byte   "IF"^     ; IF
   .byte   "RESTORE"^; RESTORE
   .byte   "GOSUB"^  ; GOSUB
   .byte   "RETURN"^ ; RETURN
   .byte   "REM"^    ; REM
   .byte   "STOP"^   ; STOP
   .byte   "ON"^     ; ON
   .byte   "WAIT"^   ; WAIT
   .byte   "LOAD"^   ; LOAD
   .byte   "SAVE"^   ; SAVE
   .byte   "VERIFY"^ ; VERIFY
   .byte   "DEF"^    ; DEF
   .byte   "POKE"^   ; POKE
   .byte   "PRINT#"^ ; PRINT#
   .byte   "PRINT"^  ; PRINT
   .byte   "CONT"^   ; CONT
   .byte   "LIST"^   ; LIST
   .byte   "CLR"^    ; CLR
   .byte   "CMD"^    ; CMD
   .byte   "SYS"^    ; SYS
   .byte   "OPEN"^   ; OPEN
   .byte   "CLOSE"^  ; CLOSE
   .byte   "GET"^    ; GET
   .byte   "NEW"^    ; NEW

; next are the secondary command keywords, these can not start a statement

   .byte   "TAB("^   ; TAB(
   .byte   "TO"^     ; TO
   .byte   "FN"^     ; FN
   .byte   "SPC("^   ; SPC(
   .byte   "THEN"^   ; THEN
   .byte   "NOT"^    ; NOT
   .byte   "STEP"^   ; STEP

; the operators

   .byte   "+"^      ; +
   .byte   "-"^      ; -
   .byte   "*"^      ; *
   .byte   "/"^      ; /
   .byte   "^"^      ; ^
   .byte   "AND"^    ; AND
   .byte   "OR"^     ; OR
   .byte   ">"^      ; >
   .byte   "="^      ; =
   .byte   "<"^      ; <

; the functions

   .byte   "SGN"^    ; SGN
   .byte   "INT"^    ; INT
   .byte   "ABS"^    ; ABS
   .byte   "USR"^    ; USR
   .byte   "FRE"^    ; FRE
   .byte   "POS"^    ; POS
   .byte   "SQR"^    ; SQR
   .byte   "RND"^    ; RND
   .byte   "LOG"^    ; LOG
   .byte   "EXP"^    ; EXP
   .byte   "COS"^    ; COS
   .byte   "SIN"^    ; SIN
   .byte   "TAN"^    ; TAN
   .byte   "ATN"^    ; ATN
   .byte   "PEEK"^   ; PEEK
   .byte   "LEN"^    ; LEN
   .byte   "STR$"^   ; STR$
   .byte   "VAL"^    ; VAL
   .byte   "ASC"^    ; ASC
   .byte   "CHR$"^   ; CHR$
   .byte   "LEFT$"^  ; LEFT$
   .byte   "RIGHT$"^ ; RIGHT$
   .byte   "MID$"^   ; MID$
   .byte   "GO"^     ; GO so that GO TO, as well as GOTO, will work
   .byte   $00       ; end marker

; error messages

ERR_01 .byte   "TOO MANY FILES"^
ERR_02 .byte   "FILE OPEN"^
```

```
ERR_03 .byte   "FILE NOT OPEN"^
ERR_04 .byte   "FILE NOT FOUND"^
ERR_05 .byte   "DEVICE NOT PRESENT"^
ERR_06 .byte   "NOT INPUT FILE"^
ERR_07 .byte   "NOT OUTPUT FILE"^
ERR_08 .byte   "MISSING FILE NAME"^
ERR_09 .byte   "ILLEGAL DEVICE NUMBER"^
ERR_0A .byte   "NEXT WITHOUT FOR"^
ERR_0B .byte   "SYNTAX"^
ERR_0C .byte   "RETURN WITHOUT GOSUB"^
ERR_0D .byte   "OUT OF DATA"^
ERR_0E .byte   "ILLEGAL QUANTITY"^
ERR_0F .byte   "OVERFLOW"^
ERR_10 .byte   "OUT OF MEMORY"^
ERR_11 .byte   "UNDEF'D STATEMENT"^
ERR_12 .byte   "BAD SUBSCRIPT"^
ERR_13 .byte   "REDIM'D ARRAY"^
ERR_14 .byte   "DIVISION BY ZERO"^
ERR_15 .byte   "ILLEGAL DIRECT"^
ERR_16 .byte   "TYPE MISMATCH"^
ERR_17 .byte   "STRING TOO LONG"^
ERR_18 .byte   "FILE DATA"^
ERR_19 .byte   "FORMULA TOO COMPLEX"^
ERR_1A .byte   "CAN'T CONTINUE"^
ERR_1B .byte   "UNDEF'D FUNCTION"^
ERR_1C .byte   "VERIFY"^
ERR_1D .byte   "LOAD"^

; error message pointer table

Basic_Msg_Tab
   .word   ERR_01  ; TOO MANY FILES
   .word   ERR_02  ; FILE OPEN
   .word   ERR_03  ; FILE NOT OPEN
   .word   ERR_04  ; FILE NOT FOUND
   .word   ERR_05  ; DEVICE NOT PRESENT
   .word   ERR_06  ; NOT INPUT FILE
   .word   ERR_07  ; NOT OUTPUT FILE
   .word   ERR_08  ; MISSING FILE NAME
   .word   ERR_09  ; ILLEGAL DEVICE NUMBER
   .word   ERR_0A  ; NEXT WITHOUT FOR
   .word   ERR_0B  ; SYNTAX
   .word   ERR_0C  ; RETURN WITHOUT GOSUB
   .word   ERR_0D  ; OUT OF DATA
   .word   ERR_0E  ; ILLEGAL QUANTITY
   .word   ERR_0F  ; OVERFLOW
   .word   ERR_10  ; OUT OF MEMORY
   .word   ERR_11  ; UNDEF'D STATEMENT
   .word   ERR_12  ; BAD SUBSCRIPT
   .word   ERR_13  ; REDIM'D ARRAY
   .word   ERR_14  ; DIVISION BY ZERO
   .word   ERR_15  ; ILLEGAL DIRECT
   .word   ERR_16  ; TYPE MISMATCH
   .word   ERR_17  ; STRING TOO LONG
   .word   ERR_18  ; FILE DATA
   .word   ERR_19  ; FORMULA TOO COMPLEX
   .word   ERR_1A  ; CAN'T CONTINUE
   .word   ERR_1B  ; UNDEF'D FUNCTION
   .word   ERR_1C  ; VERIFY
   .word   ERR_1D  ; LOAD
   .word   Msg_Break
```

```
; BASIC messages

Msg_OK    .byte   "\rOK\r",0
#if C64
Msg_Err   .byte   "  ERROR",0
#endif
#if VIC
Msg_Err   .byte   "\r ERROR",0
#endif
Msg_IN    .byte   " IN ",0
Msg_Ready .byte   "\r\nREADY.\r\n",0
Msg_CrLf  .byte   "\r\n"
Msg_Break .byte   "BREAK",0

   .byte $A0 ; unused

; ***************
  Find_Active_FOR
; ***************

   TSX               ; copy stack pointer
   INX               ; +1 pass return address
   INX               ; +2 pass return address
   INX               ; +3 pass calling routine return address
   INX               ; +4 pass calling routine return address

FAF_Loop
   LDA STACK+1,X     ; get token byte from stack
   CMP #TK_FOR       ; is it FOR token
   BNE FAF_Ret       ; exit if not FOR token
   LDA FORPNT+1      ; get FOR/NEXT variable pointer high byte
   BNE FAF_10        ; branch if defined
   LDA STACK+2,X     ; get FOR variable pointer low byte
   STA FORPNT        ; save FOR/NEXT variable pointer low byte
   LDA STACK+3,X     ; get FOR variable pointer high byte
   STA FORPNT+1      ; save FOR/NEXT variable pointer high byte

FAF_10
   CMP STACK+3,X     ; compare variable pointer with stacked variable pointer
   BNE FAF_20        ; branch if no match
   LDA FORPNT        ; get FOR/NEXT variable pointer low byte
   CMP STACK+2,X     ; compare variable pointer with stacked variable pointer
   BEQ FAF_Ret       ; exit if match found

FAF_20
   TXA               ; copy index
   CLC
   ADC #18           ; add FOR stack use size
   TAX               ; copy back to index
   BNE FAF_Loop

FAF_Ret
   RTS

; *************
  Open_Up_Space
; *************

   JSR Check_Mem_Avail
   STAY(STREND)

; **********
```

```
  Move_Block
; **********

   SEC
   LDA TMPPTB        ; get block end low byte
   SBC TMPPTC        ; subtract block start low byte
   STA INDEXA        ; save MOD(block length/$100) byte
   TAY               ; copy MOD(block length/$100) byte to Y
   LDA TMPPTB+1      ; get block end high byte
   SBC TMPPTC+1      ; subtract block start high byte
   TAX               ; copy block length high byte to X
   INX               ; +1 to allow for count=0 exit
   TYA               ; copy block length low byte to A
   BEQ MoBl_20       ; branch if length low byte=0
   LDA TMPPTB        ; get block end low byte
   SEC
   SBC INDEXA        ; subtract MOD(block length/$100) byte
   STA TMPPTB        ; save corrected old block end low byte
   BCS MoBl_10       ; if no underflow skip the high byte decrement
   DEC TMPPTB+1      ; else decrement block end high byte
   SEC

MoBl_10
   LDA TMPPTA        ; get destination end low byte
   SBC INDEXA        ; subtract MOD(block length/$100) byte
   STA TMPPTA        ; save modified new block end low byte
   BCS MoBl_Loop_X   ; if no underflow skip the high byte decrement
   DEC TMPPTA+1      ; else decrement block end high byte
   BCC MoBl_Loop_X   ; branch always

MoBl_Loop_Y
   LDA (TMPPTB),Y    ; get byte from source
   STA (TMPPTA),Y    ; copy byte to destination

MoBl_Loop_X
   DEY
   BNE MoBl_Loop_Y
   LDA (TMPPTB),Y    ; get byte from source
   STA (TMPPTA),Y    ; save byte to destination

MoBl_20
   DEC TMPPTB+1      ; decrement source pointer high byte
   DEC TMPPTA+1      ; decrement destination pointer high byte
   DEX               ; decrement block count
   BNE MoBl_Loop_X   ; loop until count = $0
   RTS

; *****************
  Check_Stack_Avail
; *****************

   ASL A
   ADC #$3E          ; need at least $3E bytes free
   BCS Error_Out_Of_Memory
   STA INDEXA        ; save result in temp byte
   TSX               ; copy stack
   CPX INDEXA        ; compare new limit with stack
   BCC Error_Out_Of_Memory
   RTS

; ***************
  Check_Mem_Avail
```

```
; ***************

   CPY FRESPC+1      ; compare with bottom of string space high byte
   BCC CMA_Ret       ; if less then OK
   BNE CMA_10        ; skip next test if greater (tested <)
   CMP FRESPC        ; compare with bottom of string space low byte
   BCC CMA_Ret       ; if less then OK

CMA_10               ; address is > string storage ptr
   PHA               ; push address low byte
   LDX #9            ; set index to save 10 bytes (FACTPA & FACTPB)
   TYA               ; copy address high byte (to push on stack)

CMA_Loop_X
   PHA               ; push byte
   LDA FACTPA,X      ; get byte from FACTPA to TMPPTC+1
   DEX
   BPL CMA_Loop_X
   JSR Garbage_Collection
   LDX #$F7          ; use zero page wrap around

CMA_Loop_2
   PLA               ; pop byte
   STA FAC1EX,X      ; save byte from FACTPA to FAC1
   INX               ; increment index
   BMI CMA_Loop_2
   PLA               ; pop address high byte
   TAY               ; copy back to Y
   PLA               ; pop address low byte
   CPY FRESPC+1      ; compare with bottom of string space high byte
   BCC CMA_Ret       ; if less then OK
   BNE Error_Out_Of_Memory
   CMP FRESPC
   BCS Error_Out_Of_Memory

CMA_Ret
   RTS

; ===================
  Error_Out_Of_Memory
; ===================

   LDX #$10          ; error code $10, out of memory error

; ===========
  Basic_Error
; ===========

   JMP (IERROR)      ; normally next statement

; =============
  Default_Error
; =============

   TXA               ; copy error number
   ASL A             ; *2
   TAX               ; copy to index
   LDA Basic_Msg_Tab-2,X  ; get error message pointer low byte
   STA INDEXA        ; save it
   LDA Basic_Msg_Tab-1,X  ; get error message pointer high byte
   STA INDEXA+1      ; save it
   JSR CLRCHN        ; Clear I/O channels
   LDA #0
   STA IOPMPT        ; clear current I/O channel, flag default
   JSR Print_CR
   JSR Print_Question_Mark
   LDY #0

DeEr_Loop
   LDA (INDEXA),Y    ; get byte from message
   PHA               ; save status
   AND #$7F          ; mask 0xxx xxxx, clear b7
   JSR Print_Char
   INY
   PLA               ; restore status
   BPL DeEr_Loop
   JSR Flush_BASIC_Stack
   LAYI(Msg_Err)

; ======================
  Display_Msg_Then_Ready
; ======================

   JSR Print_String
   LDY CURLIN+1      ; get current line number high byte
   INY               ; increment it
   BEQ Basic_Ready   ; CURLIN+1 was $FF = direct mode
   JSR Print_IN

; ===========
  Basic_Ready
; ===========

   Print_Msg(Msg_Ready)
   LDA #$80          ; set for control messages only
   JSR SETMSG        ; control kernal messages

; ==================
  Vectored_Warmstart
; ==================

   JMP (IMAIN)       ; normally next statement

; =================
  Default_Warmstart
; =================

   JSR Read_String
   STXY(TXTPTR)
   JSR CHRGET
   TAX               ; copy byte to set flags
   BEQ Vectored_Warmstart ; loop if no input

; got to interpret input line now ....

   LDX #-1           ; indicates direct mode
   STX CURLIN+1
   BCC New_Basic_Line

Direct_Call          ; used from Jiffy to identify caller
   JSR Tokenize_Line
   JMP Start_Program

; --------------
```

```
  New_Basic_Line
; --------------

   JSR Scan_Linenumber
   JSR Tokenize_Line
   STY COUNT         ; save index pointer to end of crunched line
   JSR Find_BASIC_Line
   BCC NBL_20        ; if not found skip the line delete

; line # already exists so delete it

   LDY #1            ; set index to next line pointer high byte
   LDA (TMPPTC),Y    ; get next line pointer high byte
   STA INDEXA+1      ; save it
   LDA VARTAB        ; get start of variables low byte
   STA INDEXA        ; save it
   LDA TMPPTC+1      ; get found line pointer high byte
   STA INDEXB+1      ; save it
   LDA TMPPTC        ; get found line pointer low byte
   DEY               ; decrement index
   SBC (TMPPTC),Y    ; subtract next line pointer low byte
   CLC
   ADC VARTAB        ; add start of variables low byte
   STA VARTAB        ; set start of variables low byte
   STA INDEXB        ; save destination pointer low byte
   LDA VARTAB+1      ; get start of variables high byte
   ADC #$FF          ; -1 + carry
   STA VARTAB+1      ; set start of variables high byte
   SBC TMPPTC+1      ; subtract found line pointer high byte
   TAX               ; copy to block count
   SEC
   LDA TMPPTC        ; get found line pointer low byte
   SBC VARTAB        ; subtract start of variables low byte
   TAY               ; copy to bytes in first block count
   BCS NBL_10        ; if no underflow skip the high byte decrement
   INX               ; increment block count, correct for = 0 loop exit
   DEC INDEXB+1      ; decrement destination high byte

NBL_10
   CLC
   ADC INDEXA        ; add source pointer low byte
   BCC NBL_Loop      ; if no underflow skip the high byte decrement
   DEC INDEXA+1      ; else decrement source pointer high byte
   CLC

NBL_Loop
   LDA (INDEXA),Y    ; get byte from source
   STA (INDEXB),Y    ; copy to destination
   INY
   BNE NBL_Loop      ; while <> 0 do this block
   INC INDEXA+1      ; increment source pointer high byte
   INC INDEXB+1      ; increment destination pointer high byte
   DEX               ; decrement block count
   BNE NBL_Loop      ; loop until all done

NBL_20
   JSR Reset_BASIC_Execution
   JSR Rechain
   LDA BUF           ; get first byte from buffer
   BEQ Vectored_Warmstart
   CLC               ; insert line into memory
   LDA VARTAB        ; get start of variables low byte
   STA TMPPTB        ; save as source end pointer low byte
   ADC COUNT         ; add index pointer to end of crunched line
   STA TMPPTA        ; save as destination end pointer low byte
   LDY VARTAB+1      ; get start of variables high byte
   STY TMPPTB+1      ; save as source end pointer high byte
   BCC NBL_30      ; if no carry skip the high byte increment
   INY               ; else increment the high byte
NBL_30
   STY TMPPTA+1      ; save as destination end pointer high byte
   JSR Open_Up_Space

; most of what remains to do is copy the crunched line into the space opened up in memory,
; however, before the crunched line comes the next line pointer and the line number. the
; line number is retrieved from the temporary integer and stored in memory, this
; overwrites the bottom two bytes on the stack. next the line is copied and the next line
; pointer is filled with whatever was in two bytes above the line number in the stack.
; this is ok because the line pointer gets fixed in the line chain re-build.

   LDAY(LINNUM)      ; get line number
   STA BUF-2         ; save line number low byte before crunched line
   STY BUF-1         ; save line number high byte before crunched line
   LDAY(STREND)      ; get end of arrays
   STAY(VARTAB)      ; set start of variables
   LDY COUNT         ; get index to end of crunched line
   DEY               ; -1
NBL_Copy
   LDA BUF-4,Y       ; get byte from crunched line
   STA (TMPPTC),Y    ; save byte to memory
   DEY               ; decrement index
   BPL NBL_Copy      ; loop while more to do

; =================
  Reset_And_Rechain
; =================

   JSR Reset_BASIC_Execution
   JSR Rechain
   JMP Vectored_Warmstart

; *******
  Rechain
; *******

   LDAY(TXTTAB)      ; get start of memory
   STAY(INDEXA)      ; set line start pointer
   CLC

Rech_Loop
   LDY #1            ; set index to pointer to next line high byte
   LDA (INDEXA),Y    ; get pointer to next line high byte
   BEQ Rech_Ret      ; exit if null, [EOT]
   LDY #4            ; point to first code byte of line
                     ; there is always 1 byte + [EOL] as null entries are deleted
Rech_Loop_2
   INY               ; next code byte
   LDA (INDEXA),Y    ; get byte
   BNE Rech_Loop_2   ; loop if not [EOL]
   INY               ; point to byte past [EOL], start of next line
   TYA               ; copy it
   ADC INDEXA        ; add line start pointer low byte
   TAX               ; copy to X
   LDY #0            ; point to this line's next line pointer
   STA (INDEXA),Y    ; set next line pointer low byte
   LDA INDEXA+1      ; get line start pointer high byte
   ADC #$00          ; add any overflow
   INY
   STA (INDEXA),Y    ; set next line pointer high byte
   STX INDEXA        ; set line start pointer low byte
   STA INDEXA+1      ; set line start pointer high byte
   BCC Rech_Loop     ; branch always

Rech_Ret
   RTS

; ***********
  Read_String
; ***********

   LDX #0            ; set channel 0, keyboard

ReSt_Loop
   JSR Read_Char
   CMP #CR
   BEQ ReSt_Finish
   STA BUF,X
   INX
   CPX #$59          ; compare with max+1
   BCC ReSt_Loop
   LDX #$17          ; error $17, string too long error
   JMP Basic_Error

ReSt_Finish
   JMP Terminate_BUF      ; set XY to BUF - 1 and print [CR]

; *************
  Tokenize_Line
; *************

   JMP (ICRNCH)      ; normally next statement

; ****************
  Default_Tokenize
; ****************

   LDX TXTPTR        ; get BASIC execute pointer low byte
   LDY #4
   STY GARBFL        ; clear open quote/DATA flag

Toke_Loop
   LDA BUF,X
   BPL Toke_05       ; plain text
   CMP #TK_PI
   BEQ Toke_35       ; if PI save & continue
   INX               ; next char
   BNE Toke_Loop     ; branch always

Toke_05
   CMP #' '
   BEQ Toke_35       ; if [SPACE] save & continue
   STA ENDCHR        ; save buffer byte as search character
   CMP #QUOTE
   BEQ Toke_55       ; if quote go copy quoted string
   BIT GARBFL        ; get open quote/DATA token flag
   BVS Toke_35       ; branch if b6 of Oquote set, was DATA
   CMP #'?'           ; compare with '?' character
   BNE Toke_10        ; if not "?" continue crunching
   LDA #TK_PRINT      ; replace '?' by the token for PRINT
   BNE Toke_35        ; branch always

Toke_10
   CMP #'0'           ; compare with "0"
   BCC Toke_15        ; if < "0" continue crunching
   CMP #'<'           ; compare with "<"
   BCC Toke_35        ; if <, 0123456789:; save & continue

Toke_15
   STY TMPPTD         ; copy save index
   LDY #0             ; clear table pointer
   STY COUNT          ; clear word index
   DEY                ; Y = $FF
   STX TXTPTR         ; save BASIC execute pointer low byte, buffer index
   DEX                ; adjust for pre increment loop

Toke_20
   INY                ; next table byte
   INX                ; next buffer byte

Toke_25
   LDA BUF,X          ; get byte from input buffer
   SEC
   SBC Basic_Keyword_Table,Y
   BEQ Toke_20        ; match so far
   CMP #$80           ; was it end marker match ?
   BNE Toke_60        ; if not go try the next keyword
   ORA COUNT          ; OR with word index, +$80 in A makes token

Toke_30
   LDY TMPPTD         ; restore save index

Toke_35
   INX                ; increment buffer read index
   INY                ; increment save index
   STA BUF-5,Y        ; save byte to output
   LDA BUF-5,Y        ; get byte from output, set flags
   BEQ Toke_70        ; branch if was null [EOL]
   SEC
   SBC #':'           ; subtract ":"
   BEQ Toke_40        ; branch if it was (A is now 0)
   CMP #TK_DATA-':'   ; compare with the token for DATA-':'
   BNE Toke_45        ; if not DATA go try REM

Toke_40
   STA GARBFL         ; save token-':'

Toke_45
   SEC
   SBC #TK_REM-':'    ; subtract the token for REM-':'
   BNE Toke_Loop      ; if wasn't REM go crunch next bit of line

Toke_REM              ; target from Jiffy tokenizer
   STA ENDCHR         ; else was REM so set search for [EOL]

Toke_50
   LDA BUF,X          ; get byte from input buffer
   BEQ Toke_35        ; if null [EOL] save byte then continue crunching
   CMP ENDCHR         ; compare with stored character
   BEQ Toke_35       ; if match save byte then continue crunching

Toke_55
   INY               ; increment save index
   STA BUF-5,Y       ; save byte to output
   INX               ; increment buffer index
   BNE Toke_50       ; branch always

Toke_60
   LDX TXTPTR        ; restore BASIC execute pointer low byte
   INC COUNT         ; increment word index (next word)

Toke_65
   INY               ; increment table index
   LDA Basic_Keyword_Table-1,Y; get table byte
   BPL Toke_65       ; loop if not end of word yet
   LDA Basic_Keyword_Table,Y  ; get byte from keyword table
   BNE Toke_25       ; go test next word if not zero byte, end of table
   LDA BUF,X         ; restore byte from input buffer
   BPL Toke_30       ; branch always, all unmatched bytes in the buffer are

Toke_70
   STA BUF-3,Y       ; save [EOL]
   DEC TXTPTR+1      ; decrement BASIC execute pointer high byte
   LDA #$FF          ; point to start of buffer-1
   STA TXTPTR        ; set BASIC execute pointer low byte
   RTS

; ***************
  Find_BASIC_Line
; ***************

   LDAX(TXTTAB)      ; get start of memory

; ******************
  Find_BASIC_Line_AX
; ******************

   LDY #1            ; set Y to next line link high byte
   STAX(TMPPTC)      ; save as current
   LDA (TMPPTC),Y    ; get line link high byte
   BEQ FiBL_Not_Found; 0 = end of program
   INY               ; Y = 2
   INY               ; Y = 3
   LDA LINNUM+1      ; target line # high byte
   CMP (TMPPTC),Y    ; compare with line # high byte
   BCC FiBL_Ret      ; beyond target line
   BEQ FiBL_Check    ; go check low byte if =
   DEY               ; Y = 2
   BNE FiBL_Cont     ; branch always

FiBL_Check
   LDA LINNUM        ; get target line # low byte
   DEY               ; Y = 2
   CMP (TMPPTC),Y    ; compare with line # low byte
   BCC FiBL_Ret      ; beyond target line
   BEQ FiBL_Ret      ; target line found: exit

FiBL_Cont
   DEY               ; Y = 1
   LDA (TMPPTC),Y    ; get next line link high byte
   TAX               ; copy to X
   DEY                ; Y = 0
   LDA (TMPPTC),Y     ; get next line link low byte
   BCS Find_BASIC_Line_AX ; branch always

FiBL_Not_Found
   CLC                ; clear found flag
FiBL_Ret
   RTS

; *********
  Basic_NEW
; *********

   BNE FiBL_Ret       ; exit if following byte to allow syntax error

; ===========
  Perform_NEW
; ===========

   LDA #0
   TAY                ; clear index
   STA (TXTTAB),Y     ; clear pointer to next line low byte
   INY
   STA (TXTTAB),Y     ; clear pointer to next line high byte, erase program
   LDA TXTTAB         ; get start of memory low byte
   CLC
   ADC #2             ; add null program length
   STA VARTAB         ; set start of variables low byte
   LDA TXTTAB+1       ; get start of memory high byte
   ADC #0             ; add carry
   STA VARTAB+1       ; set start of variables high byte

; ********************
  Reset_BASIC_Execution
; ********************

   JSR Reset_BASIC_Exec_Pointer
   LDA #$00           ; set Zb for CLR entry

; *********
  Basic_CLR
; *********

   BNE Flush_Ret       ; exit if following byte to allow syntax error

; *******************
  Clear_Variable_Space
; *******************

   JSR CLALL          ; close all channels and files

; ======================
  Reset_Variable_Pointer
; ======================

   LDAY(MEMSIZ)
   STAY(FRESPC)
   LDAY(VARTAB)
   STAY(ARYTAB)
   STAY(STREND)

; =======================
  Restore_And_Flush_Stack
; =======================

   JSR Basic_RESTORE ; perform RESTORE

; *****************
  Flush_BASIC_Stack
; *****************

   LDX #TEMPST       ; get descriptor stack start
   STX TEMPPT        ; set descriptor stack pointer
   PLA               ; pull return address low byte
   TAY               ; copy it
   PLA               ; pull return address high byte
   LDX #$FA          ; set cleared stack pointer
   TXS               ; set stack
   PHA               ; push return address high byte
   TYA               ; restore return address low byte
   PHA               ; push return address low byte
   LDA #0
   STA OLDTXT+1      ; clear continue pointer high byte
   STA SUBFLG        ; clear subscript/FNX flag

Flush_Ret
   RTS

; ************************
  Reset_BASIC_Exec_Pointer
; ************************

   CLC
   LDA TXTTAB        ; get start of memory low byte
   ADC #$FF          ; add -1 low byte
   STA TXTPTR        ; set BASIC execute pointer low byte
   LDA TXTTAB+1      ; get start of memory high byte
   ADC #$FF          ; add -1 high byte
   STA TXTPTR+1      ; save BASIC execute pointer high byte
   RTS

; **********
  Basic_LIST
; **********

   BCC LIST_05       ; branch if next character not token (LIST n...)
   BEQ LIST_05       ; branch if next character [NULL] (LIST)
   CMP #TK_MINUS     ; the only token allowed here (LIST -m)
   BNE Flush_Ret

LIST_05
   JSR Scan_Linenumber
   JSR Find_BASIC_Line
   JSR CHRGOT
   BEQ LIST_10       ; branch if no more chrs
   CMP #TK_MINUS     ; compare with "-"
   BNE FiBL_Ret      ; return if not "-" (will be SN error)
   JSR CHRGET        ; LIST [n]-m
   JSR Scan_Linenumber
   BNE FiBL_Ret      ; exit if not ok

LIST_10
   PLA               ; dump return address low byte, exit via warm start
   PLA               ; dump return address high byte
   LDA LINNUM        ; get temporary integer low byte
   ORA LINNUM+1      ; OR temporary integer high byte
   BNE LIST_15       ; branch if start set

LIST_12              ; entry for Jiffy Basic file list
   LDA #$FF          ; set last line to $FFFF if not specified
   STA LINNUM
   STA LINNUM+1

LIST_15
   LDY #1
   STY GARBFL        ; clear open quote flag
   LDA (TMPPTC),Y    ; get next line pointer high byte
   BEQ LIST_45       ; if null all done so exit
   JSR Check_STOP

LIST_17              ; entry for Jiffy Basic file list
   JSR Print_CR
   INY
   LDA (TMPPTC),Y    ; get line number low byte
   TAX
   INY
   LDA (TMPPTC),Y    ; get line number high byte
   CMP LINNUM+1      ; compare with temporary integer high byte
   BNE LIST_20       ; branch if no high byte match
   CPX LINNUM        ; compare with temporary integer low byte
   BEQ LIST_25       ; branch if = last line to do, < will pass next branch

LIST_20              ; else ...
   BCS LIST_45       ; if greater all done so exit

LIST_25
   STY FORPNT        ; save index for line
   JSR Print_Integer_XA
   LDA #' '          ; print [SPACE]�after line #

LIST_30
   LDY FORPNT        ; get index for line
   AND #$7F          ; mask top out bit of character

LIST_35
   JSR Print_Char
   CMP #QUOTE
   BNE LIST_40       ; if not skip the quote handle
   LDA GARBFL        ; get open quote flag
   EOR #$FF          ; toggle it
   STA GARBFL        ; save it back

LIST_40
   INY
   BEQ LIST_45       ; line too long so just bail out and do a warm start
   LDA (TMPPTC),Y    ; get next byte
   BNE Vectored_Detokenize      ; if not [EOL] (go print character)
   TAY               ; else clear index
   LDA (TMPPTC),Y    ; get next line pointer low byte
   TAX               ; copy to X
   INY
   LDA (TMPPTC),Y    ; get next line pointer high byte
   STX TMPPTC        ; set pointer to line low byte
   STA TMPPTC+1      ; set pointer to line high byte
   BNE LIST_15       ; go do next line if not [EOT]
```

```
LIST_45

#if C64
   JMP Vectored_Basic_Ready
#endif

#if VIC
   JMP Basic_Ready
#endif

; *******************
  Vectored_Detokenize
; *******************

   JMP (IQPLOP)      ; normally next statement

; ******************
  Default_Detokenize
; ******************

   BPL LIST_35       ; print it if not token byte
   CMP #TK_PI        ; compare with the token for PI
   BEQ LIST_35       ; just print it if so
   BIT GARBFL        ; test the open quote flag
   BMI LIST_35       ; just go print character if open quote set
   SEC               ; else set carry for subtract
   SBC #$7F          ; convert token to inex
   TAX               ; copy token # to X
   STY FORPNT        ; save Y
   LDY #$FF          ; start from -1, adjust for pre increment

DeTo_10
   DEX               ; decrement token #
   BEQ DeTo_30       ; if not found go do printing

DeTo_20
   INY
   LDA Basic_Keyword_Table,Y
   BPL DeTo_20      ; loop until keyword end marker
   BMI DeTo_10      ; branch always

DeTo_30
   INY
   LDA Basic_Keyword_Table,Y
   BMI LIST_30       ; go restore index, mask byte and print
   JSR Print_Char
   BNE DeTo_30       ; go get next character, branch always

; *********
  Basic_FOR
; *********

   LDA #$80
   STA SUBFLG        ; set FNX flag
   JSR Basic_LET
   JSR Find_Active_FOR
   BNE BaFO_10       ; branch if this FOR variable was not found
   TXA               ; dump the old one
   ADC #15           ; add FOR structure size-2
   TAX               ; copy to index
   TXS               ; set stack (dump FOR structure (-2 bytes))
```

```
BaFO_10
   PLA               ; pull return address
   PLA               ; pull return address
   LDA #$09          ; we need 18d bytes !
   JSR Check_Stack_Avail
   JSR Next_Statement
   CLC
   TYA               ; copy index to A
   ADC TXTPTR        ; add BASIC execute pointer low byte
   PHA               ; push onto stack
   LDA TXTPTR+1      ; get BASIC execute pointer high byte
   ADC #$00          ; add carry
   PHA               ; push onto stack
   PUSHW(CURLIN)     ; push current line number
   LDA #TK_TO        ; set "TO" token
   JSR Need_A
   JSR Is_Numeric
   JSR Eval_Numeric
   LDA FAC1SI        ; get FAC1 sign (b7)
   ORA #$7F          ; set all non sign bits
   AND FAC1M1        ; and FAC1 mantissa 1
   STA FAC1M1        ; save FAC1 mantissa 1
   LAYI(BaFO_20)
   STAY(INDEXA)      ; the following jump returns to this address
   JMP Round_And_Push_FAC1

BaFO_20
   LAYI(REAL_1)      ; default STEP value = 1.0
   JSR Load_FAC1_AY
   JSR CHRGOT
   CMP #TK_STEP
   BNE BaFO_30
   JSR CHRGET
   JSR Eval_Numeric  ; Get STEP value

BaFO_30
   JSR Get_FAC1_Sign
   JSR Push_FAC1
   PUSHW(FORPNT)     ; push FOR variable on stack
   LDA #TK_FOR       ; push FOR token
   PHA

; ================
  Interpreter_Loop
; ================

   JSR Check_STOP
   LDAY(TXTPTR)      ; get BASIC execute pointer
   CPY #>BUF         ; direct mode ?
   NOP               ; unused byte
   BEQ InLo_10       ; in direct mode skip the continue pointer save
   STAY(OLDTXT)      ; save the continue pointer

InLo_10
   LDY #0
   LDA (TXTPTR),Y    ; get BASIC byte
   BNE Inte_20      ; if not [EOL] go test for ":"
   LDY #2
   LDA (TXTPTR),Y    ; get next line pointer high byte
   CLC
   BNE InLo_20       ; branch if not end of program
   JMP End_Of_Exec
```

```
InLo_20
   INY
   LDA (TXTPTR),Y    ; get line number low byte
   STA CURLIN        ; save current line number low byte
   INY
   LDA (TXTPTR),Y    ; get line # high byte
   STA CURLIN+1      ; save current line number high byte
   TYA               ; A now = 4
   ADC TXTPTR        ; add BASIC execute pointer low byte, now points to code
   STA TXTPTR        ; save BASIC execute pointer low byte
   BCC Start_Program      ; if no overflow skip the high byte increment
   INC TXTPTR+1      ; else increment BASIC execute pointer high byte

; =============
  Start_Program
; =============

   JMP (IGONE)       ; normally following code

; =============
  Default_Start
; =============

   JSR CHRGET
   JSR Interpret
   JMP Interpreter_Loop

; *********
  Interpret
; *********

   BEQ BaRE_Ret      ; if the first byte is null just exit

  Interpret_10
; ------------
   SBC #$80          ; normalise the token
   BCC Inte_10       ; if wasn't token go do LET
   CMP #TK_TAB-$80   ; compare with token for TAB(-$80
   BCS Inte_40       ; branch if >= TAB(
   ASL A             ; *2 bytes per vector
   TAY               ; copy to index
   LDA Basic_Statement_Table+1,Y; get vector high byte
   PHA               ; push on stack
   LDA Basic_Statement_Table,Y  ; get vector low byte
   PHA               ; push on stack
   JMP CHRGET        ; the return from CHRGET calls the command code

Inte_10
   JMP Basic_LET     ; perform LET

Inte_20
   CMP #':'          ; comapre with ":"
   BEQ Start_Program ; if ":" go execute new code

Inte_30
   JMP Syntax_Error

Inte_40
   CMP #TK_GO-$80    ; compare with token for GO
   BNE Inte_30       ; if not "GO" do syntax error then warm start
   JSR CHRGET
   LDA #TK_TO        ; set "TO" token
   JSR Need_A
   JMP Basic_GOTO    ; perform GOTO

; *************
  Basic_RESTORE
; *************

   SEC
   LDA TXTTAB        ; get start of memory low byte
   SBC #$01          ; -1
   LDY TXTTAB+1      ; get start of memory high byte
   BCS Store_DATPTR
   DEY               ; else decrement high byte

Store_DATPTR
   STAY(DATPTR)

BaRE_Ret
   RTS

; **********
  Check_STOP
; **********

   JSR STOP          ; Check if stop key is pressed

; **********
  Basic_STOP
; **********

   BCS BaEN_10       ; if carry set do BREAK instead of just END

; *********
  Basic_END
; *********

   CLC

BaEN_10
   BNE BaCO_Ret      ; return if wasn't CTRL-C
   LDAY(TXTPTR)      ; get BASIC execute pointer
   LDX CURLIN+1      ; get current line number high byte
   INX               ; increment it
   BEQ BaEN_20       ; branch if was direct mode
   STAY(OLDTXT)      ; save continue pointer
   LDAY(CURLIN)      ; get current line number
   STAY(OLDLIN)      ; save break line number

BaEN_20
   PLA               ; dump return address low byte
   PLA               ; dump return address high byte

  End_Of_Exec
; -----------
   LAYI(Msg_CrLf)
   BCC BaEN_30       ; branch if it was program end
   JMP Display_Msg_Then_Ready

BaEN_30

#if C64
```

```
   JMP Vectored_Basic_Ready
#endif
#if VIC
   JMP Basic_Ready
#endif

; **********
  Basic_CONT
; **********

   BNE BaCO_Ret      ; exit if following byte to allow syntax error
   LDX #$1A          ; error code $1A, can't continue error
   LDY OLDTXT+1      ; get continue pointer high byte
   BNE BaCO_10       ; go do continue if we can
   JMP Basic_Error

BaCO_10
   LDA OLDTXT        ; get continue pointer low byte
   STAY(TXTPTR)      ; save BASIC execute pointer
   LDAY(OLDLIN)      ; get break line
   STAY(CURLIN)      ; set current line

BaCO_Ret
   RTS

; *********
  Basic_RUN
; *********

   PHP
   LDA #0            ; no control or kernal messages
   JSR SETMSG
   PLP
   BNE BaRU_10       ; branch if RUN n
   JMP Reset_BASIC_Execution

BaRU_10
   JSR Clear_Variable_Space
   JMP Goto_Line

; ***********
  Basic_GOSUB
; ***********

   LDA #3            ; need 6 bytes for GOSUB
   JSR Check_Stack_Avail
   PUSHW(TXTPTR)     ; push BASIC execute pointer
   PUSHW(CURLIN)     ; push current line number
   LDA #TK_GOSUB     ; token for GOSUB
   PHA               ; save it

  Goto_Line
; ---------
   JSR CHRGOT
   JSR Basic_GOTO
   JMP Interpreter_Loop

; **********
  Basic_GOTO
; **********

   JSR Scan_Linenumber
   JSR Next_Line
   SEC
   LDA CURLIN        ; get current line number low byte
   SBC LINNUM        ; subtract temporary integer low byte
   LDA CURLIN+1      ; get current line number high byte
   SBC LINNUM+1      ; subtract temporary integer high byte
   BCS BaGO_10       ; if current line # >= temporary search from start
   TYA               ; else copy line index to A
   SEC               ; set carry (+1)
   ADC TXTPTR        ; add BASIC execute pointer low byte
   LDX TXTPTR+1      ; get BASIC execute pointer high byte
   BCC BaGO_20       ; if no overflow skip the high byte increment
   INX               ; increment high byte
   BCS BaGO_20       ; go find the line, branch always

BaGO_10
   LDAX(TXTTAB)      ; get start of memory

BaGO_20
   JSR Find_BASIC_Line_AX
   BCC Undefined_Statement
   LDA TMPPTC        ; get pointer low byte
   SBC #1            ; -1
   STA TXTPTR        ; save BASIC execute pointer low byte
   LDA TMPPTC+1      ; get pointer high byte
   SBC #0            ; subtract carry
   STA TXTPTR+1      ; save BASIC execute pointer high byte

BaGO_Ret
   RTS

; ************
  Basic_RETURN
; ************

   BNE BaGO_Ret      ; exit if following token to allow syntax error
   LDA #$FF          ; set byte so no match possible
   STA FORPNT+1      ; save FOR/NEXT variable pointer high byte
   JSR Find_Active_FOR
   TXS               ; correct the stack
   CMP #TK_GOSUB     ; compare with GOSUB token
   BEQ BaRE_20       ; if matching GOSUB go continue RETURN
   LDX #$0C          ; else error code $04, return without gosub error
   .byte $2C         ; skip next statement

; ===================
  Undefined_Statement
; ===================

   LDX #$11          ; error code $11, undefined statement error
   JMP Basic_Error

BaRE_10
   JMP Syntax_Error

BaRE_20
   PLA               ; dump token byte
   PULLW(CURLIN)     ; pull current line number
   PULLW(TXTPTR)     ; pull BASIC execute pointer

; *********
  Basic_DATA
```

```
; **********

   JSR Next_Statement

; **************************
  Add_Y_To_Execution_Pointer
; **************************

   TYA               ; copy index to A

Add_To_TXTPTR
   CLC
   ADC TXTPTR        ; add BASIC execute pointer low byte
   STA TXTPTR        ; save BASIC execute pointer low byte
   BCC AYTE_Ret      ; skip increment if no carry
   INC TXTPTR+1      ; else increment BASIC execute pointer high byte

AYTE_Ret
   RTS

; **************
  Next_Statement
; **************

   LDX #':'          ; look for colon
   .byte   $2C       ; skip "LDX #0" command

; *********
  Next_Line
; *********

   LDX #0            ; look for 0 [EOL]
   STX CHARAC        ; store alternate search character
   LDY #0            ; set search character = [EOL]
   STY ENDCHR        ; save the search character

NeLi_10
   LDA ENDCHR        ; get search character
   LDX CHARAC        ; get alternate search character
   STA CHARAC        ; make search character = alternate search character

NeLi_15
   STX ENDCHR        ; make alternate search character = search character

NeLi_20
   LDA (TXTPTR),Y    ; get BASIC byte
   BEQ AYTE_Ret      ; exit if null [EOL]
   CMP ENDCHR        ; compare with search character
   BEQ AYTE_Ret      ; exit if found
   INY               ; else increment index
   CMP #QUOTE
   BNE NeLi_20       ; if found go swap search character for alternate search
   BEQ NeLi_10       ; loop for next character, branch always

; ********
  Basic_IF
; ********

   JSR Eval_Expression
   JSR CHRGOT
   CMP #TK_GOTO
   BEQ BaIF_10       ; do IF ... GOTO
   LDA #TK_THEN
   JSR Need_A

BaIF_10
   LDA FAC1EX        ; get FAC1 exponent
   BNE BaIF_20      ; if result was non zero continue execution

; *********
  Basic_REM
; *********

   JSR Next_Line
   BEQ Add_Y_To_Execution_Pointer ; branch always

BaIF_20              ; Basic_IF continued
   JSR CHRGOT
   BCS BaIF_30      ; if not numeric character, is variable or keyword
   JMP Basic_GOTO    ; else perform GOTO n

BaIF_30
   JMP Interpret

; ********
  Basic_ON
; ********

   JSR Get_Byte_Value
   PHA               ; push next character
   CMP #TK_GOSUB     ; compare with GOSUB token
   BEQ BaON_20       ; if GOSUB go see if it should be executed

BaON_10
   CMP #TK_GOTO      ; compare with GOTO token
   BNE BaRE_10       ; if not GOTO do syntax error then warm start

BaON_20
   DEC FAC1M4        ; decrement the byte value
   BNE BaON_30       ; if not zero go see if another line number exists
   PLA               ; pull keyword token
   JMP Interpret_10

BaON_30
   JSR CHRGET
   JSR Scan_Linenumber
   CMP #','          ; compare next character with ","
   BEQ BaON_20       ; loop if ","
   PLA               ; else pull keyword token, ran out of options

BaON_Ret
   RTS

; ***************
  Scan_Linenumber
; ***************

   LDX #0
   STX LINNUM
   STX LINNUM+1

ScLi_Loop
   BCS BaON_Ret      ; return if carry set, end of scan, character was not 0-9
   SBC #$2F          ; subtract $30, $2F+carry, from byte
   STA CHARAC        ; store # OPT: TAX
   LDA LINNUM+1      ; get temporary integer high byte
   STA INDEXA        ; save it for now
   CMP #$19          ; compare with $19
   BCS BaON_10       ; branch if >= this makes the maximum line number 63999
                     ; because the next bit does $1900 * $0A = $FA00 = 64000
                     ; decimal. the branch target is really the SYNTAX error
                     ; at BaRE_10 but that is too far so an intermediate
                     ; compare and branch to that location is used. the problem
                     ; with this is that line number that gives a partial result
                     ; from $8900 to $89FF, 35072x to 35327x, will pass the new
                     ; target compare and will try to execute the remainder of
                     ; the ON n GOTO/GOSUB. a solution to this is to copy the
                     ; byte in A before the branch to X and then branch to
                     ; BaRE_10 skipping the second compare

   LDA LINNUM        ; get temporary integer low byte
   ASL A             ; *2 low byte
   ROL INDEXA        ; *2 high byte
   ASL A             ; *2 low byte
   ROL INDEXA        ; *2 high byte (*4)
   ADC LINNUM        ; + low byte (*5)
   STA LINNUM        ; save it
   LDA INDEXA        ; get high byte temp
   ADC LINNUM+1      ; + high byte (*5)
   STA LINNUM+1      ; save it
   ASL LINNUM        ; *2 low byte (*10d)
   ROL LINNUM+1      ; *2 high byte (*10d)
   LDA LINNUM        ; get low byte OPT: TXA
   ADC CHARAC        ; add #        OPT: ADC LINNUM
   STA LINNUM        ; save low byte
   BCC ScLi_10       ; if no overflow skip high byte increment
   INC LINNUM+1      ; else increment high byte

ScLi_10
   JSR CHRGET
   JMP ScLi_Loop      ; OPT: BCC ScLi_Loop : RTS

; *********
  Basic_LET
; *********

   JSR Get_Scalar_Address
   STAY(FORPNT)      ; save variable address
   LDA #TK_EQUAL
   JSR Need_A        ; '=' is needed
   LDA INTFLG        ; get data type flag, $80 = integer, $00 = float
   PHA               ; push data type flag
   LDA VALTYP        ; get data type flag, $FF = string, $00 = numeric
   PHA               ; push data type flag
   JSR Eval_Expression
   PLA               ; pop data type flag
   ROL A             ; string bit into carry
   JSR Check_Var_Type
   BNE LET_20        ; if string go assign a string value
   PLA               ; pop integer/float data type flag

; **********************
  Assign_Numeric_variable
; **********************

   BPL LET_10        ; if float go assign a floating value
   JSR Round_FAC1_Checked
   JSR Eval_Integer
   LDY #0
   LDA FAC1M3        ; get FAC1 mantissa 3
   STA (FORPNT),Y    ; save as integer variable low byte
   INY
   LDA FAC1M4        ; get FAC1 mantissa 4
   STA (FORPNT),Y    ; save as integer variable high byte
   RTS

LET_10
   JMP Assign_FAC1_To_FOR_Index

LET_20
   PLA               ; dump integer/float data type flag

; **********************
  Assign_String_Variable
; **********************

   LDY FORPNT+1      ; get variable pointer high byte
   CPY #>NULL_Descriptor ; TI$
   BNE Assign_String
   JSR Get_String_Descriptor
   CMP #6            ; TI$ = "hhmmss"
   BNE Jump_Illegal_Quantity
   LDY #0
   STY FAC1EX        ; clear FAC1 exponent
   STY FAC1SI        ; clear FAC1 sign (b7)

LET_30
   STY TMPPTD        ; save index
   JSR Eval_Digit    ; check and evaluate numeric digit
   JSR Multiply_FAC1_BY_10
   INC TMPPTD        ; increment index
   LDY TMPPTD        ; restore index
   JSR Eval_Digit    ; check and evaluate numeric digit
   JSR FAC1_Round_And_Copy_To_FAC2
   TAX               ; copy FAC1 exponent
   BEQ LET_40        ; branch if FAC1 zero
   INX               ; increment index, * 2
   TXA               ; copy back to A
   JSR Multiply_FAC1_By_4

LET_40
   LDY TMPPTD        ; get index
   INY
   CPY #6            ; max. 6 digits "hhmmss"
   BNE LET_30

   JSR Multiply_FAC1_BY_10
   JSR FAC1_To_Integer
   LDX FAC1M3        ; get FAC1 mantissa 3
   LDY FAC1M2        ; get FAC1 mantissa 2
   LDA FAC1M4        ; get FAC1 mantissa 4
   JMP SETTIM        ; Set the system clock

; **********
  Eval_Digit
; **********

   LDA (INDEXA),Y    ; get byte from string
   JSR ISNUM
   BCC EvDi_10       ; branch if numeric

Jump_Illegal_Quantity
   JMP Illegal_Quantity

EvDi_10
   SBC #$2F          ; subtract $2F + carry = '0'
   JMP Add_A_To_FAC1

; -------------
  Assign_String
; -------------

   LDY #2            ; index to string pointer high byte
   LDA (FAC1M3),Y    ; get string pointer high byte
   CMP FRESPC+1      ; compare with bottom of string space high byte
   BCC AsSt_20       ; branch if string pointer < bottom of string space
   BNE AsSt_10       ; branch if string pointer > bottom of string space
   DEY               ; Y = 1
   LDA (FAC1M3),Y    ; get string pointer low byte
   CMP FRESPC        ; compare with bottom of string space low byte
   BCC AsSt_20       ; branch if string pointer < bottom of string space

AsSt_10
   LDY FAC1M4        ; get descriptor pointer high byte
   CPY VARTAB+1      ; compare with start of variables high byte
   BCC AsSt_20       ; branch if less, is on string stack
   BNE AsSt_30       ; if greater make space and copy string
   LDA FAC1M3        ; get descriptor pointer low byte
   CMP VARTAB        ; compare with start of variables low byte
   BCS AsSt_30       ; if greater or equal make space and copy string

AsSt_20
   LDA FAC1M3        ; get descriptor pointer low byte
   LDY FAC1M4        ; get descriptor pointer high byte
   JMP AsSt_40       ; go copy descriptor to variable

AsSt_30
   LDY #0
   LDA (FAC1M3),Y    ; get string length
   JSR Allocate_String_FAC1
   LDAY(DESCPT)
   STAY(STRPTR)
   JSR Store_String_STRPTR
   LAYI(FAC1EX)

AsSt_40
   STAY( DESCPT)     ; save descriptor pointer
   JSR Pop_Descriptor_Stack
   LDY #0
   LDA (DESCPT),Y    ; get string length from new descriptor
   STA (FORPNT),Y    ; copy string length to variable
   INY
   LDA (DESCPT),Y    ; get string pointer low byte from new descriptor
   STA (FORPNT),Y    ; copy string pointer low byte to variable
   INY
   LDA (DESCPT),Y    ; get string pointer high byte from new descriptor
   STA (FORPNT),Y    ; copy string pointer high byte to variable
   RTS

; ***********
```

```
  Basic_PRINTN
; ************

   JSR Basic_CMD
   JMP Set_Default_Channels

; *********
  Basic_CMD
; *********

   JSR Get_Byte_Value
   BEQ BCMD_10       ; branch if following byte is ":" or [EOT]
   LDA #','
   JSR Need_A

BCMD_10
   PHP               ; save status
   STX IOPMPT        ; set current I/O channel
   JSR Select_Output_Channel
   PLP               ; restore status
   JMP Basic_PRINT   ; perform PRINT

BaPR_00
   JSR Print_String_From_Descriptor

BaPR_05
   JSR CHRGOT

; ***********
  Basic_PRINT
; ***********

   BEQ Print_CR

BaPR_10
   BEQ Invert_Ret      ; if nothing following exit, end of PRINT branch
   CMP #TK_TAB       ; compare with token for TAB(
   BEQ TAB_20      ; if TAB( go handle it
   CMP #TK_SPC       ; compare with token for SPC(
   CLC               ; flag SPC(
   BEQ TAB_20      ; if SPC( go handle it
   CMP #','
   BEQ TAB_Jump
   CMP #SEMIC
   BEQ TAB_60      ; if ";" go continue the print loop
   JSR Eval_Expression
   BIT VALTYP        ; test data type flag, $FF = string, $00 = numeric
   BMI BaPR_00      ; if string go print string, scan memory and continue PRINT
   JSR Format_FAC1
   JSR Create_String_Descriptor
   JSR Print_String_From_Descriptor
   JSR Cursor_Right_Or_Space
   BNE BaPR_05      ; go scan memory and continue PRINT, branch always

; =============
  Terminate_BUF
; =============

   LDA #0
   STA BUF,X         ; terminate string with 0 byte
   LDX #<[BUF-1]
   LDY #>[BUF-1]
   LDA IOPMPT        ; get current I/O channel
   BNE Invert_Ret      ; exit if not default channel

; ********
  Print_CR
; ********

   LDA #CR
   JSR Print_Char
   BIT IOPMPT        ; test current I/O channel
   BPL Invert_A      ; needless, because this is always true
   LDA #LF
   JSR Print_Char

; --------
  Invert_A
; --------

   EOR #$FF          ; invert A

Invert_Ret
   RTS

; ========
  TAB_Jump
; ========

   SEC               ; set Cb for read cursor position
   JSR PLOT          ; Read cursor location
   TYA               ; copy cursor Y
   SEC

TAB_10

#if C64
   SBC #10           ; subtract one TAB length
#endif
#if VIC
   SBC #11           ; subtract one TAB length
#endif

   BCS TAB_10
   EOR #$FF          ; complement it
   ADC #1
   BNE TAB_30        ; print A spaces, branch always

TAB_20
   PHP               ; save TAB( or SPC( status
   SEC               ; set Cb for read cursor position
   JSR PLOT          ; Read or set cursor location
   STY TRMPOS        ; save current cursor position
   JSR Get_Next_Byte_Value
   CMP #$29          ; compare with ")"
   BNE BaIn_30       ; if not ")" do syntax error
   PLP               ; restore TAB( or SPC( status
   BCC TAB_40        ; branch if was SPC(
   TXA               ; copy TAB() byte to A
   SBC TRMPOS        ; subtract current cursor position
   BCC TAB_60        ; go loop for next if already past requited position

TAB_30
   TAX               ; copy [SPACE] count to X
```

```
TAB_40
   INX               ; increment count

TAB_50
   DEX               ; decrement count
   BNE TAB_70        ; branch if count was not zero

TAB_60
   JSR CHRGET
   JMP BaPR_10       ; continue print loop

TAB_70
   JSR Cursor_Right_Or_Space
   BNE TAB_50        ; loop, branch always

; ************
  Print_String
; ************

   JSR Create_String_Descriptor

; ----------------------------
  Print_String_From_Descriptor
; ----------------------------

   JSR Get_String_Descriptor
   TAX               ; copy length
   LDY #0
   INX               ; increment length, for pre decrement loop

PSFD_Loop
   DEX               ; decrement length
   BEQ Invert_Ret    ; exit if done
   LDA (INDEXA),Y    ; get byte from string
   JSR Print_Char
   INY
   CMP #$0D          ; compare byte with [CR]
   BNE PSFD_Loop
   JSR Invert_A      ; nonsense
   JMP PSFD_Loop

; ********************
  Cursor_Right_Or_Space
; ********************

   LDA IOPMPT        ; get current I/O channel
   BEQ CROS_10       ; if default channel load [CURSOR RIGHT]
   LDA #' '          ; else load [SPACE]
   .byte $2C         ; skip until Print_Char
CROS_10
   LDA #$1D          ; load [CURSOR RIGHT]
   .byte $2C         ; skip until Print_Char

; ******************
  Print_Question_Mark
; ******************

   LDA #'?'

; *********
  Print_Char
```

```
; **********

   JSR CHROUT_Checked
   AND #$FF          ; set flags
   RTS

; =========
  Bad_Input
; =========

   LDA INPFLG        ; get INPUT mode flag, $00 = INPUT, $40 = GET, $98 = READ
   BEQ BaIn_40       ; branch if INPUT
   BMI BaIn_10       ; branch if READ
   LDY #$FF          ; set current line high byte to -1, indicate immediate mode
   BNE BaIn_20       ; branch always

BaIn_10
   LDAY(DATLIN)      ; get current DATA line number

BaIn_20
   STAY(CURLIN)      ; set current line number

BaIn_30
   JMP Syntax_Error

BaIn_40
   LDA IOPMPT        ; get current I/O channel
   BEQ BaIn_50       ; if default channel go do "?REDO FROM START" message
   LDX #$18          ; else error $18, file data error
   JMP Basic_Error

BaIn_50
   Print_Msg(Msg_Redo_From_Start)
   LDAY(OLDTXT)      ; get continue pointer
   STAY(TXTPTR)      ; save BASIC execute pointer
   RTS

; *********
  Basic_GET
; *********

   JSR Assert_Non_Direct
   CMP #'#'          ; compare with "#"
   BNE BaGE_10       ; branch if not GET#
   JSR CHRGET
   JSR Get_Byte_Value
   LDA #','
   JSR Need_A
   STX IOPMPT        ; set current I/O channel
   JSR CHKIN_Checked

BaGE_10
   LDX #<[BUF+1]     ; set BUF+1 pointer low byte
   LDY #>[BUF+1]     ; set BUF+1 pointer high byte
   LDA #0
   STA BUF+1         ; ensure null terminator
   LDA #$40          ; input mode = GET
   JSR Read_Get
   LDX IOPMPT        ; get current I/O channel
   BNE BaIN_10       ; if not default channel go do channel close and return
   RTS
```

```
; ************
  Basic_INPUTN
; ************

   JSR Get_Byte_Value
   LDA #','
   JSR Need_A
   STX IOPMPT        ; set current I/O channel
   JSR CHKIN_Checked
   JSR Input_String

; --------------------
  Set_Default_Channels
; --------------------

   LDA IOPMPT        ; get current I/O channel

BaIN_10
   JSR CLRCHN        ; Clear I/O channels
   LDX #0
   STX IOPMPT        ; clear current I/O channel
   RTS

; ***********
  Basic_INPUT
; ***********

   CMP #QUOTE
   BNE Input_String
   JSR Make_String_Descriptor_From_Code
   LDA #SEMIC
   JSR Need_A
   JSR Print_String_From_Descriptor

; ------------
  Input_String
; ------------

   JSR Assert_Non_Direct
   LDA #','
   STA BUF-1         ; save to start of buffer - 1

BaIN_20
   JSR Prompt_And_Input
   LDA IOPMPT        ; get current I/O channel
   BEQ BaIN_30       ; branch if default I/O channel
   JSR READST        ; read I/O status word
   AND #$02          ; mask no DSR/timeout
   BEQ BaIN_30       ; branch if not error
   JSR Set_Default_Channels
   JMP Basic_DATA    ; perform DATA

BaIN_30
   LDA BUF           ; get first byte in input buffer
   BNE BaIN_50
   LDA IOPMPT        ; get current I/O channel
   BNE BaIN_20       ; if not default channel go get BASIC input
   JSR Next_Statement
   JMP Add_Y_To_Execution_Pointer

; ****************
  Prompt_And_Input
```

```
; ****************

   LDA IOPMPT        ; get current I/O channel
   BNE BaIN_40       ; skip "?" prompt if not default channel
   JSR Print_Question_Mark
   JSR Cursor_Right_Or_Space
BaIN_40
   JMP Read_String

; **********
  Basic_READ
; **********

   LDXY(DATPTR)
   LDA #$98          ; set input mode = READ
   .byte   $2C       ; skip next statement
BaIN_50
   LDA #$00          ; set input mode = INPUT

; --------
  Read_Get
; --------

   STA INPFLG        ; $00 = INPUT, $40 = GET, $98 = READ
   STXY(INPPTR)

READ_Loop_Var
   JSR Get_Scalar_Address
   STAY(FORPNT)      ; save variable address
   LDAY(TXTPTR)      ; get BASIC execute pointer
   STAY(YSAVE)       ; save BASIC execute pointer
   LDXY(INPPTR)      ; get READ pointer
   STXY(TXTPTR)      ; save as BASIC execute pointer
   JSR CHRGOT
   BNE READ_20       ; branch if not null
   BIT INPFLG        ; $00 = INPUT, $40 = GET, $98 = READ
   BVC READ_05       ; branch if not GET
   JSR GETIN_Checked
   STA BUF           ; save to buffer
   LDX #<[BUF-1]     ; set BUF-1 pointer low byte
   LDY #>[BUF-1]     ; set BUF-1 pointer high byte
   BNE READ_15       ; branch always

READ_05
   BMI READ_60      ; branch if READ else it's do INPUT
   LDA IOPMPT        ; get current I/O channel
   BNE READ_10       ; skip "?" prompt if not default channel
   JSR Print_Question_Mark

READ_10
   JSR Prompt_And_Input

READ_15
   STXY(TXTPTR)

READ_20
   JSR CHRGET        ; execute pointer now points to start of next data
   BIT VALTYP        ; test data type flag, $FF = string, $00 = numeric
   BPL READ_45      ; branch if numeric
   BIT INPFLG        ; $00 = INPUT, $40 = GET, $98 = READ
   BVC READ_25       ; branch if not GET
   INX               ; GET string
   STX TXTPTR        ; save BASIC execute pointer low byte
   LDA #0
   STA CHARAC        ; clear search character
   BEQ READ_30       ; branch always

                     ; is string INPUT or string READ
READ_25
   STA CHARAC        ; save search character
   CMP #QUOTE
   BEQ READ_35       ; if quote only search for "..." string
   LDA #':'          ; set ":"
   STA CHARAC        ; set search character
   LDA #','

READ_30
   CLC

READ_35
   STA ENDCHR        ; set scan quotes flag
   LDAY(TXTPTR)      ; get BASIC execute pointer
   ADC #0            ; INPUT and READ increment by carry
   BCC READ_40       ; if no carry skip the high byte increment
   INY               ; else increment pointer high byte

READ_40
   JSR Create_String_Descriptor_AY
   JSR Restore_Execution_Pointer
   JSR Assign_String_Variable
   JMP READ_50       ; continue processing command

                     ; GET, INPUT or READ is numeric
READ_45
   JSR Load_FAC1_From_String
   LDA INTFLG        ; get data type flag, $80 = integer, $00 = float
   JSR Assign_Numeric_variable

READ_50
   JSR CHRGOT
   BEQ READ_55       ; if ":" or [EOL] go handle the string end
   CMP #','          ; comparte with ","
   BEQ READ_55       ; if "," go handle the string end
   JMP Bad_Input

READ_55
   LDAY(TXTPTR)      ; get BASIC execute pointer
   STAY(INPPTR)      ; save READ pointer
   LDAY(YSAVE)       ; get saved BASIC execute pointer
   STAY(TXTPTR)      ; restore BASIC execute pointer
   JSR CHRGOT
   BEQ READ_70       ; branch if ":" or [EOL]
   JSR Need_Comma
   JMP READ_Loop_Var

READ_60
   JSR Next_Statement
   INY
   TAX               ; copy byte to X
   BNE READ_65       ; if ":" go look for the next DATA
   LDX #$0D          ; else set error $0D, out of data error
   INY
   LDA (TXTPTR),Y    ; get next line pointer high byte
   BEQ NEXT_20      ; if program end go do error, eventually does error X
   INY
   LDA (TXTPTR),Y    ; get next line # low byte
   STA DATLIN        ; save current DATA line low byte
   INY
   LDA (TXTPTR),Y    ; get next line # high byte
   INY
   STA DATLIN+1      ; save current DATA line high byte

READ_65
   JSR Add_Y_To_Execution_Pointer
   JSR CHRGOT
   TAX               ; copy byte
   CPX #TK_DATA      ; compare with token for DATA
   BNE READ_60       ; loop if not DATA
   JMP READ_20       ; continue evaluating READ

READ_70
   LDA INPPTR        ; get READ pointer low byte
   LDY INPPTR+1      ; get READ pointer high byte
   LDX INPFLG        ; get INPUT mode flag, $00 = INPUT, $40 = GET, $98 = READ
   BPL READ_75       ; if INPUT or GET go exit or ignore extra input
   JMP Store_DATPTR

READ_75
   LDY #0
   LDA (INPPTR),Y    ; get READ byte
   BEQ READ_Ret      ; exit if [EOL]
   LDA IOPMPT        ; get current I/O channel
   BNE READ_Ret      ; exit if not default channel
   LAYI(Msg_Extra_Ignored)
   JMP Print_String

READ_Ret
   RTS

Msg_Extra_Ignored   .byte "?EXTRA IGNORED\r",0
Msg_Redo_From_Start .byte "?REDO FROM START\r",0

; **********
  Basic_NEXT
; **********

   BNE Find_NEXT_Variable
   LDY #0
   BEQ NEXT_10      ; use any variable, branch always

; ******************
  Find_NEXT_Variable
; ******************

   JSR Get_Scalar_Address

NEXT_10
   STAY(FORPNT)      ; save FOR/NEXT variable pointer
   JSR Find_Active_FOR
   BEQ NEXT_30       ; if FOR found continue
   LDX #$0A          ; else set error $0A, next without for error

NEXT_20
   JMP Basic_Error

; found this FOR variable
```

```
NEXT_30
   TXS               ; update stack pointer
   TXA               ; copy stack pointer
   CLC
   ADC #$04          ; point to STEP value
   PHA               ; save it
   ADC #$06          ; point to TO value
   STA INDEXB        ; save pointer to TO variable for compare
   PLA               ; restore pointer to STEP value
   LDY #$01          ; point to stack page
   JSR Load_FAC1_AY
   TSX               ; get stack pointer back
   LDA STACK+9,X     ; get step sign
   STA FAC1SI        ; save FAC1 sign (b7)
   LDA FORPNT        ; get FOR/NEXT variable pointer low byte
   LDY FORPNT+1      ; get FOR/NEXT variable pointer high byte
   JSR Add_Var_AY_To_FAC1
   JSR Assign_FAC1_To_FOR_Index
   LDY #$01          ; point to stack page
   JSR Compare_FAC1_INDEXB_Y ; compare FAC1 with TO value
   TSX               ; get stack pointer back
   SEC
   SBC STACK+9,X     ; subtract step sign
   BEQ NEXT_50       ; if = loop complete, go unstack the FOR
   LDA STACK+$0F,X   ; get FOR line low byte
   STA CURLIN        ; save current line number low byte
   LDA STACK+$10,X   ; get FOR line high byte
   STA CURLIN+1      ; save current line number high byte
   LDA STACK+$12,X   ; get BASIC execute pointer low byte
   STA TXTPTR        ; save BASIC execute pointer low byte
   LDA STACK+$11,X   ; get BASIC execute pointer high byte
   STA TXTPTR+1      ; save BASIC execute pointer high byte

NEXT_40
   JMP Interpreter_Loop

NEXT_50
   TXA               ; stack copy to A
   ADC #$11          ; add $12, $11 + carry, to dump FOR structure
   TAX               ; copy back to index
   TXS               ; copy to stack pointer
   JSR CHRGOT
   CMP #','
   BNE NEXT_40       ; if not "," go do interpreter inner loop
   JSR CHRGET
   JSR Find_NEXT_Variable

; ************
  Eval_Numeric
; ************

   JSR Eval_Expression

; **********
  Is_Numeric
; **********

   CLC
   .byte   $24       ; skip next byte

; ******************
```

```
  Assert_String_Type
; ******************

   SEC                  ; string required

; **************
  Check_Var_Type
; **************

   BIT VALTYP           ; test data type flag, $FF = string, $00 = numeric
   BMI CVT_20           ; branch to string check
   BCS Type_Missmatch

CVT_10                  ; OK
   RTS

CVT_20
   BCS CVT_10           ; exit if string is required

Type_Missmatch
   LDX #$16             ; error code $16, type missmatch error
   JMP Basic_Error

; ***************
  Eval_Expression
; ***************

   LDX TXTPTR           ; get BASIC execute pointer low byte
   BNE EvEx_05          ; skip next if not zero
   DEC TXTPTR+1         ; else decrement BASIC execute pointer high byte

EvEx_05
   DEC TXTPTR           ; decrement BASIC execute pointer low byte
   LDX #$00             ; set null precedence, flag done
   .byte   $24          ; makes next line BIT VARPNT+1

EvEx_10
   PHA                  ; push compare evaluation byte if branch to here
   TXA                  ; copy precedence byte
   PHA                  ; push precedence byte
   LDA #$01             ; 2 bytes
   JSR Check_Stack_Avail
   JSR Evaluate
   LDA #0
   STA ACCSYM           ; clear comparrison evaluation flag

EvEx_15
   JSR CHRGOT

EvEx_20
   SEC
   SBC #TK_GT           ; subtract token for ">"
   BCC EvEx_25          ; if < ">" skip comparrison test check
   CMP #$03             ; compare with ">" to +3
   BCS EvEx_25          ; if >= 3 skip comparrison test check
   CMP #$01             ; compare with token for =
   ROL A                ; *2, b0 = carry (=1 if token was = or <)
   EOR #$01             ; toggle b0
   EOR ACCSYM           ; EOR with comparrison evaluation flag
   CMP ACCSYM           ; compare with comparrison evaluation flag
   BCC Jump_Syntax_Error         ; if < saved flag do syntax error then warm start
   STA ACCSYM           ; save new comparrison evaluation flag
   JSR CHRGET
   JMP EvEx_20       ; go do next character

EvEx_25
   LDX ACCSYM        ; get comparrison evaluation flag
   BNE EvEx_50       ; if compare function flagged go evaluate right hand side
   BCS Right_Operand      ; apply operator
   ADC #$07          ; add # of operators (+, -, *, /, ^, AND or OR)
   BCC Right_Operand      ; if < + operator go do the function
   ADC VALTYP        ; add data type flag, $FF = string, $00 = numeric
   BNE EvEx_30       ; if not string or not + token skip concatenate
   JMP Concatenate   ; add strings, string 1 is in the descriptor, string 2

EvEx_30
   ADC #$FF          ; -1 (corrects for carry add)
   STA INDEXA        ; save it
   ASL A             ; *2
   ADC INDEXA        ; *3
   TAY               ; copy to index

EvEx_35
   PLA               ; pull previous precedence
   CMP Basic_Operator_Table,Y  ; compare with precedence byte
   BCS RiOp_20      ; if A >= go do the function
   JSR Is_Numeric

EvEx_40
   PHA               ; save precedence

EvEx_45
   JSR Call_Operator_Function
   PLA               ; restore precedence
   LDY YSAVE         ; get precedence stacked flag
   BPL EvEx_60       ; if stacked values go check the precedence
   TAX               ; copy precedence, set flags
   BEQ RiOp_10      ; exit if done
   BNE RiOp_40      ; else pop FAC2 and return, branch always

EvEx_50
   LSR VALTYP        ; clear data type flag, $FF = string, $00 = numeric
   TXA               ; copy compare function flag
   ROL A             ; <<1, shift data type flag into b0, 1 = string, 0 = num
   LDX TXTPTR        ; get BASIC execute pointer low byte
   BNE EvEx_55       ; if no underflow skip the high byte decrement
   DEC TXTPTR+1      ; else decrement BASIC execute pointer high byte

EvEx_55
   DEC TXTPTR        ; decrement BASIC execute pointer low byte
   LDY #$1B          ; set offset to = operator precedence entry
   STA ACCSYM        ; save new comparrison evaluation flag
   BNE EvEx_35       ; branch always

EvEx_60
   CMP Basic_Operator_Table,Y  ; compare with stacked function precedence
   BCS RiOp_40      ; if A >=, pop FAC2 and return
   BCC EvEx_40       ; else go stack this one and continue, branch always

; **********************
  Call_Operator_Function
; **********************

   LDA Basic_Operator_Table+2,Y
   PHA
   LDA Basic_Operator_Table+1,Y
   PHA
   JSR Apply_Operator
   LDA ACCSYM        ; get comparrison evaluation flag
   JMP EvEx_10       ; continue evaluating expression

Jump_Syntax_Error
   JMP Syntax_Error

; **************
  Apply_Operator
; **************

   LDA FAC1SI        ; get FAC1 sign (b7)
   LDX Basic_Operator_Table,Y  ; get precedence byte

; *********
  Push_FAC1
; *********

   TAY               ; copy sign
   PLA               ; get return address low byte
   STA INDEXA        ; save it
   INC INDEXA        ; increment it as return-1 is pushed
   PLA               ; get return address high byte
   STA INDEXA+1      ; save it
   TYA               ; restore sign
   PHA               ; push sign

; ===================
  Round_And_Push_FAC1
; ===================

   JSR Round_FAC1_Checked
   PUSHW(FAC1M3)     ; push mantissa 4 & 3
   PUSHW(FAC1M1)     ; push mantissa 2 & 1
   LDA FAC1EX        ; get FAC1 exponent
   PHA               ; push it
   JMP (INDEXA)      ; return, sort of

; =============
  Right_Operand
; =============

   LDY #$FF          ; flag function
   PLA               ; pull precedence byte

RiOp_10
   BEQ RiOp_50       ; exit if done

RiOp_20
   CMP #$64          ; compare previous precedence with $64
   BEQ RiOp_30       ; if was $64 (< function) skip the type check
   JSR Is_Numeric

RiOp_30
   STY YSAVE         ; save precedence stacked flag

RiOp_40
   PLA               ; pop byte
   LSR A             ; shift out comparison evaluation lowest bit
   STA TANSGN        ; save the comparison evaluation flag
   PLA               ; pop exponent
   STA FAC2EX        ; save FAC2 exponent
   PULLW(FAC2M1)     ; pull FAC2 mantissa 1 & 2
   PULLW(FAC2M3)     ; pull FAC2 mantissa 3 & 4
   PLA               ; pop sign
   STA FAC2SI        ; save FAC2 sign (b7)
   EOR FAC1SI        ; EOR FAC1 sign (b7)
   STA STRPTR        ; save sign compare (FAC1 EOR FAC2)

RiOp_50
   LDA FAC1EX        ; get FAC1 exponent
   RTS

; ********
  Evaluate
; ********

   JMP (IEVAL)       ; normally Default_EVAL

; ************
  Default_EVAL
; ************

   LDA #0
   STA VALTYP        ; clear data type flag, $FF = string, $00 = numeric

EVA_10
   JSR CHRGET
   BCS EVA_30        ; if not numeric character continue

EVA_20
   JMP Load_FAC1_From_String

EVA_30
   JSR Is_Alpha
   BCC EVA_40
   JMP Get_Var       ; variable name set-up and return

EVA_40
   CMP #TK_PI
   BNE EVA_50        ; if not PI continue
   LAYI(Float_PI)
   JSR Load_FAC1_AY
   JMP CHRGET

Float_PI .real 3.141592653

EVA_50
   CMP #'.'
   BEQ EVA_20        ; if so get FAC1 from string and return, e.g. was .123
   CMP #TK_MINUS
   BEQ Prep_Minus_Operation
   CMP #TK_PLUS
   BEQ EVA_10        ; if + token ignore the leading +, +1 = 1
   CMP #QUOTE
   BNE EVA_70        ; if not open quote continue

; ********************************
  Make_String_Descriptor_From_Code
; ********************************
```

```
   LDAY(TXTPTR)      ; get BASIC execute pointer
   ADC #0            ; add carry to low byte
   BCC EVA_60        ; branch if no overflow
   INY               ; increment high byte

EVA_60
   JSR Create_String_Descriptor
   JMP Restore_Execution_Pointer

EVA_70
   CMP #TK_NOT       ; compare with token for NOT
   BNE EVA_80        ; if not token for NOT continue
   LDY #$18          ; offset to NOT function
   BNE Prep_Operation      ; do set-up for function then execute, branch always

; ***********
  Basic_EQUAL
; ***********

   JSR Eval_Integer
   LDA FAC1M4        ; get FAC1 mantissa 4
   EOR #$FF          ; invert it
   TAY               ; copy it
   LDA FAC1M3        ; get FAC1 mantissa 3
   EOR #$FF          ; invert it
   JMP Integer_To_Float

EVA_80
   CMP #TK_FN        ; compare with token for FN
   BNE EVA_90        ; if not token for FN continue
   JMP Eval_FNX      ; else go evaluate FNx

EVA_90
   CMP #TK_SGN       ; compare with token for SGN
   BCC Eval_In_Parenthesis
   JMP Function_Call

; *******************
  Eval_In_Parenthesis
; *******************

   JSR Need_Left_Parenthesis
   JSR Eval_Expression

; **********************
  Need_Right_Parenthesis
; **********************

   LDA #')'
   .byte   $2C       ; skip until Need_A

; *********************
  Need_Left_Parenthesis
; *********************

   LDA #'('
   .byte   $2C       ; skip until Need_A

; **********
  Need_Comma
; **********
```

```
   LDA #','

; ******
  Need_A
; ******

   LDY #0
   CMP (TXTPTR),Y    ; compare with BASIC byte
   BNE Syntax_Error
   JMP CHRGET        ; else next program byte and return

; ============
  Syntax_Error
; ============

   LDX #$0B          ; error code $0B, syntax error
   JMP Basic_Error

; ====================
  Prep_Minus_Operation
; ====================

   LDY #$15          ; set offset from base to > operator

Prep_Operation
   PLA               ; dump return address low byte
   PLA               ; dump return address high byte
   JMP EvEx_45      ; execute function then continue evaluation

; ******************
  Is_Inside_BASIC_ROM
; ******************

   SEC
   LDA FAC1M3        ; get variable address low byte
   SBC #<BASIC_ROM   ; subtract BASIC_ROM low byte
   LDA FAC1M4        ; get variable address high byte
   SBC #>BASIC_ROM   ; subtract BASIC_ROM high byte
   BCC IIBR_Ret      ; exit if address < BASIC_ROM
   LDA #<CHRGET_ROM  ; get end of BASIC marker low byte
   SBC FAC1M3        ; subtract variable address low byte
   LDA #>CHRGET_ROM  ; get end of BASIC marker high byte
   SBC FAC1M4        ; subtract variable address high byte

IIBR_Ret
   RTS

; =======
  Get_Var
; =======

   JSR Get_Scalar_Address
   STAY(FAC1M3)      ; save variable pointer low byte
   LDXY(VARNAM)      ; get current variable name first character
   LDA VALTYP        ; get data type flag, $FF = string, $00 = numeric
   BEQ Load_Value    ; if numeric go handle a numeric variable
   LDA #0
   STA FAC1M5        ; clear FAC1 rounding byte
   JSR Is_Inside_BASIC_ROM
   BCC GeVa_Ret      ; exit if not in BASIC ROM
   CPX #'T'          ; compare variable name first character with "T"
   BNE GeVa_Ret      ; exit if not "T"
   CPY #'I'+$80      ; compare variable name second character with "I$"
   BNE GeVa_Ret      ; exit if not "I$"
   JSR Load_Jiffyclock
   STY TMPVA2        ; clear exponent count adjust
   DEY               ; Y = $FF
   STY TMPPTD        ; set output string index, -1 to allow for pre increment
   LDY #6            ; HH:MM:SS is six digits
   STY TMPVA1        ; set number of characters before the decimal point
   LDY #Jiffy_Conversion_Table-Decimal_Conversion_Table
   JSR Format_Jiffyclock
   JMP BaST_10      ; exit via STR$() code tail

GeVa_Ret
   RTS

; ==========
  Load_Value
; ==========

   BIT INTFLG        ; test data type flag, $80 = integer, $00 = float
   BPL Load_Float    ; if float go handle float
   LDY #0
   LDA (FAC1M3),Y    ; get integer variable low byte
   TAX               ; copy to X
   INY
   LDA (FAC1M3),Y    ; get integer variable high byte
   TAY               ; copy to Y
   TXA               ; copy loa byte to A
   JMP Integer_To_Float

; ==========
  Load_Float
; ==========

   JSR Is_Inside_BASIC_ROM
   BCC Load_Float_Var ; if not get pointer and unpack into FAC1
   CPX #'T'
   BNE Check_ST_Var
   CPY #'I'          ; is it "TI" ?
   BNE Load_Float_Var
   JSR Load_Jiffyclock
   TYA               ; clear A
   LDX #$A0          ; set exponent to 32 bit value
   JMP CITF_10       ; set exponent = X and normalise FAC1

; ***************
  Load_Jiffyclock
; ***************

   JSR RDTIM         ; Read system clock
   STX FAC1M3        ; save jiffy clock mid byte as  FAC1 mantissa 3
   STY FAC1M2        ; save jiffy clock high byte as  FAC1 mantissa 2
   STA FAC1M4        ; save jiffy clock low byte as  FAC1 mantissa 4
   LDY #$00          ; clear Y
   STY FAC1M1        ; clear FAC1 mantissa 1
   RTS

; ============
  Check_ST_Var
; ============

   CPX #'S'
   BNE Load_Float_Var
   CPY #'T'          ; is it "ST" ?
   BNE Load_Float_Var
   JSR READST
   JMP A_To_FAC1

; ==============
  Load_Float_Var
; ==============

   LDAY(FAC1M3)      ; get variable pointer
   JMP Load_FAC1_AY

; =============
  Function_Call
; =============

   ASL A             ; offset = 2 * (token  - $80) : bit 7 shifted out
   PHA               ; save function offset
   TAX               ; copy function offset
   JSR CHRGET
   CPX #[[TK_CHRS - $80] * 2] + 1 ; chr$ index + 1
   BCC FuCa_10       ; branch if not left$, right$, mid$
   JSR Need_Left_Parenthesis
   JSR Eval_Expression
   JSR Need_Comma
   JSR Assert_String_Type
   PLA               ; restore function offset
   TAX               ; copy it
   PUSHW(FAC1M3)     ; push string pointer
   TXA               ; restore function offset
   PHA               ; save function offset
   JSR Get_Byte_Value
   PLA               ; restore function offset
   TAY               ; copy function offset
   TXA               ; copy byte parameter to A
   PHA               ; push byte parameter
   JMP FuCa_20       ; go call function

FuCa_10
   JSR Eval_In_Parenthesis
   PLA               ; restore function offset
   TAY               ; copy to index

FuCa_20
   LDA Basic_Function_Table-2*[TK_SGN-$80],Y    ; .. -$68
   STA FUNJMP
   LDA Basic_Function_Table-2*[TK_SGN-$80]+1,Y  ; .. -$67
   STA FUNJMP+1
   JSR JUMPER
   JMP Is_Numeric

; ********
  Basic_OR
; ********

   LDY #$FF          ; set Y for OR
   .byte   $2C       ; skip next statement

; *********
  Basic_AND
; *********
```

```
   LDY #$00           ; clear Y for AND
   STY COUNT          ; set AND/OR invert value
   JSR Eval_Integer
   LDA FAC1M3         ; get FAC1 mantissa 3
   EOR COUNT          ; EOR low byte
   STA CHARAC         ; save it
   LDA FAC1M4         ; get FAC1 mantissa 4
   EOR COUNT          ; EOR high byte
   STA ENDCHR         ; save it
   JSR FAC2_To_FAC1
   JSR Eval_Integer
   LDA FAC1M4         ; get FAC1 mantissa 4
   EOR COUNT          ; EOR high byte
   AND ENDCHR         ; AND with expression 1 high byte
   EOR COUNT          ; EOR result high byte
   TAY                ; save in Y
   LDA FAC1M3         ; get FAC1 mantissa 3
   EOR COUNT          ; EOR low byte
   AND CHARAC         ; AND with expression 1 low byte
   EOR COUNT          ; EOR result low byte
   JMP Integer_To_Float

; **********
  Basic_LESS
; **********

   JSR Check_Var_Type
   BCS BaLE_10        ; if string go do string compare
   LDA FAC2SI         ; get FAC2 sign (b7)
   ORA #$7F           ; set all non sign bits
   AND FAC2M1         ; and FAC2 mantissa 1 (AND in sign bit)
   STA FAC2M1         ; save FAC2 mantissa 1
   LAYI(FAC2EX)
   JSR Compare_FAC1_AY
   TAX                ; copy the result
   JMP BaLE_40        ; go evaluate result

BaLE_10               ; compare strings
   LDA #0
   STA VALTYP         ; clear data type flag, $FF = string, $00 = numeric
   DEC ACCSYM         ; clear < bit in comparrison evaluation flag
   JSR Get_String_Descriptor
   STA FAC1EX         ; save length
   STX FAC1M1         ; save string pointer low byte
   STY FAC1M2         ; save string pointer high byte
   LDA FAC2M3         ; get descriptor pointer low byte
   LDY FAC2M4         ; get descriptor pointer high byte
   JSR Get_String_Descriptor_AY
   STX FAC2M3         ; save string pointer low byte
   STY FAC2M4         ; save string pointer high byte
   TAX                ; copy length
   SEC
   SBC FAC1EX         ; subtract string 1 length
   BEQ BaLE_20        ; if str 1 length = string 2 length go compare the strings
   LDA #1             ; set str 1 length > string 2 length
   BCC BaLE_20        ; if so return + 1 if otherwise equal
   LDX FAC1EX         ; get string 1 length
   LDA #$FF           ; set str 1 length < string 2 length

BaLE_20
   STA FAC1SI         ; save length compare
   LDY #$FF          ; set index
   INX               ; adjust for loop

BaLE_30
   INY
   DEX               ; decrement count
   BNE BaLE_50       ; if still bytes to do go compare them
   LDX FAC1SI        ; get length compare back

BaLE_40
   BMI BaLE_60       ; branch if str 1 < str 2
   CLC               ; flag str 1 <= str 2
   BCC BaLE_60       ; go evaluate result, branch always

BaLE_50
   LDA (FAC2M3),Y    ; get string 2 byte
   CMP (FAC1M1),Y    ; compare with string 1 byte
   BEQ BaLE_30       ; loop if bytes =
   LDX #$FF          ; set str 1 < string 2
   BCS BaLE_60       ; branch if so

   LDX #$01          ; set str 1 > string 2
BaLE_60
   INX               ; x = 0, 1 or 2
   TXA               ; copy to A
   ROL A             ; * 2 (1, 2 or 4)
   AND TANSGN        ; AND with the comparison evaluation flag
   BEQ BaLE_70       ; branch if 0 (compare is false)

   LDA #$FF          ; else set result true
BaLE_70
   JMP A_To_FAC1

DIM_00
   JSR Need_Comma

; *********
  Basic_DIM
; *********

   TAX               ; copy "DIM" flag to X
   JSR Get_Array_Address
   JSR CHRGOT
   BNE DIM_00      ; scan for "," and loop if not null
   RTS

; ******************
  Get_Scalar_Address
; ******************

   LDX #$00          ; set DIM flag = $00
   JSR CHRGOT        ; 1st. character

; -----------------
  Get_Array_Address
; -----------------

   STX DIMFLG        ; save DIM flag

; --------------
  Get_FN_Address
; --------------
```

```
   STA VARNAM        ; save 1st character
   JSR CHRGOT
   JSR Is_Alpha
   BCS Get_Address      ; if ok continue

Var_Syntax_Error
   JMP Syntax_Error

; ===========
  Get_Address
; ===========

   LDX #0            ; clear 2nd character temp
   STX VALTYP        ; clear data type flag, $FF = string, $00 = numeric
   STX INTFLG        ; clear data type flag, $80 = integer, $00 = float
   JSR CHRGET        ; 2nd character
   BCC GAdd_05       ; if character = "0"-"9" (ok) go save 2nd character
   JSR Is_Alpha
   BCC GAdd_15       ; if <"A" or >"Z" go check if string

GAdd_05
   TAX               ; copy 2nd character

GAdd_10
   JSR CHRGET        ; 3rd character
   BCC GAdd_10       ; loop if character = "0"-"9" (ignore)
   JSR Is_Alpha
   BCS GAdd_10       ; loop if character = "A"-"Z" (ignore)

GAdd_15
   CMP #'$'
   BNE GAdd_20       ; if not string go check integer
   LDA #$FF          ; set data type = string
   STA VALTYP        ; set data type flag, $FF = string, $00 = numeric
   BNE GAdd_25       ; branch always

GAdd_20
   CMP #'%'
   BNE GAdd_30       ; if not integer go check for an array
   LDA SUBFLG        ; get subscript/FNX flag
   BNE Var_Syntax_Error      ; if ?? do syntax error then warm start
   LDA #$80          ; set integer type
   STA INTFLG        ; set data type = integer
   ORA VARNAM        ; OR current variable name first byte
   STA VARNAM        ; save current variable name first byte

GAdd_25
   TXA               ; get 2nd character back
   ORA #$80          ; set top bit, indicate string or integer variable
   TAX               ; copy back to 2nd character temp
   JSR CHRGET

GAdd_30
   STX VARNAM+1      ; save 2nd character
   SEC
   ORA SUBFLG        ; or with subscript/FNX flag - or FN name
   SBC #'('
   BNE GAdd_35       ; if not "(" go find a plain numeric variable
   JMP Find_Array

GAdd_35
```

```
   LDY #0
   STY SUBFLG        ; clear subscript/FNX flag
   LDAX(VARTAB)      ; get start of variables

GAdd_40
   STX TMPPTC+1      ; save search address high byte

GAdd_45
   STA TMPPTC        ; save search address low byte
   CPX ARYTAB+1      ; compare with end of variables high byte
   BNE GAdd_50       ; skip next compare if <>
   CMP ARYTAB        ; compare low address with end of variables low byte
   BEQ Create_Var      ; if not found go make new variable

GAdd_50
   LDA VARNAM        ; get 1st character of variable to find
   CMP (TMPPTC),Y    ; compare with variable name 1st character
   BNE GAdd_55       ; if no match go try the next variable
   LDA VARNAM+1      ; get 2nd character of variable to find
   INY               ; index to point to variable name 2nd character
   CMP (TMPPTC),Y    ; compare with variable name 2nd character
   BEQ CrVa_70      ; if match go return the variable
   DEY               ; else decrement index (now = $00)

GAdd_55
   CLC
   LDA TMPPTC        ; get search address low byte
   ADC #7            ; +7, offset to next variable name
   BCC GAdd_45       ; loop if no overflow to high byte
   INX               ; else increment high byte
   BNE GAdd_40       ; loop always, RAM doesn't extend to $FFFF

; ********
  Is_Alpha
; ********

   CMP #'A'
   BCC IA_RET
   SBC #$5B          ; subtract "Z"+1
   SEC
   SBC #$A5          ; subtract $A5 (restore byte)
                     ; carry clear if byte > $5A
IA_RET
   RTS

; ==========
  Create_Var
; ==========

   PLA               ; pop return address low byte
   PHA               ; push return address low byte
   CMP #<[Get_Var+2] ; compare with expected calling routine return low byte
   BNE CrVa_20       ; if not get variable go create new variable

; this will only drop through if the call was from Get_Var and is only
; called from there if it is searching for a variable from the right
; hand side of a LET a=b statement, it prevents the creation of
; variables not assigned a value. value returned by this is either
; numeric zero, exponent byte is 0, or null string, descriptor length
; byte is 0. in fact a pointer to any 0 byte would have done.

CrVa_10
```

```
   LAYI(NULL_Descriptor)
   RTS

CrVa_20
   LDAY(VARNAM)      ; get variable name first character
   CMP #'T'
   BNE CrVa_40
   CPY #'I'+$80      ; is it TI$ ?
   BEQ CrVa_10
   CPY #'I'
   BNE CrVa_40       ; is it TI ?

CrVa_30
   JMP Syntax_Error

CrVa_40
   CMP #'S'          ; compare first character with "S"
   BNE CrVa_50      ; if not "S" continue
   CPY #'T'          ; compare second character with "T"
   BEQ CrVa_30       ; if name is "ST" do syntax error

CrVa_50
   LDAY(ARYTAB)
   STAY(TMPPTC)
   LDAY(STREND)
   STAY(TMPPTB)      ; save old block end
   CLC
   ADC #7            ; +7, space for one variable
   BCC CrVa_60       ; if no overflow skip the high byte increment
   INY               ; else increment high byte

CrVa_60
   STAY(TMPPTA)      ; set new block end
   JSR Open_Up_Space
   LDAY(TMPPTA)      ; get new start
   INY               ; correct high byte
   STAY(ARYTAB)      ; set end of variables
   LDY #0
   LDA VARNAM        ; get variable name 1st character
   STA (TMPPTC),Y    ; save variable name 1st character
   INY
   LDA VARNAM+1      ; get variable name 2nd character
   STA (TMPPTC),Y    ; save variable name 2nd character
   LDA #0
   INY
   STA (TMPPTC),Y    ; initialise variable byte
   INY
   STA (TMPPTC),Y    ; initialise variable byte
   INY
   STA (TMPPTC),Y    ; initialise variable byte
   INY
   STA (TMPPTC),Y    ; initialise variable byte
   INY
   STA (TMPPTC),Y    ; initialise variable byte

CrVa_70
   LDA TMPPTC        ; get variable address low byte
   CLC
   ADC #$02          ; +2, offset past variable name bytes
   LDY TMPPTC+1      ; get variable address high byte
   BCC CrVa_80       ; if no overflow skip the high byte increment
   INY               ; else increment high byte
```

```
CrVa_80
   STAY(VARPNT)
   RTS

; **********************
  Array_Pointer_To_First
; **********************

   LDA COUNT         ; get # of dimensions (1, 2 or 3)
   ASL A             ; *2 (also clears the carry !)
   ADC #$05          ; +5 (result is 7, 9 or 11 here)
   ADC TMPPTC        ; add array start pointer low byte
   LDY TMPPTC+1      ; get array pointer high byte
   BCC APTF_10       ; if no overflow skip the high byte increment
   INY               ; else increment high byte

APTF_10
   STAY(TMPPTA)      ; save array data pointer
   RTS

Float_M32768 .real -32768

; ****************
  Float_To_Integer
; ****************

   JSR Eval_Integer
   LDA FAC1M3        ; get result low byte
   LDY FAC1M4        ; get result high byte
   RTS

; *********************
  Eval_Positive_Integer
; *********************

   JSR CHRGET
   JSR Eval_Expression

; ***************************
  Eval_Positive_Integer_Check
; ***************************

   JSR Is_Numeric
   LDA FAC1SI        ; get FAC1 sign (b7)
   BMI EvIn_10       ; do illegal quantity error negative

; evaluate integer expression, no sign check

; ************
  Eval_Integer
; ************

   LDA FAC1EX        ; get FAC1 exponent
   CMP #$90          ; compare with exponent = 2^16 (n>2^15)
   BCC EvIn_20       ; if n<2^16 go convert FAC1 floating to fixed and return
   LAYI(Float_M32768); set pointer -32768
   JSR Compare_FAC1_AY

EvIn_10
   BNE Illegal_Quantity
```

```
EvIn_20
   JMP FAC1_To_Integer

; **********
  Find_Array
; **********

   LDA DIMFLG        ; get DIM flag
   ORA INTFLG        ; OR with data type flag
   PHA               ; push it
   LDA VALTYP        ; get data type flag, $FF = string, $00 = numeric
   PHA               ; push it
   LDY #0            ; clear dimensions count

FiAr_05              ; get the array dimensions and stack them
   TYA               ; copy dimensions count
   PHA               ; save it
   PUSHW(VARNAM)     ; push array name
   JSR Eval_Positive_Integer
   PULLW(VARNAM)     ; pull array name
   PLA               ; pull dimensions count
   TAY               ; restore it
   TSX               ; copy stack pointer
   LDA STACK+2,X     ; get DIM flag
   PHA               ; push it
   LDA STACK+1,X     ; get data type flag
   PHA               ; push it
   LDA FAC1M3        ; get this dimension size high byte
   STA STACK+2,X     ; stack before flag bytes
   LDA FAC1M4        ; get this dimension size low byte
   STA STACK+1,X     ; stack before flag bytes
   INY               ; increment dimensions count
   JSR CHRGOT
   CMP #','
   BEQ FiAr_05       ; if found go do next dimension
   STY COUNT         ; store dimensions count
   JSR Need_Right_Parenthesis
   PLA               ; pull data type flag
   STA VALTYP        ; restore data type flag, $FF = string, $00 = numeric
   PLA               ; pull data type flag
   STA INTFLG        ; restore data type flag, $80 = integer, $00 = float
   AND #$7F          ; mask dim flag
   STA DIMFLG        ; restore DIM flag
   LDX ARYTAB        ; set end of variables low byte
   LDA ARYTAB+1      ; set end of variables high byte

FiAr_10
   STX TMPPTC        ; save as array start pointer low byte
   STA TMPPTC+1      ; save as array start pointer high byte
   CMP STREND+1      ; compare with end of arrays high byte
   BNE FiAr_15       ; if not reached array memory end continue searching
   CPX STREND        ; else compare with end of arrays low byte
   BEQ FiAr_30       ; go build array if not found

FiAr_15
   LDY #0
   LDA (TMPPTC),Y    ; get array name first byte
   INY
   CMP VARNAM        ; compare with this array name first byte
   BNE FiAr_20       ; if no match go try the next array
   LDA VARNAM+1      ; else get this array name second byte
   CMP (TMPPTC),Y    ; compare with array name second byte
   BEQ FiAr_25       ; array found so branch

FiAr_20
   INY
   LDA (TMPPTC),Y    ; get array size low byte
   CLC
   ADC TMPPTC        ; add array start pointer low byte
   TAX               ; copy low byte to X
   INY
   LDA (TMPPTC),Y    ; get array size high byte
   ADC TMPPTC+1      ; add array memory pointer high byte
   BCC FiAr_10      ; if no overflow go check next array

; =============
  Bad_Subscript
; =============

   LDX #$12          ; error $12, bad subscript error
   .byte   $2C       ; skip next statement

; ================
  Illegal_Quantity
; ================

   LDX #$0E          ; error $0E, illegal quantity error

Jump_Basic_Error
   JMP Basic_Error

FiAr_25
   LDX #$13          ; set error $13, double dimension error
   LDA DIMFLG        ; get DIM flag
   BNE Jump_Basic_Error
   JSR Array_Pointer_To_First
   LDA COUNT         ; get dimensions count
   LDY #4            ; set index to array's # of dimensions
   CMP (TMPPTC),Y    ; compare with no of dimensions
   BNE Bad_Subscript ; if wrong do bad subscript error
   JMP Find_Array_Element

FiAr_30
   JSR Array_Pointer_To_First
   JSR Check_Mem_Avail
   LDY #0
   STY TMPPTD+1      ; clear array data size high byte
   LDX #5            ; set default element size
   LDA VARNAM        ; get variable name 1st byte
   STA (TMPPTC),Y    ; save array name 1st byte
   BPL FiAr_35       ; branch if not string or floating point array
   DEX               ; decrement element size, $04

FiAr_35
   INY
   LDA VARNAM+1      ; get variable name 2nd byte
   STA (TMPPTC),Y    ; save array name 2nd byte
   BPL FiAr_40       ; branch if not integer or string
   DEX               ; decrement element size, $03
   DEX               ; decrement element size, $02

FiAr_40
   STX TMPPTD        ; save element size
   LDA COUNT         ; get dimensions count
   INY
   INY               ; .. to array  ..
   INY               ; .. dimension count
   STA (TMPPTC),Y    ; save array dimension count

FiAr_45
   LDX #11           ; set default dimension size low byte (0:10)
   LDA #0            ; set default dimension size high byte
   BIT DIMFLG        ; test DIM flag
   BVC FiAr_50       ; if default to be used don't pull a dimension
   PLA               ; pull dimension size low byte
   CLC
   ADC #1            ; add 1, allow for zeroeth element
   TAX               ; copy low byte to X
   PLA               ; pull dimension size high byte
   ADC #0            ; add carry to high byte

FiAr_50
   INY               ; incement index to dimension size high byte
   STA (TMPPTC),Y    ; save dimension size high byte
   INY               ; incement index to dimension size low byte
   TXA               ; copy dimension size low byte
   STA (TMPPTC),Y    ; save dimension size low byte
   JSR Compute_Array_Size
   STX TMPPTD        ; save result low byte
   STA TMPPTD+1      ; save result high byte
   LDY INDEXA        ; restore index
   DEC COUNT         ; decrement dimensions count
   BNE FiAr_45       ; loop if not all done
   ADC TMPPTA+1      ; add array data pointer high byte
   BCS FiAE_30       ; if overflow do out of memory error then warm start
   STA TMPPTA+1      ; save array data pointer high byte
   TAY               ; copy array data pointer high byte
   TXA               ; copy array size low byte
   ADC TMPPTA        ; add array data pointer low byte
   BCC FiAr_55       ; if no rollover skip the high byte increment
   INY               ; else increment next array pointer high byte
   BEQ FiAE_30       ; if rolled over do out of memory error then warm start

FiAr_55
   JSR Check_Mem_Avail
   STAY(STREND)      ; now we need to zero all the elements in it
   LDA #0            ; for array clear
   INC TMPPTD+1      ; increment array size high byte, now block count
   LDY TMPPTD        ; get array size low byte, now index to block
   BEQ FiAr_65       ; if $00 go do the high byte decrement

FiAr_60
   DEY               ; decrement index, do 0 to n-1
   STA (TMPPTA),Y    ; clear array element byte
   BNE FiAr_60       ; loop until this block done

FiAr_65
   DEC TMPPTA+1      ; decrement array pointer high byte
   DEC TMPPTD+1      ; decrement block count high byte
   BNE FiAr_60       ; loop until all blocks done
   INC TMPPTA+1      ; correct for last loop
   SEC
   LDA STREND        ; get end of arrays low byte
   SBC TMPPTC        ; subtract array start low byte
   LDY #$02          ; index to array size low byte
   STA (TMPPTC),Y    ; save array size low byte
   LDA STREND+1      ; get end of arrays high byte
   INY               ; index to array size high byte
   SBC TMPPTC+1      ; subtract array start high byte
   STA (TMPPTC),Y    ; save array size high byte
   LDA DIMFLG        ; get default DIM flag
   BNE FiAE_Ret      ; exit if this was a DIM command
   INY               ; set index to # of dimensions, the dimension indeces
                     ; are on the stack and will be removed as the position
                     ; of the array element is calculated

; ==================
  Find_Array_Element
; ==================

   LDA (TMPPTC),Y    ; get array's dimension count
   STA COUNT         ; save it
   LDA #0
   STA TMPPTD        ; clear array data pointer low byte

FiAE_10
   STA TMPPTD+1      ; save array data pointer high byte
   INY
   PLA               ; pull array index low byte
   TAX               ; copy to X
   STA FAC1M3        ; save index low byte to FAC1 mantissa 3
   PLA               ; pull array index high byte
   STA FAC1M4        ; save index high byte to FAC1 mantissa 4
   CMP (TMPPTC),Y    ; compare with array bound high byte
   BCC FiAE_40       ; if within bounds continue
   BNE FiAE_20       ; if outside bounds do bad subscript error
   INY               ; index to array bound low byte
   TXA               ; get array index low byte
   CMP (TMPPTC),Y    ; compare with array bound low byte
   BCC FiAE_50       ; if within bounds continue

FiAE_20
   JMP Bad_Subscript

FiAE_30
   JMP Error_Out_Of_Memory

FiAE_40
   INY               ; index to array bound low byte

FiAE_50
   LDA TMPPTD+1      ; get array data pointer high byte
   ORA TMPPTD        ; OR with array data pointer low byte
   CLC
   BEQ FiAE_60       ; if array data pointer = null skip the multiply
   JSR Compute_Array_Size
   TXA               ; get result low byte
   ADC FAC1M3        ; add index low byte from FAC1 mantissa 3
   TAX               ; save result low byte
   TYA               ; get result high byte
   LDY INDEXA        ; restore index

FiAE_60
   ADC FAC1M4        ; add index high byte from FAC1 mantissa 4
   STX TMPPTD        ; save array data pointer low byte
   DEC COUNT         ; decrement dimensions count
   BNE FiAE_10       ; loop if dimensions still to do
   STA TMPPTD+1      ; save array data pointer high byte
   LDX #$05          ; set default element size
   LDA VARNAM        ; get variable name 1st byte
   BPL FiAE_70       ; branch if not string or floating point array
   DEX               ; decrement element size, $04

FiAE_70
   LDA VARNAM+1      ; get variable name 2nd byte
   BPL FiAE_80       ; branch if not integer or string
   DEX               ; decrement element size, $03
   DEX               ; decrement element size, $02

FiAE_80
   STX FAC3+3       ; save dimension size low byte
   LDA #$00          ; clear dimension size high byte
   JSR Compute_Array_Size_A
   TXA               ; copy array size low byte
   ADC TMPPTA        ; add array data start pointer low byte
   STA VARPNT        ; save as current variable pointer low byte
   TYA               ; copy array size high byte
   ADC TMPPTA+1      ; add array data start pointer high byte
   STA VARPNT+1      ; save as current variable pointer high byte
   TAY               ; copy high byte to Y
   LDA VARPNT        ; get current variable pointer low byte
                     ; pointer to element is now in AY
FiAE_Ret
   RTS

; ******************
  Compute_Array_Size
; ******************

   STY INDEXA        ; save index
   LDA (TMPPTC),Y    ; get dimension size low byte
   STA FAC3+3       ; save dimension size low byte
   DEY               ; decrement index
   LDA (TMPPTC),Y    ; get dimension size high byte

; --------------------
  Compute_Array_Size_A
; --------------------

   STA FAC3+4       ; save dimension size high byte
   LDA #$10          ; count = $10 (16 bit multiply)
   STA TMPVA1        ; save bit count
   LDX #$00          ; clear result low byte
   LDY #$00          ; clear result high byte

CAS_10
   TXA               ; get result low byte
   ASL A             ; *2
   TAX               ; save result low byte
   TYA               ; get result high byte
   ROL A             ; *2
   TAY               ; save result high byte
   BCS FiAE_30       ; if overflow go do "Out of memory" error
   ASL TMPPTD        ; shift element size low byte
   ROL TMPPTD+1      ; shift element size high byte
   BCC CAS_20        ; skip add if no carry
   CLC               ; else clear carry for add
   TXA               ; get result low byte
   ADC FAC3+3       ; add dimension size low byte
   TAX               ; save result low byte
   TYA                ; get result high byte
   ADC FAC3+4        ; add dimension size high byte
   TAY                ; save result high byte
   BCS FiAE_30        ; if overflow go do "Out of memory" error

CAS_20
   DEC TMPVA1         ; decrement bit count
   BNE CAS_10         ; loop until all done
   RTS

; *********
  Basic_FRE
; *********

   LDA VALTYP         ; get data type flag, $FF = string, $00 = numeric
   BEQ FRE_10         ; if numeric don't pop the string
   JSR Get_String_Descriptor
                      ; FRE(n) was numeric so do this
FRE_10
   JSR Garbage_Collection
   SEC
   LDA FRESPC         ; get bottom of string space low byte
   SBC STREND         ; subtract end of arrays low byte
   TAY                ; copy result to Y
   LDA FRESPC+1       ; get bottom of string space high byte
   SBC STREND+1       ; subtract end of arrays high byte

; ***************
  Integer_To_Float
; ***************

   LDX #$00           ; set type = numeric
   STX VALTYP         ; clear data type flag, $FF = string, $00 = numeric
   STA FAC1M1         ; save FAC1 mantissa 1
   STY FAC1M2         ; save FAC1 mantissa 2
   LDX #$90           ; set exponent=2^16 (integer)
   JMP Int_To_Float_Exp_X      ; set exp = X, clear FAC1 3 and 4, normalise and return

; *********
  Basic_POS
; *********

   SEC                ; set Cb for read cursor position
   JSR PLOT           ; Read or set cursor location

; ==========
  Y_To_Float
; ==========

   LDA #0             ; clear high byte
   BEQ Integer_To_Float

; ****************
  Assert_Non_Direct
; ****************

   LDX CURLIN+1       ; get current line number high byte
   INX                ; increment it
   BNE FiAE_Ret       ; return if not direct mode
   LDX #$15           ; error $15, illegal direct error
   .byte   $2C        ; skip next statement
```

```
; ==================
  Undefined_Function
; ==================

   LDX #$1B          ; error $1B, undefined function error
   JMP Basic_Error

; *********
  Basic_DEF
; *********

   JSR Get_FN
   JSR Assert_Non_Direct
   JSR Need_Left_Parenthesis
   LDA #$80          ; set flag for FNx
   STA SUBFLG        ; save subscript/FNx flag
   JSR Get_Scalar_Address
   JSR Is_Numeric
   JSR Need_Right_Parenthesis
   LDA #TK_EQUAL     ; get = token
   JSR Need_A
   PHA               ; push next character
   PUSHW(VARPNT)     ; push current variable pointer
   PUSHW(TXTPTR)     ; push BASIC execute pointer
   JSR Basic_DATA    ; perform DATA
   JMP EvFN_30      ; put execute pointer and variable pointer into function

; ******
  Get_FN
; ******

   LDA #TK_FN        ; set FN token
   JSR Need_A
   ORA #$80          ; set FN flag bit
   STA SUBFLG        ; save FN name
   JSR Get_FN_Address
   STAY(FUNCPT)
   JMP Is_Numeric

; ********
  Eval_FNX
; ********

   JSR Get_FN
   PUSHW(FUNCPT)     ; push function pointer
   JSR Eval_In_Parenthesis
   JSR Is_Numeric
   PULLW(FUNCPT)     ; pull it
   LDY #$02          ; index to variable pointer high byte
   LDA (FUNCPT),Y    ; get variable address low byte
   STA VARPNT        ; save current variable pointer low byte
   TAX               ; copy address low byte
   INY               ; index to variable address high byte
   LDA (FUNCPT),Y    ; get variable pointer high byte
   BEQ Undefined_Function      ; if high byte zero go do undefined function error
   STA VARPNT+1      ; save current variable pointer high byte
   INY               ; index to mantissa 3

EvFN_10
   LDA (VARPNT),Y    ; get byte from variable
   PHA               ; stack it
   DEY               ; decrement index
   BPL EvFN_10      ; loop until variable stacked
   LDY VARPNT+1      ; get current variable pointer high byte
   JSR Assign_FAC1_To_Var
   PUSHW(TXTPTR)     ; push BASIC execute pointer
   LDA (FUNCPT),Y    ; get function execute pointer low byte
   STA TXTPTR        ; save BASIC execute pointer low byte
   INY               ; index to high byte
   LDA (FUNCPT),Y    ; get function execute pointer high byte
   STA TXTPTR+1      ; save BASIC execute pointer high byte
   PUSHW(VARPNT)     ; push current variable pointer
   JSR Eval_Numeric
   PULLW(FUNCPT)     ; pull variable address
   JSR CHRGOT
   BEQ EvFN_20      ; if null (should be [EOL] marker) continue
   JMP Syntax_Error

EvFN_20
   PULLW(TXTPTR)     ; pull BASIC execute pointer

EvFN_30
   LDY #0
   PLA               ; pull BASIC execute pointer low byte
   STA (FUNCPT),Y    ; save to function
   PLA               ; pull BASIC execute pointer high byte
   INY
   STA (FUNCPT),Y    ; save to function
   PLA               ; pull current variable address low byte
   INY
   STA (FUNCPT),Y    ; save to function
   PLA               ; pull current variable address high byte
   INY
   STA (FUNCPT),Y    ; save to function
   PLA               ; pull ??
   INY
   STA (FUNCPT),Y    ; save to function
   RTS

; *********
  Basic_STR
; *********

   JSR Is_Numeric
   LDY #0
   JSR Format_FAC1_Y
   PLA               ; dump return address (skip type check)
   PLA               ; dump return address (skip type check)

BaST_10
   LAYI(BASSTO)      ; set result string
   BEQ Create_String_Descriptor

; ********************
  Allocate_String_FAC1
; ********************

   LDXY(FAC1M3)
   STXY(DESCPT)

; *****************
  Allocate_String_A
; *****************
```

```
   JSR Allocate_String_Space
   STX FAC1M1        ; save string pointer low byte
   STY FAC1M2        ; save string pointer high byte
   STA FAC1EX        ; save length
   RTS

; ************************
  Create_String_Descriptor
; ************************

   LDX #QUOTE
   STX CHARAC        ; set terminator 1
   STX ENDCHR        ; set terminator 2

; --------------------------
  Create_String_Descriptor_AY
; --------------------------

   STAY(STRPTR)
   STAY(FAC1M1)
   LDY #$FF          ; set length to -1

CSD_10
   INY               ; increment length
   LDA (STRPTR),Y    ; get byte from string
   BEQ CSD_30        ; exit loop if null byte [EOS]
   CMP CHARAC        ; compare with search character, terminator 1
   BEQ CSD_20        ; branch if terminator
   CMP ENDCHR        ; compare with terminator 2
   BNE CSD_10        ; loop if not terminator 2

CSD_20
   CMP #QUOTE
   BEQ CSD_40        ; branch if " (carry set if = !)

CSD_30
   CLC

CSD_40
   STY FAC1EX        ; save length in FAC1 exponent
   TYA               ; copy length to A
   ADC STRPTR        ; add string start low byte
   STA TMPPTD        ; save string end low byte
   LDX STRPTR+1      ; get string start high byte
   BCC CSD_50        ; if no low byte overflow skip the high byte increment
   INX               ; else increment high byte

CSD_50
   STX TMPPTD+1      ; save string end high byte
   LDA STRPTR+1      ; get string start high byte
   BEQ CSD_60        ; branch if in utility area
   CMP #$02          ; compare with input buffer memory high byte
   BNE Push_String_Descriptor

CSD_60
   TYA               ; copy length to A
   JSR Allocate_String_FAC1
   LDXY(STRPTR)      ; get string start

Store_And_Push_String
   JSR Store_String_XY
```

```
; ***********************
   Push_String_Descriptor
; ***********************

   LDX TEMPPT        ; get descriptor stack pointer
   CPX #QUOTE
   BNE PSD_20        ; branch if space on string stack
   LDX #$19          ; error $19, string too complex error

PSD_10
   JMP Basic_Error

PSD_20
   LDA FAC1EX        ; get string length
   STA 0,X           ; put on string stack
   LDA FAC1M1        ; get string pointer low byte
   STA 1,X           ; put on string stack
   LDA FAC1M2        ; get string pointer high byte
   STA 2,X           ; put on string stack
   LDY #0            ; clear Y
   STXY(FAC1M3)      ; save string descriptor pointer
   STY FAC1M5        ; clear FAC1 rounding byte
   DEY               ; Y = $FF
   STY VALTYP        ; save data type flag, $FF = string
   STX LASTPT        ; save current descriptor stack item pointer low byte
   INX               ; update stack pointer
   INX               ; update stack pointer
   INX               ; update stack pointer
   STX TEMPPT        ; set new descriptor stack pointer
   RTS

; *********************
  Allocate_String_Space
; *********************

   LSR GARBFL        ; clear garbage collected flag (b7)

ASS_10
   PHA               ; save string length
   EOR #$FF          ; complement it
   SEC               ; set carry for subtract, two's complement add
   ADC FRESPC        ; add bottom of string space low byte, subtract length
   LDY FRESPC+1      ; get bottom of string space high byte
   BCS ASS_20        ; skip decrement if no underflow
   DEY               ; decrement bottom of string space high byte

ASS_20
   CPY STREND+1      ; compare with end of arrays high byte
   BCC ASS_40        ; do out of memory error if less
   BNE ASS_30        ; if not = skip next test
   CMP STREND        ; compare with end of arrays low byte
   BCC ASS_40        ; do out of memory error if less

ASS_30
   STAY(FRESPC)
   STAY(UTLSTP)      ; save string utility ptr
   TAX               ; copy low byte to X
   PLA               ; get string length back
   RTS

ASS_40
   LDX #$10          ; error code $10, out of memory error
   LDA GARBFL        ; get garbage collected flag
   BMI PSD_10        ; if set then do error code X
   JSR Garbage_Collection
   LDA #$80          ; flag for garbage collected
   STA GARBFL        ; set garbage collected flag
   PLA               ; pull length
   BNE ASS_10        ; go try again (loop always, length should never be = $00)

; ******************
  Garbage_Collection
; ******************

; This routine marks all strings as uncollected by setting the bottom of
; string space FRESPC to MEMSIZ, the top of string space.
; Then it scans through all string descriptors starting with those on the
; string stack, continuing with scalar string variables and finally all
; string arrays. The string with the highest address is then moved to the
; top of string space, FRESPC is adjusted and the iteration continues
; with searching for the string with the next highest address.

   LDX MEMSIZ        ; get end of memory low byte
   LDA MEMSIZ+1      ; get end of memory high byte

GaCo_Iter
   STX FRESPC        ; set bottom of string space low byte
   STA FRESPC+1      ; set bottom of string space high byte
   LDY #0
   STY FUNCPT+1      ; clear working pointer high byte
   STY FUNCPT        ; clear working pointer low byte
   LDAX(STREND)      ; get end of arrays
   STAX(TMPPTC)      ; save as highest uncollected string pointer
   LDA #TEMPST       ; set descriptor stack pointer
   LDX #0            ; check first descriptors on string stack
   STAX(INDEXA)      ; save descriptor stack pointer

GaCo_Loop_1
   CMP TEMPPT        ; compare with descriptor stack pointer
   BEQ GaCo_10       ; branch if descripor on stack
   JSR Check_String  ;
   BEQ GaCo_Loop_1   ; loop always (Check_String returns with LDY #0)

GaCo_10              ; done stacked strings, now do string variables
   LDA #7            ; set step size = 7, collecting variables
   STA GARBSS        ; save garbage collection step size
   LDAX(VARTAB)      ; get start of variables
   STAX(INDEXA)      ; save as pointer

GaCo_20
   CPX ARYTAB+1      ; compare end of variables high byte,
   BNE GaCo_30       ; branch if no high byte match
   CMP ARYTAB        ; else compare end of variables low byte,
   BEQ GaCo_40       ; branch if = variable memory end

GaCo_30
   JSR Check_Variable
   BEQ GaCo_20       ; loop always

GaCo_40
   STAX(TMPPTA)      ; save start of arrays low byte as working pointer
   LDA #3            ; set step size, collecting descriptors
   STA GARBSS        ; save step size
```

```
GaCo_50
   LDAX(TMPPTA)      ; get pointer

GaCo_60
   CPX STREND+1      ; compare with end of arrays high byte
   BNE GaCo_70       ; branch if not at end
   CMP STREND        ; else compare with end of arrays low byte
   BNE GaCo_70       ; branch if not at end
   JMP Collect_String

GaCo_70
   STAX(INDEXA)      ; save pointer
   LDY #0
   LDA (INDEXA),Y    ; get array name first byte
   TAX               ; copy it
   INY
   LDA (INDEXA),Y    ; get array name second byte
   PHP               ; push the flags
   INY
   LDA (INDEXA),Y    ; get array size low byte
   ADC TMPPTA        ; add start of this array low byte
   STA TMPPTA        ; save start of next array low byte
   INY
   LDA (INDEXA),Y    ; get array size high byte
   ADC TMPPTA+1      ; add start of this array high byte
   STA TMPPTA+1      ; save start of next array high byte
   PLP               ; restore the flags
   BPL GaCo_50       ; skip if not string array
   TXA               ; get name first byte back
   BMI GaCo_50       ; skip if not string array
   INY
   LDA (INDEXA),Y    ; get # of dimensions
   LDY #0
   ASL A             ; *2
   ADC #5            ; +5 (array header size)
   ADC INDEXA        ; add pointer low byte
   STA INDEXA        ; save pointer low byte
   BCC GaCo_80       ; if no rollover skip the high byte increment
   INC INDEXA+1      ; else increment pointer hgih byte

GaCo_80
   LDX INDEXA+1      ; get pointer high byte

GaCo_90
   CPX TMPPTA+1      ; compare pointer high byte with end of this array high byte
   BNE GaCo_95       ; branch if not there yet
   CMP TMPPTA        ; compare pointer low byte with end of this array low byte
   BEQ GaCo_60       ; if at end of this array go check next array

GaCo_95
   JSR Check_String
   BEQ GaCo_90      ; loop

; **************
  Check_Variable
; **************

   LDA (INDEXA),Y    ; get variable name first byte
   BMI ChSt_30       ; add step and exit if not string
   INY
   LDA (INDEXA),Y    ; get variable name second byte
   BPL ChSt_30       ; add step and exit if not string
   INY

; ************
  Check_String
; ************

; INDEXA points to the string descriptor to be checked.
; Following cases are examined for the string address (X/A)
; (1) : The length (INDEXA) is zero        -> next string
; (2) : (X/A) > FRESPC (already collected) -> next string
; (3) : (X/A) < TMPPTC                     -> next string
; (4) ; (X/A) > TMMPTC                     -> TPMPTC = (X/A)

; INDEXA is updated to point to the next string descriptor by adding
; GARBSS which may be 7 for scanning string variables or 3 for scanning
; string arrays.

; On return (A/X) holds the updated INDEXA, Y=0, Z flag set

   LDA (INDEXA),Y    ; get string length
   BEQ ChSt_30       ; add step and exit if null string
   INY
   LDA (INDEXA),Y    ; get string pointer low byte
   TAX               ; copy to X
   INY
   LDA (INDEXA),Y    ; get string pointer high byte
   CMP FRESPC+1      ; compare string pointer high byte with bottom of string
   BCC ChSt_10       ; if less go test against highest
   BNE ChSt_30       ; bottom of string space less string has been collected
   CPX FRESPC        ; compare string pointer low byte with bottom of string
   BCS ChSt_30       ; if bottom of string space less string has been collected

ChSt_10
   CMP TMPPTC+1      ; compare string pointer high byte with highest uncollected
   BCC ChSt_30       ; if highest uncollected string is greater then go update
   BNE ChSt_20       ; if highest uncollected string is less then go set this
   CPX TMPPTC        ; compare string pointer low byte with highest uncollected
   BCC ChSt_30       ; if highest uncollected string is greater then go update

ChSt_20
   STX TMPPTC        ; save string pointer low byte as highest uncollected string
   STA TMPPTC+1      ; save string pointer high byte as highest uncollected
   LDAX(INDEXA)      ; get descriptor pointer
   STAX(FUNCPT)      ; save working pointer
   LDA GARBSS        ; get step size
   STA FUNJMP        ; copy step size

ChSt_30
   LDA GARBSS        ; get step size (7 or 3)
   CLC
   ADC INDEXA        ; add pointer low byte
   STA INDEXA        ; save pointer low byte
   BCC ChSt_40       ; if no rollover skip the high byte increment
   INC INDEXA+1      ; else increment pointer high byte

ChSt_40
   LDX INDEXA+1      ; get pointer high byte
   LDY #0
   RTS

; ==============
  Collect_String
```

```
; ==============

   LDA FUNCPT+1      ; get working pointer low byte
   ORA FUNCPT        ; OR working pointer high byte
   BEQ ChSt_40       ; exit if nothing to collect
   LDA FUNJMP        ; get copied step size
   AND #4            ; mask step size, 4 for variables, 0 for array or stack
   LSR A             ; 2 for variables, 0 for descriptors
   TAY               ; copy to index
   STA FUNJMP        ; save offset to descriptor start
   LDA (FUNCPT),Y    ; get string length
   ADC TMPPTC        ; add string start low byte
   STA TMPPTB        ; set block end low byte
   LDA TMPPTC+1      ; get string start high byte
   ADC #0            ; add carry
   STA TMPPTB+1      ; set block end high byte
   LDAX(FRESPC)      ; get bottom of string space
   STAX(TMPPTA)      ; save destination end
   JSR Move_Block
   LDY FUNJMP        ; restore offset to descriptor start
   INY
   LDA TMPPTA        ; get new string pointer low byte
   STA (FUNCPT),Y    ; save new string pointer low byte
   TAX               ; copy string pointer low byte
   INC TMPPTA+1      ; increment new string pointer high byte
   LDA TMPPTA+1      ; get new string pointer high byte
   INY
   STA (FUNCPT),Y    ; save new string pointer high byte
   JMP GaCo_Iter     ; XA holds new bottom of string memory pointer

; ===========
  Concatenate
; ===========

; add strings, the first string is in the descriptor, the second string is in line

   PUSHW(FAC1M3)     ; push descriptor pointer
   JSR Evaluate
   JSR Assert_String_Type
   PULLW(STRPTR)     ; pull pointer
   LDY #0
   LDA (STRPTR),Y    ; get length of first string from descriptor
   CLC
   ADC (FAC1M3),Y    ; add length of second string
   BCC Conc_10       ; if no overflow continue
   LDX #$17          ; else error $17, string too long error
   JMP Basic_Error

Conc_10
   JSR Allocate_String_FAC1
   JSR Store_String_STRPTR
   LDAY(DESCPT)      ; get descriptor pointer
   JSR Get_String_Descriptor_AY
   JSR Store_String_INDEXA
   LDAY(STRPTR)      ; get descriptor pointer
   JSR Get_String_Descriptor_AY
   JSR Push_String_Descriptor
   JMP EvEx_15      ; continue evaluation

; ******************
  Store_String_STRPTR
; ******************
```

```
   LDY #0
   LDA (STRPTR),Y    ; get string length
   PHA               ; save it
   INY
   LDA (STRPTR),Y    ; get string pointer low byte
   TAX               ; copy to X
   INY
   LDA (STRPTR),Y    ; get string pointer high byte
   TAY               ; copy to Y
   PLA               ; get length back

; ***************
  Store_String_XY
; ***************

   STXY(INDEXA)

; *******************
  Store_String_INDEXA
; *******************

   TAY               ; copy length as index
   BEQ SSIN_20       ; branch if null string
   PHA               ; save length

SSIN_10
   DEY               ; decrement length/index
   LDA (INDEXA),Y    ; get byte from string
   STA (UTLSTP),Y    ; save byte to destination
   TYA               ; y = 0 ?
   BNE SSIN_10       ; loop if not all done yet
   PLA               ; restore length

SSIN_20
   CLC
   ADC UTLSTP        ; add string utility ptr low byte
   STA UTLSTP        ; save string utility ptr low byte
   BCC SSIN_30       ; if no rollover skip the high byte increment
   INC UTLSTP+1      ; increment string utility ptr high byte

SSIN_30
   RTS

; ***********
  Eval_String
; ***********

   JSR Assert_String_Type

; *********************
  Get_String_Descriptor
; *********************

; pop string off descriptor stack, or from of string space
; returns with A = length, X = pointer low byte, Y = pointer high byte

   LDAY(FAC1M3)      ; get descriptor pointer

; ------------------------
  Get_String_Descriptor_AY
; ------------------------
```

```
   STAY(INDEXA)      ; save descriptor pointer
   JSR Pop_Descriptor_Stack
   PHP               ; save status flags
   LDY #0
   LDA (INDEXA),Y    ; get length from string descriptor
   PHA
   INY
   LDA (INDEXA),Y    ; get string pointer low byte from descriptor
   TAX
   INY
   LDA (INDEXA),Y    ; get string pointer high byte from descriptor
   TAY
   PLA               ; get string length back
   PLP               ; restore status
   BNE GSD_20        ; branch if pointer not popped
   CPY FRESPC+1      ; compare with bottom of string space high byte
   BNE GSD_20        ; branch if <>
   CPX FRESPC        ; else compare with bottom of string space low byte
   BNE GSD_20        ; branch if <>
   PHA               ; push string length
   CLC               ; string address is identical to FRESPC,
   ADC FRESPC        ; so we can free that memory easyli.
   STA FRESPC
   BCC GSD_10
   INC FRESPC+1

GSD_10
   PLA               ; pull string length

GSD_20
   STXY(INDEXA)
   RTS

; ********************
  Pop_Descriptor_Stack
; ********************

   CPY LASTPT+1      ; compare high byte with current descriptor stack item
   BNE PDS_Ret
   CMP LASTPT        ; compare low byte with current descriptor stack item
   BNE PDS_Ret
   STA TEMPPT        ; set descriptor stack pointer
   SBC #3            ; update last string pointer low byte
   STA LASTPT        ; save current descriptor stack item pointer low byte
   LDY #0            ; set Z flag : descriptor popped

PDS_Ret
   RTS

; *********
  Basic_CHR
; *********

   JSR Eval_Byte
   TXA               ; copy to A
   PHA               ; save character
   LDA #$01          ; string is single byte
   JSR Allocate_String_A
   PLA               ; get character back
   LDY #0
   STA (FAC1M1),Y    ; save byte in string - byte IS string!
   PLA               ; dump return address (skip type check)
   PLA               ; dump return address (skip type check)
   JMP Push_String_Descriptor

; **********
  Basic_LEFT
; **********

   JSR Pop_String_Descriptor_And_Byte
   CMP (DESCPT),Y    ; compare byte parameter with string length
   TYA               ; clear A

LEFT_10
   BCC LEFT_20       ; branch if string length > byte parameter
   LDA (DESCPT),Y    ; else make parameter = length
   TAX               ; copy to byte parameter copy
   TYA               ; clear string start offset

LEFT_20
   PHA               ; save string start offset

LEFT_30
   TXA               ; copy byte parameter (or string length if <)

LEFT_40
   PHA               ; save string length
   JSR Allocate_String_A
   LDAY(DESCPT)      ; get descriptor pointer low byte
   JSR Get_String_Descriptor_AY
   PLA               ; get string length back
   TAY               ; copy length to Y
   PLA               ; get string start offset back
   CLC
   ADC INDEXA        ; add start offset to string start pointer low byte
   STA INDEXA        ; save string start pointer low byte
   BCC LEFT_50       ; if no overflow skip the high byte increment
   INC INDEXA+1      ; else increment string start pointer high byte

LEFT_50
   TYA               ; copy length to A
   JSR Store_String_INDEXA
   JMP Push_String_Descriptor

; ***********
  Basic_RIGHT
; ***********

   JSR Pop_String_Descriptor_And_Byte
   CLC
   SBC (DESCPT),Y    ; subtract string length
   EOR #$FF          ; invert it (A=LEN(expression$)-l)
   JMP LEFT_10      ; go do rest of LEFT$()

; *********
  Basic_MID
; *********

   LDA #$FF          ; set default length = 255
   STA FAC1M4        ; save default length
   JSR CHRGOT
   CMP #')'
   BEQ MID_10        ; no 2nd. byte
   JSR Need_Comma
   JSR Get_Byte_Value

MID_10
   JSR Pop_String_Descriptor_And_Byte
   BEQ Jump_To_Illegal_Quantity
   DEX              ; decrement start index
   TXA              ; copy to A
   PHA              ; save string start offset
   CLC
   LDX #0           ; clear output string length
   SBC (DESCPT),Y   ; start - string length
   BCS LEFT_30      ; if start > string length go do null string
   EOR #$FF         ; complement -length
   CMP FAC1M4       ; compare with length
   BCC LEFT_40      ; if length > remaining string go do RIGHT$
   LDA FAC1M4       ; get length byte
   BCS LEFT_40      ; go do string copy, branch always

; ******************************
  Pop_String_Descriptor_And_Byte
; ******************************

   JSR Need_Right_Parenthesis
   PLA
   TAY              ; save return address low byte
   PLA
   STA FUNJMP       ; save return address high byte
   PLA              ; dump call to function vector low byte
   PLA              ; dump call to function vector high byte
   PLA              ; pull byte parameter
   TAX              ; copy byte parameter to X
   PULLW(DESCPT)    ; pull string pointer
   LDA FUNJMP       ; get return address high byte
   PHA              ; back on stack
   TYA              ; get return address low byte
   PHA              ; back on stack
   LDY #0
   TXA              ; copy byte parameter
   RTS

; *********
  Basic_LEN
; *********

   JSR Eval_String_And_Len
   JMP Y_To_Float     ; convert Y to byte in FAC1 and return

; ******************
  Eval_String_And_Len
; ******************

   JSR Eval_String
   LDX #$00         ; set data type = numeric
   STX VALTYP       ; clear data type flag, $FF = string, $00 = numeric
   TAY              ; copy length to Y
   RTS

; *********
  Basic_ASC
; *********
```

```
   JSR Eval_String_And_Len
   BEQ Jump_To_Illegal_Quantity
   LDY #0
   LDA (INDEXA),Y    ; get 1st. byte
   TAY               ; copy to Y
   JMP Y_To_Float

; ========================
  Jump_To_Illegal_Quantity
; ========================

   JMP Illegal_Quantity

; *******************
  Get_Next_Byte_Value
; *******************

   JSR CHRGET

; --------------
  Get_Byte_Value
; --------------

   JSR Eval_Numeric

; *********
  Eval_Byte
; *********

   JSR Eval_Positive_Integer_Check
   LDX FAC1M3        ; high byte must be 0
   BNE Jump_To_Illegal_Quantity
   LDX FAC1M4
   JMP CHRGOT

; *********
  Basic_VAL
; *********

   JSR Eval_String_And_Len
   BNE VAL_10        ; if not a null string go evaluate it
   JMP Clear_FAC1_Exp_And_Sign

VAL_10
   LDXY(TXTPTR)
   STXY(TMPPTD)
   LDX INDEXA        ; get string pointer low byte
   STX TXTPTR        ; save BASIC execute pointer low byte
   CLC
   ADC INDEXA        ; add string length
   STA INDEXB        ; save string end low byte
   LDX INDEXA+1      ; get string pointer high byte
   STX TXTPTR+1      ; save BASIC execute pointer high byte
   BCC VAL_20        ; if no rollover skip the high byte increment
   INX               ; increment string end high byte

VAL_20
   STX INDEXB+1      ; save string end high byte
   LDY #0
   LDA (INDEXB),Y    ; get string end byte
   PHA               ; push it
   TYA               ; clear A
   STA (INDEXB),Y    ; terminate string with 0
   JSR CHRGOT
   JSR Load_FAC1_From_String
   PLA               ; restore string end byte
   LDY #0
   STA (INDEXB),Y    ; put string end byte back

; -------------------------
  Restore_Execution_Pointer
; -------------------------

   LDXY(TMPPTD)
   STXY(TXTPTR)
   RTS

; *****************
  Get_Word_And_Byte
; *****************

   JSR Eval_Numeric
   JSR FAC1_To_LINNUM

; *******************
  Need_Comma_Get_Byte
; *******************

   JSR Need_Comma
   JMP Get_Byte_Value

; **************
  FAC1_To_LINNUM
; **************

   LDA FAC1SI        ; get FAC1 sign
   BMI Jump_To_Illegal_Quantity
   LDA FAC1EX        ; get FAC1 exponent
   CMP #$91          ; compare with exponent = 2^16
   BCS Jump_To_Illegal_Quantity
   JSR FAC1_To_Integer
   LDA FAC1M3        ; get FAC1 mantissa 3
   LDY FAC1M4        ; get FAC1 mantissa 4
   STY LINNUM        ; save temporary integer low byte
   STA LINNUM+1      ; save temporary integer high byte
   RTS

; **********
  Basic_PEEK
; **********

   PUSHW(LINNUM)
   JSR FAC1_To_LINNUM
   LDY #0
   LDA (LINNUM),Y    ; read byte
   TAY               ; copy byte to Y
   PULLW(LINNUM)
   JMP Y_To_Float

; **********
  Basic_POKE
; **********

   JSR Get_Word_And_Byte
   TXA               ; copy byte to A
   LDY #0
   STA (LINNUM),Y    ; write byte
   RTS

; **********
  Basic_WAIT
; **********

   JSR Get_Word_And_Byte
   STX FORPNT        ; save byte
   LDX #0            ; clear mask
   JSR CHRGOT
   BEQ WAIT_10      ; skip if no third argument
   JSR Need_Comma_Get_Byte

WAIT_10
   STX FORPNT+1      ; save EOR argument
   LDY #0

WAIT_20
   LDA (LINNUM),Y    ; get byte via temporary integer (address)
   EOR FORPNT+1      ; EOR with second argument       (mask)
   AND FORPNT        ; AND with first argument        (byte)
   BEQ WAIT_20       ; loop if result is zero

WAIT_Ret
   RTS

; ***************
  Add_0_5_To_FAC1
; ***************

   LAYI(Float_0_5)
   JMP Add_Var_AY_To_FAC1

; *************
  AY_Minus_FAC1
; *************

   JSR Load_FAC2_From_AY

; ***********
  Basic_MINUS
; ***********

   LDA FAC1SI        ; get FAC1 sign (b7)
   EOR #$FF          ; complement it
   STA FAC1SI        ; save FAC1 sign (b7)
   EOR FAC2SI        ; EOR with FAC2 sign (b7)
   STA STRPTR        ; save sign compare (FAC1 EOR FAC2)
   LDA FAC1EX        ; get FAC1 exponent
   JMP Basic_PLUS    ; add FAC2 to FAC1 and return

PLUS_00
   JSR Shift_FACX_A
   BCC PLUS_20      ; go subtract the mantissas, branch always

; ******************
  Add_Var_AY_To_FAC1
; ******************
```

```
   JSR Load_FAC2_From_AY

; **********
  Basic_PLUS
; **********

   BNE PLUS_05      ; if FAC1 is not zero continue
   JMP FAC2_To_FAC1

PLUS_05
   LDX FAC1M5        ; get FAC1 rounding byte
   STX FAC2M5        ; put FAC2 rounding byte
   LDX #FAC2EX       ; set index to FAC2
   LDA FAC2EX        ; get FAC2 exponent

; ****************
  Add_FAC2_To_FAC1
; ****************

   TAY               ; copy exponent
   BEQ WAIT_Ret      ; exit if FAC2 is zero
   SEC
   SBC FAC1EX        ; FAC2 exponent - FAC1 exponent
   BEQ PLUS_20       ; if equal go add mantissas
   BCC PLUS_10       ; if FAC2 < FAC1 then shift FAC2 right
   STY FAC1EX        ; else                shift FAC1 right
   LDY FAC2SI        ; get FAC2 sign (b7)
   STY FAC1SI        ; put FAC1 sign (b7)
   EOR #$FF          ; complement A
   ADC #$00          ; +1, twos complement, carry is set
   LDY #0
   STY FAC2M5        ; clear FAC2 rounding byte
   LDX #FAC1EX       ; set index to FAC1
   BNE PLUS_15       ; branch always

PLUS_10
   LDY #0
   STY FAC1M5        ; clear FAC1 rounding byte

PLUS_15              ; shift FAC with lower exponent
   CMP #$F9          ; compare exponent diff with $F9
   BMI PLUS_00       ; branch if range $79-$F8
   TAY               ; copy exponent difference to Y
   LDA FAC1M5        ; get FAC1 rounding byte
   LSR 1,X           ; shift FAC mantissa 1
   JSR Shift_FACX_Right_Y

PLUS_20
   BIT STRPTR        ; test sign compare (FAC1 EOR FAC2)
   BPL PLUS_50      ; if = add FAC2 mantissa to FAC1 mantissa and return
   LDY #FAC1EX       ; set index to FAC1 exponent address
   CPX #FAC2EX       ; compare X to FAC2 exponent address
   BEQ PLUS_25       ; branch if equal
   LDY #FAC2EX       ; else set index to FAC2 exponent address

PLUS_25
   SEC               ; compute FACY - FACX
   EOR #$FF          ; ones complement A
   ADC FAC2M5        ; add FAC2 rounding byte
   STA FAC1M5        ; put FAC1 rounding byte
   LDA 4,Y
   SBC 4,X
   STA FAC1M4
   LDA 3,Y
   SBC 3,X
   STA FAC1M3
   LDA 2,Y
   SBC 2,X
   STA FAC1M2
   LDA 1,Y
   SBC 1,X
   STA FAC1M1

PLUS_30
   BCS Normalise_FAC1
   JSR Negate_FAC1

; **************
  Normalise_FAC1
; **************

   LDY #0
   TYA
   CLC

PLUS_35
   LDX FAC1M1        ; get FAC1 mantissa 1
   BNE PLUS_60      ; if not zero normalise FAC1
   LDX FAC1M2        ; get FAC1 mantissa 2
   STX FAC1M1        ; save FAC1 mantissa 1
   LDX FAC1M3        ; get FAC1 mantissa 3
   STX FAC1M2        ; save FAC1 mantissa 2
   LDX FAC1M4        ; get FAC1 mantissa 4
   STX FAC1M3        ; save FAC1 mantissa 3
   LDX FAC1M5        ; get FAC1 rounding byte
   STX FAC1M4        ; save FAC1 mantissa 4
   STY FAC1M5        ; clear FAC1 rounding byte
   ADC #8            ; add x to exponent offset
   CMP #$20          ; compare with $20, max offset, all bits would be = 0
   BNE PLUS_35       ; loop if not max

; =======================
  Clear_FAC1_Exp_And_Sign
; =======================

   LDA #0
PLUS_40
   STA FAC1EX        ; set FAC1 exponent

PLUS_45
   STA FAC1SI        ; save FAC1 sign (b7)
   RTS

PLUS_50
   ADC FAC2M5
   STA FAC1M5
   LDA FAC1M4
   ADC FAC2M4
   STA FAC1M4
   LDA FAC1M3
   ADC FAC2M3
   STA FAC1M3
   LDA FAC1M2
   ADC FAC2M2
   STA FAC1M2
   LDA FAC1M1
   ADC FAC2M1
   STA FAC1M1
   JMP Test_And_Normalize_FAC1

PLUS_55
   ADC #1
   ASL FAC1M5
   ROL FAC1M4
   ROL FAC1M3
   ROL FAC1M2
   ROL FAC1M1

PLUS_60
   BPL PLUS_55      ; loop if not normalised
   SEC
   SBC FAC1EX        ; subtract FAC1 exponent
   BCS Clear_FAC1_Exp_And_Sign ; branch if underflow (set result = $0)
   EOR #$FF          ; complement exponent
   ADC #$01          ; +1 (twos complement)
   STA FAC1EX        ; save FAC1 exponent

; =======================
  Test_And_Normalize_FAC1
; =======================

   BCC TANF_Ret      ; exit if no overflow

TANF_10
   INC FAC1EX        ; increment FAC1 exponent
   BEQ Overflow_Error
   ROR FAC1M1        ; shift FAC1 mantissa 1
   ROR FAC1M2        ; shift FAC1 mantissa 2
   ROR FAC1M3        ; shift FAC1 mantissa 3
   ROR FAC1M4        ; shift FAC1 mantissa 4
   ROR FAC1M5        ; shift FAC1 rounding byte

TANF_Ret
   RTS

; ***********
  Negate_FAC1
; ***********

   LDA FAC1SI        ; get FAC1 sign (b7)
   EOR #$FF          ; complement it
   STA FAC1SI        ; save FAC1 sign (b7)

; ********************
  Negate_FAC1_Mantissa
; ********************

   LDA FAC1M1
   EOR #$FF
   STA FAC1M1
   LDA FAC1M2
   EOR #$FF
   STA FAC1M2
   LDA FAC1M3
   EOR #$FF
   STA FAC1M3
   LDA FAC1M4
   EOR #$FF
   STA FAC1M4
   LDA FAC1M5
   EOR #$FF
   STA FAC1M5
   INC FAC1M5
   BNE IFM_Ret

; *****************
  Inc_FAC1_Mantissa
; *****************

   INC FAC1M4
   BNE IFM_Ret
   INC FAC1M3
   BNE IFM_Ret
   INC FAC1M2
   BNE IFM_Ret
   INC FAC1M1

IFM_Ret
   RTS

; ==============
  Overflow_Error
; ==============

   LDX #$0F          ; error $0F, overflow error
   JMP Basic_Error

; ==========
  Shift_FAC3
; ==========

   LDX #FAC3         ; apply shift routines on FAC3

; ==========
  Shift_FACX
; ==========

   LDY 4,X
   STY FAC1M5      ; mantissa 4 -> rounding byte
   LDY 3,X
   STY 4,X           ; mantissa 3 -> 4
   LDY 2,X
   STY 3,X           ; mantissa 2 -> 3
   LDY 1,X
   STY 2,X           ; mantissa 1 -> 2
   LDY FAC1OV
   STY 1,X           ; overflow -> mantissa 1

; ************
  Shift_FACX_A
; ************

   ADC #8            ; add 8 to shift count
   BMI Shift_FACX    ; if still negative shift byte wise
   BEQ Shift_FACX    ; 8 shifts to do
   SBC #8            ; reverse the addition
   TAY               ; save shift count to Y
   LDA FAC1M5      ; get FAC1 rounding byte
   BCS ShFA_30

ShFA_10
   ASL 1,X           ; shift sign to carry, bit0 set to 0
   BCC ShFA_20       ; branch if positive
   INC 1,X           ; bit0 set to 1

ShFA_20              ; bit0 equals now sign (in carry)
   ROR 1,X           ; shift FACX mantissa 1 with sign extension
   ROR 1,X           ; shift FACX mantissa 1 with sign extension

; ******************
  Shift_FACX_Right_Y
; ******************

   ROR 2,X           ; shift FACX mantissa 2
   ROR 3,X           ; shift FACX mantissa 3
   ROR 4,X           ; shift FACX mantissa 4
   ROR A             ; shift FACX rounding byte
   INY               ; increment exponent diff
   BNE ShFA_10       ; branch if range adjust not complete

ShFA_30
   CLC               ; just clear it
   RTS

; constants and series for LOG(n)

REAL_1 .real 1

VLOG_A
   .byte   $03            ; series counter
   .real   0.4342559419
   .real   0.5765845413
   .real   0.9618007592
   .real   2.8853900731

HALF_SQRT_2 .real   0.7071067812  ; 0.5 * sqrt(2.0)
SQRT_2      .real   1.4142135624  ; sqrt(2.0)
MINUS_0_5   .real  -0.5
LN_2        .real   0.6931471807  ; ln(2.0)

; *********
  Basic_LOG
; *********

   JSR Get_FAC1_Sign
   BEQ LOG_10        ; if zero do illegal quantity
   BPL LOG_20        ; skip error if positive

LOG_10
   JMP Illegal_Quantity      ; do illegal quantity error then warm start

LOG_20
   LDA FAC1EX        ; get FAC1 exponent
   SBC #$7F          ; normalise it
   PHA               ; save it
   LDA #$80          ; set exponent to zero
   STA FAC1EX        ; save FAC1 exponent
   LAYI(HALF_SQRT_2)
   JSR Add_Var_AY_To_FAC1
   LAYI(SQRT_2)
   JSR AY_Divided_By_FAC1
   LAYI(REAL_1)
   JSR AY_Minus_FAC1
   LAYI(VLOG_A)
   JSR Square_And_Series_Eval
   LAYI(MINUS_0_5)
   JSR Add_Var_AY_To_FAC1
   PLA               ; restore FAC1 exponent
   JSR Add_A_To_FAC1
   LAYI(LN_2)

; *********************
  Multiply_FAC1_With_AY
; *********************

   JSR Load_FAC2_From_AY

; **************
  Basic_MULTIPLY
; **************

   BNE MULT_10       ; multiply FAC1 by FAC2 ??
   JMP Mult_Sub_Ret  ; exit if zero

MULT_10
   JSR Check_FACs
   LDA #0
   STA FAC3+1
   STA FAC3+2
   STA FAC3+3
   STA FAC3+4
   LDA FAC1M5
   JSR Mult_SubA
   LDA FAC1M4
   JSR Mult_SubA
   LDA FAC1M3
   JSR Mult_SubA
   LDA FAC1M2
   JSR Mult_SubA
   LDA FAC1M1
   JSR Mult_SubB
   JMP FAC3_To_FAC1

; =========
  Mult_SubA
; =========

   BNE Mult_SubB
   JMP Shift_FAC3

; =========
  Mult_SubB
; =========

   LSR A             ; shift byte
   ORA #$80          ; set top bit (mark for 8 times)

MULT_20
   TAY               ; copy result
   BCC MULT_30       ; skip next if bit was zero
   CLC
   LDA FAC3+4
   ADC FAC2M4
   STA FAC3+4
   LDA FAC3+3
   ADC FAC2M3
   STA FAC3+3
   LDA FAC3+2
   ADC FAC2M2
   STA FAC3+2
   LDA FAC3+1
   ADC FAC2M1
   STA FAC3+1

MULT_30
   ROR FAC3+1
   ROR FAC3+2
   ROR FAC3+3
   ROR FAC3+4
   ROR FAC1M5
   TYA                ; get byte back
   LSR A              ; shift byte
   BNE MULT_20        ; loop if all bits not done

Mult_Sub_Ret
   RTS

; *****************
  Load_FAC2_From_AY
; *****************

   STAY(INDEXA)
   LDY #4             ; 5 bytes to get (0-4)
   LDA (INDEXA),Y     ; get mantissa 4
   STA FAC2M4         ; save FAC2 mantissa 4
   DEY                ; decrement index
   LDA (INDEXA),Y     ; get mantissa 3
   STA FAC2M3         ; save FAC2 mantissa 3
   DEY                ; decrement index
   LDA (INDEXA),Y     ; get mantissa 2
   STA FAC2M2         ; save FAC2 mantissa 2
   DEY                ; decrement index
   LDA (INDEXA),Y     ; get mantissa 1 + sign
   STA FAC2SI         ; save FAC2 sign (b7)
   EOR FAC1SI         ; EOR with FAC1 sign (b7)
   STA STRPTR         ; save sign compare (FAC1 EOR FAC2)
   LDA FAC2SI         ; recover FAC2 sign (b7)
   ORA #$80           ; set 1xxx xxx (set normal bit)
   STA FAC2M1         ; save FAC2 mantissa 1
   DEY                ; decrement index
   LDA (INDEXA),Y     ; get exponent byte
   STA FAC2EX         ; save FAC2 exponent
   LDA FAC1EX         ; get FAC1 exponent
   RTS

; **********
  Check_FACs
; **********

   LDA FAC2EX         ; get FAC2 exponent

; ------------
  Check_FACs_A
; ------------
```

```
   BEQ ChFA_30       ; branch if FAC2 = $00 (handle underflow)
   CLC
   ADC FAC1EX        ; add FAC1 exponent
   BCC ChFA_10       ; branch if sum of exponents < $0100
   BMI ChFA_40       ; do overflow error
   CLC
   .byte   $2C       ; skip next statement

ChFA_10
   BPL ChFA_30       ; if positive go handle underflow
   ADC #$80          ; adjust exponent
   STA FAC1EX        ; save FAC1 exponent
   BNE ChFA_20       ; branch if not zero
   JMP PLUS_45       ; save FAC1 sign and return

ChFA_20
   LDA STRPTR        ; get sign compare (FAC1 EOR FAC2)
   STA FAC1SI        ; save FAC1 sign (b7)
   RTS

; **************
  Check_Overflow
; **************

   LDA FAC1SI        ; get FAC1 sign (b7)
   EOR #$FF          ; complement it
   BMI ChFA_40       ; do overflow error

ChFA_30
   PLA               ; pop return address low byte
   PLA               ; pop return address high byte
   JMP Clear_FAC1_Exp_And_Sign

ChFA_40
   JMP Overflow_Error

; ******************
  Multiply_FAC1_BY_10
; ******************

   JSR FAC1_Round_And_Copy_To_FAC2
   TAX               ; copy exponent (set the flags)
   BEQ Mu10_Ret      ; exit if zero
   CLC
   ADC #$02          ; add two to exponent (*4)
   BCS ChFA_40       ; do overflow error if > $FF

; ------------------
  Multiply_FAC1_By_4
; ------------------

   LDX #0
   STX STRPTR        ; clear sign compare (FAC1 EOR FAC2)
   JSR Add_FAC2_To_FAC1
   INC FAC1EX        ; increment FAC1 exponent (*2)
   BEQ ChFA_40       ; if exponent now zero go do overflow error

Mu10_Ret
   RTS

Float_10 .real 10
```

```
; *****************
  Divide_FAC1_By_10
; *****************

   JSR FAC1_Round_And_Copy_To_FAC2
   LAYI(Float_10)
   LDX #0            ; clear sign

; *****************
  Divide_FAC2_By_AY
; *****************

   STX STRPTR        ; save sign compare (FAC1 EOR FAC2)
   JSR Load_FAC1_AY
   JMP Basic_DIVIDE  ; do FAC2/FAC1

; ******************
  AY_Divided_By_FAC1
; ******************

   JSR Load_FAC2_From_AY

; ************
  Basic_DIVIDE
; ************

   BEQ Divide_By_Zero; if zero go do /0 error
   JSR Round_FAC1_Checked
   LDA #0
   SEC
   SBC FAC1EX        ; subtract FAC1 exponent (2s complement)
   STA FAC1EX        ; save FAC1 exponent
   JSR Check_FACs
   INC FAC1EX        ; increment FAC1 exponent
   BEQ ChFA_40       ; if zero do overflow error
   LDX #$FC          ; set index to FAC temp
   LDA #$01          ; set byte

DIVI_10
   LDY FAC2M1        ; compare mantissa
   CPY FAC1M1
   BNE DIVI_20
   LDY FAC2M2
   CPY FAC1M2
   BNE DIVI_20
   LDY FAC2M3
   CPY FAC1M3
   BNE DIVI_20
   LDY FAC2M4
   CPY FAC1M4

DIVI_20
   PHP               ; save the FAC2-FAC1 compare status
   ROL A             ; shift byte
   BCC DIVI_30       ; skip next if no carry
   INX               ; increment index to FAC temp
   STA FAC3+4,X
   BEQ DIVI_60
   BPL DIVI_70
   LDA #1
```

```
DIVI_30
   PLP                  ; restore FAC2-FAC1 compare status
   BCS DIVI_50          ; if FAC2 >= FAC1 then do subtract

DIVI_40
   ASL FAC2M4           ; shift FAC2 mantissa 4
   ROL FAC2M3           ; shift FAC2 mantissa 3
   ROL FAC2M2           ; shift FAC2 mantissa 2
   ROL FAC2M1           ; shift FAC2 mantissa 1
   BCS DIVI_20          ; loop with no compare
   BMI DIVI_10          ; loop with compare
   BPL DIVI_20          ; loop always with no compare

DIVI_50
   TAY                  ; save FAC2-FAC1 compare status
   LDA FAC2M4           ; FAC2 = FAC2 - FAC1
   SBC FAC1M4
   STA FAC2M4
   LDA FAC2M3
   SBC FAC1M3
   STA FAC2M3
   LDA FAC2M2
   SBC FAC1M2
   STA FAC2M2
   LDA FAC2M1
   SBC FAC1M1
   STA FAC2M1
   TYA                  ; restore FAC2-FAC1 compare status
   JMP DIVI_40          ; go shift FAC2

DIVI_60
   LDA #$40
   BNE DIVI_30

DIVI_70
   ASL A
   ASL A
   ASL A
   ASL A
   ASL A
   ASL A
   STA FAC1M5           ; save FAC1 rounding byte
   PLP                  ; dump FAC2-FAC1 compare status
   JMP FAC3_To_FAC1

; ==============
  Divide_By_Zero
; ==============

   LDX #$14             ; error $14, divide by zero error
   JMP Basic_Error

; ============
  FAC3_To_FAC1
; ============

   LDA FAC3+1
   STA FAC1M1
   LDA FAC3+2
   STA FAC1M2
   LDA FAC3+3
   STA FAC1M3
```

```
    LDA FAC3+4
    STA FAC1M4
    JMP Normalise_FAC1

; ************
  Load_FAC1_AY
; ************

    STAY(INDEXA)
    LDY #$04          ; 5 bytes to do
    LDA (INDEXA),Y    ; get fifth byte
    STA FAC1M4        ; save FAC1 mantissa 4
    DEY               ; decrement index
    LDA (INDEXA),Y    ; get fourth byte
    STA FAC1M3        ; save FAC1 mantissa 3
    DEY               ; decrement index
    LDA (INDEXA),Y    ; get third byte
    STA FAC1M2        ; save FAC1 mantissa 2
    DEY               ; decrement index
    LDA (INDEXA),Y    ; get second byte
    STA FAC1SI        ; save FAC1 sign (b7)
    ORA #$80          ; set 1xxx xxxx (add normal bit)
    STA FAC1M1        ; save FAC1 mantissa 1
    DEY               ; decrement index
    LDA (INDEXA),Y    ; get first byte (exponent)
    STA FAC1EX        ; save FAC1 exponent
    STY FAC1M5        ; clear FAC1 rounding byte
    RTS

; **************
  FAC1_To_FACTPB
; **************

    LDX #<FACTPB      ; set pointer low byte
    .byte   $2C       ; skip next statement

; **************
  FAC1_To_FACTPA
; **************

    LDX #<FACTPA      ; set pointer low byte
    LDY #>FACTPA      ; set pointer high byte
    BEQ Assign_FAC1_To_Var

; ************************
  Assign_FAC1_To_FOR_Index
; ************************

    LDXY(FORPNT)

; ******************
  Assign_FAC1_To_Var
; ******************

    JSR Round_FAC1_Checked
    STXY(INDEXA)
    LDY #$04          ; set index
    LDA FAC1M4        ; get FAC1 mantissa 4
    STA (INDEXA),Y    ; store in destination
    DEY               ; decrement index
    LDA FAC1M3        ; get FAC1 mantissa 3
    STA (INDEXA),Y    ; store in destination
```

```
   DEY               ; decrement index
   LDA FAC1M2        ; get FAC1 mantissa 2
   STA (INDEXA),Y    ; store in destination
   DEY               ; decrement index
   LDA FAC1SI        ; get FAC1 sign (b7)
   ORA #$7F          ; set bits x111 1111
   AND FAC1M1        ; AND in FAC1 mantissa 1
   STA (INDEXA),Y    ; store in destination
   DEY               ; decrement index
   LDA FAC1EX        ; get FAC1 exponent
   STA (INDEXA),Y    ; store in destination
   STY FAC1M5        ; clear FAC1 rounding byte
   RTS

; ************
  FAC2_To_FAC1
; ************

   LDA FAC2SI        ; get FAC2 sign (b7)

; ---------------------
  Copy_ABS_FAC2_To_FAC1
; ---------------------

   STA FAC1SI        ; save FAC1 sign (b7)
   LDX #5            ; 5 bytes to copy

F2F1_Loop
   LDA FAC2EX-1,X
   STA FAC1EX-1,X
   DEX
   BNE F2F1_Loop
   STX FAC1M5        ; clear FAC1 rounding byte
   RTS

; ***************************
  FAC1_Round_And_Copy_To_FAC2
; ***************************

   JSR Round_FAC1_Checked

; ************
  FAC1_To_FAC2
; ************

   LDX #6

F1F2_Loop
   LDA FAC1EX-1,X
   STA FAC2EX-1,X
   DEX
   BNE F1F2_Loop
   STX FAC1M5        ; clear FAC1 rounding byte

F1F2_Ret
   RTS

; ******************
  Round_FAC1_Checked
; ******************

   LDA FAC1EX        ; get FAC1 exponent
   BEQ F1F2_Ret      ; exit if zero
   ASL FAC1M5        ; shift FAC1 rounding byte
   BCC F1F2_Ret      ; exit if no overflow

; ----------
  Round_FAC1
; ----------

   JSR Inc_FAC1_Mantissa
   BNE F1F2_Ret      ; branch if no overflow
   JMP TANF_10       ; normalise FAC1 for C=1 and return

; *************
  Get_FAC1_Sign
; *************

   LDA FAC1EX        ; get FAC1 exponent
   BEQ GFS_Ret       ; exit if zero

GFS_10
   LDA FAC1SI        ; else get FAC1 sign (b7)

GFS_20
   ROL A             ; move sign bit to carry
   LDA #$FF          ; set byte for negative result
   BCS GFS_Ret       ; return if sign was set (negative)

   LDA #1            ; else set byte for positive result
GFS_Ret
   RTS

; *********
  Basic_SGN
; *********

   JSR Get_FAC1_Sign

; *********
  A_To_FAC1
; *********

   STA FAC1M1        ; save FAC1 mantissa 1
   LDA #0
   STA FAC1M2        ; clear FAC1 mantissa 2
   LDX #$88          ; set exponent

; ==================
  Int_To_Float_Exp_X
; ==================

   LDA FAC1M1        ; get FAC1 mantissa 1
   EOR #$FF          ; complement it
   ROL A             ; sign bit into carry

; ****************
  Convert_Integer_To_Float
; ****************

   LDA #0
   STA FAC1M4        ; clear FAC1 mantissa 4
   STA FAC1M3        ; clear FAC1 mantissa 3
```

```
; set exponent = X and normalise FAC1

CITF_10
   STX FAC1EX        ; set FAC1 exponent
   STA FAC1M5        ; clear FAC1 rounding byte
   STA FAC1SI        ; clear FAC1 sign (b7)
   JMP PLUS_30       ; do ABS and normalise FAC1

; *********
  Basic_ABS
; *********

   LSR FAC1SI        ; clear FAC1 sign, put zero in b7
   RTS

; ***************
  Compare_FAC1_AY
; ***************

   STA INDEXB        ; save pointer low byte

; returns A =  0 if FAC1 = (AY)
; returns A =  1 if FAC1 > (AY)
; returns A = -1 if FAC1 < (AY)

; ---------------------
  Compare_FAC1_INDEXB_Y
; ---------------------

   STY INDEXB+1      ; save pointer high byte
   LDY #0
   LDA (INDEXB),Y    ; get exponent
   INY
   TAX               ; copy (AY) exponent to X
   BEQ Get_FAC1_Sign
   LDA (INDEXB),Y    ; get (AY) mantissa 1, with sign
   EOR FAC1SI        ; EOR FAC1 sign (b7)
   BMI GFS_10        ; if signs <> do return A = $FF, Cb = 1/negative
   CPX FAC1EX        ; compare (AY) exponent with FAC1 exponent
   BNE CPFA_10       ; branch if different
   LDA (INDEXB),Y    ; get (AY) mantissa 1, with sign
   ORA #$80          ; normalise top bit
   CMP FAC1M1        ; compare with FAC1 mantissa 1
   BNE CPFA_10       ; branch if different
   INY
   LDA (INDEXB),Y    ; get mantissa 2
   CMP FAC1M2        ; compare with FAC1 mantissa 2
   BNE CPFA_10       ; branch if different
   INY
   LDA (INDEXB),Y    ; get mantissa 3
   CMP FAC1M3        ; compare with FAC1 mantissa 3
   BNE CPFA_10       ; branch if different
   INY
   LDA #$7F          ; set for 1/2 value rounding byte
   CMP FAC1M5        ; compare with FAC1 rounding byte (set carry)
   LDA (INDEXB),Y    ; get mantissa 4
   SBC FAC1M4        ; subtract FAC1 mantissa 4
   BEQ FATI_20       ; exit if mantissa 4 equal

; gets here if number <> FAC1

CPFA_10
```

```
   LDA FAC1SI        ; get FAC1 sign (b7)
   BCC CPFA_20       ; branch if FAC1 > (AY)
   EOR #$FF          ; else toggle FAC1 sign

CPFA_20
   JMP GFS_20        ; return A = $FF, Cb = 1/negative A = $01, Cb = 0/positive

; ***************
  FAC1_To_Integer
; ***************

   LDA FAC1EX
   BEQ Clear_FAC1
   SEC
   SBC #$A0          ; subtract maximum integer range exponent
   BIT FAC1SI        ; test FAC1 sign (b7)
   BPL FATI_10       ; branch if FAC1 positive
   TAX               ; copy subtracted exponent
   LDA #$FF          ; overflow for negative number
   STA FAC1OV        ; set FAC1 overflow byte
   JSR Negate_FAC1_Mantissa
   TXA               ; restore subtracted exponent

FATI_10
   LDX #FAC1EX
   CMP #$F9          ; compare exponent result
   BPL FATI_30       ; less than 8 shifts
   JSR Shift_FACX_A
   STY FAC1OV        ; clear FAC1 overflow byte

FATI_20
   RTS

FATI_30
   TAY               ; copy shift count
   LDA FAC1SI        ; get FAC1 sign (b7)
   AND #$80          ; mask sign bit only (x000 0000)
   LSR FAC1M1        ; shift FAC1 mantissa 1
   ORA FAC1M1        ; OR sign in b7 FAC1 mantissa 1
   STA FAC1M1        ; save FAC1 mantissa 1
   JSR Shift_FACX_Right_Y
   STY FAC1OV        ; clear FAC1 overflow byte
   RTS

; *********
  Basic_INT
; *********

   LDA FAC1EX        ; get FAC1 exponent
   CMP #$A0          ; compare with max int
   BCS ClF1_Ret      ; exit if >= (allready int, too big for fractional part!)
   JSR FAC1_To_Integer
   STY FAC1M5        ; save FAC1 rounding byte
   LDA FAC1SI        ; get FAC1 sign (b7)
   STY FAC1SI        ; save FAC1 sign (b7)
   EOR #$80          ; toggle FAC1 sign
   ROL A             ; shift into carry
   LDA #$A0          ; set new exponent
   STA FAC1EX        ; save FAC1 exponent
   LDA FAC1M4        ; get FAC1 mantissa 4
   STA CHARAC        ; save FAC1 mantissa 4 for power function
   JMP PLUS_30      ; do ABS and normalise FAC1
```

```
; ==========
  Clear_FAC1
; ==========

   STA FAC1M1
   STA FAC1M2
   STA FAC1M3
   STA FAC1M4
   TAY

ClF1_Ret
   RTS

; ********************
  Load_FAC1_From_String
; ********************

   LDY #0
   LDX #10

LFFS_05
   STY TMPVA1,X      ; clear 10 bytes TMPVA1 & FAC1 ($5D - $66)
   DEX
   BPL LFFS_05
   BCC LFFS_20       ; branch if first character is numeric
   CMP #'-'          ; else compare with "-"
   BNE LFFS_10       ; branch if not "-"
   STX SGNFLG        ; set flag for negative n (X = $FF)
   BEQ LFFS_15       ; branch always

LFFS_10
   CMP #'+'          ; else compare with "+"
   BNE LFFS_25       ; branch if not "+"

LFFS_15
   JSR CHRGET        ; next char

LFFS_20
   BCC LFFS_75       ; branch if numeric character

LFFS_25
   CMP #'.'
   BEQ LFFS_40
   CMP #'E'
   BNE LFFS_45
   JSR CHRGET        ; read exponent
   BCC LFFS_37       ; branch if numeric character
   CMP #TK_MINUS
   BEQ LFFS_30
   CMP #'-'
   BEQ LFFS_30
   CMP #TK_PLUS
   BEQ LFFS_35
   CMP #'+'
   BEQ LFFS_35
   BNE LFFS_38       ; branch always

LFFS_30
   ROR TMPPTC+1      ; set exponent negative flag (ror carry into sign)

LFFS_35
```

```
   JSR CHRGET        ; next char of exponent

LFFS_37
   BCC LFFS_85       ; branch if numeric character

LFFS_38
   BIT TMPPTC+1      ; test exponent negative flag
   BPL LFFS_45       ; if positive go evaluate exponent
   LDA #0
   SEC
   SBC TMPVA2        ; negate exponent
   JMP LFFS_50       ; go evaluate exponent

LFFS_40
   ROR TMPPTC        ; set decimal point flag
   BIT TMPPTC        ; test decimal point flag
   BVC LFFS_15       ; branch if only one decimal point so far

LFFS_45
   LDA TMPVA2        ; get exponent count byte

LFFS_50
   SEC
   SBC TMPVA1        ; subtract numerator exponent
   STA TMPVA2        ; save exponent count byte
   BEQ LFFS_65       ; branch if no adjustment
   BPL LFFS_60       ; else if positive go do FAC1*10^expcnt

LFFS_55
   JSR Divide_FAC1_By_10
   INC TMPVA2        ; increment exponent count byte
   BNE LFFS_55       ; loop until all done
   BEQ LFFS_65       ; branch always

LFFS_60
   JSR Multiply_FAC1_BY_10
   DEC TMPVA2        ; decrement exponent count byte
   BNE LFFS_60       ; loop until all done

LFFS_65
   LDA SGNFLG        ; get negative flag
   BMI LFFS_70       ; if negative do - FAC1 and return
   RTS

LFFS_70
   JMP Basic_GREATER ; do - FAC1

LFFS_75
   PHA               ; save character
   BIT TMPPTC        ; test decimal point flag
   BPL LFFS_80       ; skip exponent increment if not set
   INC TMPVA1        ; else increment number exponent

LFFS_80
   JSR Multiply_FAC1_BY_10
   PLA               ; restore character
   SEC
   SBC #'0'          ; convert to binary
   JSR Add_A_To_FAC1
   JMP LFFS_15      ; go do next character

; *************
```

```
  Add_A_To_FAC1
; *************

   PHA               ; save digit
   JSR FAC1_Round_And_Copy_To_FAC2
   PLA               ; restore digit
   JSR A_To_FAC1
   LDA FAC2SI        ; get FAC2 sign (b7)
   EOR FAC1SI        ; toggle with FAC1 sign (b7)
   STA STRPTR        ; save sign compare (FAC1 EOR FAC2)
   LDX FAC1EX        ; get FAC1 exponent
   JMP Basic_PLUS    ; add FAC2 to FAC1 and return

; evaluate next character of exponential part of number

LFFS_85
   LDA TMPVA2        ; get exponent count byte
   CMP #10           ; compare with 10 decimal
   BCC LFFS_90       ; branch if less
   LDA #$64          ; make all negative exponents = -100 decimal (causes underflow)
   BIT TMPPTC+1      ; test exponent negative flag
   BMI LFFS_95       ; branch if negative
   JMP Overflow_Error

LFFS_90
   ASL A             ; *2
   ASL A             ; *4
   CLC
   ADC TMPVA2        ; *5
   ASL A             ; *10
   CLC
   LDY #0
   ADC (TXTPTR),Y    ; add character (will be $30 too much!)
   SEC
   SBC #'0'          ; convert character to binary

LFFS_95
   STA TMPVA2        ; save exponent count byte
   JMP LFFS_35       ; go get next character

MAXREAL_A .real   99999999.90625
MAXREAL_B .real  999999999.25
MAXREAL_C .real 1000000000

; ********
  Print_IN
; ********

   LAYI(Msg_IN)
   JSR To_Print_String
   LDA CURLIN+1      ; get the current line number high byte
   LDX CURLIN        ; get the current line number low byte

; ****************
  Print_Integer_XA
; ****************

   STA FAC1M1        ; save high byte as FAC1 mantissa1
   STX FAC1M2        ; save low byte as FAC1 mantissa2
   LDX #$90          ; set exponent to 16d bits
   SEC               ; set integer is positive flag
   JSR Convert_Integer_To_Float
   JSR Format_FAC1_Y

To_Print_String
   JMP Print_String

; ***********
  Format_FAC1
; ***********

   LDY #1
; -------------
  Format_FAC1_Y
; -------------

   LDA #' '          ; character = " " (assume positive)
   BIT FAC1SI        ; test FAC1 sign (b7)
   BPL FoFA_02       ; if positive skip the - sign set
   LDA #'-'          ; else character = "-"

FoFA_02
   STA BASSTO,Y      ; save leading character (" " or "-")
   STA FAC1SI        ; save FAC1 sign (b7)
   STY TMPPTD        ; save the index
   INY
   LDA #'0'          ; set character = "0"
   LDX FAC1EX        ; get FAC1 exponent
   BNE FoFA_04       ; if FAC1 not zero format it
   JMP FoFA_52       ; just 0

FoFA_04
   LDA #$00          ; clear (number exponent count)
   CPX #$80          ; compare FAC1 exponent with $80 (<1.00000)
   BEQ FoFA_06       ; branch if 0.5 <= FAC1 < 1.0
   BCS FoFA_08       ; branch if FAC1=>1

FoFA_06
   LAYI(MAXREAL_C)   ; set 1000000000 pointer
   JSR Multiply_FAC1_With_AY
   LDA #$F7          ; set number exponent count

FoFA_08
   STA TMPVA1        ; save number exponent count

FoFA_10
   LAYI(MAXREAL_B)   ; set 999999999.25 pointer (max before sci note)
   JSR Compare_FAC1_AY
   BEQ FoFA_20       ; exit if FAC1 = (AY)
   BPL FoFA_16       ; go do /10 if FAC1 > (AY)

FoFA_12
   LAYI(MAXREAL_A)   ; set 99999999.90625 pointer
   JSR Compare_FAC1_AY
   BEQ FoFA_14       ; branch if FAC1 = (AY) (allow decimal places)
   BPL FoFA_18       ; branch if FAC1 > (AY) (no decimal places)

FoFA_14
   JSR Multiply_FAC1_BY_10
   DEC TMPVA1        ; decrement number exponent count
   BNE FoFA_12       ; go test again, branch always

FoFA_16
   JSR Divide_FAC1_By_10
   INC TMPVA1        ; increment number exponent count
   BNE FoFA_10       ; go test again, branch always

FoFA_18
   JSR Add_0_5_To_FAC1

FoFA_20
   JSR FAC1_To_Integer
   LDX #$01          ; set default digits before dp = 1
   LDA TMPVA1        ; get number exponent count
   CLC
   ADC #$0A          ; up to 9 digits before point
   BMI FoFA_22       ; if negative then 1 digit before dp
   CMP #$0B          ; A>=$0B if n>=1E9
   BCS FoFA_24       ; branch if >= $0B
   ADC #$FF          ; take 1 from digit count
   TAX               ; copy to X
   LDA #$02          ; set the exponent adjust

FoFA_22
   SEC

FoFA_24
   SBC #$02          ; -2
   STA TMPVA2        ; save the exponent adjust
   STX TMPVA1        ; save digits before dp count
   TXA               ; copy digits before dp count to A
   BEQ FoFA_26       ; if no digits before the dp go do the "."
   BPL FoFA_30       ; if there are digits before the dp go do them

FoFA_26
   LDY TMPPTD        ; get the output string index
   LDA #'.'          ; character "."
   INY               ; increment the index
   STA STACK-1,Y     ; save the "." to the output string
   TXA               ; copy digits before dp count to A
   BEQ FoFA_28       ; if no digits before the dp skip the "0"
   LDA #'0'          ; character "0"
   INY
   STA STACK-1,Y     ; save the "0" to the output string

FoFA_28
   STY TMPPTD        ; save the output string index

FoFA_30
   LDY #0            ; clear the powers of 10 index (point to -100,000,000)

; -----------------
  Format_Jiffyclock
; -----------------

   LDX #$80          ; clear the digit, set the test sense
FoFA_32
   LDA FAC1M4        ; get FAC1 mantissa 4
   CLC
   ADC Decimal_Conversion_Table+3,Y
   STA FAC1M4        ; save FAC1 mantissa4
   LDA FAC1M3        ; get FAC1 mantissa 3
   ADC Decimal_Conversion_Table+2,Y
   STA FAC1M3        ; save FAC1 mantissa3
   LDA FAC1M2        ; get FAC1 mantissa 2
   ADC Decimal_Conversion_Table+1,Y
   STA FAC1M2        ; save FAC1 mantissa2
   LDA FAC1M1        ; get FAC1 mantissa 1
   ADC Decimal_Conversion_Table+0,Y
   STA FAC1M1        ; save FAC1 mantissa1
   INX               ; increment the digit, set the sign on the test sense bit
   BCS FoFA_34       ; if the carry is set go test if the result was positive
   BPL FoFA_32       ; not negative so try again
   BMI FoFA_36       ; else done so return the digit

FoFA_34
   BMI FoFA_32       ; not positive so try again

FoFA_36
   TXA               ; copy the digit
   BCC FoFA_38       ; if Cb=0 just use it
   EOR #$FF          ; else make the 2's complement ..
   ADC #$0A          ; .. and subtract it from 10

FoFA_38
   ADC #'0'-1        ; add "0"-1 to result
   INY               ; increment ..
   INY               ; .. index to..
   INY               ; .. next less ..
   INY               ; .. power of ten
   STY VARPNT        ; save the powers of ten table index
   LDY TMPPTD        ; get output string index
   INY               ; increment output string index
   TAX               ; copy character to X
   AND #$7F          ; mask out top bit
   STA STACK-1,Y     ; save to output string
   DEC TMPVA1        ; decrement # of characters before the dp
   BNE FoFA_40       ; if still characters to do skip the decimal point
   LDA #'.'          ; character "."
   INY               ; increment output string index
   STA STACK-1,Y     ; save to output string

FoFA_40
   STY TMPPTD        ; save the output string index
   LDY VARPNT        ; get the powers of ten table index
   TXA               ; get the character back
   EOR #$FF          ; toggle the test sense bit
   AND #$80          ; clear the digit
   TAX               ; copy it to the new digit
   CPY #Jiffy_Conversion_Table-Decimal_Conversion_Table
   BEQ FoFA_42       ; if at the max exit the digit loop
   CPY #End_Of_Conversion-Decimal_Conversion_Table
   BNE FoFA_32       ; loop if not at the max

FoFA_42
   LDY TMPPTD        ; restore the output string index
FoFA_44
   LDA STACK-1,Y     ; get character from output string
   DEY               ; decrement output string index
   CMP #'0'          ; compare with "0"
   BEQ FoFA_44       ; loop until non "0" character found
   CMP #'.'          ; compare with "."
   BEQ FoFA_46       ; branch if was dp
   INY               ; increment output string index

FoFA_46
   LDA #'+'          ; character "+"
   LDX TMPVA2        ; get exponent count
   BEQ FoFA_54       ; if zero go set null terminator and exit
   BPL FoFA_48       ; branch if exponent count positive
   LDA #0
   SEC
   SBC TMPVA2        ; subtract exponent count adjust (convert negative to positive)
   TAX               ; copy exponent count to X
   LDA #'-'          ; character "-"

FoFA_48
   STA STACK+1,Y     ; save to output string
   LDA #'E'          ; character "E"
   STA STACK,Y       ; save exponent sign to output string
   TXA               ; get exponent count back
   LDX #$2F          ; one less than "0" character
   SEC

FoFA_50
   INX               ; increment 10's character
   SBC #$0A          ; subtract 10 from exponent count
   BCS FoFA_50       ; loop while still >= 0
   ADC #':'          ; add character ":" ($30+$0A, result is 10 less that value)
   STA STACK+3,Y     ; save to output string
   TXA               ; copy 10's character
   STA STACK+2,Y     ; save to output string
   LDA #$00          ; set null terminator
   STA STACK+4,Y     ; save to output string
   BEQ FoFA_56       ; go set string pointer (AY) and exit, branch always

FoFA_52
   STA STACK-1,Y     ; save last character to output string

FoFA_54
   LDA #$00          ; set null terminator
   STA STACK,Y       ; save after last character

FoFA_56
   LAYI(STACK)
   RTS

Float_0_5       .byte $80,$00 ; 0.5 (including next 3 bytes)
NULL_Descriptor .byte 0,0,0 ; null descriptor for undefined variables

; ========================
  Decimal_Conversion_Table
; ========================

   .quad -100000000
   .quad  +10000000
   .quad   -1000000
   .quad    +100000
   .quad     -10000
   .quad      +1000
   .quad       -100
   .quad        +10
   .quad         -1

; ======================
  Jiffy_Conversion_Table
; ======================

   .quad -2160000 ; 10s hours
   .quad  +216000 ;     hours
   .quad   -36000 ; 10s mins
   .quad    +3600 ;     mins
   .quad     -600 ; 10s secs
   .quad      +60 ;     secs

End_Of_Conversion

#if C64
   .byte $EC
#endif
#if VIC
   .byte $BF
#endif

   .fill 30 ($AA)

; *********
  Basic_SQR
; *********

   JSR FAC1_Round_And_Copy_To_FAC2
   LAYI(Float_0_5)
   JSR Load_FAC1_AY

; ***********
  Basic_POWER
; ***********

   BEQ Basic_EXP     ; perform EXP()
   LDA FAC2EX        ; get FAC2 exponent
   BNE POW_10        ; branch if FAC2<>0
   JMP PLUS_40       ; clear FAC1 exponent and sign and return

POW_10
   LDX #<FUNCPT      ; set destination pointer low byte
   LDY #>FUNCPT      ; set destination pointer high byte
   JSR Assign_FAC1_To_Var
   LDA FAC2SI        ; get FAC2 sign (b7)
   BPL POW_20        ; branch if FAC2>0
   JSR Basic_INT     ; perform INT()
   LAYI(FUNCPT)      ; set source pointer
   JSR Compare_FAC1_AY
   BNE POW_20        ; branch if FAC1 <> (AY) to allow Function Call error
   TYA               ; clear sign b7
   LDY CHARAC        ; get FAC1 mantissa 4 from INT() function as sign in
                     ; Y for possible later negation, b0 only needed
POW_20
   JSR Copy_ABS_FAC2_To_FAC1
   TYA               ; copy sign back ..
   PHA               ; .. and save it
   JSR Basic_LOG     ; perform LOG()
   LAYI(FUNCPT)
   JSR Multiply_FAC1_With_AY
   JSR Basic_EXP     ; perform EXP()
   PLA               ; pull sign from stack
   LSR A             ; b0 is to be tested
   BCC GREA_Ret      ; if no bit then exit

; *************
  Basic_GREATER
; *************
```

```
   LDA FAC1EX        ; get FAC1 exponent
   BEQ GREA_Ret      ; exit if FAC1_e = $00
   LDA FAC1SI        ; get FAC1 sign (b7)
   EOR #$FF          ; complement it
   STA FAC1SI        ; save FAC1 sign (b7)

GREA_Ret
   RTS

REV_LOG_2 .real 1.4426950408

VAR_EXP
   .byte   7         ; series count

   .real  2.1498763705E-5
   .real  1.4352314041E-4
   .real  1.3422634825E-3
   .real  9.6140170140E-3
   .real  5.5505126870E-2
   .real  2.4022638465E-1
   .real  6.9314718640E-1
   .real  1.0

; *********
  Basic_EXP
; *********

   LAYI(REV_LOG_2)   ; point to 1.0/ln(2.0) = 1.443
   JSR Multiply_FAC1_With_AY
   LDA FAC1M5        ; get FAC1 rounding byte
   ADC #$50          ; +$50/$100
   BCC EXP_10        ; skip rounding if no carry
   JSR Round_FAC1

EXP_10

#if C64
   JMP C64_Kernal_ROM
   .org  $E000
C64_Kernal_ROM
#endif

#if C64
   .store  *,$2000,"kernal_64.rom"
#endif
#if VIC
   .store  $e000,$2000,"kernal_20.rom"
#endif

   STA FAC2M5        ; save FAC2 rounding byte
   JSR FAC1_To_FAC2
   LDA FAC1EX        ; get FAC1 exponent
   CMP #$88          ; compare with EXP limit (256d)
   BCC EXP_30        ; branch if less

EXP_20
   JSR Check_Overflow

EXP_30
   JSR Basic_INT     ; perform INT()
   LDA CHARAC        ; get mantissa 4 from INT()
   CLC
   ADC #$81          ; normalise +1
   BEQ EXP_20        ; if $00 result has overflowed so go handle it
   SEC
   SBC #$01          ; exponent now correct
   PHA               ; save FAC2 exponent
   LDX #$05          ; 4 bytes to do

EXP_40
   LDA FAC2EX,X      ; get FAC2,X
   LDY FAC1EX,X      ; get FAC1,X
   STA FAC1EX,X      ; save FAC1,X
   STY FAC2EX,X      ; save FAC2,X
   DEX               ; decrement count/index
   BPL EXP_40        ; loop if not all done
   LDA FAC2M5        ; get FAC2 rounding byte
   STA FAC1M5        ; save as FAC1 rounding byte
   JSR Basic_MINUS   ; perform subtraction, FAC2 from FAC1
   JSR Basic_GREATER ; do - FAC1
   LAYI(VAR_EXP)     ; set counter pointer
   JSR Eval_Series_AY
   LDA #0
   STA STRPTR        ; clear sign compare (FAC1 EOR FAC2)
   PLA               ; pull the saved FAC2 exponent
   JSR Check_FACs_A
   RTS

; *********************
  Square_And_Series_Eval
; *********************

   STAY(TMPPTD)
   JSR FAC1_To_FACTPA
   LDA #<FACTPA      ; set pointer low byte (Y already $00)
   JSR Multiply_FAC1_With_AY
   JSR Eval_Series
   LAYI(FACTPA)      ; pointer to original
   JMP Multiply_FAC1_With_AY

; **************
  Eval_Series_AY
; **************

   STAY(TMPPTD)

; -----------
  Eval_Series
; -----------

   JSR FAC1_To_FACTPB
   LDA (TMPPTD),Y    ; get constants count
   STA SGNFLG        ; save constants count
   LDY TMPPTD        ; get count pointer low byte
   INY               ; increment it (now constants pointer)
   TYA               ; copy it
   BNE EvSe_10       ; skip next if no overflow
   INC TMPPTD+1      ; else increment high byte

EvSe_10
   STA TMPPTD        ; save low byte
   LDY TMPPTD+1      ; get high byte

EvSe_20
```

```
   JSR Multiply_FAC1_With_AY
   LDAY(TMPPTD)
   CLC
   ADC #$05          ; +5 to low pointer (5 bytes per constant)
   BCC EvSe_30       ; skip next if no overflow
   INY               ; increment high byte

EvSe_30
   STAY(TMPPTD)
   JSR Add_Var_AY_To_FAC1
   LAYI(FACTPB)      ; set pointer to partial
   DEC SGNFLG        ; decrement constants count
   BNE EvSe_20       ; loop until all done
   RTS

RND_VA .real 11879546
RND_VB .real 3.927677739E-8

; *********
  Basic_RND
; *********

   JSR Get_FAC1_Sign
   BMI RND_20        ; if      n<0 copy byte swapped FAC1 into RND() seed
   BNE RND_10        ; else if n>0 get next number in RND() sequence
   JSR IOBASE        ; else    n=0 so get the RND() from VIA 1 timers
   STXY(INDEXA)      ; save pointer low byte
   LDY #$04          ; set index to T1 low byte
   LDA (INDEXA),Y    ; get T1 low byte
   STA FAC1M1        ; save FAC1 mantissa 1
   INY
   LDA (INDEXA),Y    ; get T1 high byte
   STA FAC1M3        ; save FAC1 mantissa 3
   LDY #$08          ; set index to T2 low byte
   LDA (INDEXA),Y    ; get T2 low byte
   STA FAC1M2        ; save FAC1 mantissa 2
   INY
   LDA (INDEXA),Y    ; get T2 high byte
   STA FAC1M4        ; save FAC1 mantissa 4
   JMP RND_30        ; set exponent and exit

RND_10
   LAYI(RNDX)        ; set seed pointer
   JSR Load_FAC1_AY
   LAYI(RND_VA)      ; set 11879546 pointer
   JSR Multiply_FAC1_With_AY
   LAYI(RND_VB)      ; set 3.927677739E-8 pointer
   JSR Add_Var_AY_To_FAC1

RND_20
   LDX FAC1M4        ; get FAC1 mantissa 4
   LDA FAC1M1        ; get FAC1 mantissa 1
   STA FAC1M4        ; save FAC1 mantissa 4
   STX FAC1M1        ; save FAC1 mantissa 1
   LDX FAC1M2        ; get FAC1 mantissa 2
   LDA FAC1M3        ; get FAC1 mantissa 3
   STA FAC1M2        ; save FAC1 mantissa 2
   STX FAC1M3        ; save FAC1 mantissa 3

RND_30
   LDA #0
   STA FAC1SI        ; clear FAC1 sign (always positive)
   LDA FAC1EX        ; get FAC1 exponent
   STA FAC1M5        ; save FAC1 rounding byte
   LDA #$80          ; set exponent = $80
   STA FAC1EX        ; save FAC1 exponent
   JSR Normalise_FAC1
   LDX #<RNDX        ; set seed pointer low address
   LDY #>RNDX        ; set seed pointer high address

; --------------------
  Go_Assign_FAC1_To_Var
; --------------------

   JMP Assign_FAC1_To_Var

; =============
  Error_Handler
; =============

   CMP #$F0          ; compare error with $F0
   BNE ErHa_10       ; branch if not $F0
   STY MEMSIZ+1      ; set end of memory high byte
   STX MEMSIZ        ; set end of memory low byte
   JMP Reset_Variable_Pointer

ErHa_10
   TAX               ; copy error #
   BNE ErHa_20       ; branch if not $00
   LDX #$1E          ; else error $1E, break error

ErHa_20
   JMP Basic_Error

; **************
  CHROUT_Checked
; **************

   JSR CHROUT        ; Output a character
   BCS Error_Handler
   RTS

; *********
  Read_Char
; *********

   JSR CHRIN
   BCS Error_Handler
   RTS

; *********************
  Select_Output_Channel
; *********************

#if C64
   JSR CHKOUT_Checked
#endif

#if VIC
   JSR CHKOUT        ; open channel for output
#endif

   BCS Error_Handler
   RTS
```

```
; *************
  CHKIN_Checked
; *************

   JSR CHKIN         ; open channel for input
   BCS Error_Handler
   RTS

; *************
  GETIN_Checked
; *************

   JSR GETIN         ; get character from input device
   BCS Error_Handler
   RTS

; *********
  Basic_SYS
; *********

   JSR Eval_Numeric
   JSR FAC1_To_LINNUM
   LDA #>[SYS_Ret-1] ; get return address high byte
   PHA               ; push as return address
   LDA #<[SYS_Ret-1] ; get return address low byte
   PHA               ; push as return address
   LDA SPREG         ; get saved status register
   PHA               ; put on stack
   LDA SAREG         ; get saved A
   LDX SXREG         ; get saved X
   LDY SYREG         ; get saved Y
   PLP               ; pull processor status
   JMP (LINNUM)      ; call SYS address

; -------
  SYS_Ret
; -------

; the SYS_Ret is needed because the following code is to be executed once the user code
; returns. this is done by pushing the target return address - 1 onto the stack

   PHP               ; save status
   STA SAREG         ; save returned A
   STX SXREG         ; save returned X
   STY SYREG         ; save returned Y
   PLA               ; restore saved status
   STA SPREG         ; save status
   RTS

; **********
  Basic_SAVE
; **********

   JSR Get_Load_Save_Params

Jiffy_SAVE
   LDXY(VARTAB)      ; get start of variables
   LDA #TXTTAB       ; index to start of program memory
   JSR SAVE          ; save RAM to device, A = index to start address, XY = end
   BCS Error_Handler
   RTS
```

```
; ************
  Basic_VERIFY
; ************

   LDA #1            ; flag verify
   .byte   $2C       ; skip next statement

; **********
  Basic_LOAD
; **********

   LDA #0            ; flag load
   STA VERCK         ; set load/verify flag
   JSR Get_Load_Save_Params
   LDA VERCK         ; get load/verify flag
   LDXY(TXTTAB)      ; get start of memory
   JSR LOAD          ; load RAM from a device
   BCS Jump_Error_Handler
   LDA VERCK         ; get load/verify flag
   BEQ LOAD_30       ; branch if load

LOAD_05
   LDX #$1C          ; error $1C, verify error
   JSR READST        ; read I/O status word
   AND #$10          ; mask for tape read error

#if C64
   BNE LOAD_40       ; branch if read error
#endif
#if VIC
   BEQ LOAD_10       ; branch if no read error
   JMP Basic_Error
#endif

LOAD_10
   LDA TXTPTR        ; get BASIC execute pointer low byte
   CMP #2            ; BUG! should be LDA TXTPTR+1:CMP #>BUF:BNE LOAD20
   BEQ LOAD_20
   LAYI(Msg_OK)
   JMP Print_String

LOAD_20
   RTS

LOAD_30
   JSR READST        ; read I/O status word
   AND #$BF          ; mask x0xx xxxx, clear read error
   BEQ LOAD_50       ; branch if no errors
   LDX #$1D          ; error $1D, load error

LOAD_40
   JMP Basic_Error

LOAD_50
   LDA TXTPTR+1      ; get BASIC execute pointer high byte
   CMP #>BUF
   BNE LOAD_60       ; branch if not direct mode

LOAD_55
   STXY(VARTAB)
   Print_Msg(Msg_Ready)
   JMP Reset_And_Rechain

LOAD_60
   JSR Reset_BASIC_Exec_Pointer

#if C64
   JSR Rechain
   JMP Restore_And_Flush_Stack      ; do RESTORE, clear stack and return
#endif
#if VIC
   JMP Rebuild_andRestore      ; rebuild BASIC line chaining, do RESTORE and return
#endif

; **********
  Basic_OPEN
; **********

   JSR Get_Open_Close_Params
   JSR OPEN          ; open a logical file
   BCS Jump_Error_Handler      ; branch if error

   RTS

; ***********
  Basic_CLOSE
; ***********

   JSR Get_Open_Close_Params
   LDA FORPNT        ; get logical file number
   JSR CLOSE         ; close a specified logical file
   BCC LOAD_20      ; exit if no error

Jump_Error_Handler
   JMP Error_Handler

; ********************
  Get_Load_Save_Params
; ********************

   LDA #$00          ; clear file name length
   JSR SETNAM        ; clear filename
   LDX #$01          ; set default device number, cassette
   LDY #$00          ; set default command
#if JIFFY
   JSR Jiffy_SETLFS
#else
   JSR SETLFS        ; set logical, first and second addresses
#endif
   JSR Exit_On_EOS
   JSR Set_Filename
   JSR Exit_On_EOS
   JSR Get_Byte_Param
   LDY #$00          ; clear command
   STX FORPNT        ; save device number
   JSR SETLFS        ; set logical, first and second addresses
   JSR Exit_On_EOS
   JSR Get_Byte_Param
   TXA               ; copy command to A
   TAY               ; copy command to Y
   LDX FORPNT        ; get device number back
   JMP SETLFS        ; set logical, first and second addresses and return
```

```
; **************
  Get_Byte_Param
; **************

   JSR Read_Comma_And_Byte
   JMP Get_Byte_Value

; ***********
  Exit_On_EOS
; ***********

   JSR CHRGOT
   BNE EOE_Ret       ; branch if not [EOL] or ":"
   PLA               ; dump return address low byte
   PLA               ; dump return address high byte
EOE_Ret
   RTS

; *******************
  Read_Comma_And_Byte
; *******************

   JSR Need_Comma

; *********
  Need_Byte
; *********

   JSR CHRGOT
   BNE EOE_Ret      ; exit if following byte
   JMP Syntax_Error

; *********************
  Get_Open_Close_Params
; *********************

   LDA #$00          ; clear file name length
   JSR SETNAM        ; clear filename
   JSR Need_Byte
   JSR Get_Byte_Value
   STX FORPNT        ; save logical file number
   TXA               ; copy logical file number to A
   LDX #$01          ; set default device number, cassette

GOCP_05
   LDY #$00          ; set default command
   JSR SETLFS        ; set logical, first and second addresses
   JSR Exit_On_EOS
   JSR Get_Byte_Param
   STX FORPNT+1      ; save device number
   LDY #$00          ; clear command
   LDA FORPNT        ; get logical file number
   CPX #$03          ; compare device number with screen
   BCC GOCP_10       ; branch if less than screen
   DEY               ; else decrement command

GOCP_10
   JSR SETLFS        ; set logical, first and second addresses
   JSR Exit_On_EOS
   JSR Get_Byte_Param
   TXA               ; copy command to A
   TAY               ; copy command to Y
   LDX FORPNT+1      ; get device number
   LDA FORPNT        ; get logical file number
   JSR SETLFS        ; set logical, first and second addresses
   JSR Exit_On_EOS
   JSR Read_Comma_And_Byte

; ************
  Set_Filename
; ************

   JSR Eval_Expression

Set_Filename_From_String
   JSR Eval_String
   LDX INDEXA        ; get string pointer low byte
   LDY INDEXA+1      ; get string pointer high byte
   JMP SETNAM        ; set filename and return

; *********
  Basic_COS
; *********

   LAYI(PI_Half)    ; set pi/2 pointer
   JSR Add_Var_AY_To_FAC1

; *********
  Basic_SIN
; *********

   JSR FAC1_Round_And_Copy_To_FAC2
   LAYI(Two_PI)
   LDX FAC2SI        ; get FAC2 sign (b7)
   JSR Divide_FAC2_By_AY
   JSR FAC1_Round_And_Copy_To_FAC2
   JSR Basic_INT
   LDA #0
   STA STRPTR        ; clear sign compare (FAC1 EOR FAC2)
   JSR Basic_MINUS   ; perform subtraction, FAC2 from FAC1
   LAYI(Float_0_25)
   JSR AY_Minus_FAC1
   LDA FAC1SI        ; get FAC1 sign (b7)
   PHA               ; save FAC1 sign
   BPL SIN_10        ; branch if positive
   JSR Add_0_5_To_FAC1
   LDA FAC1SI        ; get FAC1 sign (b7)
   BMI SIN_20        ; branch if negative
   LDA TANSGN        ; get the comparison evaluation flag
   EOR #$FF          ; toggle flag
   STA TANSGN        ; save the comparison evaluation flag

SIN_10
   JSR Basic_GREATER ; do - FAC1

SIN_20
   LAYI(Float_0_25)  ; set 0.25 pointer
   JSR Add_Var_AY_To_FAC1
   PLA               ; restore FAC1 sign
   BPL SIN_30        ; branch if was positive
   JSR Basic_GREATER ; do - FAC1

SIN_30
   LAYI(VAR_SIN)     ; set pointer to counter
   JMP Square_And_Series_Eval

; *********
  Basic_TAN
; *********

   JSR FAC1_To_FACTPA
   LDA #0
   STA TANSGN        ; clear the comparison evaluation flag
   JSR Basic_SIN     ; perform SIN()
   LDX #<FUNCPT      ; set sin(n) pointer low byte
   LDY #>FUNCPT      ; set sin(n) pointer high byte
   JSR Go_Assign_FAC1_To_Var
   LAYI(FACTPA)      ; set n pointer
   JSR Load_FAC1_AY
   LDA #0
   STA FAC1SI        ; clear FAC1 sign (b7)
   LDA TANSGN        ; get the comparison evaluation flag
   JSR TAN_10
   LAYI(FUNCPT)      ; set sin(n) pointer
   JMP AY_Divided_By_FAC1

; ======
  TAN_10
; ======

   PHA               ; save comparison flag
   JMP SIN_10      ; add 0.25, ^2 then series evaluation

PI_Half    .real 1.5707963271
Two_PI     .real 6.283185307
Float_0_25 .real 0.25

VAR_SIN
   .byte   $05       ; series counter
   .real -14.381390673
   .real  42.00779713
   .real -76.70417028
   .real  81.60522370
   .real -41.34170211
   .real   6.283185308

; *********
  Basic_ATN
; *********

   LDA FAC1SI        ; get FAC1 sign (b7)
   PHA               ; save sign
   BPL ATN_10        ; branch if positive
   JSR Basic_GREATER ; else do - FAC1

ATN_10
   LDA FAC1EX        ; get FAC1 exponent
   PHA               ; push exponent
   CMP #$81          ; compare with 1
   BCC ATN_20        ; branch if FAC1 < 1
   LAYI(REAL_1)
   JSR AY_Divided_By_FAC1

ATN_20
   LAYI(VAR_ATN)     ; pointer to series
   JSR Square_And_Series_Eval
   PLA                ; restore old FAC1 exponent
   CMP #$81           ; compare with 1
   BCC ATN_30         ; branch if FAC1 < 1
   LAYI(PI_Half)      ; pointer to (pi/2)
   JSR AY_Minus_FAC1

ATN_30
   PLA                ; restore FAC1 sign
   BPL ATN_Ret        ; exit if was positive
   JMP Basic_GREATER ; else do - FAC1 and return

ATN_Ret
   RTS

VAR_ATN
   .byte 11        ; series counter

   .real -6.8479391200E-4
   .real  4.8509421570E-3
   .real -1.6111701850E-2
   .real  3.4209638050E-2
   .real -5.4279132770E-2
   .real  7.2457196550E-2
   .real -8.9802395400E-2
   .real  1.1093241345E-1
   .real -0.1428398077
   .real  0.1999991205
   .real -0.3333333157
   .real  1.0

#if C64
Basic_Warm_Start
   JSR CLRCHN         ; Clear I/O channels
   LDA #0
   STA IOPMPT         ; set current I/O channel, flag default
   JSR Flush_BASIC_Stack
   CLI                ; enable interrupts
Vectored_Basic_Ready
   LDX #$80
   JMP (IERROR)       ; normally next statement
Back_To_Prompt
   TXA
   BMI Jump_READY
   JMP Default_Error       ; print error message

Jump_READY
   JMP Basic_Ready
#endif

Basic_Cold_Start
#if JIFFY
   JSR Jiffy_Jump_Vectors
#else
   JSR Init_BASIC_Jump_Vectors
#endif
   JSR Init_BASIC_RAM_Vectors
   JSR Print_Startup_Message
   LDX #$FB           ; value for start stack
   TXS                ; set stack pointer
#if C64
   BNE Vectored_Basic_Ready        ; branch always
#endif
```

```
#if VIC
   JMP Basic_Ready
#endif

; **********
  CHRGET_ROM
; **********

   INC TXTPTR        ; increment BASIC execute pointer low byte
   BNE CHRO_10       ; branch if no carry
   INC TXTPTR+1      ; increment BASIC execute pointer high byte

CHRO_10
   LDA $EA60         ; get byte to scan, address set by call routine
   CMP #':'          ; compare with ":"
   BCS CHRO_Ret      ; exit if>=
   CMP #' '          ; compare with " "
   BEQ CHRGET_ROM    ; if " " go do next
   SEC               ; set carry for SBC
   SBC #'0'          ; subtract "0"
   SEC               ; set carry for SBC
   SBC #$D0          ; subtract -"0"
                     ; clear carry if byte = "0"-"9"
CHRO_Ret
   RTS

   .byte   $80,$4F,$C7,$52,$58 ; 0.811635157

; *********************
  Init_BASIC_RAM_Vectors
; *********************

   LDA #$4C          ; opcode for JMP
   STA JUMPER        ; save for functions vector jump
   STA Basic_USR
   LAYI(Illegal_Quantity)
   STAY(USRVEC)
   LAYI(Integer_To_Float)
   STAY(ADRAY2)      ; save fixed to float vector low byte
   LAYI(Float_To_Integer)    ; set float to fixed vector
   STAY(ADRAY1)       ; save float to fixed vector low byte
   LDX #$1C          ; set byte count

IBRV_10
   LDA CHRGET_ROM,X  ; get byte from table
   STA CHRGET,X      ; save byte in page zero
   DEX               ; decrement count
   BPL IBRV_10       ; loop if not all done
   LDA #$03          ; set step size, collecting descriptors
   STA GARBSS        ; save garbage collection step size
   LDA #0
   STA FAC1OV        ; clear FAC1 overflow byte
   STA IOPMPT        ; clear current I/O channel, flag default
   STA LASTPT+1      ; clear current descriptor stack item pointer high byte
   LDX #$01          ; set X
   STX BUF-3         ; set chain link pointer low byte
   STX BUF-4         ; set chain link pointer high byte
   LDX #TEMPST       ; initial value for descriptor stack
   STX TEMPPT        ; set descriptor stack pointer
   SEC               ; set Cb = 1 to read the bottom of memory
   JSR MEMBOT        ; read/set the bottom of memory
   STXY(TXTTAB)      ; save start of memory
   SEC               ; set Cb = 1 to read the top of memory
   JSR MEMTOP        ; read/set the top of memory
   STXY(MEMSIZ)      ; save end of memory
   STXY(FRESPC)      ; set bottom of string space
   LDY #0
   TYA               ; clear A
   STA (TXTTAB),Y    ; clear first byte of memory
   INC TXTTAB        ; increment start of memory low byte
   BNE IBRV_Ret      ; branch if no rollover
   INC TXTTAB+1      ; increment start of memory high byte

IBRV_Ret
   RTS

; *********************
  Print_Startup_Message
; *********************

   LDAY(TXTTAB)      ; get start of memory
   JSR Check_Mem_Avail
   Print_Msg(Start_Message)
   LDA MEMSIZ        ; get end of memory low byte
   SEC
   SBC TXTTAB        ; subtract start of memory low byte
   TAX               ; copy result to X
   LDA MEMSIZ+1      ; get end of memory high byte
   SBC TXTTAB+1      ; subtract start of memory high byte
   JSR Print_Integer_XA
   Print_Msg(Bytes_Free_Message)
   JMP Perform_NEW

#if VIC
Bytes_Free_Message
   .byte   " BYTES FREE",$0D,$00

Start_Message
#if JIFFY
   .byte   $93," JIFFYDOS (C)1989 CMD ",$0D,$00
#else
   .byte   $93,"**** CBM BASIC V2 ****",$0D,$00
#endif
#endif

; BASIC vectors, these are copied to RAM from IERROR   onwards

Basic_Vectors

#if C64
#if JIFFY
   .word   Jiffy_Dispatch     ; error message            IERROR
#else
   .word   Back_To_Prompt     ; error message            IERROR
#endif
#endif

#if VIC
#if JIFFY
   .word   Jiffy_Dispatch     ; error message            IERROR
#else
   .word   Default_Error      ; error message            IERROR
#endif
```

```
#endif

   .word   Default_Warmstart  ; BASIC warm start         IMAIN
#if JIFFY
   .word   Jiffy_Tokenize     ; crunch BASIC tokens      ICRNCH
#else
   .word   Default_Tokenize   ; crunch BASIC tokens      ICRNCH
#endif
   .word   Default_Detokenize ; uncrunch BASIC tokens    IQPLOP
   .word   Default_Start      ; start new BASIC code     IGONE
   .word   Default_EVAL       ; get arithmetic element   IEVAL

; ***********************
  Init_BASIC_Jump_Vectors
; ***********************

   LDX #$0B          ; set byte count

IBJV_10
   LDA Basic_Vectors,X
   STA IERROR,X      ; save byte to RAM
   DEX               ; decrement index
   BPL IBJV_10       ; loop if more to do
   RTS

#if C64

   .BYTE 0

Bytes_Free_Message
   .BYTE " BASIC BYTES FREE\r",0

Start_Message
   .BYTE $93
#if JIFFY
   .BYTE "\r       JIFFYDOS V6.01 (C)1989 CMD  \r"
   .BYTE "\r C-64 BASIC V2   ",0
#else
   .BYTE "\r    **** COMMODORE 64 BASIC V2 ****\r"
   .BYTE "\r 64K RAM SYSTEM  ",0
#endif
   .BYTE $81

; **************
  CHKOUT_Checked
; **************

   PHA
   JSR CHKOUT
   TAX
   PLA
   BCC CHCh_Ret
   TXA

CHCh_Ret
   RTS

#endif

#if VIC
; ****************
  Basic_Warm_Start
```

```
; ****************

   JSR CLRCHN          ; Clear I/O channels
   LDA #0
   STA IOPMPT          ; set current I/O channel, flag default
   JSR Flush_BASIC_Stack
   CLI                 ; enable interrupts
   JMP Basic_Ready

; checksum byte, not referenced

#if PAL
   .byte   $e8         ; [PAL]
#else
   .byte   $41         ; [NTSC]
#endif

; rebuild BASIC line chaining and do RESTORE

Rebuild_andRestore
   JSR Rechain
   JMP Restore_And_Flush_Stack
#endif

#if C64
#if JIFFY
; ******************
  Jiffy_Jump_Vectors
; ******************

   JSR Init_BASIC_Jump_Vectors
   LDA #<Jiffy_F1
   STA CMPO
   LDA #>Jiffy_F1
   STA CMPO+1

Jiffy_Inx_PRTY
   INX
   STX PRTY
   RTS

; ***********
  Jiffy_CHRIN
; ***********

   LDA #$6f
   JSR Jiffy_CHKIN
   JSR CHRIN
   CMP #'5'
   RTS

   TAX
   TAX

#else
   .fill 28 ($aa)
#endif

Je4d3
   STA RINONE
   LDA #1
   STA RIPRTY
   RTS

STA_COLOR
   LDA COLOR
   STA (USER),Y
   RTS

; *************
  Delay_2JiffyM
; *************

   ADC #2
Be4e2
   LDY STKEY
   INY
   BNE Be4eb
   CMP JIFFYM
   BNE Be4e2
Be4eb
   RTS

; ********
  BaudNTSC
; ********

   .WORD $2619       ;       9753
   .WORD $1944       ;       6468
   .WORD $111a       ;       4378
   .WORD $0de8       ;       3560
   .WORD $0c70       ;       3184
   .WORD $0606       ;       1542
   .WORD $02d1       ;        721
   .WORD $0137       ;        311
   .WORD $00ae       ;        174
   .WORD $0069       ;        105

#endif

#if VIC
#if JIFFY
; ******************
  Jiffy_Jump_Vectors
; ******************

   JSR Init_BASIC_Jump_Vectors
   LDA #<Jiffy_F1
   STA CMPO
   LDA #>Jiffy_F1
   STA CMPO+1

Jiffy_Inx_PRTY
   INX
   STX PRTY
   RTS

; ***********
  Jiffy_CHRIN
; ***********

   LDA #$6f
   JSR Jiffy_CHKIN
   JSR CHRIN
   CMP #'5'
   RTS

  Jiffy_e496

   PLA
   PLA
   PLA
   PLA
   PLA
   RTS

  CIA2_PRA
  VIC_SPR_ENA
  VIC_CONTROL_1
  VIC_RASTER
   .fill 4 (-1)
#else
   .fill 36 (-1)
#endif

; set serial data out high

; ***********
  CLR_IEC_DAT        ; set serial data high (clear bit)
; ***********

   LDA IEC_PCR
   AND #~IEC_DAT_BIT
   STA IEC_PCR
   RTS

; set serial data out low

SET_IEC_DAT
   LDA IEC_PCR       ; get VIA 2 PCR
   ORA #$20          ; set CB2 high, serial data out low
   STA IEC_PCR       ; set VIA 2 PCR
   RTS

; ***********
  GET_IEC_CLK        ; get serial clock status
; ***********

   LDA IEC_DRAN      ; get VIA 1 DRA, no handshake
   CMP IEC_DRAN      ; compare with self
   BNE GET_IEC_CLK   ; loop if changing
   LSR A             ; shift serial clock to Cb
   RTS

; **********************
  Get_SA_Print_Searching
; **********************

   LDX SA
   JMP Print_Searching

; ****************
  Set_Load_Address
; ****************

   TXA               ; copy secondary address
   BNE SLA_10        ; load location not set in LOAD call, so
   LDA MEMUSS        ; get load address low byte
   STA EAL           ; save program start address low byte
   LDA MEMUSS+1      ; get load address high byte
   STA EAL+1         ; save program start address high byte

SLA_10
   JMP Display_LOADING_Or_VERIFYING

; **********
  Close_Patch
; **********

#if JIFFY
; ****************
  Jiffy_Test_Device
; ****************

   STX FA

; ************
  Jiffy_Test_FA
; ************

   TYA
   PHA
   JSR Jiffy_Open_Command_Channel    ; open 15,x,15
   JSR JiDi_60    ; set command channel (15) as output
   PHP
   JSR Jiffy_Close_15
   PLP
   PLA
   TAY
   LDX FA
   RTS

; *********
  Jiffy_STOP
; *********

   TXA
   PHA
   TSX
   LDA STACK+7,X
   CMP #$f6
   BNE JTD_09
   LDA STACK+6,X
   CMP #$35
   BEQ JTD_11
   CMP #$2f
   BEQ JTD_11

JTD_09
   PLA
   TAX
   JMP (ISTOP)

JTD_11
   JMP Jiffy_e496

#else
   JSR Init_Tape_Write
   BCC ClPa_10       ; branch if no error
   PLA               ; else dump stacked exit code
   LDA #$00          ; clear exit code

ClPa_10
   JMP KeCL_50      ; go do I/O close

   .FILL 38 (-1)     ; spare bytes, not referenced
#endif
#endif

; *************
  Kernal_IOBASE      ; return base address of I/O devices
; *************

   LDX #<IO_Base_Address
   LDY #>IO_Base_Address
   RTS

; *************
  Kernal_SCREEN      ; Return screen format
; *************

   LDX #COLS
   LDY #ROWS
   RTS

; ***********
  Kernal_PLOT        ; read (C=1) or set (C=0) X,Y cursor position
; ***********

   BCS PLOT_10       ; if read cursor skip the set cursor

PLOT_05
   STX TBLX          ; save cursor row
   STY CSRIDX        ; save cursor column
   JSR Adjust_Line   ; set screen pointers for cursor row, column

PLOT_10
   LDX TBLX          ; get cursor row
   LDY CSRIDX        ; get cursor column
   RTS

; *******************
  Initialise_Hardware
; *******************

   JSR Set_Default_Devices
#if VIC
   LDA SCNMPG        ; get screen memory page
   AND #$FD          ; mask xxxx xx0x, all but va9
   ASL A             ; << 1 xxxx x0x0
   ASL A             ; << 2 xxxx 0x00
   ORA #$80          ; set  1xxx 0x00
   STA VIC_R5        ; set screen and character memory location
   LDA SCNMPG        ; get screen memory page
   AND #$02          ; mask bit 9
   BEQ InHa_10       ; if zero just go normalise screen
   LDA #$80          ; set b7
   ORA VIC_R2        ; OR in as video address 9
   STA VIC_R2        ; save new va9
#endif
```

```
InHa_10
   LDA #0
   STA MODE          ; clear shift mode switch
   STA BLNON         ; clear cursor blink phase
   LDA #<Keyboard_Decoder
   STA KEYLOG
   LDA #>Keyboard_Decoder
   STA KEYLOG+1
   LDA #$0A          ; 10d
   STA KBMAXL        ; set maximum size of keyboard buffer
   STA KRPTDL        ; set repeat delay counter
   LDA #Default_Color
   STA COLOR         ; set current colour code
   LDA #$04          ; speed 4
   STA KRPTSP        ; set repeat speed counter
   LDA #$0C          ; cursor flash timing
   STA BLNCT         ; set cursor timing countdown
   STA BLNSW         ; set cursor enable, $00 = flash cursor

; ************
  Clear_Screen
; ************

   LDA SCNMPG        ; get screen memory page
   ORA #$80          ; set high bit, flag every line is logical line start
   TAY               ; copy to Y
   LDA #$00          ; clear line start low byte
   TAX               ; clear index

ClSc_10
   STY SLLTBL,X      ; save start of line X pointer high byte
   CLC
   ADC #COLS         ; add line length to low byte
   BCC ClSc_20       ; if no rollover skip the high byte increment
   INY               ; else increment high byte

ClSc_20
   INX               ; increment line index
   CPX #ROWS+1       ; compare with number of lines + 1
   BNE ClSc_10       ; loop if not all done
   LDA #$FF          ; end of table marker ??
   STA SLLTBL,X      ; mark end of table
   LDX #ROWS-1       ; set line count

ClSc_30
   JSR Clear_Screen_Row_X
   DEX               ; decrement count
   BPL ClSc_30       ; loop if more to do

; *******
  Do_Home
; *******

   LDY #0
   STY CSRIDX        ; clear cursor column
   STY TBLX          ; clear cursor row

Adjust_Line
   LDX TBLX          ; get cursor row
   LDA CSRIDX        ; get cursor column
```

```
Home_10
   LDY SLLTBL,X      ; get start of line X pointer high byte
   BMI Home_20       ; continue if logical line start
   CLC               ; else clear carry for add
   ADC #COLS         ; add one line length
   STA CSRIDX        ; save cursor column
   DEX               ; decrement cursor row
   BPL Home_10       ; loop, branch always

Home_20
#if C64
   JSR Start_Of_Line
#endif
#if VIC
   LDA SLLTBL,X      ; get start of line X pointer high byte
   AND #$03          ; mask 0000 00xx, line memory page
   ORA SCNMPG        ; OR with screen memory page
   STA LINPTR+1      ; set current screen line pointer high byte
   LDA Line_Adress_Low,X    ; get start of line low byte from ROM table
   STA LINPTR        ; set current screen line pointer low byte
#endif
   LDA #COLS-1       ; set line length
   INX               ; increment cursor row

Home_30
   LDY SLLTBL,X      ; get start of line X pointer high byte
   BMI Home_40       ; exit if logical line start
   CLC               ; else clear carry for add
   ADC #COLS         ; add one line length to current line length
   INX               ; increment cursor row
   BPL Home_30       ; loop, branch always

Home_40
   STA LINLEN        ; save current screen line length
#if VIC
   RTS
#endif

#if C64
   JMP Set_COLRAM_Pointer

Home_50
   CPX ICRROW
   BEQ Home_Ret
   JMP Set_Pointer_To_Start_Of_Logical_Row_X

Home_Ret
   RTS

   NOP
#endif

   JSR Set_Default_Devices
   JMP Do_Home

; *******************
  Set_Default_Devices
; *******************

   LDA #3            ; set screen
   STA DFLTO         ; set output device number
   LDA #0            ; set keyboard
   STA DFLTN         ; set input device number

; *************
  Init_VIC_Chip
; *************

   LDX #VIC_REGS

IVC_Loop
   LDA VIC_INIT-1,X  ; get byte from setup table
   STA VIC_BASE-1,X  ; save byte to Vic chip
   DEX               ; decrement count/index
   BNE IVC_Loop      ; loop if more to do
   RTS

; *****************************
  Get_Char_From_Keyboard_Buffer
; *****************************

   LDY KBUFFR        ; get current character from buffer
   LDX #0

GCFK_Loop
   LDA KBUFFR+1,X    ; get next character,X from buffer
   STA KBUFFR,X      ; save as current character,X in buffer
   INX               ; increment index
   CPX NDX           ; compare with keyboard buffer index
   BNE GCFK_Loop     ; loop if more to do
   DEC NDX           ; decrement keyboard buffer index
   TYA               ; copy key to A
   CLI               ; enable interrupts
   CLC               ; flag got byte
   RTS

; ===================
  Display_And_Get_Key
; ===================

   JSR Screen_CHROUT ; output character

; =======
  Get_Key
; =======

   LDA NDX           ; get keyboard buffer index
   STA BLNSW         ; cursor enable, $00 = flash cursor, $xx = no flash
   STA AUTODN        ; screen scrolling flag, $00 = scroll, $xx = no scroll
   BEQ Get_Key       ; loop if buffer empty
   SEI               ; disable interrupts
   LDA BLNON         ; get cursor blink phase
   BEQ GETK_10       ; branch if cursor phase
   LDA GDBLN         ; get character under cursor
   LDX CSRCLR        ; get colour under cursor
   LDY #$00          ; clear Y
   STY BLNON         ; clear cursor blink phase
   JSR Display_Char_A_And_Color_X

GETK_10
#if JIFFY
   JSR Jiffy_f9e5
#else
   JSR Get_Char_From_Keyboard_Buffer
```

```
#endif
   CMP #$83          ; compare with [SHIFT][RUN]
   BNE GETK_30       ; branch if not [SHIFT][RUN]
   LDX #9            ; set byte count
   SEI               ; disable interrupts
   STX NDX           ; set keyboard buffer index

GETK_20
   LDA RUNKEY-1,X    ; get byte from auto load/run table
   STA KBUFFR-1,X    ; save to keyboard buffer
   DEX               ; decrement count/index
   BNE GETK_20       ; loop while more to do
   BEQ Get_Key       ; loop for next key, branch always

GETK_30
   CMP #CR
   BNE Display_And_Get_Key
   LDY LINLEN        ; get current screen line length
   STY INSRC         ; input from keyboard or screen, $xx = screen,

GETK_40
   LDA (LINPTR),Y    ; get character from current screen line
   CMP #' '          ; compare with [SPACE]
   BNE GETK_50       ; branch if not [SPACE]
   DEY               ; else eliminate the space, decrement end of input line
   BNE GETK_40       ; loop, branch always

GETK_50
   INY               ; increment past last non space character on line
   STY INDX          ; save input [EOL] pointer
   LDY #0
   STY AUTODN        ; clear screen scrolling flag, $00 = scroll, $xx = no scroll
   STY CSRIDX        ; clear cursor column
   STY CSRMOD        ; clear cursor quote flag, $xx = quote, $00 = no quote
   LDA ICRROW        ; get input cursor row
   BMI Get_Screen
   LDX TBLX          ; get cursor row
#if C64
   JSR Home_50
#endif
#if VIC
   JSR Set_Pointer_To_Start_Of_Logical_Row_X
#endif
   CPX ICRROW        ; compare with input cursor row
   BNE Get_Screen
#if VIC
   BNE Get_Screen    ; obsolete
#endif
   LDA ICRCOL        ; get input cursor column
   STA CSRIDX        ; save cursor column
   CMP INDX          ; compare with input [EOL] pointer
   BCC Get_Screen    ; branch if less, cursor is in line
   BCS GetS_20      ; else cursor is beyond the line end, branch always

; ========================
  CHRIN_Keyboard_Or_Screen
; ========================

   TYA               ; copy Y
   PHA               ; save Y
   TXA               ; copy X
   PHA               ; save X
   LDA INSRC         ; input from keyboard or screen, $xx = screen,
   BEQ Get_Key       ; if keyboard go wait for key

; ==========
  Get_Screen
; ==========

   LDY CSRIDX        ; get cursor column
   LDA (LINPTR),Y    ; get character from the current screen line
#if VIC
   .fill 23 ($EA)    ; NOP's
#endif
   STA LASTKY        ; save temporary last character
   AND #$3F          ; mask key bits
   ASL LASTKY        ; << temporary last character
   BIT LASTKY        ; test it
   BPL GetS_05       ; branch if not [NO KEY]
   ORA #$80

GetS_05
   BCC GetS_10
   LDX CSRMOD        ; get cursor quote flag, $xx = quote, $00 = no quote
   BNE GetS_15       ; branch if in quote mode

GetS_10
   BVS GetS_15
   ORA #$40

GetS_15
   INC CSRIDX        ; increment cursor column
   JSR If_Quote_Toggle_Flag
   CPY INDX          ; compare with input [EOL] pointer
   BNE GetS_35       ; branch if not at line end

GetS_20
   LDA #$00
   STA INSRC         ; clear input from keyboard or screen, $xx = screen,
   LDA #$0D          ; set character [CR]
   LDX DFLTN         ; get input device number
   CPX #$03          ; compare with screen
   BEQ GetS_25       ; branch if screen
   LDX DFLTO         ; get output device number
   CPX #$03          ; compare with screen
   BEQ GetS_30       ; branch if screen

GetS_25
   JSR Screen_CHROUT ; output character

GetS_30
   LDA #$0D          ; set character [CR]

GetS_35
   STA LASTKY        ; save character
   PLA               ; pull X
   TAX               ; restore X
   PLA               ; pull Y
   TAY               ; restore Y
   LDA LASTKY        ; restore character
   CMP #$DE
   BNE GetS_40
   LDA #$FF
```

```
GetS_40
   CLC
   RTS

; *******************
  If_Quote_Toggle_Flag
; *******************

   CMP #QUOTE
   BNE IQTF_Ret       ; exit if not "
   LDA CSRMOD         ; get cursor quote flag, $xx = quote, $00 = no quote
   EOR #1             ; toggle it
   STA CSRMOD         ; save cursor quote flag
   LDA #QUOTE

IQTF_Ret
   RTS

; ===========
  Insert_Char
; ===========

   ORA #$40           ; change to uppercase/graphic

InsC_10
   LDX RVS            ; get reverse flag
   BEQ InsC_30        ; branch if not reverse

InsC_20
   ORA #$80           ; reverse character

InsC_30
   LDX INSRTO         ; get insert count
   BEQ InsC_40        ; branch if none
   DEC INSRTO         ; else decrement insert count

InsC_40
   LDX COLOR          ; get current colour code
   JSR Display_Char_A_And_Color_X
   JSR Advance_Cursor

InsC_50
   PLA                ; pull Y
   TAY                ; restore Y
   LDA INSRTO         ; get insert count
   BEQ InsC_60        ; skip quote flag clear if inserts to do
   LSR CSRMOD         ; clear cursor quote flag, $xx = quote, $00 = no quote

InsC_60
   PLA                ; pull X
   TAX                ; restore X
   PLA                ; restore A
   CLC
   CLI                ; enable interrupts
   RTS

; **************
  Advance_Cursor
; **************

   JSR Test_Line_Inc
   INC CSRIDX         ; increment cursor column
   LDA LINLEN        ; get current screen line length
   CMP CSRIDX        ; compare with cursor column
   BCS SPTS_Ret      ; exit if line length >= cursor column
   CMP #COLMAX-1     ; compare with max length
   BEQ SPTS_20       ; if at max clear column, back cursor up and do newline
   LDA AUTODN        ; get autoscroll flag
   BEQ AdCu_10       ; branch if autoscroll on
   JMP InsL_10      ; else open space on screen

AdCu_10
   LDX TBLX          ; get cursor row
   CPX #ROWS         ; compare with max + 1
   BCC Expand_Logical_Line
   JSR Scroll_Screen
   DEC TBLX          ; decrement cursor row
   LDX TBLX          ; get cursor row

; *******************
  Expand_Logical_Line
; *******************

   ASL SLLTBL,X      ; shift start of line X pointer high byte
   LSR SLLTBL,X      ; clears bit 7
#if C64
   INX               ; increment screen row
   LDA SLLTBL,X      ; get start of line X pointer high byte
   ORA #$80          ; mark as start of logical line
   STA SLLTBL,X      ; set start of line X pointer high byte
   DEX               ; restore screen row
   LDA LINLEN        ; get current screen line length
   CLC
#endif
#if VIC
   JMP ELL_10        ; make next screen line start of logical line, increment
#endif

; add one line length and set pointers for start of line

ELL_20
   ADC #COLS         ; add one line length
   STA LINLEN        ; save current screen line length

; *************************************
  Set_Pointer_To_Start_Of_Logical_Row_X
; *************************************

   LDA SLLTBL,X      ; get start of line X pointer high byte
   BMI SPTS_10       ; exit loop if start of logical line
   DEX               ; else back up one line
   BNE Set_Pointer_To_Start_Of_Logical_Row_X

SPTS_10
   JMP Start_Of_Line

SPTS_20
   DEC TBLX          ; decrement cursor row. if the cursor was incremented past
   JSR Do_Newline
   LDA #0
   STA CSRIDX        ; clear cursor column

SPTS_Ret
   RTS
```

```
; *************
  Previous_Line
; *************

   LDX TBLX           ; get cursor row
   BNE PreL_10        ; branch if not top row
   STX CSRIDX         ; clear cursor column
   PLA                ; dump return address low byte
   PLA                ; dump return address high byte
   BNE InsC_50        ; restore registers, set quote flag and exit, branch always

PreL_10
   DEX                ; decrement cursor row
   STX TBLX           ; save cursor row
   JSR Adjust_Line    ; set screen pointers for cursor row, column
   LDY LINLEN         ; get current screen line length
   STY CSRIDX         ; save as cursor column
   RTS

; *************
  Screen_CHROUT
; *************

   PHA                ; save character
   STA LASTKY         ; save temporary last character
   TXA                ; copy X
   PHA                ; save X
   TYA                ; copy Y
   PHA                ; save Y
   LDA #0
   STA INSRC          ; clear input from keyboard or screen, $xx = screen,
   LDY CSRIDX         ; get cursor column
   LDA LASTKY         ; restore last character
   BPL ScrO_02        ; branch if unshifted
   JMP ScrO_42        ; do shifted characters and return

ScrO_02
   CMP #$0D           ; compare with [CR]
   BNE ScrO_04        ; branch if not [CR]
   JMP Screen_Return

ScrO_04
   CMP #' '           ; compare with [SPACE]
   BCC ScrO_10        ; branch if < [SPACE]
   CMP #$60
   BCC ScrO_06        ; branch if $20 to $5F
   AND #$DF
   BNE ScrO_08

ScrO_06
   AND #$3F

ScrO_08
   JSR If_Quote_Toggle_Flag
   JMP InsC_10

ScrO_10
   LDX INSRTO         ; get insert count
   BEQ ScrO_12        ; branch if no characters to insert
   JMP InsC_20        ; insert reversed character
```

```
ScrO_12
   CMP #$14          ; compare with [INSERT]/[DELETE]
   BNE ScrO_20       ; branch if not [INSERT]/[DELETE]
   TYA
   BNE ScrO_14
   JSR Previous_Line
   JMP ScrO_18

ScrO_14
   JSR Test_Line_Dec
   DEY               ; decrement index to previous character
   STY CSRIDX        ; save cursor column
   JSR Set_COLRAM_Pointer

ScrO_16
   INY
   LDA (LINPTR),Y    ; get character from current screen line
   DEY               ; decrement index to previous character
   STA (LINPTR),Y    ; save character to current screen line
   INY
   LDA (USER),Y      ; get colour RAM byte
   DEY               ; decrement index to previous character
   STA (USER),Y      ; save colour RAM byte
   INY
   CPY LINLEN        ; compare with current screen line length
   BNE ScrO_16       ; loop if not there yet

ScrO_18
   LDA #' '          ; set [SPACE]
   STA (LINPTR),Y    ; clear last character on current screen line
   LDA COLOR         ; get current colour code
   STA (USER),Y      ; save to colour RAM
   BPL ScrO_38       ; branch always

ScrO_20
   LDX CSRMOD        ; get cursor quote flag, $xx = quote, $00 = no quote
   BEQ ScrO_22       ; branch if not quote mode
   JMP InsC_20       ; insert reversed character

ScrO_22
   CMP #$12          ; compare with [RVS ON]
   BNE ScrO_24       ; branch if not [RVS ON]
   STA RVS           ; set reverse flag

ScrO_24
   CMP #$13          ; compare with [CLR HOME]
   BNE ScrO_26       ; branch if not [CLR HOME]
   JSR Do_Home

ScrO_26
   CMP #$1D          ; compare with [CURSOR RIGHT]
   BNE ScrO_32       ; branch if not [CURSOR RIGHT]
   INY               ; increment cursor column
   JSR Test_Line_Inc
   STY CSRIDX        ; save cursor column
   DEY               ; decrement cursor column
   CPY LINLEN        ; compare cursor column with current screen line length
   BCC ScrO_30       ; exit if less
   DEC TBLX          ; decrement cursor row
   JSR Do_Newline
   LDY #$00          ; clear cursor column
```

```
ScrO_28
   STY CSRIDX        ; save cursor column

ScrO_30
   JMP InsC_50       ; restore registers, set quote flag and exit

ScrO_32
   CMP #$11          ; compare with [CURSOR DOWN]
   BNE ScrO_40       ; branch if not [CURSOR DOWN]
   CLC
   TYA               ; copy cursor column
   ADC #COLS         ; add one line
   TAY               ; copy back to A
   INC TBLX          ; increment cursor row
   CMP LINLEN        ; compare cursor column with current screen line length
   BCC ScrO_28       ; save cursor column and exit if less
   BEQ ScrO_28       ; save cursor column and exit if equal
   DEC TBLX          ; decrement cursor row

ScrO_34
   SBC #COLS         ; subtract one line
   BCC ScrO_36       ; exit loop if on previous line
   STA CSRIDX        ; else save cursor column
   BNE ScrO_34       ; loop if not at start of line

ScrO_36
   JSR Do_Newline

ScrO_38
   JMP InsC_50       ; restore registers, set quote flag and exit

ScrO_40
   JSR Set_Color
   JMP Switch_Text_Graphics

ScrO_42
#if VIC
   .fill 21 ($EA)    ; NOP's
#endif
   AND #$7F          ; mask 0xxx xxxx, clear b7
   CMP #$7F          ; was it $FF before the mask
   BNE ScrO_44       ; branch if not
   LDA #$5E          ; else make it $5E
ScrO_44
#if VIC
   NOP
   NOP
   NOP
   NOP
   NOP
   NOP
#endif
   CMP #' '          ; compare with [SPACE]
   BCC ScrO_46       ; branch if < [SPACE]
   JMP Insert_Char

ScrO_46
   CMP #$0D          ; compare with [CR]
   BNE ScrO_48       ; branch if not [CR]
   JMP Screen_Return      ; else output [CR] and return
                     ; was not [CR]
ScrO_48
```

```
   LDX CSRMOD        ; get cursor quote flag, $xx = quote, $00 = no quote
   BNE ScrO_60       ; branch if quote mode
   CMP #$14          ; compare with [INSERT DELETE]
   BNE ScrO_58       ; branch if not [INSERT DELETE]
   LDY LINLEN        ; get current screen line length
   LDA (LINPTR),Y    ; get character from current screen line
   CMP #' '          ; compare with [SPACE]
   BNE ScrO_50       ; branch if not [SPACE]
   CPY CSRIDX        ; compare current column with cursor column
   BNE ScrO_52       ; if not cursor column go open up space on line

ScrO_50
   CPY #COLMAX-1     ; compare current column with max line length
   BEQ ScrO_56       ; exit if at line end
   JSR Insert_Line

ScrO_52
   LDY LINLEN        ; get current screen line length
   JSR Set_COLRAM_Pointer

ScrO_54
   DEY               ; decrement index to previous character
   LDA (LINPTR),Y    ; get character from current screen line
   INY
   STA (LINPTR),Y    ; save character to current screen line
   DEY               ; decrement index to previous character
   LDA (USER),Y      ; get current screen line colour RAM byte
   INY
   STA (USER),Y      ; save current screen line colour RAM byte
   DEY               ; decrement index to previous character
   CPY CSRIDX        ; compare with cursor column
   BNE ScrO_54       ; loop if not there yet
   LDA #' '          ; set [SPACE]
   STA (LINPTR),Y    ; clear character at cursor position on current screen line
   LDA COLOR         ; get current colour code
   STA (USER),Y      ; save to cursor position on current screen line colour RAM
   INC INSRTO        ; increment insert count

ScrO_56
   JMP InsC_50       ; restore registers, set quote flag and exit

ScrO_58
   LDX INSRTO        ; get insert count
   BEQ ScrO_62       ; branch if no insert space

ScrO_60
   ORA #$40          ; change to uppercase/graphic
   JMP InsC_20       ; insert reversed character

ScrO_62
   CMP #$11          ; compare with [CURSOR UP]
   BNE ScrO_66       ; branch if not [CURSOR UP]
   LDX TBLX          ; get cursor row
   BEQ ScrO_74       ; branch if on top line
   DEC TBLX          ; decrement cursor row
   LDA CSRIDX        ; get cursor column
   SEC
   SBC #COLS         ; subtract one line length
   BCC ScrO_64       ; branch if stepped back to previous line
   STA CSRIDX        ; else save cursor column ..
   BPL ScrO_74       ; .. and exit, branch always
```

```
ScrO_64
   JSR Adjust_Line   ; set screen pointers for cursor row, column ..
   BNE ScrO_74       ; .. and exit, branch always

ScrO_66
   CMP #$12          ; compare with [RVS OFF]
   BNE ScrO_68       ; branch if not [RVS OFF]
   LDA #0
   STA RVS           ; clear reverse flag

ScrO_68
   CMP #$1D          ; compare with [CURSOR LEFT]
   BNE ScrO_72       ; branch if not [CURSOR LEFT]
   TYA               ; copy cursor column
   BEQ ScrO_70       ; branch if at start of line
   JSR Test_Line_Dec
   DEY               ; decrement cursor column
   STY CSRIDX        ; save cursor column
   JMP InsC_50       ; restore registers, set quote flag and exit

ScrO_70
   JSR Previous_Line
   JMP InsC_50       ; restore registers, set quote flag and exit

ScrO_72
   CMP #$13          ; compare with [CLR]
   BNE ScrO_76       ; branch if not [CLR]
   JSR Clear_Screen

ScrO_74
   JMP InsC_50       ; restore registers, set quote flag and exit

ScrO_76
   ORA #$80          ; restore b7, colour can only be black, cyan, magenta
   JSR Set_Color
   JMP STG_10

; **********
  Do_Newline
; **********

   LSR ICRROW        ; shift >> input cursor row
   LDX TBLX          ; get cursor row

NewL_10
   INX               ; increment row
   CPX #ROWS         ; compare with last row + 1
   BNE NewL_20       ; branch if not last row + 1
   JSR Scroll_Screen

NewL_20
   LDA SLLTBL,X      ; get start of line X pointer high byte
   BPL NewL_10       ; loop if not start of logical line
   STX TBLX          ; else save cursor row
   JMP Adjust_Line   ; set screen pointers for cursor row, column and return

; =============
  Screen_Return
; =============

   LDX #0
   STX INSRTO        ; clear insert count
   STX RVS           ; clear reverse flag
   STX CSRMOD        ; clear cursor quote flag, $xx = quote, $00 = no quote
   STX CSRIDX        ; clear cursor column
   JSR Do_Newline
   JMP InsC_50      ; restore registers, set quote flag and exit

; *************
  Test_Line_Dec
; *************

   LDX #COLINK       ; set count
   LDA #$00          ; set column

TLD_10
   CMP CSRIDX        ; compare with cursor column
   BEQ TLD_20        ; branch if at start of line
   CLC               ; else clear carry for add
   ADC #COLS         ; increment to next line
   DEX               ; decrement loop count
   BNE TLD_10        ; loop if more to test
   RTS

TLD_20
   DEC TBLX          ; else decrement cursor row
   RTS

; *************
  Test_Line_Inc
; *************

   LDX #COLINK       ; set count
   LDA #COLS-1       ; set column

TLI_10
   CMP CSRIDX        ; compare with cursor column
   BEQ TLI_20        ; if at end of line test and possibly increment cursor row
   CLC               ; else clear carry for add
   ADC #COLS         ; increment to next line
   DEX               ; decrement loop count
   BNE TLI_10        ; loop if more to test
   RTS

                     ; cursor is at end of line
TLI_20
   LDX TBLX          ; get cursor row
   CPX #ROWS
   BEQ TLI_30        ; exit if end of screen
   INC TBLX          ; else increment cursor row

TLI_30
   RTS

; *********
  Set_Color
; *********

   LDX #Color_Codes_End-Color_Codes-1

SeCo_Loop
   CMP Color_Codes,X ; compare the character with the table code
   BEQ SeCo_10       ; if a match go save the colour and exit
   DEX               ; else decrement the index
   BPL SeCo_Loop     ; loop if more to do
   RTS

SeCo_10
   STX COLOR         ; set current colour code
   RTS

Color_Codes

   .byte $90         ; black
   .byte $05         ; white
   .byte $1C         ; red
   .byte $9F         ; cyan
   .byte $9C         ; magenta
   .byte $1E         ; green
   .byte $1F         ; blue
   .byte $9E         ; yellow

#if C64
   .byte $81         ; orange
   .byte $95         ; brown
   .byte $96         ; light red
   .byte $97         ; grey 1
   .byte $98         ; grey 2
   .byte $99         ; light green
   .byte $9a         ; light blue
   .byte $9b         ; grey 3
#endif

Color_Codes_End

; 76 bytes of unused VIC data

#if VIC
   .byte   $EF,$A1,$DF,$A6,$E1,$B1,$E2,$B2,$E3,$B3,$E4,$B4,$E5,$B5,$E6,$B6
   .byte   $E7,$B7,$E8,$B8,$E9,$B9,$FA,$BA,$FB,$BB,$FC,$BC,$EC,$BD,$FE,$BE
   .byte   $84,$BF,$F7,$C0,$F8,$DB,$F9,$DD,$EA,$DE,$5E,$E0,$5B,$E1,$5D,$E2
   .byte   $40,$B0,$61,$B1,$78,$DB,$79,$DD,$66,$B6,$77,$C0,$70,$F0,$71,$F1
   .byte   $72,$F2,$73,$F3,$74,$F4,$75,$F5,$76,$F6,$7D,$FD
#endif

; *************
  Scroll_Screen
; *************

   LDA SAL           ; save SAL & EAL
   PHA
   LDA SAL+1
   PHA
   LDA EAL
   PHA
   LDA EAL+1
   PHA

ScSc_05
   LDX #-1           ; set for pre increment loop
   DEC TBLX          ; decrement cursor row
   DEC ICRROW        ; decrement input cursor row
   DEC SCROWM        ; decrement screen row marker

ScSc_10
   INX               ; increment line number
   JSR Start_Of_Line
   CPX #ROWS-1      ; compare with last line
   BCS ScSc_15      ; branch if on last line
   LDA Line_Adress_Low+1,X
   STA SAL          ; save next line pointer low byte
   LDA SLLTBL+1,X   ; get start of next line pointer high byte
   JSR Shift_Row
   BMI ScSc_10      ; loop, branch always

ScSc_15
   JSR Clear_Screen_Row_X
   LDX #0

ScSc_20
   LDA SLLTBL,X     ; get start of line X pointer high byte
   AND #$7F         ; clear line X start of logical line bit
   LDY SLLTBL+1,X   ; get start of next line pointer high byte
   BPL ScSc_25      ; branch if next line not start of line
   ORA #$80         ; set line X start of logical line bit

ScSc_25
   STA SLLTBL,X     ; set start of line X pointer high byte
   INX              ; increment line number
   CPX #ROWS-1      ; compare with last line
   BNE ScSc_20      ; loop if not last line
   LDA SLLTBL+ROWS-1 ; get start of last line pointer high byte
   ORA #$80         ; mark as start of logical line
   STA SLLTBL+ROWS-1 ; set start of last line pointer high byte
   LDA SLLTBL       ; get start of first line pointer high byte
   BPL ScSc_05      ; if not start of logical line loop back and
   INC TBLX         ; increment cursor row
   INC SCROWM       ; increment screen row marker
ScSc_27
#if JIFFY
#if C64
   JSR KeSc_70      ; scan standard column
   LDA KEYB_ROW     ; get VIA/CIA keyboard row
   CMP #CTRL_ROW    ; compare with row of [CTRL] key
#endif
#if VIC
   PHP
   SEI
   JMP TASB_20

Jiffy_e9c6
   PLP
   CPX #$fe
#endif
#else
   LDA #CTRL_COL    ; set keyboard column for [CTRL]�key
   STA KEYB_COL     ; set VIA/CIA keyboard column
   LDA KEYB_ROW     ; get VIA/CIA keyboard row
   CMP #CTRL_ROW    ; compare with row of [CTRL] key
#endif
#if JIFFY
   BNE ScSc_40      ; no [CTRL]
   LDX NDX          ; chars in keyboard buffer
   BEQ ScSc_27      ; none
   LDA KBUFFR-1,X   ; last key
   SBC #$13         ; [CTRL]�S
   BNE ScSc_40
   STA NDX          ; clear keyboard buffer
```

```
ScSc_28
   CLI
   CMP NDX
   BEQ ScSc_28       ; wait until key pressed
   STA NDX
#else
   PHP               ; save status
   LDA #STND_COL     ; set standard keyboard col
   STA KEYB_COL      ; set VIA/CIA keyboard column
   PLP               ; restore status
   BNE ScSc_40       ; skip if no [CTRL]�key down
   LDY #0            ; delay scrolling if [CTRL]�key down

ScSc_30
   NOP               ; waste cycles
   DEX               ; decrement inner loop count
   BNE ScSc_30       ; loop if not all done
   DEY               ; decrement outer loop count
   BNE ScSc_30       ; loop if not all done
   STY NDX           ; clear keyboard buffer index
#endif

ScSc_40
   LDX TBLX          ; get cursor row

ScSc_45
   PLA               ; restore EAL & SAL
   STA EAL+1
   PLA
   STA EAL
   PLA
   STA SAL+1
   PLA
   STA SAL
   RTS

; ***********
  Insert_Line
; ***********

   LDX TBLX          ; get cursor row

InsL_10
   INX               ; increment row
   LDA SLLTBL,X      ; get start of line X pointer high byte
   BPL InsL_10       ; branch if not start of logical line
   STX SCROWM        ; set screen row marker
   CPX #ROWS-1       ; compare with last line
   BEQ InsL_20       ; branch if = last line
   BCC InsL_20       ; branch if < last line
   JSR Scroll_Screen
   LDX SCROWM        ; get screen row marker
   DEX               ; decrement screen row marker
   DEC TBLX          ; decrement cursor row
   JMP Expand_Logical_Line

InsL_20
   LDA SAL           ; copy tape buffer pointer
   PHA               ; save it
   LDA SAL+1         ; copy tape buffer pointer
   PHA               ; save it
   LDA EAL           ; copy tape buffer end pointer
   PHA               ; save it
   LDA EAL+1         ; copy tape buffer end pointer
   PHA               ; save it
   LDX #ROWS         ; set to end line + 1 for predecrement loop

InsL_30
   DEX               ; decrement line number
   JSR Start_Of_Line
   CPX SCROWM        ; compare with screen row marker
   BCC InsL_40       ; branch if < screen row marker
   BEQ InsL_40       ; branch if = screen row marker
   LDA Line_Adress_Low-1,X  ; else get start of previous line low byte from ROM table
   STA SAL           ; save previous line pointer low byte
   LDA SLLTBL-1,X    ; get start of previous line pointer high byte
   JSR Shift_Row
   BMI InsL_30       ; loop, branch always

InsL_40
   JSR Clear_Screen_Row_X
   LDX #ROWS-2

InsL_50
   CPX SCROWM        ;.compare with screen row marker
   BCC InsL_70
   LDA SLLTBL+1,X
   AND #$7F
   LDY SLLTBL,X
   BPL InsL_60
   ORA #$80

InsL_60
   STA SLLTBL+1,X
   DEX
   BNE InsL_50

InsL_70
   LDX SCROWM        ;.get screen row marker
   JSR Expand_Logical_Line

#if C64
   JMP ScSc_45
#endif
#if VIC
   PLA               ; pull tape buffer end pointer
   STA EAL+1         ; restore it
   PLA               ; pull tape buffer end pointer
   STA EAL           ; restore it
   PLA               ; pull tape buffer pointer
   STA SAL+1         ; restore it
   PLA               ; pull tape buffer pointer
   STA SAL           ; restore it
   RTS
#endif

; *********
  Shift_Row
; *********

   AND #$03          ; mask 0000 00xx, line memory page
   ORA SCNMPG        ; OR with screen memory page
   STA SAL+1         ; save next/previous line pointer high byte
   JSR Update_Color_RAM_Pointer

ShRo_10
   LDY #COLS-1       ; set column count

ShRo_20
   LDA (SAL),Y       ; get character from next/previous screen line
   STA (LINPTR),Y    ; save character to current screen line
   LDA (EAL),Y       ; get colour from next/previous screen line colour RAM
   STA (USER),Y      ; save colour to current screen line colour RAM
   DEY               ; decrement column index/count
   BPL ShRo_20
   RTS

; ************************
  Update_Color_RAM_Pointer
; ************************

   JSR Set_COLRAM_Pointer
   LDA SAL
   STA EAL
   LDA SAL+1
   AND #$03
#if C64
   ORA #$D8          ; C64 color RAM = $D800
#endif
#if VIC
   ORA #$94          ; VIC color RAM = $9400
#endif
   STA EAL+1
   RTS

; *************
  Start_Of_Line
; *************

   LDA Line_Adress_Low,X
   STA LINPTR        ; set current screen line pointer low byte
   LDA SLLTBL,X      ; get start of line high byte from RAM table
   AND #$03          ; mask 0000 00xx, line memory page
   ORA SCNMPG        ; OR with screen memory page
   STA LINPTR+1      ; set current screen line pointer high byte
   RTS

; ******************
  Clear_Screen_Row_X
; ******************

   LDY #COLS-1       ; set number of columns to clear
   JSR Start_Of_Line
   JSR Set_COLRAM_Pointer

CSRX_Loop
#if C64
   JSR STA_COLOR
#endif
   LDA #' '          ; set [SPACE]
   STA (LINPTR),Y    ; clear character in current screen line
#if VIC
   LDA #1            ; set colour, blue on white
   STA (USER),Y      ; set colour RAM in current screen line
#endif
```

```
   DEY
   BPL CSRX_Loop
   RTS


#if C64
   NOP
#endif

; *************************
  Display_Char_A_And_Color_X
; *************************

   TAY               ; copy character
   LDA #$02          ; set count to $02, usually $14 ??
   STA BLNCT         ; set cursor countdown
   JSR Set_COLRAM_Pointer
   TYA               ; get character back

; *********************
  Display_Char_And_Color
; *********************

   LDY CSRIDX        ; get cursor column
   STA (LINPTR),Y    ; save character from current screen line
   TXA               ; copy colour to A
   STA (USER),Y      ; save to colour RAM
   RTS

; *****************
  Set_COLRAM_Pointer
; *****************

   LDA LINPTR        ; get current screen line pointer low byte
   STA USER          ; save pointer to colour RAM low byte
   LDA LINPTR+1      ; get current screen line pointer high byte
   AND #$03          ; mask 0000 00xx, line memory page
   ORA #COLRAM_PAGE  ; set  1001 01xx, colour memory page
   STA USER+1        ; save pointer to colour RAM high byte
   RTS

; **********
  Default_IRQ
; **********

   JSR UDTIM         ; Update the system clock
   LDA BLNSW         ; get cursor enable
   BNE DIRQ_20       ; branch if not flash cursor
   DEC BLNCT         ; else decrement cursor timing countdown
   BNE DIRQ_20       ; branch if not done
   LDA #$14          ; set count
   STA BLNCT         ; save cursor timing countdown
   LDY CSRIDX        ; get cursor column
   LSR BLNON         ; shift b0 cursor blink phase into carry
   LDX CSRCLR        ; get colour under cursor
   LDA (LINPTR),Y    ; get character from current screen line
   BCS DIRQ_10       ; branch if cursor phase b0 was 1
   INC BLNON         ; set cursor blink phase to 1
   STA GDBLN         ; save character under cursor
   JSR Set_COLRAM_Pointer
   LDA (USER),Y      ; get colour RAM byte
   STA CSRCLR        ; save colour under cursor
   LDX COLOR         ; get current colour code
   LDA GDBLN         ; get character under cursor

DIRQ_10
   EOR #$80          ; toggle b7 of character under cursor
   JSR Display_Char_And_Color

DIRQ_20
#if JIFFY
   JMP DIRQ_50

; **************
  Jiffy_Tokenize
; **************

   PLA
   PHA
   CMP #<[Direct_Call+2]  ; called from direct mode
   BEQ JiTo_20

JiTo_10
   JMP Default_Tokenize

JiTo_20
   JSR Jiffy_Test_Command
   BNE JiTo_10
   LDX TXTPTR
   LDY #4
   TYA
   JMP Toke_REM
#if C64
   .byte 1
#endif

#if VIC
Jiffy_eb08
   JSR JiDi_60
   JMP PrSe_10
   .byte $03
   STA VIA1_PCR
#endif


#else

#if C64
   LDA R6510
   AND #$10          ; mask cassette switch sense
#endif
#if VIC
   LDA IEC_DRAN      ; get VIA 1 DRA, no handshake
   AND #$40          ; mask cassette switch sense
#endif

   BEQ DIRQ_30      ; branch if cassette sense low
   LDY #0
   STY CAS1          ; clear the tape motor interlock

#if C64
   LDA R6510
   ORA #$20          ; set CA2 high, turn off motor
#endif
```

```
#if VIC
   LDA VIA1_PCR      ; get VIA 1 PCR
   ORA #$02          ; set CA2 high, turn off motor
#endif

   BNE DIRQ_40      ; branch always

DIRQ_30
   LDA CAS1          ; get tape motor interlock
   BNE DIRQ_50      ; if cassette interlock <> 0 don't turn on motor

#if C64
   LDA $01           ; R6510
   AND #$1F          ; turn on motor
#endif
#if VIC
   LDA VIA1_PCR      ; get VIA 1 PCR
   AND #$FD          ; set CA2 low, turn on motor
#endif

DIRQ_40

#if C64
   STA R6510
#endif
#if VIC
   BIT VIA1_IER      ; test VIA 1 IER
   BVS DIRQ_50       ; if T1 interrupt enabled don't change motor state
   STA VIA1_PCR      ; set VIA 1 PCR, set CA2 high/low
#endif

#endif               ; JIFFY

DIRQ_50
   JSR Kernal_SCNKEY ; scan keyboard

#if C64
   LDA CIA1_ICR      ; CIA1 Interrupt Control Register
#endif
#if VIC
   BIT VIA2_T1CL     ; clear the timer interrupt flag
#endif

   PLA
   TAY
   PLA
   TAX
   PLA
   RTI

; *************
  Kernal_SCNKEY
; *************

; 1) check if key pressed, if not then exit the routine
; 2) init I/O ports of VIA 2 for keyboard scan and set pointers to
;    decode table 1. clear the character counter
; 3) set one line of port B low and test for a closed key on port A by
;    shifting the byte read from the port. if the carry is clear then a
;    key is closed so save the count which is incremented on each shift.
;    check for shift/stop/cbm keys and flag if closed
; 4) repeat step 3 for the whole matrix
; 5) evaluate the SHIFT/CTRL/C= keys, this may change the decode table
;    selected
; 6) use the key count saved in step 3 as an index into the table
;    selected in step 5
; 7) check for key repeat operation
; 8) save the decoded key to the buffer if first press or repeat

   LDA #0
   STA SHFLAG        ; clear keyboard shift/control/c= flag
   LDY #$40          ; set no key
   STY SFDX          ; save which key
   STA KEYB_COL      ; clear keyboard column
   LDX KEYB_ROW      ; get keyboard row
   CPX #$FF          ; compare with all bits set
   BEQ KeSc_50       ; if no key pressed clear current key and exit

#if C64
   TAY               ; clear key count
#endif
#if VIC
   LDA #$FE
   STA KEYB_COL      ; select keyboard col 0
   LDY #0            ; clear key count
#endif

   LDA #<KBD_NORMAL  ; get decode table low byte
   STA KBDPTR        ; set keyboard pointer low byte
   LDA #>KBD_NORMAL  ; get decode table high byte
   STA KBDPTR+1      ; set keyboard pointer high byte

#if C64
   LDA #$FE
   STA KEYB_COL      ; select keyboard col 0
#endif

KeSc_05
   LDX #8            ; set row count

#if C64
   PHA
KeSc_10
#endif

   LDA KEYB_ROW      ; get VIA/CIA keyboard row
   CMP KEYB_ROW      ; compare with itself

#if C64
   BNE KeSc_10       ; loop if changing
#endif
#if VIC
   BNE KeSc_05      ; loop if changing
#endif

KeSc_15
   LSR A             ; shift row to Cb
   BCS KeSc_30       ; if no key closed on this row go do next row
   PHA               ; save row
   LDA (KBDPTR),Y    ; get character from decode table
   CMP #$05          ; compare with $05, there is no $05 key but the control
   BCS KeSc_20       ; if not shift/control/c=/stop go save key count
   CMP #$03          ; compare with $03, stop
   BEQ KeSc_20       ; if stop go save key count and continue
   ORA SHFLAG        ; OR keyboard shift/control/c= flag
   STA SHFLAG        ; save keyboard shift/control/c= flag
   BPL KeSc_25       ; skip save key, branch always

KeSc_20
   STY SFDX          ; save key count

KeSc_25
   PLA               ; restore row

KeSc_30
   INY               ; increment key count
   CPY #$41          ; compare with max+1
   BCS KeSc_35       ; exit loop if >= max+1
   DEX               ; decrement row count
   BNE KeSc_15       ; loop if more rows to do
   SEC               ; set carry for keyboard column shift

#if C64
   PLA
   ROL A
   STA KEYB_COL
#endif
#if VIC
   ROL KEYB_COL      ; shift VIA 2 DRB, keyboard column
#endif

   BNE KeSc_05      ; loop for next column, branch always

KeSc_35
#if C64
   PLA
#endif

   JMP (KEYLOG)      ; normally Keyboard_Decoder

; key decoding continues here after the SHIFT/CTRL/C= keys are evaluated

KeSc_40
   LDY SFDX          ; get saved key count
   LDA (KBDPTR),Y    ; get character from decode table
   TAX               ; copy character to X
   CPY LSTX          ; compare key count with last key count
   BEQ KeSc_45       ; if this key = current key, key held, go test repeat
   LDY #$10          ; set repeat delay count
   STY KRPTDL        ; save repeat delay count
   BNE KeSc_65       ; go save key to buffer and exit, branch always

KeSc_45
   AND #$7F          ; clear b7
   BIT KEYRPT        ; test key repeat
   BMI KeSc_55       ; branch if repeat all
   BVS KeSc_70       ; branch if repeat none
   CMP #$7F          ; compare with end marker

KeSc_50
   BEQ KeSc_65       ; if $00/end marker go save key to buffer and exit
   CMP #$14          ; compare with [INSERT]/[DELETE]
   BEQ KeSc_55       ; if [INSERT]/[DELETE] go test for repeat
   CMP #' '          ; compare with [SPACE]
   BEQ KeSc_55       ; if [SPACE] go test for repeat
   CMP #$1D          ; compare with [CURSOR RIGHT]
   BEQ KeSc_55       ; if [CURSOR RIGHT] go test for repeat
   CMP #$11          ; compare with [CURSOR DOWN]
   BNE KeSc_70       ; if not [CURSOR DOWN] just exit

KeSc_55
   LDY KRPTDL        ; get repeat delay counter
   BEQ KeSc_60       ; branch if delay expired
   DEC KRPTDL        ; else decrement repeat delay counter
   BNE KeSc_70       ; branch if delay not expired

KeSc_60
   DEC KRPTSP        ; decrement repeat speed counter
   BNE KeSc_70       ; branch if repeat speed count not expired
   LDY #$04          ; set for 4/60ths of a second
   STY KRPTSP        ; set repeat speed counter
   LDY NDX           ; get keyboard buffer index
   DEY               ; decrement it
   BPL KeSc_70       ; if the buffer isn't empty just exit

KeSc_65
   LDY SFDX          ; get the key count
   STY LSTX          ; save as the current key count
   LDY SHFLAG        ; get keyboard shift/control/c= flag
   STY LSTSHF        ; save as last keyboard shift pattern
   CPX #$FF          ; compare character with table end marker or no key
   BEQ KeSc_70       ; if table end marker or no key just exit
   TXA               ; copy character to A
   LDX NDX           ; get keyboard buffer index
   CPX KBMAXL        ; compare with keyboard buffer size
   BCS KeSc_70       ; if buffer full just exit
   STA KBUFFR,X      ; save character to keyboard buffer
   INX               ; increment index
   STX NDX           ; save keyboard buffer index

KeSc_70
   LDA #STND_COL     ; col 3 on VIC / col 7 on C64
   STA KEYB_COL      ; set VIA/CIA keyboard column
   RTS

; ****************
  Keyboard_Decoder
; ****************

   LDA SHFLAG        ; get keyboard shift/control/c= flag
   CMP #$03          ; compare with [SHIFT][C=]
   BNE KeDe_10       ; branch if not
   CMP LSTSHF        ; compare with last
   BEQ KeSc_70       ; exit if still the same
   LDA MODE          ; get shift mode switch $00 = enabled, $80 = locked
   BMI KeDe_30       ; if locked continue keyboard decode

#if VIC
#if !JIFFY
   .fill 19 ($EA)    ; NOP's
#endif
#endif
                     ; switch character ROM
#if C64
   LDA MEM_CONTROL   ; get start of character memory, ROM
   EOR #$02          ; toggle $8000,$8800
   STA MEM_CONTROL   ; set start of character memory, ROM
#endif
```

```
#if VIC
   LDA VIC_R5        ; get start of character memory, ROM
   EOR #$02          ; toggle $8000,$8800
   STA VIC_R5        ; set start of character memory, ROM
#if !JIFFY
   .fill  4 ($EA)    ; NOP's
#endif
#endif

#if VIC & JIFFY
   JMP KeSc_40       ; continue keyboard decode
#else
   JMP KeDe_30       ; continue keyboard decode
#endif

KeDe_10
   ASL A             ; convert flag to index
   CMP #8            ; compare with [CTRL]
   BCC KeDe_20       ; branch if not [CTRL] pressed
   LDA #6            ; [CTRL] : table 3 : index 6
#if VIC
#if !JIFFY
   .fill 2 ($EA)     ; NOP's
#endif
#endif

KeDe_20
#if VIC
#if !JIFFY
   .fill 32 ($EA)    ; NOP's
#endif
#endif
   TAX               ; copy index to X
   LDA KBD_Decode_Pointer,X
   STA KBDPTR
   LDA KBD_Decode_Pointer+1,X
   STA KBDPTR+1
KeDe_30
   JMP KeSc_40       ; continue keyboard decode

#if VIC & JIFFY
KeDe_60
   LDA #4
   JSR LISTEN
   LDA MEM_CONTROL
   AND #2
   BEQ KeDe_62
   LDA #7

KeDe_62
   ORA #$60
   JSR SECOND
   LDA CSRIDX
   PHA
   LDA TBLX
   PHA
   JMP $edaa

   .fill 15 ($ea)

   TAX               ; copy index to X
   LDA KBD_Decode_Pointer,X
   STA KBDPTR
   LDA KBD_Decode_Pointer+1,X
   STA KBDPTR+1
   JMP KeSc_40       ; continue keyboard decode

#endif

; =================
  KBD_Decode_Pointer
; =================

   .word   KBD_NORMAL  ; 0   normal
   .word   KBD_SHIFTED ; 1   shifted
   .word   KBD_CBMKEY  ; 2   commodore
#if VIC & JIFFY
   .word   KBD_COMMCON ; 6   commodore control
#else
   .word   KBD_CONTROL ; 3   control
#endif

#if VIC
#if JIFFY
Jiffy_Combine_Nibbles
   LDA TAPE1+1
   AND #15
   STA TAPE1+1
   LDA CAS1
   ASL A
   ASL A
   ASL A
   ASL A
   ORA TAPE1+1
   RTS
   .byte $ed
#else
   .word   KBD_NORMAL  ; 4   control
   .word   KBD_SHIFTED ; 5   shift - control
   .word   KBD_COMMCON ; 6   commodore control
   .word   KBD_CONTROL ; 7   shift - commdore - control
   .word   Switch_Text_Graphics    ; 8   unused
   .word   KBD_COMMCON ; 9   unused
   .word   KBD_COMMCON ; a   unused
   .word   KBD_CONTROL ; b   unused
#endif
#endif


KBD_NORMAL           ; keyboard decode table - unshifted

#if C64
   .BYTE $14,$0d,$1d,$88,$85,$86,$87,$11
   .BYTE $33,$57,$41,$34,$5a,$53,$45,$01
   .BYTE $35,$52,$44,$36,$43,$46,$54,$58
   .BYTE $37,$59,$47,$38,$42,$48,$55,$56
   .BYTE $39,$49,$4a,$30,$4d,$4b,$4f,$4e
   .BYTE $2b,$50,$4c,$2d,$2e,$3a,$40,$2c
   .BYTE $5c,$2a,$3b,$13,$01,$3d,$5e,$2f
   .BYTE $31,$5f,$04,$32,$20,$02,$51,$03
   .BYTE $ff
#endif

#if VIC
```

```
   .byte $31,$33,$35,$37,$39,$2B,$5C,$14
   .byte $5F,$57,$52,$59,$49,$50,$2A,$0D
   .byte $04,$41,$44,$47,$4A,$4C,$3B,$1D
   .byte $03,$01,$58,$56,$4E,$2C,$2F,$11
   .byte $20,$5A,$43,$42,$4D,$2E,$01,$85
   .byte $02,$53,$46,$48,$4B,$3A,$3D,$86
   .byte $51,$45,$54,$55,$4F,$40,$5E,$87
   .byte $32,$34,$36,$38,$30,$2D,$13,$88
   .byte $FF
#endif

KBD_SHIFTED          ; keyboard decode table - shifted

#if C64
   .BYTE $94,$8d,$9d,$8c,$89,$8a,$8b,$91
   .BYTE $23,$d7,$c1,$24,$da,$d3,$c5,$01
   .BYTE $25,$d2,$c4,$26,$c3,$c6,$d4,$d8
   .BYTE $27,$d9,$c7,$28,$c2,$c8,$d5,$d6
   .BYTE $29,$c9,$ca,$30,$cd,$cb,$cf,$ce
   .BYTE $db,$d0,$cc,$dd,$3e,$5b,$ba,$3c
   .BYTE $a9,$c0,$5d,$93,$01,$3d,$de,$3f
   .BYTE $21,$5f,$04,$22,$a0,$02,$d1,$83
   .BYTE $ff
#endif

#if VIC
   .byte $21,$23,$25,$27,$29,$DB,$A9,$94
   .byte $5F,$D7,$D2,$D9,$C9,$D0,$C0,$8D
   .byte $04,$C1,$C4,$C7,$CA,$CC,$5D,$9D
   .byte $83,$01,$D8,$D6,$CE,$3C,$3F,$91
   .byte $A0,$DA,$C3,$C2,$CD,$3E,$01,$89
   .byte $02,$D3,$C6,$C8,$CB,$5B,$3D,$8A
   .byte $D1,$C5,$D4,$D5,$CF,$BA,$DE,$8B
   .byte $22,$24,$26,$28,$30,$DD,$93,$8C
   .byte $FF
#endif

KBD_CBMKEY           ; keyboard decode table - commodore

#if C64
   .BYTE $94,$8d,$9d,$8c,$89,$8a,$8b,$91
   .BYTE $96,$b3,$b0,$97,$ad,$ae,$b1,$01
   .BYTE $98,$b2,$ac,$99,$bc,$bb,$a3,$bd
   .BYTE $9a,$b7,$a5,$9b,$bf,$b4,$b8,$be
   .BYTE $29,$a2,$b5,$30,$a7,$a1,$b9,$aa
   .BYTE $a6,$af,$b6,$dc,$3e,$5b,$a4,$3c
   .BYTE $a8,$df,$5d,$93,$01,$3d,$de,$3f
   .BYTE $81,$5f,$04,$95,$a0,$02,$ab,$83
   .BYTE $ff
#endif

#if VIC
   .byte $21,$23,$25,$27,$29,$A6,$A8,$94
   .byte $5F,$B3,$B2,$B7,$A2,$AF,$DF,$8D
   .byte $04,$B0,$AC,$A5,$B5,$B6,$5D,$9D
   .byte $83,$01,$BD,$BE,$AA,$3C,$3F,$91
   .byte $A0,$AD,$BC,$BF,$A7,$3E,$01,$89
   .byte $02,$AE,$BB,$B4,$A1,$5B,$3D,$8A
   .byte $AB,$B1,$A3,$B8,$B9,$A4,$DE,$8B
   .byte $22,$24,$26,$28,$30,$DC,$93,$8C
   .byte $FF
#endif
```

```
; ====================
  Switch_Text_Graphics
; ====================

   CMP #$0E           ; compare with [SWITCH TO LOWER CASE]
   BNE STG_10         ; branch if not [SWITCH TO LOWER CASE]

#if C64
   LDA MEM_CONTROL
   ORA #2
   BNE STG_20
#endif
#if VIC
   LDA #$02           ; set for $8800, lower case characters
   ORA MEM_CONTROL    ; OR with start of character memory, ROM
   STA MEM_CONTROL    ; save start of character memory, ROM
   JMP InsC_50        ; restore registers, set quote flag and exit
#endif

STG_10
   CMP #$8E           ; compare with [SWITCH TO UPPER CASE]
   BNE STG_40         ; branch if not [SWITCH TO UPPER CASE]

#if C64
   LDA MEM_CONTROL
   AND #$FD
#endif
#if VIC
   LDA #$FD           ; set for $8000, upper case characters
   AND MEM_CONTROL    ; AND with start of character memory, ROM
#endif

STG_20
   STA MEM_CONTROL    ; save start of character memory, ROM

STG_30
   JMP InsC_50        ; restore registers, set quote flag and exit

STG_40
   CMP #$08           ; compare with disable [SHIFT][C=]
   BNE STG_50         ; branch if not disable [SHIFT][C=]
   LDA #$80           ; set to lock shift mode switch
   ORA MODE           ; OR with shift mode switch, $00 = enabled, $80 = locked

#if C64
   BMI STG_60
#endif
#if VIC
   STA MODE           ; save shift mode switch
   BMI STG_30         ; branch always
#endif

STG_50
   CMP #$09           ; compare with enable [SHIFT][C=]
   BNE STG_30         ; exit if not enable [SHIFT][C=]    ##### start ####
   LDA #$7F           ; set to unlock shift mode switch
   AND MODE           ; AND with shift mode switch, $00 = enabled, $80 = locked

STG_60
   STA MODE           ; save shift mode switch
```

```
#if C64
   JMP InsC_50
#endif
#if VIC

   BPL STG_30        ; branch always

; VC-20 patch for "Expand Logical Line": make next screen line start of
; logical line, increment line length and set pointers

ELL_10
   INX               ; increment screen row
   LDA SLLTBL,X      ; get start of line X pointer high byte
   ORA #$80          ; mark as start of logical line
   STA SLLTBL,X      ; set start of line X pointer high byte
   DEX               ; restore screen row
   LDA LINLEN        ; get current screen line length
   CLC
   JMP ELL_20      ; add one line length, set pointers for start of line and
                     ; return
#endif

#if C64

KBD_CONTROL          ; keyboard decode table - control

  .BYTE $ff,$ff,$ff,$ff,$ff,$ff,$ff,$ff
  .BYTE $1c,$17,$01,$9f,$1a,$13,$05,$ff
  .BYTE $9c,$12,$04,$1e,$03,$06,$14,$18
  .BYTE $1f,$19,$07,$9e,$02,$08,$15,$16
  .BYTE $12,$09,$0a,$92,$0d,$0b,$0f,$0e
  .BYTE $ff,$10,$0c,$ff,$ff,$1b,$00,$ff
  .BYTE $1c,$ff,$1d,$ff,$ff,$1f,$1e,$ff
  .BYTE $90,$06,$ff,$05,$ff,$ff,$11,$ff
  .BYTE $ff

VIC_INIT             ; initialise VIC registers

   .BYTE $00,$00,$00,$00,$00,$00,$00,$00
   .BYTE $00,$00,$00,$00,$00,$00,$00,$00
   .BYTE $00,$9b,$37,$00,$00,$00,$08,$00
   .BYTE $14,$0f,$00,$00,$00,$00,$00,$00
   .BYTE $0e,$06,$01,$02,$03,$04,$00,$01
   .BYTE $02,$03,$04,$05,$06,$07
#endif

#if VIC
KBD_COMMCON         ; keyboard decode table - cbm - control
#if JIFFY

   .byte $90,$1c,$9c,$1f,$12,$ff,$1c,$ff
   .byte $06,$17,$12,$19,$09,$10,$ff,$ff
   .byte $ff,$01,$04,$07,$0a,$0c,$1d,$ff
   .byte $ff,$ff,$18,$16,$0e,$ff,$ff,$ff
   .byte $ff,$1a,$03,$02,$0d,$ff,$ff,$ff
   .byte $ff,$13,$06,$08,$0b,$1b,$1f,$ff
   .byte $11,$05,$14,$15,$0f,$00,$1e,$ff
   .byte $05,$9f,$1e,$9e,$92,$ff,$ff,$ff
   .byte $ff

JAAA
   LDY #0
   STY CSRMOD
   JSR PLOT_05
   INC LINLEN

JAAA_10
   JSR CHRI_07
   JSR Kernal_CIOUT
   CMP #13
   BNE JAAA_10
   INX
   CPX #$17
   BCS JAAA_20
   ASL LINLEN
   BPL JAAA
   INX
   BNE JAAA

JAAA_20
   JSR UNLSN
   PLA
   TAX
   PLA
   TAY
   JSR PLOT_05

JAAA_30
   PLA

JAAA_40
   RTS
#else
   .byte   $FF,$FF,$FF,$FF,$FF,$FF,$FF,$FF
   .byte   $FF,$04,$FF,$FF,$FF,$FF,$FF,$E2
   .byte   $9D,$83,$01,$FF,$FF,$FF,$FF,$FF
   .byte   $91,$A0,$FF,$FF,$FF,$FF,$EE,$01
   .byte   $89,$02,$FF,$FF,$FF,$FF,$E1,$FD
   .byte   $8A,$FF,$FF,$FF,$FF,$FF,$B0,$E0
   .byte   $8B,$F2,$F4,$F6,$FF,$F0,$ED,$93
   .byte   $8C,$FF

KBD_CONTROL          ; keyboard decode table - control

   .byte   $90,$1C,$9C,$1F,$12,$FF,$FF,$FF
   .byte   $06,$FF,$12,$FF,$FF,$FF,$FF,$FF
   .byte   $FF,$FF,$FF,$FF,$FF,$FF,$FF,$FF
   .byte   $FF,$FF,$FF,$FF,$FF,$FF,$FF,$FF
   .byte   $FF,$FF,$FF,$FF,$FF,$FF,$FF,$FF
   .byte   $FF,$FF,$FF,$FF,$FF,$FF,$FF,$FF
   .byte   $FF,$FF
#endif

   .byte $FF,$FF,$FF,$FF,$FF,$FF,$05
   .byte $9F,$1E,$9E,$92,$FF,$FF,$FF,$FF

VIC_INIT             ; initial values for VIC registers

#if PAL
   .byte   $0C       ; horizontal offset [PAL]
   .byte   $26       ; vertical origin [PAL]
#else
   .byte   $05       ; horizontal offset [NTSC]
   .byte   $19       ; vertical origin [NTSC]
```

```
#endif

   .byte   $16       ; video address and colums, $9400 for colour RAM
                     ; bit   function
                     ; ---   --------
                     ;  7   video address va9
                     ; 6-0   number of columns
   .byte   $2E       ; rows and character size
                     ; bit   function
                     ; ---   --------
                     ;  7   b9 raster line
                     ; 6-1   number of rows
                     ;  0   8x16 / 8x8 characters
   .byte   $00       ; raster line
   .byte   $C0       ; video memory addresses, RAM $1000, ROM $8000
                     ; bit   function
                     ; ---   --------
                     ;  7   must be 1
                     ; 6-4   video memory address va12-va10
                     ; 3-0   character memory start address

                     ; 0000 ROM   $8000   set 1 - we use this
                     ; 0001  "   $8400
                     ; 0010  "   $8800 set 2
                     ; 0011  "   $8C00
                     ; 1100 RAM   $1000
                     ; 1101  "   $1400
                     ; 1110  "   $1800
                     ; 1111  "   $1C00

   .byte   $00       ; light pen horizontal position
   .byte   $00       ; light pen vertical position

   .byte   $00       ; paddle X
   .byte   $00       ; paddle Y
   .byte   $00       ; oscillator 1 frequency
   .byte   $00       ; oscillator 2 frequency
   .byte   $00       ; oscillator 3 frequency
   .byte   $00       ; noise source frequency
   .byte   $00       ; aux colour and volume
                     ; bit   function
                     ; ---   --------
                     ; 7-4   auxiliary colour information
                     ; 3-0   volume
   .byte   $1B       ; screen and border colour
                     ; bit   function
                     ; ---   --------
                     ; 7-4   background colour
                     ;  3   inverted or normal mode
                     ; 2-0   border colour
#endif

RUNKEY .byte "LOAD",$0D,"RUN",$0D

; ***************
  Line_Adress_Low
; ***************

#if C64
   .byte $00,$28,$50,$78,$a0,$c8,$f0,$18
   .byte $40,$68,$90,$b8,$e0,$08,$30,$58
   .byte $80,$a8,$d0,$f8,$20,$48,$70,$98
   .byte $c0
#endif
#if VIC
   .byte $00,$16,$2C,$42,$58,$6E,$84,$9A
   .byte $B0,$C6,$DC,$F2,$08,$1E,$34,$4A
   .byte $60,$76,$8C,$A2,$B8,$CE,$E4
#endif

; ***********
  Kernal_TALK
; ***********

   ORA #$40          ; OR with the TALK command
   .byte   $2C       ; skip next 2 bytes

; *************
  Kernal_LISTEN
; *************

   ORA #$20          ; OR with the LISTEN command
   JSR RS232_Stop

; *********************
  IEC_Send_Control_Byte
; *********************

   PHA               ; save device address
   BIT C3PO          ; test deferred character flag
   BPL ISCB_10       ; branch if no defered character
   SEC               ; flag EOI
   ROR TSFCNT        ; rotate into EOI flag byte
#if JIFFY
   JSR Jiffy_Send_Byte
#else
   JSR IEC_Send_Byte
#endif
   LSR C3PO          ; clear deferred character flag
   LSR TSFCNT        ; clear EOI flag

ISCB_10
   PLA               ; device address OR'ed with command
   STA BSOUR         ; save as serial defered character
#if C64
   SEI
#endif
#if JIFFY
   JSR Jiffy_CLR_DAT
#else
   JSR CLR_IEC_DAT   ; set IEC data out high (0)
#endif
   CMP #$3F          ; compare read byte with $3F
   BNE ISCB_20       ; branch if not $3F, this branch will always be taken
   JSR CLR_IEC_CLK   ; set IEC clock out high (0)

ISCB_20
   LDA IEC_DRAN      ; get VIA 1 DRA, no handshake
   ORA #IEC_ATN_BIT  ; set IEC ATN low (1)
   STA IEC_DRAN      ; set VIA 1 DRA, no handshake

; **********************
  IEC_Delay_And_Send_Byte
; **********************
```

```
#if C64
   SEI
#endif
   JSR SET_IEC_CLK   ; set IEC clock out low
   JSR CLR_IEC_DAT   ; set IEC data  out high
   JSR WAIT_1MS      ; 1ms delay

; *************
  IEC_Send_Byte
; *************

   SEI               ; disable interrupts
   JSR CLR_IEC_DAT   ; set serial data out high
   JSR GET_IEC_CLK   ; get serial clock status
#if VIC
   LSR A             ; shift serial data to Cb
#endif
   BCS Device_Not_Present
   JSR CLR_IEC_CLK   ; set serial clock high
   BIT TSFCNT        ; test EOI flag
   BPL ISCB_50       ; branch if not EOI

; I think this is the EOI sequence so the serial clock has been released
; and the serial data is being held low by the peripherals. first up
; wait for the serial data to rise

ISCB_30
   JSR GET_IEC_CLK   ; get serial clock status
#if VIC
   LSR A             ; shift serial data to Cb
#endif
   BCC ISCB_30       ; loop if data low

; now the data is high, EOI is signalled by waiting for at least 200us
; without pulling the serial clock line low again. the listener should
; respond by pulling the serial data line low

ISCB_40
   JSR GET_IEC_CLK   ; get serial clock status
#if VIC
   LSR A             ; shift serial data to Cb
#endif
   BCS ISCB_40       ; loop if data high

; the serial data has gone low ending the EOI sequence, now just wait
; for the serial data line to go high again or, if this isn't an EOI
; sequence, just wait for the serial data to go high the first time

ISCB_50
   JSR GET_IEC_CLK   ; get serial clock status
#if VIC
   LSR A             ; shift serial data to Cb
#endif
   BCC ISCB_50       ; loop if data low

; serial data is high now pull the clock low, preferably within 60us

   JSR SET_IEC_CLK   ; set serial clock low

; now the Vic has to send the eight bits, LSB first. first it sets the
; serial data line to reflect the bit in the byte, then it sets the
; serial clock to high. The serial clock is left high for 26 cycles,
; 23us on a PAL Vic, before it is again pulled low and the serial data
; is allowed high again

; The jiffy routine detecs Jiffy devices within the routine
; Jiffy_Detect_Device and X=2

#if JIFFY
   TXA
   PHA                  ; save X
   LDX #8               ; eight bits to do

ISCB_51
#if C64
   PHA                  ; waste 7 cycles
   PLA
   BIT IEC_DRAN
   BMI ISCB_52          ; IEC clock low (1) ?
#endif
#if VIC
   LDA VIA1_DATN
   AND #2
   BNE ISCB_52          ; IEC clock low (1) ?
#endif
   PLA                  ; no
   TAX                  ; restore X
   JMP IEC_Timeout

ISCB_52
   JSR CLR_IEC_DAT      ; set IEC data high (0)
   ROR BSOUR            ; rotate bit to send into carry
   BCS ISCB_54          ; branch if bit = 1
   JSR SET_IEC_DAT      ; set IEC data low (1)

ISCB_54
   JSR CLR_IEC_CLK      ; set IEC clock high (0)
   LDA IEC_PCR
#if C64
   AND #$df             ; set data high (0)
   ORA #$10             ; set clock low (1)
#endif
#if VIC
   AND #$dd             ; set data high (0)
   ORA #$02             ; set clock low (1)
#endif
   PHP                  ;
   PHA
   JSR Jiffy_Detect_Device
   PLA
   PLP
   DEX
   BNE ISCB_51          ; next bit
   PLA                  ; restore X
   TAX
#if VIC
   NOP
#endif
#else
   LDA #$08             ; eight bits to do
   STA CNTDN            ; set serial bus bit count

ISCB_60
```

```
   LDA IEC_DRAN      ; get VIA 1 DRA, no handshake
   CMP IEC_DRAN      ; compare with self
   BNE ISCB_60       ; loop if changing

#if C64
   ASL A             ; serial clock to carry
#endif
#if VIC
   LSR A             ; serial clock to carry
   LSR A             ; serial data to carry
#endif
   BCC IEC_Timeout
   ROR BSOUR         ; rotate transmit byte
   BCS ISCB_70       ; branch if bit = 1
   JSR SET_IEC_DAT   ; else set serial data out low
   BNE ISCB_80       ; branch always

ISCB_70
   JSR CLR_IEC_DAT   ; set serial data out high

ISCB_80
   JSR CLR_IEC_CLK   ; set serial clock high
   NOP
   NOP
   NOP
   NOP
   LDA IEC_PCR       ; get VIA/CIA PCR
   AND #$DF          ; set CB2 low, serial data out high
   ORA #IEC_CLK_BIT  ; set CA2 high, serial clock out low
   STA IEC_PCR
   DEC CNTDN         ; decrement serial bus bit count
   BNE ISCB_60       ; loop if not all done

; now all eight bits have been sent it's up to the peripheral to signal the byte was
; received by pulling the serial data low. this should be done within one milisecond

#endif
   LDA #4            ; wait for up to about 1ms
   STA IEC_TIM_H     ; set VIA/CIA timer high

ISCB_90
#if C64
   LDA #$19
   STA CIA1_CRB      ; CIA1 Control Register B
   LDA CIA1_ICR      ; CIA1 Interrupt Control Register

ISCB_95
   LDA CIA1_ICR      ; CIA1 Interrupt Control Register
   AND #2
   BNE IEC_Timeout
   JSR GET_IEC_CLK   ; get serial clock status
   BCS ISCB_95
#endif
#if VIC
   LDA IEC_IFR       ; get VIA 2 IFR
   AND #$20          ; mask T2 interrupt
   BNE IEC_Timeout      ; if T2 interrupt do timeout on serial bus
   JSR GET_IEC_CLK   ; get serial clock status
   LSR A             ; shift serial data to Cb
   BCS ISCB_90       ; if data high go wait some more
#endif
   CLI
   RTS

; ==================
  Device_Not_Present
; ==================

   LDA #$80          ; error $80, device not present
   .byte   $2C       ; skip next statement

; ===========
  IEC_Timeout
; ===========

   LDA #$03          ; error $03, write timeout

; ==============
  Set_IEC_Status
; ==============

   JSR Ora_Status
   CLI               ; enable interrupts
   CLC               ; clear for branch
   BCC KeUN_10      ; ATN high, delay, clock high then data high, branch always

; *************
  Kernal_SECOND
; *************

   STA BSOUR         ; save defered byte
   JSR IEC_Delay_And_Send_Byte

; ************
  IEC_ATN_High
; ************

   LDA IEC_DRAN      ; get VIA 1 DRA, no handshake
#if C64
   AND #$F7          ; set serial ATN high
#endif
#if VIC
   AND #$7F          ; set serial ATN high
#endif
   STA IEC_DRAN      ; set VIA 1 DRA, no handshake
   RTS

; ***********
  Kernal_TKSA
; ***********

   STA BSOUR         ; save the secondary address byte to transmit
   JSR IEC_Delay_And_Send_Byte

; ***************
  IEC_Finish_Send
; ***************

   SEI               ; disable interrupts
   JSR SET_IEC_DAT   ; set serial data out low
   JSR IEC_ATN_High  ; set serial ATN high
   JSR CLR_IEC_CLK   ; set serial clock high

IFS_10
```

```
#if JIFFY
#if C64
   BIT IEC_DRAN
   BVS IFS_10
#else
   JSR GET_IEC_CLK
   BCS IFS_10
#endif
#else
   JSR GET_IEC_CLK   ; get serial clock status
#if C64
   BMI IFS_10        ; branch if clock high
#endif
#if VIC
   BCS IFS_10        ; branch if clock high
#endif
#endif               ; JIFFY
   CLI               ; enable interrupts
   RTS

; ************
  Kernal_CIOUT
; ************

   BIT C3PO          ; test deferred character flag
   BMI KeCI_10       ; branch if defered character
   SEC               ; set carry
   ROR C3PO          ; shift into deferred character flag
   BNE KeCI_20       ; save byte and exit, branch always

KeCI_10
   PHA               ; save byte
#if JIFFY
   JSR Jiffy_Send_Byte
#else
   JSR IEC_Send_Byte
#endif
   PLA               ; restore byte

KeCI_20
   STA BSOUR         ; save defered byte
   CLC               ; flag ok
   RTS

; ************
  Kernal_UNTLK
; ************

#if C64
   SEI
#endif
#if !JIFFY
   JSR SET_IEC_CLK
#endif
   LDA IEC_DRAN      ; get VIA 1 DRA, no handshake
   ORA #IEC_ATN_BIT  ; set serial ATN low
   STA IEC_DRAN      ; set VIA 1 DRA, no handshake
#if JIFFY
   JSR SET_IEC_CLK
#endif
   LDA #$5F          ; set the UNTALK command
   .byte   $2C       ; skip next two bytes
```

```
; ************
  Kernal_UNLSN
; ************

   LDA #$3F          ; set the UNLISTEN command
   JSR IEC_Send_Control_Byte

KeUN_10
   JSR IEC_ATN_High  ; set serial ATN high

; ****************************
  IEC_Delay_CLK_High_DATA_High
; ****************************

   TXA               ; save device number
#if C64
   LDX #10           ; short delay
#endif
#if VIC
   LDX #11           ; short delay
#endif
IDel_10
   DEX               ; decrement count
   BNE IDel_10       ; loop if not all done
   TAX               ; restore device number
   JSR CLR_IEC_CLK   ; set serial clock high
   JMP CLR_IEC_DAT   ; set serial data out high and return

; ************
  Kernal_ACPTR
; ************

#if JIFFY
   JMP Jiffy_ACPTR
#else
   SEI               ; disable interrupts
   LDA #0
#endif

KeAC_03
   STA CNTDN         ; clear serial bus bit count
   JSR CLR_IEC_CLK   ; set serial clock high

KeAC_05
   JSR GET_IEC_CLK   ; get serial clock status
#if C64
   BPL KeAC_05       ; loop while clock low
#endif
#if VIC
   BCC KeAC_05       ; loop while clock low
   JSR CLR_IEC_DAT   ; set serial data out high
#endif

KeAC_10
   LDA #$01          ; set timeout count high byte
   STA IEC_TIM_H

KeAC_15
#if C64
   LDA #$19
   STA CIA1_CRB      ; CIA1 Control Register B
   JSR CLR_IEC_DAT   ; set serial data out high
   LDA CIA1_ICR      ; CIA1 Interrupt Control Register
KeAC_20
#endif

   LDA IEC_IFR       ; get VIA 2 IFR
   AND #IEC_IFR_BIT  ; mask T2 interrupt
   BNE KeAC_25       ; branch if T2 interrupt
   JSR GET_IEC_CLK   ; get serial clock status

#if C64
   BMI KeAC_20       ; loop if clock high
   BPL KeAC_35       ; else go se 8 bits to do, branch always
#endif
#if VIC
   BCS KeAC_15       ; loop if clock high
   BCC KeAC_35       ; else go se 8 bits to do, branch always
#endif

KeAC_25
   LDA CNTDN         ; get serial bus bit count
   BEQ KeAC_30       ; if not already EOI then go flag EOI
   LDA #$02          ; error $02, read timeour
   JMP Set_IEC_Status

KeAC_30
   JSR SET_IEC_DAT    ; set serial data out low

#if C64
   JSR CLR_IEC_CLK
#endif
#if VIC
   JSR IEC_Delay_CLK_High_DATA_High      ; 1ms delay, clock high then data high
#endif
   LDA #$40          ; set EOI
   JSR Ora_Status
   INC CNTDN         ; increment serial bus bit count, do error on next timeout
   BNE KeAC_10       ; go try again

KeAC_35
   LDA #$08          ; 8 bits to do
   STA CNTDN         ; set serial bus bit count

KeAC_40
   LDA IEC_DRAN      ; get VIA 1 DRA, no handshake
   CMP IEC_DRAN      ; compare with self
   BNE KeAC_40       ; loop if changing

#if C64
   ASL A             ; serial clock into carry
   BPL KeAC_40       ; loop while serial clock low
#endif
#if VIC
   LSR A             ; serial clock into carry
   BCC KeAC_40       ; loop while serial clock low
   LSR A             ; serial data into carry
#endif

   ROR TBTCNT        ; shift data bit into receive byte
KeAC_45
   LDA IEC_DRAN      ; get VIA 1 DRA, no handshake
   CMP IEC_DRAN      ; compare with self
   BNE KeAC_45       ; loop if changing

#if C64
   ASL A             ; serial clock into carry
   BMI KeAC_45       ; loop while serial clock high
#endif
#if VIC
   LSR A             ; serial clock into carry
   BCS KeAC_45       ; loop while serial clock high
#endif

   DEC CNTDN         ; decrement serial bus bit count
   BNE KeAC_40       ; loop if not all done
   JSR SET_IEC_DAT   ; set serial data out low

#if C64
   BIT STATUS        ; get serial status byte
   BVC KeAC_50       ; branch if no error
#endif
#if VIC
   LDA STATUS        ; get serial status byte
   BEQ KeAC_50       ; branch if no error
#endif

   JSR IEC_Delay_CLK_High_DATA_High

KeAC_50
   LDA TBTCNT        ; get receive byte
   CLI               ; enable interrupts
   CLC
   RTS

; ***********
  CLR_IEC_CLK        ; set serial clock high (clear bit)
; ***********

   LDA IEC_PCR
   AND #~IEC_CLK_BIT
   STA IEC_PCR
   RTS

; ***********
  SET_IEC_CLK        ; set serial clock low (set bit)
; ***********

   LDA IEC_PCR
   ORA #IEC_CLK_BIT
   STA IEC_PCR
   RTS

#if C64

; ***********
  CLR_IEC_DAT        ; set serial data high (clear bit)
; ***********

   LDA IEC_PCR
   AND #~IEC_DAT_BIT
   STA IEC_PCR
   RTS

; ***********
```

```
  SET_IEC_DAT        ; set serial data low (set bit)
; ***********

   LDA IEC_PCR
   ORA #IEC_DAT_BIT
   STA IEC_PCR
   RTS

; ***********
  GET_IEC_CLK        ; set serial clock
; ***********

   LDA IEC_PCR
   CMP IEC_PCR
   BNE GET_IEC_CLK
   ASL A             ; C = data   bit 7 = clock
   RTS

#endif

; ********
  WAIT_1MS           ; wait one millesecond
; ********

#if C64
   TXA
   LDX #$B8

W1MS_10
   DEX
   BNE W1MS_10
   TAX
   RTS
#endif

#if VIC
   LDA #$04          ; set for 1024 cycles
   STA VIA2_T2CH     ; set VIA 2 T2C_h

W1MS_20
   LDA IEC_IFR       ; get VIA 2 IFR
   AND #$20          ; mask T2 interrupt
   BEQ W1MS_20      ; loop until T2 interrupt
   RTS
#endif

; *****************
  RS232_NMI_Transmit
; *****************

   LDA BITTS         ; get RS232 bit count
   BEQ RS232_Setup_Next_Byte_To_Send
   BMI RTra_40       ; if negative go do stop bit(s)
   LSR RODATA        ; shift RS232 output byte buffer
   LDX #$00          ; set $00 for bit = 0
   BCC RTra_05       ; branch if bit was 0
   DEX               ; set $FF for bit = 1

RTra_05
   TXA               ; copy bit to A
   EOR ROPRTY        ; EOR with RS232 parity byte
   STA ROPRTY        ; save RS232 parity byte
   DEC BITTS         ; decrement RS232 bit count
   BEQ RTra_15       ; if RS232 bit count now zero go do parity bit

RTra_10
   TXA               ; copy bit to A
   AND #RS232_C_BIT  ; mask for CB2 control bit
   STA NXTBIT        ; save RS232 next bit to send
   RTS

RTra_15
   LDA #$20          ; mask 00x0 0000, parity enable bit
   BIT M51CDR        ; test pseudo 6551 command register
   BEQ RTra_30       ; branch if parity disabled
   BMI RTra_35       ; branch if fixed mark or space parity
   BVS RTra_32       ; branch if even parity
   LDA ROPRTY        ; get RS232 parity byte
   BNE RTra_25       ; if parity not zero leave parity bit = 0

RTra_20
   DEX               ; make parity bit = 1

RTra_25
   DEC BITTS         ; decrement RS232 bit count, 1 stop bit
   LDA M51CTR        ; get pseudo 6551 control register
   BPL RTra_10       ; if 1 stop bit save parity bit and exit
   DEC BITTS         ; decrement RS232 bit count, 2 stop bits
   BNE RTra_10       ; save bit and exit, branch always

RTra_30
   INC BITTS         ; increment RS232 bit count, = -1 stop bit
   BNE RTra_20       ; set stop bit = 1 and exit

RTra_32
   LDA ROPRTY        ; get RS232 parity byte
   BEQ RTra_25       ; if parity zero leave parity bit = 0
   BNE RTra_20       ; else make parity bit = 1, branch always

RTra_35
   BVS RTra_25       ; if fixed space parity leave parity bit = 0
   BVC RTra_20       ; else fixed mark parity make parity bit = 1, branch always

; decrement stop bit count, set stop bit = 1 and exit. $FF is one stop
; bit, $FE is two stop bits

RTra_40
   INC BITTS         ; decrement RS232 bit count
   LDX #$FF          ; set stop bit = 1
   BNE RTra_10       ; save stop bit and exit, branch always

; *****************************
  RS232_Setup_Next_Byte_To_Send
; *****************************

   LDA M51CDR        ; get 6551 pseudo command register
   LSR A             ; handshake bit inot Cb
   BCC RSNB_10       ; branch if 3 line interface

#if C64
   BIT RS2_DSR_CTS
#endif
#if VIC
   BIT $9120         ; test VIA 2 DRB, this is wrong
```

```
#endif

   BPL RS232_No_DSR_Signal
   BVC RNDS_10      ; if CTS = 0 set CTS signal not present and exit

RSNB_10
   LDA #0
   STA ROPRTY        ; clear RS232 parity byte
   STA NXTBIT        ; clear RS232 next bit to send
   LDX BITNUM        ; get number of bits to be sent/received
   STX BITTS         ; set RS232 bit count
   LDY RODBS         ; get index to Tx buffer start
   CPY RODBE         ; compare with index to Tx buffer end
   BEQ RNDS_20      ; if all done go disable T1 interrupt and return
   LDA (TXPTR),Y     ; else get byte from buffer
   STA RODATA        ; save to RS232 output byte buffer
   INC RODBS         ; increment index to Tx buffer start
   RTS

; ******************
  RS232_No_DSR_Signal
; ******************

   LDA #$40          ; set DSR signal not present
   .byte   $2C

RNDS_10
   LDA #$10          ; set CTS signal not present
   ORA RSSTAT        ; OR with RS232 status register
   STA RSSTAT        ; save RS232 status register

; disable T1 interrupt

RNDS_20

#if C64
   LDA #1

RNDS_30
   STA CIA2_ICR
   EOR ENABL
   ORA #$80
   STA ENABL
   STA CIA2_ICR
#endif
#if VIC
   LDA #$40          ; disable T1 interrupt
   STA VIA1_IER      ; set VIA 1 IER
#endif

   RTS

; ******************
  RS232_Set_Data_Bits
; ******************

   LDX #$09          ; set bit count to 9, 8 data + 1 stop bit
   LDA #$20          ; mask for 8/7 data bits
   BIT M51CTR        ; test pseudo 6551 control register
   BEQ RSDB_10       ; branch if 8 bits
   DEX               ; else decrement count for 7 data bits
```

```
RSDB_10
   BVC RSDB_20       ; branch if 7 bits
   DEX               ; else decrement count ..
   DEX               ; .. for 5 data bits

RSDB_20
   RTS

; *****************
  RS232_NMI_Receive
; *****************

   LDX RINONE        ; get start bit check flag
   BNE RRec_25       ; branch if no start bit received
   DEC BITCI         ; decrement receiver bit count in
   BEQ RRec_30
   BMI RRec_15
   LDA INBIT
   EOR RIPRTY
   STA RIPRTY
   LSR INBIT         ; shift receiver input bit temporary storage
   ROR RIDATA

RRec_05
   RTS

RRec_10
   DEC BITCI         ; decrement receiver bit count in

RRec_15
   LDA INBIT         ; get receiver input bit temporary storage
   BEQ RRec_65
   LDA M51CTR        ; get pseudo 6551 control register
   ASL A
   LDA #$01
   ADC BITCI         ; add receiver bit count in
   BNE RRec_05

RRec_20
   LDA #$90          ; enable CB1 interrupt
   STA RS2_IRQ_REG   ; set VIA 1 IER
#if C64
   ORA ENABL
   STA ENABL
   STA RINONE
   LDA #2
   JMP RNDS_30
#endif
#if VIC
   STA RINONE        ; set start bit check flag, set no start bit received
   LDA #$20          ; disable T2 interrupt
   STA RS2_IRQ_REG   ; set VIA 1 IER
   RTS
#endif

RRec_25
   LDA INBIT         ; get receiver input bit temporary storage
   BNE RRec_20

#if C64
   JMP Je4d3
#endif
```

```
#if VIC
   STA RINONE        ; set start bit check flag, set start bit received
   RTS
#endif

RRec_30
   LDY RIDBE         ; get index to Rx buffer end
   INY
   CPY RIDBS         ; compare with index to Rx buffer start
   BEQ RRec_50       ; if buffer full go do Rx overrun error
   STY RIDBE         ; save index to Rx buffer end
   DEY               ; decrement index
   LDA RIDATA        ; get assembled byte
   LDX BITNUM        ; get bit count

RRec_35
   CPX #$09          ; compare with byte + stop
   BEQ RRec_40       ; branch if all nine bits received
   LSR A             ; else shift byte
   INX               ; increment bit count
   BNE RRec_35       ; loop, branch always

RRec_40
   STA (RXPTR),Y     ; save received byte to Rx buffer
   LDA #$20          ; mask 00x0 0000, parity enable bit
   BIT M51CDR        ; test pseudo 6551 command register
   BEQ RRec_10       ; branch if parity disabled
   BMI RRec_05       ; branch if mark or space parity
   LDA INBIT         ; get receiver input bit temporary storage
   EOR RIPRTY
   BEQ RRec_45
   BVS RRec_05
   .byte   $2C

RRec_45
   BVC RRec_05
   LDA #$01          ; set Rx parity error
   .byte   $2C

RRec_50
   LDA #$04          ; set Rx overrun error
   .byte   $2C

RRec_55
   LDA #$80          ; Rx break error
   .byte   $2C

RRec_60
   LDA #$02          ; Rx frame error
   ORA RSSTAT        ; OR with RS232 status byte
   STA RSSTAT        ; save RS232 status byte
   JMP RRec_20

RRec_65
   LDA RIDATA
   BNE RRec_60      ; if ?? do frame error
   BEQ RRec_55      ; else do break error, branch always

RRec_70
#if VIC
   JMP Illegal_Jiffy_Device
#endif
```

```
; ============
  RS232_CHKOUT
; ============

   STA DFLTO         ; save output device number
   LDA M51CDR        ; get pseudo 6551 command register
   LSR A             ; shift handshake bit to carry
   BCC RSSN_10       ; branch if 3 line interface
   LDA #$02          ; mask for RTS out
   BIT RS2_DSR_CTS   ; test VIA 1 DRB
   BPL RS232_Set_Status_No_Signal
   BNE RSSN_10       ; if RTS = 1 just exit

RCHO_10
#if C64
   LDA ENABL
   AND #2
#endif
#if VIC
   LDA VIA1_IER      ; get VIA 1 IER
   AND #$30          ; mask 00xx 0000, T2 and CB1 interrupts
#endif
   BNE RCHO_10       ; loop while either enabled

RCHO_20
   BIT RS2_DSR_CTS   ; test VIA 1 DRB
   BVS RCHO_20       ; loop while CTS high
   LDA RS2_DSR_CTS   ; get VIA 1 DRB
   ORA #$02          ; set RTS high
   STA RS2_DSR_CTS   ; save VIA 1 DRB

RCHO_30
   BIT RS2_DSR_CTS   ; test VIA 1 DRB
   BVS RSSN_10       ; exit if CTS high
   BMI RCHO_30       ; loop while DSR high

; **************************
  RS232_Set_Status_No_Signal
; **************************

#if C64
   LDA #$40
   STA RSSTAT
#endif
#if VIC
   JSR RS232_No_DSR_Signal
#endif
RSSN_10
   CLC               ; flag ok
   RTS

; ************************
  RS232_Put_Byte_To_Buffer
; ************************

#if C64
   JSR RPBB_20

RPBB_10
#endif
   LDY RODBE         ; get index to Tx buffer end
   INY               ; + 1
   CPY RODBS         ; compare with index to Tx buffer start
   BEQ RS232_Put_Byte_To_Buffer
   STY RODBE         ; set index to Tx buffer end
   DEY               ; index to available buffer byte
#if C64
   LDA PTR1
#endif
   STA (TXPTR),Y     ; save byte to buffer

RPBB_20
#if C64
   LDA ENABL
   LSR A
   BCS RPBB_40
   LDA #$10
   STA CIA2_CRA
#endif
#if VIC
   BIT VIA1_IER      ; test VIA 1 IER
   BVC RPBB_30       ; branch if T1 not enabled
   RTS
#endif

RPBB_30
   LDA BAUDOF        ; get baud rate bit time low byte
   STA RS2_TIM_LOW   ; set VIA 1 T1C_l
   LDA BAUDOF+1      ; get baud rate bit time high byte
   STA RS2_TIM_HIG   ; set VIA 1 T1C_h
#if C64
   LDA #$81
   JSR RNDS_30
   JSR RS232_Setup_Next_Byte_To_Send
   LDA #$11
   STA CIA2_CRA

RPBB_40
   RTS
#endif
#if VIC
   LDA #$C0          ; enable T1 interrupt
   STA VIA1_IER      ; set VIA 1 IER
   JMP RS232_Setup_Next_Byte_To_Send
#endif

; ===========
  RS232_CHKIN
; ===========

   STA DFLTN         ; save input device number
   LDA M51CDR        ; get pseudo 6551 command register
   LSR A
   BCC RCHI_40       ; branch if 3 line interface
   AND #$08          ; mask duplex bit, pseudo 6551 command is >> 1
   BEQ RCHI_40       ; branch if full duplex
   LDA #2
   BIT RS2_DSR_CTS   ; test VIA 1 DRB
   BPL RS232_Set_Status_No_Signal
   BEQ RCHI_50

RCHI_10
#if C64
```

```
   LDA ENABL
   LSR A
   BCS RCHI_10
#endif
#if VIC
   BIT VIA1_IER      ; test VIA 1 IER
   BVS RCHI_10       ; loop while T1 interrupt enabled
#endif
   LDA RS2_DSR_CTS   ; get VIA 1 DRB
   AND #$FD          ; mask xxxx xx0x, clear RTS out
   STA RS2_DSR_CTS   ; save VIA 1 DRB

RCHI_20
   LDA RS2_DSR_CTS   ; get VIA 1 DRB
   AND #$04          ; mask xxxx x1xx, DTR
   BEQ RCHI_20       ; loop while DTR low

RCHI_30
   LDA #$90          ; enable CB1 interrupt
#if C64
   CLC
   JMP RNDS_30

RCHI_40
   LDA ENABL
   AND #$12
   BEQ RCHI_30
#endif
#if VIC
   STA VIA1_IER      ; set VIA 1 IER
#endif

RCHI_50
   CLC
   RTS

#if VIC
RCHI_40
   LDA VIA1_IER      ; get VIA 1 IER
   AND #$30          ; mask 0xx0 0000, T1 and T2 interrupts
   BEQ RCHI_30       ; if both interrupts disabled go enable CB1
   CLC
   RTS
#endif

; **************************
  RS232_Get_Byte_From_Buffer
; **************************

#if C64
   LDA RSSTAT
#endif
   LDY RIDBS         ; get index to Rx buffer start
   CPY RIDBE         ; compare with index to Rx buffer end
   BEQ RGBB_10       ; return null if buffer empty
#if C64
   AND #$F7
   STA RSSTAT
#endif
   LDA (RXPTR),Y     ; get byte from Rx buffer
   INC RIDBS         ; increment index to Rx buffer start
   RTS
```

```
RGBB_10
#if C64
   ORA #8
   STA RSSTAT
#endif
   LDA #$00          ; return null
   RTS

; **********
  RS232_Stop
; **********

   PHA
#if C64
   LDA ENABL
   BEQ RSTP_30

RSTP_10
   LDA ENABL
   AND #3
   BNE RSTP_10
#endif
#if VIC
   LDA RS2_IRQ_REG   ; get VIA 1 IER
   BEQ RSTP_30       ; branch if no interrupts enabled. this branch will

RSTP_20
   LDA RS2_IRQ_REG   ; get VIA 1 IER
   AND #$60          ; mask 0xx0 0000, T1 and T2 interrupts
   BNE RSTP_20       ; loop if T1 or T2 active
#endif
   LDA #$10          ; disable CB1 interrupt
   STA RS2_IRQ_REG   ; set VIA 1 IER
#if C64
   LDA #0
   STA ENABL
#endif

RSTP_30
   PLA
   RTS

Msg_Start     .byte "\rI/O ERROR #"^
Msg_Searching .byte "\rSEARCHING "^
Msg_FOR       .byte "FOR "^

#if JIFFY
#if C64
; *******************
  Jiffy_Clear_Sprites
; *******************

   LDA #0
   STA VIC_SPR_ENA

JCS_10
   ADC #1
   BNE JCS_10
   RTS
#endif
```

```
; ****************
  Jiffy_CHKIN_PTR2
; ****************

   LDA PTR2

; ***********
  Jiffy_CHKIN
; ***********

   PHA
   JSR CLRCHN
   PLA
   TAX
   JMP CHKIN

; *************
  Jiffy_CLR_DAT
; *************

#if VIC
   SEI
#endif
   LDA #0
   STA TSFCNT        ; clear jiffy flag
   JMP CLR_IEC_DAT   ; continue with IEC routine

; ************************
  Jiffy_Send_Drive_Command
; ************************

   TXA
   PHA
   JSR JiDi_60
   PLA
   TAX
Bf0fb
   LDA Jiffy_Transfer_01,X
   JSR CHROUT
   INX
   DEY
   BNE Bf0fb
   RTS

#if VIC
; **************
  Back_To_Prompt
; **************

   TXA
   BMI JAAB_10
   JMP Default_Error

JAAB_10
   JMP Basic_Ready
#endif

#else
Msg_Play      .byte "\rPRESS PLAY ON TAPE"^
Msg_Record    .byte "PRESS RECORD & PLAY ON TAPE"^
#endif
Msg_Loading   .byte "\rLOADING"^
```

```
Msg_Saving    .byte "\rSAVING "^
Msg_Verifying .byte "\rVERIFYING"^
Msg_Found     .byte "\rFOUND "^
Msg_ok        .byte "\rOK\r"^

; ==================
  Display_Direct_Msg
; ==================

   BIT MSGFLG        ; test message mode flag
   BPL DDM_10        ; exit if control messages off

; *************************
  Display_Kernal_IO_Message
; *************************

   LDA Msg_Start,Y    ; get byte from message table
   PHP               ; save status
   AND #$7F          ; clear b7
   JSR CHROUT        ; Output a character
   INY
   PLP               ; restore status
   BPL Display_Kernal_IO_Message

DDM_10
   CLC
   RTS

; ************
  Kernal_GETIN
; ************

   LDA DFLTN         ; get input device number
   BNE KeGE_10       ; branch if not keyboard
   LDA NDX           ; get keyboard buffer length
#if C64
   BEQ KeGE_20
#endif
#if VIC
   BEQ IGB_20      ; if buffer empty go flag no byte and return
#endif
   SEI               ; disable inter/rupts
   JMP Get_Char_From_Keyboard_Buffer

KeGE_10
   CMP #$02          ; compare device with RS232 device
   BNE CHRI_05      ; branch if not RS232 device

; **************
  RS232_Get_Byte
; **************

   STY TEMPX         ; save Y
   JSR RS232_Get_Byte_From_Buffer
   LDY TEMPX         ; restore Y

KeGE_20
   CLC               ; flag no error
   RTS

; ************
  Kernal_CHRIN       ; Get a character from the input channel
```

```
; ************

   LDA DFLTN         ; get input device number
#if JIFFY
   BNE Jiffy_f1a9
#else
   BNE CHRI_05       ; if it's not the keyboard continue
#endif
   LDA CSRIDX        ; get cursor column
   STA ICRCOL        ; set input cursor column
   LDA TBLX          ; get cursor row
   STA ICRROW        ; set input cursor row
   JMP CHRIN_Keyboard_Or_Screen

; the input device was not the keyboard

CHRI_05
   CMP #$03          ; compare device number with screen
   BNE CHRI_10       ; if it's not the screen continue

CHRI_07
   STA INSRC         ; input from keyboard or screen, $xx = screen,
   LDA LINLEN        ; get current screen line length
   STA INDX          ; save input [EOL] pointer
   JMP CHRIN_Keyboard_Or_Screen

CHRI_10
   BCS IEC_Get_Byte

; the input device is < the screen do must be the RS232 or tape device

   CMP #$02          ; compare device with RS232 device
   BEQ RS232_Read_Byte

; else there's only the tape device left ..

#if JIFFY
; *********************
  Jiffy_ACPTR_Load_Check
; *********************

   JSR Jiffy_ACPTR   ; get byte from serial bus
   PHA               ; temp store on stack
   BIT TSFCNT        ; test bit6, if serial device is a JiffyDOS device
   BVC JALC_20       ; no JiffyDOS device
   CPX #0
   BNE JALC_10
   LDA MEMUSS+1      ; load address high byte

JALC_10
   CMP #4            ; load address < $0400 ?
   BCC JALC_20       ; don't load with Jiffy
   LDY #0
   LDA (FNADR),Y     ; get first character of filename
   CMP #'$'          ; is it a directory load ?
   BEQ JALC_20       ; yes, don't load with Jiffy
   INC SA
   JSR Jiffy_Talk_TkSA
   DEC SA
   ASL TSFCNT        ; continue with Jiffy load

JALC_20
```

```
   PLA               ; recover read byte
   RTS

; ***************
  Flag_Read_Error
; ***************

  LDA #16
  JMP Ora_Status

; *****
  Vf1a3
; *****

  .WORD Back_To_Prompt
  .WORD Default_Warmstart
  .WORD Default_Tokenize

; **********
  Jiffy_f1a9
; **********

   CMP #4
   BCC CHRI_05

#else
   STX TEMPX         ; save X
   JSR TAPE_Get_Byte
   BCS CHRI_25       ; exit if error
   PHA               ; save byte
   JSR TAPE_Get_Byte
   BCS CHRI_20       ; exit if error
   BNE CHRI_15       ; branch if end reached
   LDA #$40          ; set [EOF] bit
   JSR Ora_Status

CHRI_15
   DEC BUFPNT        ; decrement tape buffer index
   LDX TEMPX         ; restore X
   PLA               ; restore saved byte
   RTS

; error exit from input character

CHRI_20
   TAX               ; copy error byte ??
   PLA               ; dump saved byte
   TXA               ; restore error byte ??
CHRI_25
   LDX TEMPX         ; restore X
   RTS

; *************
  TAPE_Get_Byte
; *************

   JSR TAPE_Advance_Buffer_Pointer
   BNE TGB_05        ; if not end get next byte and exit
   JSR TAPE_Init_Read
   BCS IGB_30      ; exit if error flagged
   LDA #0
   STA BUFPNT        ; clear tape buffer index
   BEQ TAPE_Get_Byte ; branch always

TGB_05
   LDA (TAPE1),Y     ; get next byte from buffer
   CLC               ; flag no error
   RTS

#endif               ; JIFFY

; * = $f1ad

; ============
  IEC_Get_Byte
; ============

   LDA STATUS        ; get serial status byte
   BEQ IGB_40        ; if no errors flagged go input byte and return

IGB_10
   LDA #$0D          ; else return [EOL]

IGB_20
   CLC               ; flag no error

IGB_30
   RTS

IGB_40
#if JIFFY
   JMP Jiffy_ACPTR   ; input a byte from the serial bus and return
#else
   JMP Kernal_ACPTR
#endif

; ===============
  RS232_Read_Byte
; ===============

   JSR RS232_Get_Byte
#if C64
   BCS IGB_30
#endif
#if VIC
   BCS RRB_Ret       ; branch if error, this doesn't get taken as the last
#endif
   CMP #$00          ; compare with null
#if C64
   BNE IGB_20
   LDA RSSTAT
   AND #$60
   BNE IGB_10
#endif
   BEQ RS232_Read_Byte      ; loop if null
#if VIC
   CLC               ; flag no error
RRB_Ret
   RTS
#endif

; *************
  Kernal_CHROUT      ; Output a character
; *************
```

```
   PHA                ; save the character to send
   LDA DFLTO          ; get output device number
   CMP #$03           ; compare device number with screen
   BNE KeCO_10        ; if output device not screen continue

; the output device is the screen

   PLA                ; restore character to send
   JMP Screen_CHROUT ; output character and return

; the output device was not the screen

KeCO_10
   BCC KeCO_20        ; if output device < screen continue

; the output device was > screen so it is a serial bus device

   PLA                ; restore character to send
   JMP Kernal_CIOUT  ; output a byte to the serial bus and return

; the output device is < screen

KeCO_20
#if C64
   LSR A
#endif
#if VIC
   CMP #$02           ; compare the device with RS232 device
   BEQ RS232_Send_Byte
#endif

; else the output device is the cassette

   PLA                ; restore the character to send

; **************
  TAPE_Send_Byte
; **************

#if ![VIC & JIFFY]
   STA PTR1           ; save character to character buffer
#if VIC
   PHA                ; save A
#endif
   TXA                ; copy X
   PHA                ; save X
   TYA                ; copy Y
   PHA                ; save Y
#endif
#if C64
   BCC RS232_Send_Byte
#endif
#if JIFFY
#if C64
   JMP Pop_Dev_Not_Present
#else
   JMP Dev_Not_Present
#endif

; *****************
  Jiffy_Disk_Status
```

```
; ****************

   JSR Jiffy_Open_Command_10
   JSR Jiffy_CHRIN
   CMP #'0'          ; first char of disk status
   RTS

; ***********************
  Jiffy_Set_Default_Device
; ***********************

   JSR Get_Next_Byte_Value
   STX FA
   JSR Jiffy_Validate_FA
   STX Jiffy_Device
   RTS
#else
   JSR TAPE_Advance_Buffer_Pointer
   BNE TASB_10       ; if not end save next byte and exit
   JSR Init_Tape_Write
   BCS TASB_30       ; exit if error
   LDA #$02          ; set data block type ??
   LDY #0
   STA (TAPE1),Y     ; save type to buffer ??
   INY
   STY BUFPNT        ; save tape buffer index

TASB_10
   LDA PTR1          ; restore character from character buffer
   STA (TAPE1),Y     ; save to buffer

#endif
TASB_20
#if JIFFY & VIC
   LDA #$fb
   STA KEYB_COL
   LDX KEYB_ROW
   LDA #$f7
   STA KEYB_COL
   JMP Jiffy_e9c6
   .byte 0
#else
   CLC               ; flag no error

TASB_30
   PLA               ; pull Y
   TAY               ; restore Y
   PLA               ; pull X
   TAX               ; restore X
#if C64
   LDA PTR1
#endif
#if VIC
   PLA               ; restore A
#endif
   BCC TASB_40       ; exit if no error
   LDA #$00          ; else clear A
#endif

TASB_40
   RTS
```

```
; ===============
  RS232_Send_Byte
; ===============

#if C64
   JSR RPBB_10
   JMP TASB_20
#endif
#if VIC
   PLA                ; restore character to send
   STX TEMPX          ; save X
   STY PTR1           ; save Y
   JSR RS232_Put_Byte_To_Buffer
   LDX TEMPX          ; restore Y
   LDY PTR1           ; restore X
   CLC                ; flag ok
   RTS
#endif

; *************
  Kernal_ICHKIN
; *************

   JSR Find_File_X
   BEQ KICH_10        ; branch if file opened
   JMP File_Not_Open

KICH_10
   JSR Get_LFS
   LDA FA             ; get device number
   BEQ KICH_30        ; if device was keyboard save device #, flag ok and exit
   CMP #$03           ; compare device number with screen
   BEQ KICH_30        ; if device was screen save device #, flag ok and exit
   BCS KICH_40        ; branch if serial bus device
   CMP #$02           ; compare device with RS232 device
   BNE KICH_20        ; branch if not RS 232 device
   JMP RS232_CHKIN

KICH_20
   LDX SA             ; get secondary address
   CPX #$60
   BEQ KICH_30
   JMP Not_Input_File

KICH_30
   STA DFLTN          ; save input device number
   CLC                ; flag ok
   RTS

KICH_40               ; CHKIN for IEC device
   TAX                ; copy device number to X
   JSR Kernal_TALK    ; command a serial bus device to TALK
   LDA SA             ; get secondary address
   BPL KICH_50
   JSR IEC_Finish_Send
   JMP KICH_60

KICH_50
   JSR Kernal_TKSA    ; send secondary address after TALK

KICH_60
   TXA                ; copy device back to A
   BIT STATUS        ; test serial status byte
   BPL KICH_30       ; if device present save device number and exit
   JMP Dev_Not_Present      ; do device not present error and return

; *************
  Kernal_CHKOUT      ; Open a channel for output
; *************

   JSR Find_File_X
   BEQ KCHO_10       ; branch if file found
   JMP File_Not_Open

KCHO_10
   JSR Get_LFS
   LDA FA            ; get device number
   BNE KCHO_30       ; branch if device is not keyboard

KCHO_20
   JMP Not_Output_File

KCHO_30
   CMP #$03          ; compare device number with screen
   BEQ KCHO_50       ; if screen save output device number and exit
   BCS KCHO_60       ; branch if > screen, serial bus device
   CMP #$02          ; compare device with RS232 device
   BNE KCHO_40       ; branch if not RS232 device, must be tape
   JMP RS232_CHKOUT

KCHO_40
   LDX SA            ; get secondary address
   CPX #$60
   BEQ KCHO_20       ; if ?? do not output file error and return

KCHO_50
   STA DFLTO         ; save output device number
   CLC               ; flag ok
   RTS

KCHO_60
   TAX               ; copy device number
   JSR Kernal_LISTEN ; command devices on the serial bus to LISTEN
   LDA SA            ; get secondary address
   BPL KCHO_70       ; branch if address to send
   JSR IEC_ATN_High
   BNE KCHO_80       ; branch always

KCHO_70
   JSR Kernal_SECOND ; send secondary address after LISTEN

KCHO_80
   TXA               ; copy device number back to A
   BIT STATUS        ; test serial status byte
   BPL KCHO_50       ; if device present save output device number and exit
   JMP Dev_Not_Present      ; else do device not present error and return

; ************
  Kernal_CLOSE
; ************

   JSR Find_File_A
   BEQ KeCL_10       ; if the file is found go close it
   CLC               ; else the file was closed so just flag ok
   RTS

KeCL_10
   JSR Get_LFS
   TXA               ; copy file index to A
   PHA               ; save file index
   LDA FA            ; get device number
   BEQ KeCL_80       ; if $00, keyboard, restore index and close file
   CMP #3            ; compare device number with screen
   BEQ KeCL_80       ; if screen restore index and close file
   BCS KeCL_70       ; if > screen go do serial bus device close
   CMP #2            ; compare device with RS232 device
   BNE KeCL_40       ; branch if not RS232 device
   PLA               ; restore file index
   JSR Close_File
#if C64
   JSR ROPN_50
#endif
#if VIC
   LDA #$7D          ; disable T1, T2, CB1, CB2, SR and CA2
   STA RS2_IRQ_REG   ; set VIA 1 IER
   LDA #$06          ; set DTR and RTS high
   STA RS2_DSR_CTS   ; set VIA 1 DRB
   LDA #$EE          ; CB2 high, CB1 negative edge, CA2 high, CA1 negative edge
   STA VIA1_PCR      ; set VIA 1 PCR
#endif
   JSR Read_Memtop
   LDA RXPTR+1       ; get RS232 input buffer pointer high byte
   BEQ KeCL_20       ; branch if no RS232 input buffer
   INY               ; else reclaim RS232 input buffer memory

KeCL_20
   LDA TXPTR+1       ; get RS232 output buffer pointer high byte
   BEQ KeCL_30       ; branch if no RS232 output buffer
   INY               ; else reclaim RS232 output buffer memory

KeCL_30
   LDA #0
   STA RXPTR+1       ; clear RS232 input buffer pointer high byte
   STA TXPTR+1       ; clear RS232 output buffer pointer high byte
   JMP ROPN_45       ; go set top of memory and exit

KeCL_40
#if JIFFY
   PLA
   JMP Illegal_Jiffy_Device

; **************
  Jiffy_Close_15
; **************

   JSR CLRCHN

Jiffy_Close_6f
   LDA #$6f
   JSR Find_File_A
   BNE ClFi_Ret
   JMP ClFi_05

#if C64
; *****************
  Jiffy_Test_Device
```

```
; *****************

   STX FA

; *************
  Jiffy_Test_FA
; *************

   TYA
   PHA
   JSR Jiffy_Open_Command_Channel    ; open 15,x,15
   JSR JiDi_60    ; set command channel (15) as output
   PHP
   JSR Jiffy_Close_15
   PLP
   PLA
#else
JTABC
   .byte $00,$20,$00,$20,$02,$22,$02,$22
   .byte $00,$20,$00,$20,$02,$22,$02,$22
#endif
#if C64
   TAY
   LDX FA
   RTS
   .byte $f2
#endif
#else
   LDA SA            ; get secondary address
   AND #$0F
   BEQ KeCL_80
   JSR TAPE_Get_Buffer_Address
   LDA #0
#if C64
   SEC
#endif
   JSR TAPE_Send_Byte
#if C64
   JSR Init_Tape_Write
   BCC KeCL_60
   PLA
   LDA #0
   RTS
#endif
#if VIC
   JMP Close_Patch   ; go do CLOSE tail
KeCL_50
   BCS ClFi_Ret      ; just exit if error
#endif

KeCL_60
   LDA SA            ; get secondary address
   CMP #$62
   BNE KeCL_80
   LDA #$05          ; set logical end of the tape
   JSR TAPE_Write_Header
   JMP KeCL_80       ; restore index and close file
#endif               ; JIFFY

KeCL_70
   JSR IEC_Close
```

```
KeCL_80
   PLA               ; restore file index

; **********
  Close_File
; **********

   TAX               ; copy index to file to close

ClFi_05
   DEC LDTND         ; decrement open file count
   CPX LDTND         ; compare index with open file count
   BEQ ClFi_10       ; exit if equal, last entry was closing file
   LDY LDTND         ; get open file count as index
   LDA FILTBL,Y      ; get last+1 logical file number from logical file table
   STA FILTBL,X      ; save logical file number over closed file
   LDA DEVTBL,Y      ; get last+1 device number from device number table
   STA DEVTBL,X      ; save device number over closed file
   LDA SECATB,Y      ; get last+1 secondary address from secondary address table
   STA SECATB,X      ; save secondary address over closed file

ClFi_10
   CLC

ClFi_Ret
   RTS

; ***********
  Find_File_X
; ***********

   LDA #0
   STA STATUS        ; clear serial status byte
   TXA               ; copy logical file number to A

; ***********
  Find_File_A
; ***********

   LDX LDTND         ; get open file count

FiFi_10
   DEX               ; decrememnt count to give index
   BMI GLFS_Ret      ; exit if no files
   CMP FILTBL,X      ; compare logical file number with table logical file number
   BNE FiFi_10       ; loop if no match
   RTS

; *******
  Get_LFS
; *******

   LDA FILTBL,X      ; get logical file from logical file table
   STA LA            ; set logical file
   LDA DEVTBL,X      ; get device number from device number table
   STA FA            ; set device number
   LDA SECATB,X      ; get secondary address from secondary address table
   STA SA            ; set secondary address

GLFS_Ret
   RTS
```

```
; ************
  Kernal_CLALL
; ************

   LDA #0
   STA LDTND         ; clear open file count

; *************
  Kernal_CLRCHN
; *************

   LDX #$03          ; set X to screen
   CPX DFLTO         ; compare output device number with screen
   BCS KeCC_10       ; branch if >= screen
   JSR Kernal_UNLSN  ; command the serial bus to UNLISTEN

KeCC_10
   CPX DFLTN         ; compare input device number with screen
   BCS KeCC_20       ; branch if >= screen
   JSR Kernal_UNTLK  ; command the serial bus to UNTALK

KeCC_20
   STX DFLTO         ; set output device number to screen
   LDA #$00          ; set for keyboard
   STA DFLTN         ; set input device number to keyboard
   RTS

; ***********
  Kernal_OPEN
; ***********

   LDX LA            ; get logical file number
   BNE OPEN_05       ; branch if there is a file
   JMP Not_Input_File      ; else do not input file error and return

OPEN_05
   JSR Find_File_X
   BNE OPEN_10       ; branch if file not found
   JMP File_Already_Open      ; else do file already open error and return

OPEN_10
   LDX LDTND         ; get open file count
   CPX #$0A          ; compare with max
   BCC OPEN_15       ; branch if less
   JMP Too_Many_Files      ; else do too many files error and return

OPEN_15
   INC LDTND         ; increment open file count
   LDA LA            ; get logical file number
   STA FILTBL,X      ; save to logical file table
   LDA SA            ; get secondary address
   ORA #$60          ; OR with the OPEN CHANNEL command
   STA SA            ; set secondary address
   STA SECATB,X      ; save to secondary address table
   LDA FA            ; get device number
   STA DEVTBL,X      ; save to device number table
   BEQ OPEN_60       ; do ok exit if keyboard
   CMP #$03          ; compare device number with screen
   BEQ OPEN_60       ; do ok exit if screen
   BCC OPEN_20       ; branch if < screen, tape or RS232
   JSR IEC_Send_SA_And_Filename
   BCC OPEN_60       ; do ok exit
```

```
OPEN_20
#if JIFFY
   CMP #1            ; tape ?
   BEQ Jmp_Dev_Not_Present
#else
   CMP #2            ; RS232 ?
   BNE OPEN_25
#endif
   JMP RS232_Open

#if JIFFY
Jiffy_Talk_TkSA
   JSR UNTLK
   LDA FA
   JSR TALK
#else
OPEN_25
   JSR TAPE_Get_Buffer_Address
   BCS OPEN_30       ; branch if >= $0200
   JMP Illegal_Jiffy_Device
#endif

OPEN_30
   LDA SA            ; get secondary address
#if JIFFY
   JMP TKSA

Jiffy_Transfer_01
   .BYTE "M-W",$00,$06,$1c ; memory write $0600 - $061b

Jiffy_Disk_Code_01   ; $1c bytes drive code
   LDA $0261
   STA $07
   LDA #$12
   STA $06
   LDX #0
   STX $f9
   JSR $d586
   LDY $0267
   LDA ($30),Y
   EOR #$40
   STA ($30),Y
   JMP $d58a

   .BYTE "M-E",$00,$06            ; memory execute at $0600
   .BYTE "M-W",$6a,$00,$01        ; memory write   at $006a
   .BYTE "M-W",$69,$00,$01        ; memory write   at $0069
   .BYTE $50,$6e,$01,$00,$53,$3a
#else
   AND #$0F
   BNE OPEN_45
   JSR Wait_For_Play
   BCS OPEN_Ret      ; exit if STOP was pressed
   JSR Print_Searching
   LDA FNLEN         ; get file name length
   BEQ OPEN_40       ; if null file name just go find header
   JSR TAPE_Find_Fileheader
   BCC OPEN_50       ; branch if no error
   BEQ OPEN_Ret      ; exit if ??

OPEN_35
```

```
   JMP File_Not_Found      ; do file not found error and return

OPEN_40
   JSR TAPE_Find_Any_Header
   BEQ OPEN_Ret      ; exit if end of tape found
   BCC OPEN_50
   BCS OPEN_35

OPEN_45
   JSR TAPE_Wait_For_Record
   BCS OPEN_Ret      ; exit if STOP was pressed
   LDA #$04          ; set data file header
   JSR TAPE_Write_Header

OPEN_50
   LDA #$BF
   LDY SA            ; get secondary address
   CPY #$60
   BEQ OPEN_55
   LDY #0
   LDA #$02
   STA (TAPE1),Y     ; save to tape buffer
   TYA               ; clear A
#endif

OPEN_55
   STA BUFPNT        ; save tape buffer index

OPEN_60
   CLC               ; flag ok

OPEN_Ret
   RTS

; ************************
  IEC_Send_SA_And_Filename
; ************************

   LDA SA            ; get secondary address
#if C64
   BMI OPEN_60       ; ok exit if negative
#endif
#if VIC
   BMI ISSF_40       ; ok exit if negative
#endif
   LDY FNLEN         ; get file name length
#if C64
   BEQ OPEN_60       ; ok exit if null
#endif
#if VIC
   BEQ ISSF_40       ; ok exit if null
#endif
#if C64
   LDA #0
   STA STATUS
#endif
   LDA FA            ; get device number
   JSR Kernal_LISTEN ; command devices on the serial bus to LISTEN
   LDA SA            ; get the secondary address
   ORA #$F0          ; OR with the OPEN command
   JSR Kernal_SECOND ; send secondary address after LISTEN
   LDA STATUS        ; get serial status byte
   BPL ISSF_10       ; branch if device present

Pop_Dev_Not_Present
   PLA               ; else dump calling address low byte
   PLA               ; dump calling address high byte

Jmp_Dev_Not_Present
   JMP Dev_Not_Present      ; do device not present error and return

ISSF_10
   LDA FNLEN         ; get file name length
   BEQ ISSF_30       ; branch if null name

   LDY #0
ISSF_20
   LDA (FNADR ),Y    ; get file name byte
   JSR Kernal_CIOUT  ; output a byte to the serial bus
   INY
   CPY FNLEN         ; compare with file name length
   BNE ISSF_20       ; loop if not all done

ISSF_30
#if C64
   JMP IClo_10
#endif
#if VIC
   JSR Kernal_UNLSN
ISSF_40
   CLC               ; flag ok
   RTS
#endif

; ==========
  RS232_Open
; ==========

#if C64
   JSR ROPN_50
#endif
#if VIC
   LDA #$06          ; IIII IOOI, DTR and RTS only as outputs
   STA VIA1_DDRB     ; set VIA 1 DDRB
   STA RS2_DSR_CTS   ; set VIA 1 DRB, DTR and RTS high
   LDA #$EE          ; CB2 high, CB1 negative edge, CA2 high, CA1 negative edge
   STA VIA1_PCR      ; set VIA 1 PCR
   LDY #0
#endif
   STY RSSTAT        ; clear RS232 status byte

ROPN_05
   CPY FNLEN         ; compare with file name length
   BEQ ROPN_10       ; exit loop if done
   LDA (FNADR ),Y    ; get file name byte
   STA M51CTR  ,Y    ; copy to 6551 register set
   INY
   CPY #$04          ; compare with $04
   BNE ROPN_05       ; loop if not to 4 yet

ROPN_10
   JSR RS232_Set_Data_Bits
   STX BITNUM        ; save bit count
   LDA M51CTR        ; get pseudo 6551 control register
   AND #$0F          ; mask 0000 xxxx, baud rate
#if C64
   BEQ ROPN_30
#endif
#if VIC
   BNE ROPN_15       ; quirk
#endif

ROPN_15
   ASL A             ; * 2
   TAX               ; copy to index
#if C64
   LDA TVSFLG        ; TV flag
   BNE ROPN_20       ; 0 = PAL
   LDY Baudrate-1,X
   LDA Baudrate-2,X
   JMP ROPN_25

ROPN_20
   LDY BaudNTSC-1,X
   LDA BaudNTSC-2,X

ROPN_25
   STY M51AJB+1
   STA M51AJB

ROPN_30
   LDA M51AJB
   ASL A
   JSR Set_Baud_Rate
#endif
#if VIC
   LDA Baudrate-2,X  ; get timer constant low byte
   ASL A             ; * 2
   TAY               ; copy to Y
   LDA Baudrate-1,X  ; get timer constant high byte
   ROL A             ; * 2
   PHA               ; save it
   TYA               ; get timer constant low byte back
   ADC #$C8          ; + $C8, carry cleared by previous ROL
   STA BAUDOF        ; save bit cell time low byte
   PLA               ; restore high  byte
   ADC #$00          ; add carry
   STA BAUDOF+1      ; save bit cell time high byte
#endif
   LDA M51CDR        ; get pseudo 6551 command register
   LSR A             ; shift b0 into Cb
   BCC ROPN_35       ; branch if 3 line interface
#if C64
   LDA  CIA2_PRB
#endif
#if VIC
   LDA VIA2_DATB     ; get VIA 2 DRB, this is wrong, the adress should be
#endif
   ASL A             ; shift DSR into Cb
   BCS ROPN_35       ; branch if DSR = 1
#if C64
   JSR RS232_Set_Status_No_Signal
#endif
#if VIC
   JMP RS232_No_DSR_Signal
#endif
```

```
ROPN_35
   LDA RIDBE         ; get index to Rx buffer end
   STA RIDBS         ; set index to Rx buffer start, clear Rx buffer
   LDA RODBE         ; get index to Tx buffer end
   STA RODBS         ; set index to Tx buffer start, clear Tx buffer
   JSR Read_Memtop
   LDA RXPTR+1       ; get Rx buffer pointer high byte
   BNE ROPN_40       ; branch if buffer already set
   DEY               ; decrement top of memory high byte, 256 byte buffer
   STY RXPTR+1       ; set Rx buffer pointer high byte
   STX RXPTR         ; set Rx buffer pointer low byte

ROPN_40
   LDA TXPTR+1       ; get Tx buffer pointer high byte
   BNE ROPN_45       ; branch if buffer already set
   DEY               ; decrement Rx buffer pointer high byte, 256 byte buffer
   STY TXPTR+1       ; set Tx buffer pointer high byte
   STX TXPTR         ; set Tx buffer pointer low byte

ROPN_45
   SEC
   LDA #$F0
   JMP Set_memtop

#if C64
ROPN_50
   LDA #$7F
   STA CIA2_ICR
   LDA #6
   STA CIA2_DDRB
   STA CIA2_PRB
   LDA #4
   ORA CIA2_PRA
   STA CIA2_PRA
   LDY #0
   STY ENABL
   RTS
#endif

; ***********
  Kernal_LOAD
; ***********

   STXY(MEMUSS)      ; set kernal setup pointer
   JMP (ILOAD)       ; do LOAD vector, usually points to Default_LOAD

; ************
  Default_LOAD
; ************

   STA VERCKK        ; save load/verify flag
   LDA #0
   STA STATUS        ; clear serial status byte
   LDA FA            ; get device number
   BNE DLOA_10       ; branch if not keyboard

DLOA_05
   JMP Illegal_Jiffy_Device

DLOA_10
   CMP #$03          ; compare device number with screen
   BEQ DLOA_05       ; if screen go do illegal device number and return
#if JIFFY
   BCC DLOA_05       ; no tape with jiffy dos
#else
   BCC DLOA_55       ; branch if less than screen
#endif
   LDY FNLEN         ; get file name length
   BNE DLOA_15       ; branch if not null name
#if JIFFY
   JMP Jiffy_Default_Filename
#else
   JMP Missing_File_Name
#endif

DLOA_15
#if C64
   LDX SA
   JSR Print_Searching
#endif
#if VIC
   JSR Get_SA_Print_Searching
#endif
   LDA #$60
   STA SA            ; save the secondary address
   JSR IEC_Send_SA_And_Filename
   LDA FA            ; get device number
   JSR Kernal_TALK   ; command a serial bus device to TALK
   LDA SA            ; get secondary address
   JSR Kernal_TKSA   ; send secondary address after TALK
   JSR Kernal_ACPTR  ; input a byte from the serial bus
   STA EAL           ; save program start address low byte
   LDA STATUS        ; get serial status byte
   LSR A             ; shift time out read ..
   LSR A             ; .. into carry bit
   BCS DLOA_50       ; if timed out go do file not found error and return
#if JIFFY
   JSR Jiffy_ACPTR_Load_Check
#else
   JSR Kernal_ACPTR  ; input a byte from the serial bus
#endif
   STA EAL+1         ; save program start address high byte
#if C64
   TXA
   BNE DLOA_20
   LDA MEMUSS
   STA EAL
   LDA MEMUSS+1
   STA EAL+1

DLOA_20
#if JIFFY
   JMP Jiffy_fac4
#else
   JSR Display_LOADING_Or_VERIFYING
#endif
#endif
#if VIC
#if JIFFY
   JMP Jiffy_fb05
#else
   JSR Set_Load_Address; set LOAD address if secondary address = 0
#endif
```

```
#endif

DLOA_25
#if JIFFY
   JSR STOP
   BNE DLOA_26
   JMP SAVE_50

DLOA_26
   JSR Jiffy_ACPTR
   LDA STATUS
   AND #$fd
   CMP STATUS
   STA STATUS
   BNE DLOA_25
   LDY #0
   LDX TSFCNT
   LDA TBTCNT
   CPY VERCKK
   BEQ DLOA_31
#else
   LDA #$FD          ; mask xxxx xx0x, clear time out read bit
   AND STATUS        ; mask serial status byte
   STA STATUS        ; set serial status byte
   JSR STOP          ; Check if stop key is pressed
   BNE DLOA_30       ; branch if not [STOP]
   JMP SAVE_50       ; else close the serial bus device and flag stop

DLOA_30
   JSR Kernal_ACPTR  ; input a byte from the serial bus
   TAX               ; copy byte
   LDA STATUS        ; get serial status byte
   LSR A             ; shift time out read ..
   LSR A             ; .. into carry bit
   BCS DLOA_25       ; if timed out go ??
   TXA               ; copy received byte back
   LDY VERCKK        ; get load/verify flag
   BEQ DLOA_35       ; branch if load
   LDY #0
#endif
   CMP (EAL),Y       ; compare byte with previously loaded byte
#if JIFFY
   BEQ DLOA_32
   JSR Flag_Read_Error
  .BYTE $2c

DLOA_31
   STA (EAL),Y

DLOA_32
   STX TSFCNT
#else
   BEQ DLOA_40       ; branch if match
   LDA #$10          ; flag read error
   JSR Ora_Status
   .byte   $2C

DLOA_35
   STA (EAL),Y       ; save byte to memory
#endif

DLOA_40
```

```
   INC EAL           ; increment save pointer low byte
   BNE DLOA_45       ; if no rollover skip the high byte increment
   INC EAL+1         ; else increment save pointer high byte

DLOA_45
   BIT STATUS        ; test serial status byte
   BVC DLOA_25       ; loop if not end of file

DLOA_47
   JSR Kernal_UNTLK  ; command the serial bus to UNTALK
   JSR IEC_Close
#if JIFFY & VIC
   BCC DLOA_95
#else
   BCC DLOA_94       ; if OK exit
#endif

DLOA_50
   JMP File_Not_Found

DLOA_55
#if JIFFY

; *************
  Jiffy_Exec_At
; *************

   LDA FNLEN
   BEQ JLTF_10       ; no filename
   LDA (FNADR),Y
   CMP #'$'          ; display directory ?
   BEQ JLBF_10
#if VIC
   JMP Jiffy_eb08
#else
   JMP Jiffy_fc9a
#endif

; *******************
  Jiffy_List_Text_File ; Jiffy_List_Text_File
; *******************

   TYA               ; (Y) contains the command number

JLTF_05
   PHA               ; save command
   JSR Jiffy_Open_Command_10
   PLA               ; retrieve

JLTF_10
   STA BUFPNT        ; store

JLTF_20
   JSR Jiffy_Read_Text_Line    ; input charaters to buffer (filename area)
   BNE JLTF_Ret      ; exit if errors occured
   LDA BUFPNT        ; get command number, should be $0f
   PHP
   BEQ JLTF_30
   JSR Jiffy_CHRIN
   BEQ JLTF_40       ; exit if zero

JLTF_30
```

```
   JSR JiDi_50
   JSR PrSe_10       ; print filename, ie. the input buffer
#if VIC
   LDA STKEY         ; STKEY FLAG, test if <STOP> is pressed
   LSR A
   BCC JLTF_40       ; exit
#else
   BIT STKEY         ; STKEY FLAG, test if <STOP> is pressed
   BPL JLTF_40       ; exit
#endif
   PLP
   BNE JLTF_20
   BVC JLTF_20
   .BYTE $24         ; skip PLP statement

JLTF_40
   PLP

JLTF_Ret
   RTS

; ********************
  Jiffy_List_Basic_File
; ********************

   LDX #$6c          ; get byte for SA, list basic program
   .BYTE $2c         ; skip next statement

JLBF_10
   LDX #$60          ; get byte for SA, list directory
   JSR Jiffy_Open_Command_20    ; open file with current parameters
#if C64
   LDA #$39          ; setup IERROR vector to point to $f739 (RTS)
#endif
#if VIC
   LDA #$bc          ; setup IERROR vector to point to $f739 (RTS)
#endif
   STA IERROR
   LDY #$fc          ; set up (Y) pointer to 252
   JSR Jiffy_fca6    ; skip load adddress

JLBF_20
   LDY #0

JLBF_30
   JSR Jiffy_fca6    ; read 254 bytes, store in input buffer
   BVS JLBF_40       ; exit on EOF
   CPY #2            ; exit if nothing read
   BEQ JLBF_40
   CPY #6
   BCC JLBF_30
#if C64
   LDX FNADR         ; Fr_Bot = FNADR
   STX TMPPTC
   LDX FNADR+1
   STX TMPPTC+1
   LDY #1
   STA (TMPPTC),Y
#endif
#if VIC
   LDX FNADR+1       ; Fr_Bot = FNADR
   STX TMPPTC+1
   LDA FNADR
   STA TMPPTC
   JSR Jiffy_fbc8
#endif
   JSR LIST_12       ; part of Basic LIST routine
   JSR JiDi_50
   JSR LIST_17
#if C64
   BIT STKEY
   BMI JLBF_20       ; continue if no STOP key pressed
#endif
#if VIC
   LDA STKEY
   LSR A
   BCS JLBF_20
#endif

JLBF_40
#if C64
   LDA #$63          ; restore IERROR vector to $f763
#endif
#if VIC
   LDA #$e6          ; restore IERROR vector to
#endif
   STA IERROR
   RTS
#if VIC & !JIFFY
   NOP
#endif
#else
#if C64
   LSR A
   BCS DLOA_60
   JMP Illegal_Jiffy_Device
#endif
#if VIC
   CMP #$02          ; compare device with RS232 device
   BNE DLOA_60       ; if not RS232 device continue
   JMP RRec_70       ; else do illegal device number and return
#endif

DLOA_60
   JSR TAPE_Get_Buffer_Address
   BCS DLOA_65       ; branch if >= $0200
   JMP Illegal_Jiffy_Device

DLOA_65
   JSR Wait_For_Play
   BCS DLOA_Ret      ; exit if STOP was pressed
   JSR Print_Searching

DLOA_70
   LDA FNLEN         ; get file name length
   BEQ DLOA_75
   JSR TAPE_Find_Fileheader
   BCC DLOA_80       ; if no error continue
   BEQ DLOA_Ret      ; exit if ??
   BCS DLOA_50       ; branch always

DLOA_75
   JSR TAPE_Find_Any_Header
   BEQ DLOA_Ret      ; exit if ??
   BCS DLOA_50

DLOA_80
   LDA STATUS        ; get serial status byte
   AND #$10          ; mask 000x 0000, read error
   SEC               ; flag fail
   BNE DLOA_Ret      ; if read error just exit
   CPX #$01
   BEQ DLOA_90
   CPX #$03
   BNE DLOA_70

DLOA_85
   LDY #$01
   LDA (TAPE1),Y
   STA MEMUSS
   INY
   LDA (TAPE1),Y
   STA MEMUSS+1
   BCS DLOA_92

DLOA_90
   LDA SA
   BNE DLOA_85

DLOA_92
   LDY #$03
   LDA (TAPE1),Y
   LDY #$01
   SBC (TAPE1),Y
   TAX
   LDY #$04
   LDA (TAPE1),Y
   LDY #$02
   SBC (TAPE1),Y
   TAY
   CLC
   TXA
   ADC MEMUSS
   STA EAL
   TYA
   ADC MEMUSS+1
   STA EAL+1
   LDA MEMUSS
   STA STAL
   LDA MEMUSS+1
   STA STAL+1
   JSR Display_LOADING_Or_VERIFYING
   JSR TAPE_Read
   .byte   $24       ; skip CLC statement
#endif

DLOA_94
   CLC               ; flag ok

DLOA_95
   LDX EAL           ; get the LOAD end pointer low byte
   LDY EAL+1         ; get the LOAD end pointer high byte

DLOA_Ret
   RTS
```

```
; ***************
  Print_Searching
; ***************

   LDA MSGFLG
   BPL PrSe_Ret
   LDY #Msg_Searching-Msg_Start
   JSR Display_Kernal_IO_Message
   LDA FNLEN
   BEQ PrSe_Ret
   LDY #Msg_FOR-Msg_Start
   JSR Display_Kernal_IO_Message

PrSe_10
   LDY FNLEN
   BEQ PrSe_Ret

   LDY #0
PrSe_20
   LDA (FNADR ),Y
   JSR CHROUT
   INY
   CPY FNLEN
   BNE PrSe_20

PrSe_Ret
   RTS

; ****************************
  Display_LOADING_Or_VERIFYING
; ****************************

   LDY #Msg_Loading-Msg_Start
   LDA VERCKK        ; get load/verify flag
   BEQ DLV_10        ; branch if load
   LDY #Msg_Verifying-Msg_Start

DLV_10
   JMP Display_Direct_Msg

; ***********
  Kernal_SAVE
; ***********

   STX EAL           ; save end address low byte
   STY EAL+1         ; save end address high byte
   TAX               ; copy index to start pointer
   LDA 0,X           ; get start address low byte
   STA STAL          ; set I/O start addresses low byte
   LDA 1,X           ; get start address high byte
   STA STAL+1        ; set I/O start addresses high byte
   JMP (ISAVE)       ; go save, usually points to Default_SAVE

; ************
  Default_SAVE
; ************

   LDA FA            ; get device number
   BNE SAVE_20       ; branch if not keyboard

SAVE_10
   JMP Illegal_Jiffy_Device
```

```
SAVE_20
   CMP #$03          ; compare device number with screen
   BEQ SAVE_10       ; if screen do illegal device number and return
#if JIFFY
   BCC SAVE_10
#else
   BCC IClo_30      ; branch if < screen
#endif
   LDA #$61          ; set secondary address to $01
   STA SA            ; save secondary address
   LDY FNLEN         ; get file name length
   BNE SAVE_30       ; branch if filename not null

Jmp_Missing_Filename
   JMP Missing_File_Name      ; else do missing file name error and return

SAVE_30
   JSR IEC_Send_SA_And_Filename
   JSR Display_SAVING_Filename
   LDA FA            ; get device number
   JSR Kernal_LISTEN ; command devices on the serial bus to LISTEN
   LDA SA            ; get secondary address
   JSR Kernal_SECOND ; send secondary address after LISTEN
   LDY #0
   JSR Set_IO_Start
   LDA SAL           ; get buffer address low byte
   JSR Kernal_CIOUT  ; output a byte to the serial bus
   LDA SAL+1         ; get buffer address high byte
   JSR Kernal_CIOUT  ; output a byte to the serial bus

SAVE_40
   JSR Check_IO_End
   BCS SAVE_70       ; go do UNLISTEN if at end
   LDA (SAL),Y       ; get byte from buffer
   JSR Kernal_CIOUT  ; output a byte to the serial bus
   JSR STOP          ; Check if stop key is pressed
   BNE SAVE_60       ; if stop not pressed go increment pointer and loop for next

SAVE_50
   JSR IEC_Close
   LDA #0
   SEC               ; flag stop
   RTS

SAVE_60
   JSR Inc_SAL_Word
   BNE SAVE_40       ; loop, branch always

SAVE_70
   JSR Kernal_UNLSN  ; command the serial bus to UNLISTEN

; *********
  IEC_Close
; *********

   BIT SA            ; test the secondary address
   BMI IClo_20       ; if already closed just exit
   LDA FA            ; get the device number
   JSR Kernal_LISTEN ; command devices on the serial bus to LISTEN
   LDA SA            ; get secondary address
   AND #$EF          ; mask the channel number
   ORA #$E0          ; OR with the CLOSE command
   JSR Kernal_SECOND ; send secondary address after LISTEN

IClo_10
   JSR Kernal_UNLSN  ; command the serial bus to UNLISTEN

IClo_20
   CLC               ; flag ok
   RTS

#if JIFFY
; **********************
  Jiffy_Default_Filename
; **********************

   LDA NDX
   BEQ Jmp_Missing_Filename
   LDA #2
   STA SA
   LDX #<[Jiffy_F1+2]
   LDY #>[Jiffy_F1+2]
   JSR SETNAM
   JMP DLOA_15

; *****
  Jf66b
; *****

   LDX #$33
   LDY #4
   JMP Jiffy_Char_Command

; Function key assignment

Jiffy_F1  .BYTE "@$:*\r",0           ; directory
Jiffy_F3  .BYTE "/",0                ; load
Jiffy_F5  .BYTE "^",0                ; save
Jiffy_F7  .BYTE "%",0                ; load ML
Jiffy_F2  .BYTE "@D",0
Jiffy_F4  .BYTE "@T",0
Jiffy_F6  .BYTE "_",0
Jiffy_F8  .BYTE "@  ",QUOTE,"S:",0
#if VIC
   .byte $24
#endif
#else
IClo_30
#if C64
   LSR A
   BCS IClo_40
   JMP Illegal_Jiffy_Device
#endif
#if VIC
   CMP #$02          ; compare device with RS232 device
   BNE IClo_40       ; branch if not RS232 device
   JMP RRec_70       ; else do illegal device number and return
#endif

IClo_40
   JSR TAPE_Get_Buffer_Address
   BCC SAVE_10
   JSR TAPE_Wait_For_Record
   BCS IClo_Ret      ; exit if STOP was pressed
   JSR Display_SAVING_Filename
   LDX #$03          ; set header for a non relocatable program file
   LDA SA            ; get secondary address
   AND #$01          ; mask non relocatable bit
   BNE IClo_50       ; branch if non relocatable program
   LDX #$01          ; else set header for a relocatable program file

IClo_50
   TXA               ; copy header type to A
   JSR TAPE_Write_Header
   BCS IClo_Ret      ; exit if error
   JSR TAPE_Write_With_Lead
   BCS IClo_Ret      ; exit if error
   LDA SA            ; get secondary address
   AND #$02          ; mask end of tape flag
   BEQ IClo_60       ; branch if not end of tape
   LDA #$05          ; else set logical end of the tape
   JSR TAPE_Write_Header
   .byte   $24       ; skip next command
#endif

IClo_60
   CLC               ; flag ok

IClo_Ret
   RTS

; ***********************
  Display_SAVING_Filename
; ***********************

   LDA MSGFLG        ; get message mode flag
   BPL IClo_Ret      ; exit if control messages off
   LDY #Msg_Saving-Msg_Start
   JSR Display_Kernal_IO_Message
   JMP PrSe_10      ; print file name and return

; ************
  Kernal_UDTIM       ; Update the system clock
; ************

   LDX #$00          ; clear X
   INC JIFFYL        ; increment jiffy low byte
   BNE UDTI_10       ; if no rollover skip the mid byte increment
   INC JIFFYM        ; increment jiffy mid byte
   BNE UDTI_10       ; if no rollover skip the high byte increment
   INC JIFFYH        ; increment jiffy high byte

UDTI_10
   SEC
   LDA JIFFYL        ; get jiffy clock low byte
   SBC #$01          ; subtract $4F1A01 low byte
   LDA JIFFYM        ; get jiffy clock mid byte
   SBC #$1A          ; subtract $4F1A01 mid byte
   LDA JIFFYH        ; get jiffy clock high byte
   SBC #$4F          ; subtract $4F1A01 high byte
   BCC Look_For_Special_Keys
   STX JIFFYH        ; clear jiffies high byte
   STX JIFFYM        ; clear jiffies mid byte
   STX JIFFYL        ; clear jiffies low byte
```

```
; *********************
  Look_For_Special_Keys
; *********************

   LDA KEYB_ROWN     ; get VIA 2 DRA, keyboard row, no handshake
   CMP KEYB_ROWN     ; compare with self
   BNE Look_For_Special_Keys
#if C64
   TAX
   BMI LFSK_20
   LDX #$bd
   STX KEYB_COL

LFSK_10
   LDX KEYB_ROW
   CPX KEYB_ROW
   BNE LFSK_10
   STA KEYB_COL
   INX
   BNE LFSK_Ret
#endif

LFSK_20
   STA STKEY         ; save VIA 2 DRA, keyboard row

LFSK_Ret
   RTS

; ************
  Kernal_RDTIM       ; Read system clock
; ************

   SEI               ; disable interrupts
   LDA JIFFYL        ; get jiffy clock low byte
   LDX JIFFYM        ; get jiffy clock mid byte
   LDY JIFFYH        ; get jiffy clock high byte

; *************
  Kernal_SETTIM      ; Set the system clock
; *************

   SEI               ; disable interrupts
   STA JIFFYL        ; save jiffy clock low byte
   STX JIFFYM        ; save jiffy clock mid byte
   STY JIFFYH        ; save jiffy clock high byte
   CLI               ; enable interrupts
   RTS

; ***********
  Kernal_STOP        ; Check if stop key is pressed
; ***********

   LDA STKEY         ; get keyboard row
#if C64
   CMP #$7f
#endif
#if VIC
   CMP #$FE          ; compare with r0 down
#endif
   BNE STOP_Ret      ; branch if not just r0
   PHP               ; save status
   JSR CLRCHN        ; Clear I/O channels
   STA NDX           ; save keyboard buffer length
   PLP               ; restore status

STOP_Ret
   RTS

Too_Many_Files
   LDA #$01
   .byte   $2C

File_Already_Open
   LDA #$02
   .byte   $2C

File_Not_Open
   LDA #$03
   .byte   $2C

File_Not_Found
   LDA #$04
   .byte   $2C

Dev_Not_Present
   LDA #$05
   .byte   $2C

Not_Input_File
   LDA #$06
   .byte   $2C

Not_Output_File
   LDA #$07
   .byte   $2C

Missing_File_Name
   LDA #$08
   .byte   $2C

Illegal_Jiffy_Device
   LDA #$09          ; illegal device number
   PHA               ; save error #
   JSR CLRCHN        ; Clear I/O channels
   LDY #0            ; index to "I/O ERROR #"
   BIT MSGFLG        ; test message mode flag
   BVC DIOR_10       ; exit if kernal messages off
   JSR Display_Kernal_IO_Message
   PLA               ; restore error #
   PHA               ; copy error #
   ORA #'0'          ; convert to ASCII
   JSR CHROUT        ; Output a character

DIOR_10
   PLA               ; pull error number
   SEC               ; flag error

Jiffy_RTS
   RTS

#if JIFFY

; ******************
  Jiffy_Test_Command
```

```
; *****************

   LDY #12
   JSR CHRGOT

; *******************
  Jiffy_Test_Commands
; *******************

   CMP Jiffy_Command_List,Y
   BEQ JTC_Ret
   DEY
   BPL Jiffy_Test_Commands

JTC_Ret
   RTS

; ************
  Jiffy_SETLFS
; ************

   JSR SETLFS

; *********************
  Jiffy_Test_IEC_Device
; *********************

   CLC
   PHP
   LDX Jiffy_Device
   CPX #8
   BCC JTID_20       ; device <  8

JTID_10
   CPX #$1f
   BCC JTID_30       ; device < 31

JTID_20
   PLP
   BCS Jiffy_Device_Not_Present
   SEC
   PHP
   LDX #8            ; try device 8

JTID_30
   STX Jiffy_Device
   JSR Jiffy_Test_Device
   BCC JTID_40
   INX               ; try next device
   BNE JTID_10

JTID_40
   PLA

JTID_Ret
   RTS

; *****************
  Jiffy_Validate_FA
; *****************

   JSR Jiffy_Test_FA
   BCC JTID_Ret

; ************************
  Jiffy_Device_Not_Present
; ************************

   LDX #5

; **************
  Jiffy_Dispatch
; **************

   CPX #11           ; SYNTAX ERROR
   BEQ JiDi_20

JiDi_10
   JMP Back_To_Prompt

JiDi_20
   JSR Jiffy_Test_Command  ; (Y) = command
   BNE JiDi_10             ; not a Jiffy command
   STY COMSAV
   TAX
   BMI JiDi_30
   PLA
   PLA

JiDi_30
   JSR Jiffy_Test_IEC_Device
   JSR Jiffy_At_Command
   LDA COMSAV
   LDY #0
   ASL A
   TAX
   LDA JTAB,X
   STA FUNJMP
   LDA JTAB+1,X
   STA FUNJMP+1

JiDi_40
   JSR JUMPER        ; execute JiffyDOS command
   JSR Basic_DATA    ; ignore next statement
   JSR Jiffy_Close_15
   LDA PTR2
   JSR CLOSE

; *******
  JiDi_50
; *******

   JSR CLRCHN
   LDX IOPMPT
   BEQ JTID_Ret
   .BYTE $2c         ; ignore next statement

; *******
  JiDi_60
; *******

   LDX #$6f
   JMP CHKOUT
```

```
; *************
  Jiffy_Load_ML
; *************

   TYA
   INY
   .BYTE $2c         ; skip INY and TYA

; ************
  Jiffy_Verify
; ************

   INY

; ****************
  Jiffy_Load_Basic
; ****************

   TYA
   STY SA
   LDX TXTTAB
   LDY TXTTAB+1
   JSR LOAD
   BCC JiLo_30
   JMP Error_Handler

JiLo_10
   JMP LOAD_30

JiLo_20
   JMP LOAD_05

JiLo_30
   LDA COMSAV
   CMP #11           ; verify command (�)
   BEQ JiLo_20       ; output verify OK
   BCS JiDi_40
   CMP #8            ; load ml (%)
#if C64
   BEQ JTID_Ret
   BCC JiLo_10
#endif
#if VIC
   BNE JAAC_10
   RTS

JAAC_10
   BCC JiLo_10
#endif
   STX VARTAB
   STY VARTAB+1
   PLA
   PLA
   JSR Print_CR
   JSR Rechain
   JMP Basic_RUN

; ******************
  Jiffy_Command_List
; ******************

   .byte "@"         ;  0 : disk status and command
   .byte "_"         ;  1 : <- save file
   .byte "*"         ;  2 : *  copy file
   .byte $ac         ;  3 :    copy file
   .byte QUOTE       ;  4 : nothing
   .byte $12         ;  5 : nothing
   .byte "/"         ;  6 : load Basic program
   .byte $ad         ;  7
   .byte "%"         ;  8
   .byte "^"         ;  9
   .byte $ae         ; 10
   .byte $27         ; 11
   .byte $5c         ; 12 : load file ML
   .byte "D"         ; 13 : display basic program
   .byte "L"
   .byte "T"
   .byte "#"
   .byte "B"
   .byte "F"
   .byte "O"
   .byte "P"
   .byte "Q"
   .byte "X"
   .byte "G"

; ****
  JTAB
; ****

   .WORD Jiffy_Exec_At            ;  @    0
   .WORD Jiffy_SAVE               ;  <-   1
   .WORD Jiffy_Copy               ;  *    2
   .WORD Jiffy_Copy               ;  $ac  3
   .WORD Jiffy_RTS                ;  Quo  4
   .WORD Jiffy_RTS                ;  RVS  5
   .WORD Jiffy_Load_Basic         ;  /    6
   .WORD Jiffy_Load_Basic         ;  $ad  7
   .WORD Jiffy_Load_ML            ;  %    8
   .WORD Jiffy_Load_Basic         ;  ^    9
   .WORD Jiffy_Load_Basic         ;  $ae 10
   .WORD Jiffy_Verify             ;  '   11
   .WORD Jiffy_Load_ML            ;  \   12
   .WORD Jiffy_List_Basic_File    ;  Disp   13
   .WORD Jiffy_Lock_File          ;  Lock   14
   .WORD Jiffy_List_Text_File     ;  Text   15
   .WORD Jiffy_Set_Default_Device ;  #      16
   .WORD Jiffy_No_Bump            ;  Bump   17
   .WORD Jiffy_Inx_PRTY           ;  F      18
   .WORD Jiffy_OLD                ;  Old    19
   .WORD Jiffy_Toggle_Printer     ;  Print  20
   .WORD Jiffy_Disable            ;  Quit   21
   .WORD Jiffy_Destination        ;  Xfer   22
   .WORD Jiffy_Gap                ;  Gap    23

; *********
  Jiffy_OLD
; *********

   INY               ; Y = 1
   TYA               ; A = 1
   STA (TXTTAB),Y    ; Create dummy link <> zero
   JSR Rechain
   TXA               ; X/A = end of program + 2
   ADC #2
   TAX
   LDA INDEXA+1
   ADC #0
   TAY
   JMP LOAD_55

; 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 filname 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.

; ****************
  Jiffy_At_Command
; ****************

   TYA
   BNE JAC_05
   STA FNLEN
   JSR CHRGET
   BEQ JAC_45        ; terminator found, exit
   LDY #$17          ; test 23 commands
   JSR Jiffy_Test_Commands
   BNE JAC_10        ; not a Jiffy command
   CPY #13           ; ignore commands 0 - 12
   BCC JAC_10
   STY COMSAV
   CPY #16
   BCS JAC_45        ; no filename for commands >= 16

JAC_05
   LDA #1
   JSR Add_To_TXTPTR

JAC_10
   LDY #$ff

JAC_15               ; scan filename
   INY
   LDA (TXTPTR),Y
   BEQ JAC_20
   CMP #QUOTE
   BEQ JAC_25
   CMP #':'
   BNE JAC_15

JAC_20
   BIT MSGFLG
   BPL JAC_30
   CLC
   JSR Make_String_Descriptor_From_Code
   JMP JAC_35

JAC_25
   JSR Add_Y_To_Execution_Pointer
```

```
JAC_30
   JSR Eval_Expression

JAC_35
   JSR Set_Filename_From_String
   JSR CHRGOT
   CMP #','
   BNE JAC_45
   JSR Get_Next_Byte_Value

JAC_40
   STX FA

JAC_45
   LDY #0
   BIT MSGFLG
   BPL JAC_55

JAC_50
   LDA (LINPTR),Y
   CMP #' '
   BEQ JAC_55
   LDA #13
   JSR Screen_CHROUT
   BNE JAC_50

JAC_55
   JSR Jiffy_Validate_FA
   LDA #$ff
   JSR Allocate_String_FAC1
   LDA FNLEN
   LDX FNADR
   LDY FNADR+1
   JSR Store_And_Push_String
   JSR Eval_String
   STX FNADR
   STY FNADR+1

; **************************
  Jiffy_Open_Command_Channel
; **************************

   JSR Jiffy_Close_6f; close command channel if open
   LDA FNLEN
   LDX #0
   STX FNLEN         ; FNLEN = 0
   LDX #$6f
   BNE JOC_30        ; branch always

; *********************
  Jiffy_Open_Command_10
; *********************

   LDX #$6e

; *********************
  Jiffy_Open_Command_20
; *********************

   LDA FNLEN

JOC_30
```

```
   STX SA
   STX PTR2

JOC_40
   PHA
   STX LA
   JSR CLRCHN
   JSR OPEN
   PLA
   STA FNLEN         ; restore FNLEN

JOC_Ret
;#if C64
   RTS
;#endif

; ***************
  Jiffy_Lock_File
; ***************

   JSR Jiffy_Disk_Status
   BNE JOC_Ret
   LDX #0   ; start data at "M-W...."
   LDY #$22 ; send 34 bytes
   JSR Jiffy_Drive_Command
   LDY #5   ; start data after "M-W" sequence
   LDX #$22 ; send 34 bytes

; *******************
  Jiffy_Drive_Command
; *******************

   JSR Jiffy_Send_Drive_Command
   JMP CLRCHN

; *******************
  Jiffy_Detect_Device
; *******************

#if C64
   STA CIA2_PRA
   AND #8            ; Test ATN out
   BEQ JIS_Ret
   LDA BSOUR
   ROR A
   ROR A
#endif
#if VIC
   STA VIA2_PCR
   BIT VIA1_DATN
   BPL JIS_Ret
#endif

   CPX #2
#if C64
   BNE JIS_Ret
#endif
#if VIC
   BNE JIS_Ret
   LDA #2
#endif
   LDX #$1e          ; wait for jiffy protocol
```

```
JIS_10
#if C64
   BIT CIA2_PRA
   BPL JIS_20        ; data high (0) -> Jiffy signal
#endif
#if VIC
   BIT VIA1_DATN
   BEQ JIS_20        ; data high (0) -> Jiffy signal
#endif
   DEX
   BNE JIS_10
#if C64
   BEQ JIS_30        ; no Jiffy device
#endif
#if VIC
   BEQ JIS_30
#endif

JIS_20
#if C64
   BIT CIA2_PRA
   BPL JIS_20        ; wait for end of Jiffy signal
#endif
#if VIC
   BIT IEC_DRAN
   BEQ JIS_20        ; wait for end of Jiffy signal
   LDA BSOUR
   ROR A
   ROR A
#endif

   ORA #$40          ; BSOUR >>2 | $40
   STA TSFCNT        ; Flag as Jiffy device

JIS_30
   LDX #2

JIS_Ret
   RTS

; ********************
  Jiffy_Read_Text_Line
; ********************

   LDY #0
   JSR Jiffy_CHKIN_PTR2

JRTL_10
   JSR Jfca9
   BVS JRTL_20
   BCC JRTL_10

JRTL_20
   STY FNLEN
   LDA STATUS
   AND #$82
   RTS

; *********
  Jiffy_Gap
; *********
```

```
   JSR Get_Next_Byte_Value
   TXA
   LDX #$2d          ; "M-W" 69 00 01
   BNE Jiffy_6_Char_Command

; *************
  Jiffy_No_Bump
; *************

   LDA #$85
   LDX #$27          ; "M-W" 6a 00 01

; ********************
  Jiffy_6_Char_Command
; ********************

   LDY #6

; ******************
  Jiffy_Char_Command
; ******************

   PHA
   JSR Jiffy_Send_Drive_Command
   PLA
   JMP CHROUT

; *********************
  Jiffy_Toggle_Copy_Flag
; *********************

   LDX #0
   .BYTE $2c

; ****************************
  Jiffy_Toggle_Copy_Flag_Single
; ****************************

   LDX #6
   JSR Reset_BASIC_Exec_Pointer
   LDY #5
   LDA (TXTPTR),Y    ; test 5th. character
   CMP #$12          ; <RVS ON>?
   BNE JTD_10         ; if not, directory isn't loaded
   PLA
   TXA               ; store (X), the toggle flag, on stack
   PHA
   LDY #$23          ; skip diskheader

JTCF_10
   LDX #'"'
   JSR NeLi_15       ; use part of Next_Line, to search for character
   DEY
   JSR Add_Y_To_Execution_Pointer
   PLA               ; recover flag
   PHA
   BEQ JTCF_30       ; toggle all files
   STA CSRIDX
   LDY #1

JTCF_20
```

```
   INY
   JSR CHRI_07       ; use part of CHRIN
   CMP (TXTPTR),Y
   BNE JTCF_40
   SBC #'"'
   BNE JTCF_20

JTCF_30
   TAY
   LDA (TXTPTR),Y    ; get character
   EOR #10           ; toggle between ' ' ($20) and '*' ($2a)
   STA (TXTPTR),Y
   LDY #4
   STA (LINPTR),Y

JTCF_40
   JSR Basic_DATA    ; skip rest of line
   LDY #5
   SEC
   LDA (TXTPTR),Y
   SBC #$42 ; 'B'
   BNE JTCF_10       ; next file
   LDY #2
   STA (TXTPTR),Y
   PLA
   BEQ JTCF_50
   LDA #$8d
   RTS

JTCF_50
   JMP LIST_05

; ******************
  Jiffy_Toggle_Drive
; ******************

   BIT MSGFLG
   BPL JTD_10
   TSX
   LDY STACK+7,X
   CPY #$e1
   BNE JTD_10
   CMP #4
   BNE Bf9b2
   INC Jiffy_Device
   JSR Jiffy_Test_IEC_Device
   LDA #0
   JSR Print_Integer_XA
   JSR Print_CR
   JSR JiDi_50

JTD_10
   PLA
   RTS

; *****
  Bf9b2
; *****

   CMP #1
   BEQ Jiffy_Toggle_Copy_Flag
   CMP #$17
   BEQ Jiffy_Toggle_Copy_Flag_Single
   LDY PRTY
   BNE JTD_10
   CMP #$8d
   BCS JTD_10
   CMP #$85
   BCC JTD_10
   PLA
   SBC #$85
   TAX
   BEQ Bf9d5
Bf9cc     INY
   LDA (CMPO),Y
   BNE Bf9cc
   DEX
   BNE Bf9cc
Bf9d4     INY
Bf9d5     LDA (CMPO),Y
   BEQ Bf9e2
   CMP #13
   BEQ Bf9e4
   JSR Screen_CHROUT
   BNE Bf9d4
Bf9e2     STA CSRMOD
Bf9e4     RTS

; **********
  Jiffy_f9e5
; **********

   JSR Get_Char_From_Keyboard_Buffer
   PHA
   LDX CSRMOD
#if C64
   BNE Bfa37
#endif
#if VIC
   BNE JAAD_10
#endif
   LDX INSRTO
#if C64
   BNE Bfa37
#endif
#if VIC
   BNE JAAD_10
#endif
   CMP #16
   BNE Jiffy_Toggle_Drive
#if C64
   LDA #4
#endif
#if VIC
   JMP KeDe_60
#endif

JAAD_10
#if C64
   JSR LISTEN
   LDA MEM_CONTROL
   AND #2
   BEQ Bfa03
   LDA #7
```

```
Bfa03     ORA #$60
   JSR SECOND
   LDA CSRIDX
   PHA
   LDA TBLX
   PHA
Bfa0e     LDY #0
   STY CSRMOD
   JSR PLOT_05
   INC LINLEN
Bfa17     JSR CHRI_07
   JSR Kernal_CIOUT
   CMP #13
   BNE Bfa17
   INX
   CPX #$19
   BCS Bfa2d
   ASL LINLEN
   BPL Bfa0e
   INX
   BNE Bfa0e
Bfa2d     JSR UNLSN
   PLA
   TAX
   PLA
   TAY
   JSR PLOT_05
#endif
Bfa37     PLA
Bfa38     RTS

; **********
  Jiffy_Copy
; **********

   STY FAC3+1
   JSR Jiffy_Disk_Status
   BNE Bfa38
   JSR CHRGOT
   CMP #$52 ; 'R'
   BNE Bfa5a

Bfa47
   DEC FAC3+1
   LDA FAC3+1
   JSR Jf66b
   JSR Jiffy_CHRIN
   BEQ Bfa47
   LDA #0
   JSR Jf66b
   LDA #$4c ; 'L'

Bfa5a
   PHA
   LDX MYCH
   CPX FA
   BEQ Bfa37
   JSR JAC_40
   LDX #$37 ; '7'
   LDY #2
   JSR Jiffy_Send_Drive_Command
   JSR PrSe_10
   LDA #$2c ; ','
   STA (FNADR),Y
   INY
   PLA
   STA (FNADR),Y
   INY
   LDA #$2c ; ','
   STA (FNADR),Y
   INY
   LDA FAC3+1
   PHA
   BNE Bfa83
   LDA #$57 ; 'W'

Bfa83
   STA (FNADR),Y
   INY
   STY FNLEN
   LDY #12
Bfa8a
   JSR Jfab2
   JSR Jiffy_Test_IEC_Device
   JSR Jiffy_Open_Command_Channel
   PLA
   JSR JLTF_05

; ********************
  Jiffy_Toggle_Printer
; ********************

   LDA IOPMPT
   BEQ Bfaa7
   CMP #$7f
   BNE Bfa38
   JSR BaIN_10
   LDA #$7f
   JMP CLOSE

Bfaa7
   LDX #4
   JSR CHRGET
   JSR GOCP_05
   JSR Jiffy_Validate_FA

; *****
  Jfab2
; *****

   STY SA
   LDX #$7f
   STX IOPMPT
   LDA FNLEN
   JMP JOC_40
#if C64
   TAX               ; dead code
   BNE Bfa8a
   LDA NXTBIT
   BEQ JAAE_05+1

Jiffy_fac4
   JSR Display_LOADING_Or_VERIFYING
#endif
```

```
#if VIC
Jiffy_fb05
   JSR Set_Load_Address
#endif
   TSX
   LDA STACK+2,X
#if C64
   CMP #$f7          ; high byte of return address
#endif
#if VIC
   CMP #$f8
#endif

JAAE_05
   BNE JAAE_10
   LDA EAL
   STA FUNJMP
   LDA EAL+1
   STA FUNJMP+1

JAAE_10
   BIT TSFCNT
   BMI Bfade
   JMP DLOA_25

Bfade
   SEI
#if C64
   LDY #3

Bfae1
   LDA EAL+1,Y
   PHA
   DEY
   BNE Bfae1
   LDA VIC_SPR_ENA
   STA CMPO
   JSR Jiffy_Clear_Sprites

Bfaf0
   JSR Look_For_Special_Keys
   BPL Bfb27
   LDA VIC_CONTROL_1
   AND #7
   CLC
   ADC #$2f
   STA CMPO+1
   LDA CIA2_PRA
   AND #7
   STA TAPE1
   STA CIA2_PRA
   ORA #$20
   TAX

Bfb0c
   BIT CIA2_PRA
   BVC Bfb0c
   BPL Jiffy_LOAD
   LDX #$64
Bfb15     BIT CIA2_PRA
   BVC Bfb20
#endif
```

```
#if VIC
   LDA TAPE1
   PHA
   LDY #0

JAAH_10
   JSR Look_For_Special_Keys
   CMP #$fe
   BEQ Bfb27
   LDA IEC_PCR
   AND #$dd
   TAX
   ORA #$20
   STA TAPE1
   STX IEC_PCR
   LDA #$80
   STA DPSW

JAAF_10
   LDA IEC_DRAN
   LSR A
   BCC JAAF_10
   AND #1
   BEQ Jiffy_LOAD
   LDX #$64

Bfb15
   BIT IEC_DRAN
   BEQ Bfb20
Bfaf0
#endif
   DEX
   BNE Bfb15
   LDA #$42
   .BYTE $2c

Bfb20
   LDA #$40
   JSR Ora_Status
   CLC
   .BYTE $24

Bfb27
   SEC
#if C64
   LDA CMPO
   STA VIC_SPR_ENA
   PLA
   STA CMPO
   PLA
   STA CMPO+1
#endif
   PLA
   STA TAPE1
   BCS Bfb3b
   JMP DLOA_47

Bfb3b
   JMP SAVE_50

; **********
  Jiffy_LOAD
```

```
; **********

#if C64
   BIT CIA2_PRA
   BPL Jiffy_LOAD    ; wait until data (7) = 1
   SEC               ; [2: 2]

JiLO_10
   LDA VIC_RASTER    ; [4: 4] current raster line
   SBC CMPO+1        ; [4: 8]  minus fine scroll register
   BCC JiLO_20       ; [3:11]  no bad line
   AND #7
   BEQ JiLO_10

JiLO_20
   LDA TAPE1         ; [3:14]
   STX CIA2_PRA      ; [4:18]�data (5) and clock (4) output = 0
   BIT CIA2_PRA      ; [4:22]�
   BVC Bfaf0         ; [2:24]
   NOP               ; [2:26]�wait
   STA CIA2_PRA      ; [4:30]
   ORA CIA2_PRA      ; [4:34]�get bit 0 & 1 of byte      <--
   LSR A             ; [2:36] A = .XX.....
   LSR A             ; [2:38] A = ..XX....
   NOP               ; [2:40] wait 2 cycles
   ORA CIA2_PRA      ; [4:44] get bit 2 & 3 of byte      <--
   LSR A             ; [2:46] A = .XXXX...
   LSR A             ; [2:48] A = ..XXXX..
   EOR TAPE1         ; [3:51] leave bits 2-0 unchanged
   EOR CIA2_PRA      ; [4:55] get bit 4 & 5 of byte      <--
   LSR A             ; [2:57] A = .XXXXXX.
   LSR A             ; [2:59] A = ..XXXXXX
   EOR TAPE1         ; [3:62] leave bits 2-0 unchanged
   EOR CIA2_PRA      ; [4:66] get bit 6 & 7 of byte      <--
#endif

#if VIC
   LDA #2

JiLO_05
   BIT IEC_DRAN
   BEQ JiLO_05

JiLO_10
   PHA
   PLA
   NOP
   LDA TAPE1
   STA IEC_PCR
   LDA #1
   BIT IEC_DRAN
   BEQ JAAH_10
   STX IEC_PCR
   LDA IEC_DRAN
   ROR A
   ROR A
   AND #$80
   ORA IEC_DRAN
   ROL A
   ROL A
   STA TAPE1+1
   LDA IEC_DRAN
   ROR A
   ROR A
   AND DPSW
   ORA IEC_DRAN
   ROL A
   ROL A
   STA CAS1
   JSR Jiffy_Combine_Nibbles
#endif

   CPY VERCKK
   BNE JiLO_40
   STA (EAL),Y

JiLO_30
   INC EAL
   BNE JiLO_10
   INC EAL+1
   JMP JiLO_10

JiLO_40
   CMP (EAL),Y
   BEQ JiLO_30
#if C64
   SEC
#endif
   LDA #16
   STA STATUS
   BNE JiLO_30       ; branch always

#if VIC
JTABB
   .byte $00,$00,$20,$20,$00,$00,$20,$20
   .byte $02,$02,$22,$22,$02,$02,$22,$22

Jiffy_fbc8
   LDY #0
   STA (TMPPTC),Y
   INY
   TXA
   STA (TMPPTC),Y
   RTS

   .byte $ff
#endif

; ************
  Set_IO_Start
; ************

   LDA STAL+1
   STA SAL+1
   LDA STAL
   STA SAL
   RTS

; **************************
  Jiffy_Disable_Sprite_ACPTR
; **************************

#if C64
   PHA                ; save sprite enable register
   JSR Jiffy_Clear_Sprites
   JSR Jiffy_ACPTR_10 ; call Jiffy_ACPTR
   PLA
   STA VIC_SPR_ENA    ; restore sprite enable register
   LDA TBTCNT         ; recover received byte
   RTS
#endif

; ***************
  Jiffy_Jmp_ACPTR
; ***************

   LDA #0
   JMP KeAC_03

; 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"

; video timing by Marko Makela
; ---------------------------------------------------------------
;  NTSC-M systems:
;
;           Chip      Crystal  Dot      Processor Cycles/ Lines/
;    Host    ID        freq/Hz  clock/Hz clock/Hz  line    frame
;    ------  --------  -------- -------- --------- ------- ------
;    VIC-20  6560-101  14318181  4090909   1022727      65    261
;    C64     6567R56A  14318181  8181818   1022727      64    262
;    C64     6567R8    14318181  8181818   1022727      65    263
;
;  Later NTSC-M video chips were most probably like the 6567R8.  Note
;  that the processor clock is a 14th of the crystal frequency on all
;  NTSC-M systems.
;
;  PAL-B systems:
;
;           Chip      Crystal  Dot      Processor Cycles/ Lines/
;    Host    ID        freq/Hz  clock/Hz clock/Hz  line    frame
;    ------  --------  -------- -------- --------- ------- ------
;    VIC-20  6561-101   4433618  4433618   1108405      71    312
;    C64     6569      17734472  7881988    985248      63    312
;
; ---------------------------------------------------------------
; So the Jiffy routine needs more time than one raster line, to
; receive 1 byte. Therefore we have to make sure, that the bad line
; is not nearer than 2 lines ahead!


; ***********
  Jiffy_ACPTR
; ***********

   SEI
   BIT TSFCNT          ; test to see if the device is a JiffyDOS drive
   BVC Jiffy_Jmp_ACPTR ; nope, back to normal ACPTR routine
#if C64
   LDA VIC_SPR_ENA     ; are sprites active ?
   BNE Jiffy_Disable_Sprite_ACPTR
#endif
```

```
; **************
  Jiffy_ACPTR_10
; **************

#if C64
   LDA CIA2_PRA        ; IEC bus register
   CMP #%01000000
   BCC Jiffy_ACPTR_10  ; wait until data (7) or clock (6) = 1
   AND #7              ; mask bits 2-0
   PHA                 ; save (carry is set)

JiAC_10
   LDA VIC_RASTER      ; [4: 4] current raster line
   SBC VIC_CONTROL_1   ; [4: 8]  minus fine scroll register
   AND #7              ; [2:10] modulo 7
   CMP #7              ; [2:12] compare with 7
   BCS JiAC_10         ; [2:14] we're one line before a bad line
   PLA                 ; [4:18]�restore bits 2-0
   STA CIA2_PRA        ; [4:22] data (5) and clock (4) output = 0
   STA TBTCNT          ; [3:25] save bits 2-0
   ORA #%00100000      ; [2:27] data (5) output = 1
   PHA                 ; [3:30] save
   NOP                 ; [2:32] wait 2 cycles
   NOP                 ; [2:34] wait 2 cycles
   ORA CIA2_PRA        ; [4:38] get bit 0 & 1 of byte      <--
   LSR A               ; [2:40] A = .XX.....
   LSR A               ; [2:42] A = ..XX....
   NOP                 ; [2:44] wait 2 cycles
   ORA CIA2_PRA        ; [4:48] get bit 2 & 3 of byte      <--
   LSR A               ; [2:50] A = .XXXX...
   LSR A               ; [2:52] A = ..XXXX..
   EOR TBTCNT          ; [3:55] leave bits 2-0 unchanged
   EOR CIA2_PRA        ; [4:59] get bit 4 & 5 of byte      <--
   LSR A               ; [2:61] A = .XXXXXX.
   LSR A               ; [2:63] A = ..XXXXXX
   EOR TBTCNT          ; [3:66] leave bits 2-0 unchanged
   EOR CIA2_PRA        ; [4:70] get bit 6 & 7 of byte      <--
   STA TBTCNT          ; [3:73] byte completed
   PLA                 ; [4:77] recover %00100000 OR bit 2-0
   BIT CIA2_PRA        ; [4:81] test data in (7) and clock in (6)
   STA CIA2_PRA        ; [4:85] data out (5) = 1
   BVC Jiffy_Set_OK    ; branch on clock in = 0
   BPL Jiffy_Set_EOI   ; branch on data  in = 0
   LDA #%01000010      ; EOI (6) and time out (1)
   JMP Set_IEC_Status
#endif

; ��

#if VIC
; read one byte from IEC bus
; ------------------------------------
; cycle 34: clock = bit1   data = bit0
; cycle 44: clock = bit3   data = bit2
; cycle 55: clock = bit5   data = bit4
; cycle 66: clock = bit7   data = bit6
; ------------------------------------
; cycle 77: status
; cycle 81: finish transmission

   LDA IEC_DRAN        ; IEC bus register
   AND #3
   BEQ Jiffy_ACPTR_10  ; wait until data (1) or clock (0) = 1
   LDA #$80
   STA DPSW
   TXA                 ; [2: 2]
   PHA                 ; [3: 4]�save X
   PHA                 ; [3: 7]�wait 3 cycles
   PLA                 ; [4:11]�wait 4 cycles
   LDA IEC_PCR         ; [4:15]
   AND #%11011101      ; [2:17]
   STA IEC_PCR         ; [4:21] data (5) and clock (1) = 0
   ORA #%00100000      ; [2:23]
   TAX                 ; [2:25] save mask in X
   BIT DPSW            ; [3:28] wait
   BIT DPSW            ; [3:31] wait
   BIT DPSW            ; [3:34]�wait
   LDA IEC_DRAN        ; [4:38]�get bit 0 & 1             <--
   ROR A               ; [2:40]�bit 0 (clock) -> bit 7
   ROR A               ; [2:42] bit 1 (data ) -> carry
   AND #$80            ; [2:44]�mask received bit 0
   ORA IEC_DRAN        ; [4:48]�get bit 2 & 3             <--
   ROL A               ; [2:50]�A = .....XXX
   ROL A               ; [2:52]�A = ....XXXX
   STA TAPE1+1         ; [3:55] store lower nibble
   LDA IEC_DRAN        ; [4:59]�get bit 4 & 5             <--
   ROR A               ; [2:61]�bit 4 (clock) -> bit 7
   ROR A               ; [2:63]�bit 5 (data ) -> carry
   AND DPSW            ; [3:66]�mask received bit 4
   ORA IEC_DRAN        ; [4:70] get bit 6 & 7            <--
   ROL A               ; [2:72]�A = .....XXX
   ROL A               ; [2:74]�A = ....XXXX
   STA CAS1            ; [3:77]�store upper nibble
   LDA IEC_DRAN        ; [4:81]�get status bits
   STX IEC_PCR         ; [4:85] data out (5) = 1
   STA DPSW            ; save status bits
   JSR Jiffy_Combine_Nibbles
   STA TBTCNT          ; received byte
   PLA
   TAX                 ; restore X
   LDA DPSW            ; restore status bits
   ROR A               ; (clock) -> bit 7
   ROR A               ; (data ) -> carry
   BPL Jiffy_Set_OK    ; clock = 0 -> OK
   BCC Jiffy_Set_EOI   ; data  = 0 -> EOI
   LDA #%01000010      ; EOI (6) and time out (1) ($42)
   JMP Set_IEC_Status
#endif

; ***************
  Jiffy_Send_Byte
; ***************

   SEI
   BIT TSFCNT        ; test to see if the device is a JiffyDOS drive
#if C64
   BVC Jiffy_Send_Byte_20

JSB_10
   LDA VIC_SPR_ENA
   BEQ Jiffy_Send_Byte_30
   PHA
   JSR Jiffy_Clear_Sprites
   JSR Jiffy_Send_Byte_30
   PLA
   STA VIC_SPR_ENA
   RTS
#endif

#if VIC
   BVS JSB_10
#endif

; ******************
  Jiffy_Send_Byte_20
; ******************

   LDA TSFCNT
   CMP #$a0
   BCS JSB_10
   JMP IEC_Send_Byte

; *************
  Jiffy_Set_EOI
; *************

   LDA #%01000000    ; bit 6 = EOI
   JSR Ora_Status

; ************
  Jiffy_Set_OK
; ************

   LDA TBTCNT

Bfc24
   CLI
   CLC
   RTS
#if VIC
JSB_10
#endif

; ******************
  Jiffy_Send_Byte_30
; ******************

#if C64
   TXA
   PHA
   LDA BSOUR
   AND #$f0
   PHA
   LDA BSOUR
   AND #15
   TAX
Bfc33     LDA CIA2_PRA
   BPL Bfc33
   AND #7
   STA BSOUR
   SEC
Bfc3d     LDA VIC_RASTER
   SBC VIC_CONTROL_1
   AND #7
   CMP #6
   BCS Bfc3d
   LDA BSOUR
   STA CIA2_PRA
   PLA
   ORA BSOUR
   STA CIA2_PRA
   LSR A
   LSR A
   AND #$f0
   ORA BSOUR
   STA CIA2_PRA
   LDA Vfc8a,X
   ORA BSOUR
   STA CIA2_PRA
   LSR A
   LSR A
   AND #$f0
   ORA BSOUR
   STA CIA2_PRA
   AND #15
   BIT TSFCNT
   BMI Bfc76
   ORA #16

Bfc76
   STA CIA2_PRA
   PLA
   TAX
   LDA BSOUR
   ORA #16
   STA CIA2_PRA
   BIT CIA2_PRA
   BPL Bfc24
   JMP IEC_Timeout

Vfc8a
   .BYTE $00,$80,$20,$a0,$40,$c0,$60,$e0
   .BYTE $10,$90,$30,$b0,$50,$d0,$70,$f0

Jiffy_fc9a
   JSR JiDi_60
   JMP PrSe_10

#endif

#if VIC
   TXA
   PHA
   LDA BSOUR
   LSR A
   LSR A
   LSR A
   LSR A
   TAX
   LDA JTABA,X
   PHA
   TXA
   LSR A
   LSR A
   TAX
   LDA JTABA,X
   STA TAPE1+1
   LDA BSOUR
   AND #15
   TAX
   LDA #2

JAAI_10
   BIT VIA1_DATN
   BEQ JAAI_10
   LDA IEC_PCR
   AND #$dd
   STA DPSW
   PHA
   PLA
   PHA
   PLA
   STA IEC_PCR
   PLA
   ORA DPSW
   STA IEC_PCR
   LDA TAPE1+1
   ORA DPSW
   ORA DPSW
   STA IEC_PCR
   LDA JTABB,X
   ORA DPSW
   STA IEC_PCR
   LDA JTABC,X
   ORA DPSW
   NOP
   STA IEC_PCR
   AND #$dd
   BIT TSFCNT
   BMI JAAI_20
   ORA #2

JAAI_20
   STA IEC_PCR
   PLA
   TAX
   LDA DPSW
   ORA #2
   STA IEC_PCR
   LDA IEC_DRAN
   AND #2
   BEQ Bfc24
   JMP IEC_Timeout

JTABA
   .byte $00,$02,$20,$22,$00,$02,$20,$22
   .byte $00,$02,$20,$22,$00,$02,$20,$22

#endif


; *****************
  Jiffy_Destination
; *****************

   JSR Get_Next_Byte_Value
   STX MYCH
   RTS
```

```
; **********
  Jiffy_fca6
; **********

   JSR Jiffy_CHKIN_PTR2

; *****
  Jfca9
; *****

   JSR CHRIN
   STA (FNADR),Y
   INY
   BIT STATUS
   BVS Bfcbb
   CPY #$fe
   BCS Bfcbb
   CMP #1
   BCS Jfca9
Bfcbb     RTS

; *************
  Jiffy_Disable
; *************

   LDX #5

Bfcbe
   LDA Vf1a3,X
   STA IERROR,X
   DEX
   BPL Bfcbe
   STX PRTY
   RTS

#if C64
   .byte $a5,$a5,$01,$29,$fd,$85,$01
#endif

#if VIC
   .byte $fd

   STA CINV
   LDA $fdea,X
   STA CINV+1
   RTS

   LDA VIA1_PCR
   ORA #$0e
   STA VIA1_PCR
   RTS

#endif

#else                ; JIFFY
; *******************
  TAPE_Find_Any_Header
; *******************

   LDA VERCKK        ; get load/verify flag
   PHA               ; save load/verify flag
   JSR TAPE_Init_Read
   PLA               ; restore load/verify flag
   STA VERCKK        ; save load/verify flag
   BCS TFAH_Ret      ; exit if error
   LDY #0
   LDA (TAPE1),Y     ; read first byte from tape buffer
   CMP #$05          ; compare with logical end of the tape
   BEQ TFAH_Ret      ; exit if end of the tape
   CMP #$01          ; compare with header for a relocatable program file
   BEQ TFAH_10      ; branch if program file header
   CMP #$03          ; compare with header for a non relocatable program file
   BEQ TFAH_10      ; branch if program file header
   CMP #$04          ; compare with data file header
   BNE TAPE_Find_Any_Header

TFAH_10
   TAX               ; copy header type
   BIT MSGFLG        ; get message mode flag
   BPL TFAH_30      ; exit if control messages off
   LDY #Msg_Found-Msg_Start
   JSR Display_Kernal_IO_Message
   LDY #$05          ; index to tape filename

TFAH_20
   LDA (TAPE1),Y     ; get byte from tape buffer
   JSR CHROUT        ; Output a character
   INY
   CPY #$15          ; compare with end+1
   BNE TFAH_20      ; loop if more to do
#if C64
   LDA JIFFYM
   JSR Delay_2JiffyM
   NOP
#endif

TFAH_30
   CLC               ; flag no error
   DEY               ; decrement index

TFAH_Ret
   RTS

; *****************
  TAPE_Write_Header
; *****************

   STA PTR1          ; save header type
   JSR TAPE_Get_Buffer_Address
   BCC TWH_Ret       ; exit if < $0200
   PUSHW(STAL)       ; push I/O start address
   PUSHW(EAL)        ; push tape end address
   LDY #$BF          ; index to header end
   LDA #' '          ; clear byte, [SPACE]

TWH_10
   STA (TAPE1),Y     ; clear header byte
   DEY               ; decrement index
   BNE TWH_10        ; loop if more to do
   LDA PTR1          ; get header type back
   STA (TAPE1),Y     ; write to header
   INY
   LDA STAL          ; get I/O start address low byte
   STA (TAPE1),Y     ; write to header
   INY
   LDA STAL+1        ; get I/O start address high byte
   STA (TAPE1),Y     ; write to header
   INY
   LDA EAL           ; get tape end address low byte
   STA (TAPE1),Y     ; write to header
   INY
   LDA EAL+1         ; get tape end address high byte
   STA (TAPE1),Y     ; write to header
   INY
   STY PTR2          ; save index
   LDY #$00          ; clear Y
   STY PTR1          ; clear name index

TWH_20
   LDY PTR1          ; get name index
   CPY FNLEN         ; compare with file name length
   BEQ TWH_30        ; exit loop if all done
   LDA (FNADR ),Y    ; get file name byte
   LDY PTR2          ; get buffer index
   STA (TAPE1),Y     ; save file name byte to buffer
   INC PTR1          ; increment file name index
   INC PTR2          ; increment tape buffer index
   BNE TWH_20        ; loop, branch always

TWH_30
   JSR TAPE_Set_Buffer_Pointer
   LDA #$69          ; set write lead cycle count
   STA RIPRTY        ; save write lead cycle count
   JSR TAPE_Write
   TAY
   PULLW(EAL)
   PULLW(STAL)
   TYA

TWH_Ret
   RTS

; ***********************
  TAPE_Get_Buffer_Address
; ***********************

   LDXY(TAPE1)       ; get tape buffer start pointer low byte
   CPY #2            ; compare high byte with $02xx
   RTS

; ***********************
  TAPE_Set_Buffer_Pointer
; ***********************

   JSR TAPE_Get_Buffer_Address
   TXA               ; copy tape buffer start pointer low byte
   STA STAL          ; save as I/O address pointer low byte
   CLC
   ADC #$C0          ; add buffer length low byte
   STA EAL           ; save tape buffer end pointer low byte
   TYA               ; copy tape buffer start pointer high byte
   STA STAL+1        ; save as I/O address pointer high byte
   ADC #$00          ; add buffer length high byte
   STA EAL+1         ; save tape buffer end pointer high byte
   RTS
```

```
; ********************
  TAPE_Find_Fileheader
; ********************

   JSR TAPE_Find_Any_Header
   BCS TFF_Ret       ; just exit if error
   LDY #$05          ; index to name
   STY PTR2          ; save as tape buffer index
   LDY #$00          ; clear Y
   STY PTR1          ; save as name buffer index

TFF_10
   CPY FNLEN         ; compare with file name length
   BEQ TFF_20        ; ok exit if match
   LDA (FNADR ),Y    ; get file name byte
   LDY PTR2          ; get index to tape buffer
   CMP (TAPE1),Y     ; compare with tape header name byte
   BNE TAPE_Find_Fileheader
   INC PTR1          ; else increment name buffer index
   INC PTR2          ; increment tape buffer index
   LDY PTR1          ; get name buffer index
   BNE TFF_10        ; loop, branch always

TFF_20
   CLC               ; flag ok

TFF_Ret
   RTS

; ***************************
  TAPE_Advance_Buffer_Pointer
; ***************************

   JSR TAPE_Get_Buffer_Address
   INC BUFPNT        ; increment tape buffer index
   LDY BUFPNT        ; get tape buffer index
   CPY #$C0          ; compare with buffer length
   RTS

; *************
  Wait_For_Play
; *************

   JSR TAPE_Sense
   BEQ TASe_10      ; exit if switch closed
   LDY #Msg_Play-Msg_Start

WFP_10
   JSR Display_Kernal_IO_Message

WFP_20
   JSR TAPE_Abort_On_STOP
   JSR TAPE_Sense
   BNE WFP_20        ; loop if cassette switch open
   LDY #Msg_ok-Msg_Start
   JMP Display_Kernal_IO_Message

; **********
  TAPE_Sense
; **********
```

```
#if C64
   LDA #$10
   BIT R6510
   BNE TASe_10
   BIT R6510
#endif
#if VIC
   LDA #$40           ; mask for cassette switch
   BIT IEC_DRAN       ; test VIA 1 DRA, no handshake
   BNE TASe_10        ; branch if cassette sense high
   BIT IEC_DRAN       ; test VIA 1 DRA again
#endif

TASe_10
   CLC
   RTS

; ********************
  TAPE_Wait_For_Record
; ********************

   JSR TAPE_Sense
   BEQ TASe_10        ; exit if switch closed
   LDY #Msg_Record-Msg_Start
   BNE WFP_10         ; display message and wait for switch, branch always

; **************
  TAPE_Init_Read
; **************

   LDA #0
   STA STATUS         ; clear serial status byte
   STA VERCKK         ; clear the load/verify flag
   JSR TAPE_Set_Buffer_Pointer

; *********
  TAPE_Read
; *********

   JSR Wait_For_Play
   BCS TAWR_10       ; exit if STOP was pressed, uses further BCS at target
   SEI                ; disable interrupts
   LDA #0
   STA RIDATA         ;.
   STA BITTS          ;.
   STA CMPO           ; clear tape timing constant min byte
   STA PTR1           ; clear tape pass 1 error log/char buffer
   STA PTR2           ; clear tape pass 2 error log corrected
   STA DPSW           ; clear byte received flag
#if C64
   LDA #$90
#endif
#if VIC
   LDA #$82           ; enable CA1 interrupt
#endif
   LDX #$0E           ; set index for tape read vector
   BNE TAWR_20       ; go do tape read/write, branch always

; ***************
  Init_Tape_Write
; ***************
```

```
   JSR TAPE_Set_Buffer_Pointer

; ********************
  TAPE_Write_With_Lead
; ********************

   LDA #20           ; set write lead cycle count
   STA RIPRTY        ; save write lead cycle count

; **********
  TAPE_Write
; **********

   JSR TAPE_Wait_For_Record

TAWR_10
   BCS Clear_Saved_IRQ_Address
   SEI               ; disable interrupts
#if C64
   LDA #$82
#endif
#if VIC
   LDA #$A0          ; enable VIA 2 T2 interrupt
#endif
   LDX #$08          ; set index for tape write tape leader vector

TAWR_20
   LDY #$7F          ; disable all interrupts
   STY VIA2_IER      ; set VIA 2 IER, disable interrupts
   STA VIA2_IER      ; set VIA 2 IER, enable interrupts according to A
#if C64
   LDA CIA1_CRA
   ORA #$19
   STA CIA1_CRB      ; CIA1 Control Register B
   AND #$91
   STA TODSNS
#endif
   JSR RS232_Stop
#if C64
   LDA VIC_CONTROL_1
   AND #$EF          ; clear bit 8 of raster value
   STA VIC_CONTROL_1
#endif
   LDA CINV          ; get IRQ vector low byte
   STA IRQTMP        ; save IRQ vector low byte
   LDA CINV+1        ; get IRQ vector high byte
   STA IRQTMP+1      ; save IRQ vector high byte
   JSR TAPE_Set_IRQ_Vector
   LDA #$02          ; set copies count. the first copy is the load copy, the
   STA FSBLK         ; save copies count
   JSR TAPE_New_Byte_Setup
#if C64
   LDA R6510
#endif
#if VIC
   LDA VIA1_PCR      ; get VIA 1 PCR
#endif
#if C64
   AND #$1f
   STA R6510
#endif
#if VIC
```

```
   AND #$FD          ; CA2 low, turn on tape motor
   ORA #$0C          ; manual output mode
   STA VIA1_PCR      ; set VIA 1 PCR
#endif
   STA CAS1          ; set tape motor interlock
   LDX #$FF          ; outer loop count

TAWR_30
   LDY #$FF          ; inner loop count

TAWR_40
   DEY               ; decrement inner loop count
   BNE TAWR_40      ; loop if more to do
   DEX               ; decrement outer loop count
   BNE TAWR_30      ; loop if more to do
#if VIC
   STA VIA2_T2CH     ; set VIA 2 T2C_h
#endif
   CLI               ; enable tape interrupts

TAWR_50
   LDA IRQTMP+1      ; get saved IRQ high byte
   CMP CINV+1        ; compare with the current IRQ high byte
   CLC               ; flag ok
   BEQ Clear_Saved_IRQ_Address
   JSR TAPE_Abort_On_STOP
#if C64
   JSR Look_For_Special_Keys
#endif
#if VIC
   LDA IEC_IFR       ; get VIA 2 IFR
   AND #$40          ; mask T1 interrupt
   BEQ TAWR_50      ; loop if not T1 interrupt
   LDA RS2_TIM_LOW   ; get VIA 1 T1C_l, clear T1 flag
   JSR Kernal_UDTIM  ; Update the system clock
#endif
   JMP TAWR_50      ; loop

; ******************
  TAPE_Abort_On_STOP
; ******************

   JSR STOP          ; Check if stop key is pressed
   CLC               ; flag no stop
   BNE CSIA_Ret      ; exit if no stop
   JSR Restoring_After_STOP
   SEC               ; flag stopped
   PLA               ; dump return address low byte
   PLA               ; dump return address high byte

; =======================
  Clear_Saved_IRQ_Address
; =======================

   LDA #0
   STA IRQTMP+1      ; clear saved IRQ address high byte

CSIA_Ret
   RTS

; **************
  TAPE_Set_Timer
```

```
; **************

   STX CMPO+1        ; save tape timing constant max byte
   LDA CMPO          ; get tape timing constant min byte
   ASL A             ; *2
   ASL A             ; *4
   CLC
   ADC CMPO          ; add tape timing constant min byte *5
   CLC
   ADC CMPO+1        ; add tape timing constant max byte
   STA CMPO+1        ; save tape timing constant max byte
   LDA #$00          ;.
   BIT CMPO          ; test tape timing constant min byte
   BMI TST_10        ; branch if b7 set
   ROL A             ; else shift carry into ??

TST_10
   ASL CMPO+1        ; shift tape timing constant max byte
   ROL A
   ASL CMPO+1        ; shift tape timing constant max byte
   ROL A
   TAX

TST_20
   LDA VIA2_T2CL     ; get VIA 2 T2C_l
#if C64
   CMP #$16
#endif
#if VIC
   CMP #$15
#endif
   BCC TST_20        ; loop if less
   ADC CMPO+1        ; add tape timing constant max byte
   STA VIA2_T1CL     ; set VIA 2 T1C_l
   TXA
   ADC VIA2_T2CH     ; add VIA 2 T2C_h
   STA VIA2_T1CH     ; set VIA 2 T1C_h
#if C64
   LDA TODSNS
   STA CIA1_CRA
   STA TD1IRQ
   LDA CIA1_ICR      ; CIA1 Interrupt Control Register
   AND #$10
   BEQ TST_30
   LDA #>TST_30
   PHA
   LDA #<TST_30
   PHA
   JMP Clear_BREAK_Flag

TST_30
#endif
   CLI
   RTS


;**********************************************************************;
;
;   On Commodore computers, the streams consist of four kinds of symbols
;   that denote different kinds of low-to-high-to-low transitions on the
;   read or write signals of the Commodore cassette interface.
;
```

```
;   A A break in the communications, or a pulse with very long cycle
;      time.
;
;   B A short pulse, whose cycle time typically ranges from 296 to 424
;      microseconds, depending on the computer model.
;
;   C A medium-length pulse, whose cycle time typically ranges from
;      440 to 576 microseconds, depending on the computer model.
;
;   D A long pulse, whose cycle time typically ranges from 600 to 744
;      microseconds, depending on the computer model.
;
;   The actual interpretation of the serial data takes a little more work to
; explain. The typical ROM tape loader (and the turbo loaders) will
; initialize a timer with a specified value and start it counting down. If
; either the tape data changes or the timer runs out, an IRQ will occur. The
; loader will determine which condition caused the IRQ. If the tape data
; changed before the timer ran out, we have a short pulse, or a "0" bit. If
; the timer ran out first, we have a long pulse, or a "1" bit. Doing this
; continuously and we decode the entire file.
; read tape bits, IRQ routine
; read T2C which has been counting down from $FFFF. subtract this from $FFFF

; *************
  TAPE_Read_IRQ
; *************

   LDX VIA2_T2CH     ; get VIA 2 T2C_h
   LDY #$FF
   TYA
   SBC VIA2_T2CL     ; subtract VIA 2 T2C_l
   CPX VIA2_T2CH     ; compare VIA 2 T2C_h with previous
   BNE TAPE_Read_IRQ ; loop if timer low byte rolled over
   STX CMPO+1        ; save tape timing constant max byte
   TAX               ; copy $FF - T2C_l
   STY VIA2_T2CL     ; set VIA 2 T2C_l to $FF
   STY VIA2_T2CH     ; set VIA 2 T2C_h to $FF
#if C64
   LDA #$19
   STA CIA1_CRB      ; CIA1 Control Register B
   LDA CIA1_ICR      ; CIA1 Interrupt Control Register
   STA TRDTMP
#endif
   TYA
   SBC CMPO+1        ; subtract tape timing constant max byte
   STX CMPO+1        ; save tape timing constant max byte
   LSR A             ; A = $FF - T2C_h >> 1
   ROR CMPO+1        ; shift tape timing constant max byte
   LSR A             ; A = $FF - T2C_h >> 1
   ROR CMPO+1        ; shift tape timing constant max byte
   LDA CMPO          ; get tape timing constant min byte
   CLC
   ADC #$3C
#if VIC
   BIT KEYB_ROW      ; test VIA 2 DRA, keyboard row
#endif
   CMP CMPO+1        ; compare with tape timing constant max byte
   BCS TARI_14       ; branch if min + $3C >= ($FFFF - T2C) >> 2
   LDX DPSW          ; get byte received flag
   BEQ TARI_02       ; branch if not byte received
   JMP TAPE_Store_Char      ; store tape character
```

```
TARI_02
   LDX TSFCNT        ; get EOI flag byte
   BMI TARI_04
   LDX #$00
   ADC #$30
   ADC CMPO          ; add tape timing constant min byte
   CMP CMPO+1        ; compare with tape timing constant max byte
   BCS TARI_08
   INX
   ADC #$26
   ADC CMPO          ; add tape timing constant min byte
   CMP CMPO+1        ; compare with tape timing constant max byte
   BCS TARI_10
   ADC #$2C
   ADC CMPO          ; add tape timing constant min byte
   CMP CMPO+1        ; compare with tape timing constant max byte
   BCC TARI_06

TARI_04
   JMP TARI_30

TARI_06
   LDA BITTS         ; get bit count
   BEQ TARI_14       ; branch if zero
   STA BITCI         ; save receiver bit count in
   BNE TARI_14       ; branch always

TARI_08
   INC RINONE        ; increment ?? start bit check flag
   BCS TARI_12

TARI_10
   DEC RINONE        ; decrement ?? start bit check flag

TARI_12
   SEC
   SBC #$13
   SBC CMPO+1        ; subtract tape timing constant max byte
   ADC SVXT          ; add timing constant for tape
   STA SVXT          ; save timing constant for tape
   LDA TBTCNT        ; get tape bit cycle phase
   EOR #$01
   STA TBTCNT        ; save tape bit cycle phase
   BEQ TARI_22
   STX LASTKY

TARI_14
   LDA BITTS         ; get bit count
   BEQ TARI_20       ; exit if zero

#if C64
   LDA TRDTMP
   AND #1
   BNE TARI_16
   LDA TD1IRQ
   BNE TARI_20

TARI_16
#endif
#if VIC
   BIT IEC_IFR       ; test get 2 IFR
   BVC TARI_20       ; exit if no T1 interrupt
```

```
#endif
   LDA #$00
   STA TBTCNT        ; clear tape bit cycle phase
#if C64
   STA TD1IRQ
#endif
   LDA TSFCNT        ;.get EOI flag byte
   BPL TARI_28
   BMI TARI_04

TARI_18
   LDX #$A6          ; set timimg max byte
   JSR TAPE_Set_Timer
   LDA PRTY
   BNE TARI_06

TARI_20
   JMP Exit_IRQ      ; restore registers and exit interrupt

TARI_22
   LDA SVXT          ; get timing constant for tape
   BEQ TARI_26
   BMI TARI_24
   DEC CMPO          ; decrement tape timing constant min byte
   .byte   $2C

TARI_24
   INC CMPO          ; increment tape timing constant min byte

TARI_26
   LDA #0
   STA SVXT          ; clear timing constant for tape
   CPX LASTKY
   BNE TARI_28
   TXA
   BNE TARI_06
   LDA RINONE        ; get start bit check flag
   BMI TARI_14
   CMP #$10
   BCC TARI_14
   STA SYNO          ; save cassette block synchronization number
   BCS TARI_14

TARI_28
   TXA
   EOR PRTY
   STA PRTY
   LDA BITTS
   BEQ TARI_20
   DEC TSFCNT        ; decrement EOI flag byte
   BMI TARI_18
   LSR LASTKY
   ROR MYCH          ; parity count
   LDX #$DA          ; set timimg max byte
   JSR TAPE_Set_Timer
   JMP Exit_IRQ      ; restore registers and exit interrupt

TARI_30
   LDA SYNO          ; get cassette block synchronization number
   BEQ TARI_32
   LDA BITTS
   BEQ TARI_34
```

```
TARI_32
   LDA TSFCNT        ; get EOI flag byte
#if C64
   BMI TARI_34
   JMP TARI_10
#endif
#if VIC
   BPL TARI_10
#endif

TARI_34
   LSR CMPO+1        ; shift tape timing constant max byte
   LDA #$93
   SEC
   SBC CMPO+1        ; subtract tape timing constant max byte
   ADC CMPO          ; add tape timing constant min byte
   ASL A
   TAX               ; copy timimg high byte
   JSR TAPE_Set_Timer      ; set timing
   INC DPSW
   LDA BITTS
   BNE TARI_36
   LDA SYNO          ; get cassette block synchronization number
   BEQ TARI_40
   STA BITCI         ; save receiver bit count in
   LDA #$00
   STA SYNO          ; clear cassette block synchronization number
#if C64
   LDA #$81
#endif
#if VIC
   LDA #$C0          ; enable T1 interrupt
#endif
   STA VIA2_IER      ; set VIA 2 IER
   STA BITTS

TARI_36
   LDA SYNO          ; get cassette block synchronization number
   STA NXTBIT
   BEQ TARI_38
   LDA #$00
   STA BITTS
#if C64
   LDA #1
#endif
#if VIC
   LDA #$40          ; disable T1 interrupt
#endif
   STA VIA2_IER      ; set VIA 2 IER

TARI_38
   LDA MYCH          ; parity count
   STA ROPRTY        ; save RS232 parity byte
   LDA BITCI         ; get receiver bit count in
   ORA RINONE        ; OR with start bit check flag
   STA RODATA

TARI_40
   JMP Exit_IRQ      ; restore registers and exit interrupt

; ===============
  TAPE_Store_Char
; ===============

   JSR TAPE_New_Byte_Setup
   STA DPSW          ; clear byte received flag
   LDX #$DA          ; set timimg max byte
   JSR TAPE_Set_Timer
   LDA FSBLK         ; get copies count
   BEQ TASC_10
   STA INBIT         ; save receiver input bit temporary storage

TASC_10
   LDA #$0F
   BIT RIDATA
   BPL TASC_40
   LDA NXTBIT
   BNE TASC_20
   LDX FSBLK         ; get copies count
   DEX
   BNE TASC_30       ; if ?? restore registers and exit interrupt
   LDA #$08          ; set short block
   JSR Ora_Status
   BNE TASC_30       ; restore registers and exit interrupt, branch always

TASC_20
   LDA #$00
   STA RIDATA

TASC_30
   JMP Exit_IRQ      ; restore registers and exit interrupt

TASC_40
   BVS TASC_64
   BNE TASC_60
   LDA NXTBIT
   BNE TASC_30
   LDA RODATA
   BNE TASC_30
   LDA INBIT         ; get receiver input bit temporary storage
   LSR A
   LDA ROPRTY        ; get RS232 parity byte
   BMI TASC_50
   BCC TASC_62
   CLC

TASC_50
   BCS TASC_62
   AND #$0F
   STA RIDATA

TASC_60
   DEC RIDATA
   BNE TASC_30
   LDA #$40
   STA RIDATA
   JSR Set_IO_Start
   LDA #0
   STA RIPRTY
   BEQ TASC_30      ; branch always

TASC_62
   LDA #$80
   STA RIDATA
   BNE TASC_30      ; restore registers and exit interrupt, branch always

TASC_64
   LDA NXTBIT
   BEQ TASC_66

TASC_65
   LDA #$04
   JSR Ora_Status
   LDA #$00
   JMP TASC_84

TASC_66
   JSR Check_IO_End
   BCC TASC_68
   JMP TASC_82

TASC_68
   LDX INBIT         ; get receiver input bit temporary storage
   DEX
   BEQ TASC_72
   LDA VERCKK
   BEQ TASC_70
   LDY #0
   LDA ROPRTY
   CMP (SAL),Y
   BEQ TASC_70
   LDA #$01
   STA RODATA

TASC_70
   LDA RODATA
   BEQ TASC_78
   LDX #$3D
   CPX PTR1
   BCC TASC_76
   LDX PTR1
   LDA SAL+1
   STA STACK+1,X
   LDA SAL
   STA STACK,X
   INX
   INX
   STX PTR1
   JMP TASC_78

TASC_72
   LDX PTR2
   CPX PTR1
   BEQ TASC_80
   LDA SAL
   CMP STACK,X
   BNE TASC_80
   LDA SAL+1
   CMP STACK+1,X
   BNE TASC_80
   INC PTR2
   INC PTR2
   LDA VERCKK        ; get load/verify flag
   BEQ TASC_74      ; branch if load
   LDA ROPRTY        ; get RS232 parity byte
   LDY #$00
   CMP (SAL),Y
   BEQ TASC_80
   INY
   STY RODATA

TASC_74
   LDA RODATA
   BEQ TASC_78

TASC_76
   LDA #$10
   JSR Ora_Status
   BNE TASC_80

TASC_78
   LDA VERCKK        ; get load/verify flag
   BNE TASC_80       ; branch if verify
   TAY
   LDA ROPRTY        ; get RS232 parity byte
   STA (SAL),Y

TASC_80
   JSR Inc_SAL_Word
   BNE TASC_92      ; restore registers and exit interrupt, branch always

TASC_82
   LDA #$80

TASC_84
   STA RIDATA
#if C64
   SEI
   LDX #1
   STX CIA1_ICR      ; CIA1 Interrupt Control Register
   LDX CIA1_ICR      ; CIA1 Interrupt Control Register
#endif
   LDX FSBLK         ; get copies count
   DEX
   BMI TASC_86
   STX FSBLK         ; save copies count

TASC_86
   DEC INBIT         ; decrement receiver input bit temporary storage
   BEQ TASC_88
   LDA PTR1
   BNE TASC_92       ; if ?? restore registers and exit interrupt
   STA FSBLK         ; save copies count
   BEQ TASC_92       ; restore registers and exit interrupt, branch always

TASC_88
   JSR Restoring_After_STOP
   JSR Set_IO_Start
   LDY #0
   STY RIPRTY        ; clear checksum

TASC_90
   LDA (SAL),Y       ; get byte from buffer
   EOR RIPRTY        ; XOR with checksum
   STA RIPRTY        ; save new checksum
   JSR Inc_SAL_Word
   JSR Check_IO_End
   BCC TASC_90        ; loop if not at end
   LDA RIPRTY         ; get computed checksum
   EOR ROPRTY         ; compare with stored checksum ??
   BEQ TASC_92        ; if checksum ok restore registers and exit interrupt
   LDA #$20           ; else set checksum error
   JSR Ora_Status

TASC_92
   JMP Exit_IRQ       ; restore registers and exit interrupt

; ************
  Set_IO_Start
; ************
   LDA STAL+1         ; get I/O start address high byte
   STA SAL+1          ; set buffer address high byte
   LDA STAL           ; get I/O start address low byte
   STA SAL            ; set buffer address low byte
   RTS

; ******************
  TAPE_New_Byte_Setup
; ******************

   LDA #$08           ; eight bits to do
   STA TSFCNT         ; set bit count
   LDA #0
   STA TBTCNT         ; clear tape bit cycle phase
   STA BITCI          ; clear start bit first cycle done flag
   STA PRTY           ; clear byte parity
   STA RINONE         ; clear start bit check flag, set no start bit yet
   RTS

; *************
  TAPE_Write_Bit
; *************

; this routine tests the least significant bit in the tape write byte
; and sets VIA 2 T2 depending on the state of the bit. if the bit is a 1
; a time of $00B0 cycles is set, if the bot is a 0 a time of $0060
; cycles is set. note that this routine does not shift the bits of the
; tape write byte but uses a copy of that byte, the byte itself is
; shifted elsewhere

   LDA ROPRTY         ; get tape write byte
   LSR A              ; shift lsb into Cb
   LDA #$60           ; set time constant low byte for bit = 0
   BCC TAPE_Write_Timer

; ******************
  TAPE_Timer_Bit_Is_1
; ******************

   LDA #$B0           ; set time constant low byte for bit = 1

; ***************
  TAPE_Write_Timer
; ***************

   LDX #$00           ; set time constant high byte

; ***************
  TAPE_Start_Timer
```

```
; ****************

   STA VIA2_T2CL     ; set VIA 2 T2C_l
   STX VIA2_T2CH     ; set VIA 2 T2C_h
#if C64
   LDA CIA1_ICR    ; CIA1 Interrupt Control Register
   LDA #$19
   STA CIA1_CRB    ; CIA1 Control Register B
   LDA R6510
#endif
#if VIC
   LDA KEYB_COL      ; get VIA 2 DRB, keyboard column
#endif
   EOR #$08          ; toggle tape out bit
#if C64
    STA R6510
#endif
#if VIC
   STA KEYB_COL      ; set VIA 2 DRB
#endif
   AND #$08          ; mask tape out bit
   RTS

TAST_10
   SEC               ; set carry flag
#if C64
   ROR RODATA
#endif
#if VIC
   ROR SAL+1         ; set buffer address high byte negative, flag all sync,
#endif
   BMI TAWI_15      ; restore registers and exit interrupt, branch always

; **************
  TAPE_Write_IRQ
; **************

; this is the routine that writes the bits to the tape. it is called each time VIA 2 T2
; times out and checks if the start bit is done, if so checks if the data bits are done,
; if so it checks if the byte is done, if so it checks if the synchronisation bytes are
; done, if so it checks if the data bytes are done, if so it checks if the checksum byte
; is done, if so it checks if both the load and verify copies have been done, if so it
; stops the tape

   LDA BITCI         ; get start bit first cycle done flag
   BNE TAWI_05       ; if first cycle done go do rest of byte

; each byte sent starts with two half cycles of $0110 ststem clocks and the whole block
; ends with two more such half cycles

   LDA #$10          ; set first start cycle time constant low byte
   LDX #$01          ; set first start cycle time constant high byte
   JSR TAPE_Start_Timer
   BNE TAWI_15      ; if first half cycle go restore registers and exit
   INC BITCI         ; set start bit first start cycle done flag
#if C64
   LDA RODATA
#endif
#if VIC
   LDA SAL+1         ; get buffer address high byte
#endif
   BPL TAWI_15      ; if block not complete go restore registers and exit
   JMP TAWI_55      ; else do tape routine, block complete exit

; continue tape byte write. the first start cycle, both half cycles of it, is complete
; so the routine drops straight through to here

TAWI_05
   LDA RINONE        ; get start bit check flag
   BNE TAWI_10       ; if the start bit is complete go send the byte bits

; after the two half cycles of $0110 ststem clocks the start bit is completed with two
; half cycles of $00B0 system clocks. this is the same as the first part of a 1 bit

   JSR TAPE_Timer_Bit_Is_1
   BNE TAWI_15       ; if first half cycle go restore registers and exit
   INC RINONE        ; set start bit check flag
   BNE TAWI_15       ; restore registers and exit interrupt, branch always

; continue tape byte write. the start bit, both cycles of it, is complete so the routine
; drops straight through to here. now the cycle pairs for each bit, and the parity bit,
; are sent

TAWI_10
   JSR TAPE_Write_Bit
   BNE TAWI_15       ; if first half cycle go restore registers and exit
   LDA TBTCNT        ; get tape bit cycle phase
   EOR #$01          ; toggle b0
   STA TBTCNT        ; save tape bit cycle phase
   BEQ TAWI_20       ; if bit cycle phase complete go setup for next bit

; each bit is written as two full cycles. a 1 is sent as a full cycle of $0160 system
; clocks then a full cycle of $00C0 system clocks. a 0 is sent as a full cycle of $00C0
; system clocks then a full cycle of $0160 system clocks. to do this each bit from the
; write byte is inverted during the second bit cycle phase. as the bit is inverted it
; is also added to the, one bit, parity count for this byte

   LDA ROPRTY        ; get tape write byte
   EOR #$01          ; invert bit being sent
   STA ROPRTY        ; save tape write byte
   AND #$01          ; mask b0
   EOR PRTY          ; EOR with tape write byte parity bit
   STA PRTY          ; save tape write byte parity bit

TAWI_15
   JMP Exit_IRQ      ; restore registers and exit interrupt

; the bit cycle phase is complete so shift out the just written bit and test for byte
; end

TAWI_20
   LSR ROPRTY        ; shift bit out of tape write byte
   DEC TSFCNT        ; decrement tape write bit count
   LDA TSFCNT        ; get tape write bit count
   BEQ TAWI_45       ; if all the data bits have been written go setup for
   BPL TAWI_15       ; if all the data bits are not yet sent just restore the

; do next tape byte
; the byte is complete. the start bit, data bits and parity bit have been written to
; the tape so setup for the next byte

TAWI_25
   JSR TAPE_New_Byte_Setup
   CLI               ; enable interrupts
   LDA CNTDN         ; get cassette synchronization character count
   BEQ TAWI_35       ; if synchronisation characters done go do block data

; at the start of each block sent to tape there are a number of synchronisation bytes
; that count down to the actual data. the commodore tape system saves two copies of all
; the tape data, the first is loaded and is indicated by the synchronisation bytes
; having b7 set, and the second copy is indicated by the synchronisation bytes having b7
; clear. the sequence goes $09, $08, ..... $02, $01, data bytes

   LDX #$00          ; clear X
   STX LASTKY        ; clear checksum byte
   DEC CNTDN         ; decrement cassette synchronization byte count
   LDX FSBLK         ; get cassette copies count
   CPX #$02          ; compare with load block indicator
   BNE TAWI_30       ; branch if not the load block
   ORA #$80          ; this is the load block so make the synchronisation count
                     ; go $89, $88, ..... $82, $81
TAWI_30
   STA ROPRTY        ; save the synchronisation byte as the tape write byte
   BNE TAWI_15       ; restore registers and exit interrupt, branch always

; the synchronization bytes have been done so now check and do the actual block data

TAWI_35
   JSR Check_IO_End
   BCC TAWI_40       ; if not all done yet go get the byte to send
   BNE TAST_10       ; if pointer > end go flag block done and exit interrupt
   INC SAL+1         ; increment buffer pointer high byte, this means the block
   LDA LASTKY        ; get checksum byte
   STA ROPRTY        ; save checksum as tape write byte
   BCS TAWI_15       ; restore registers and exit interrupt, branch always

; the block isn't finished so get the next byte to write to tape

TAWI_40
   LDY #0
   LDA (SAL),Y       ; get byte from buffer
   STA ROPRTY        ; save as tape write byte
   EOR LASTKY        ; XOR with checksum byte
   STA LASTKY        ; save new checksum byte
   JSR Inc_SAL_Word
   BNE TAWI_15       ; restore registers and exit interrupt, branch always

; set parity as next bit and exit interrupt

TAWI_45
   LDA PRTY          ; get parity bit
   EOR #$01          ; toggle it
   STA ROPRTY        ; save as tape write byte

TAWI_50
   JMP Exit_IRQ      ; restore registers and exit interrupt

; tape routine, block complete exit

TAWI_55
   DEC FSBLK         ; decrement copies remaining to read/write
   BNE TAWI_60       ; branch if more to do
   JSR TAPE_Stop_Motor

TAWI_60
   LDA #$50          ; set tape write leader count
   STA INBIT         ; save tape write leader count
   LDX #$08          ; set index for write tape leader vector
   SEI               ; disable interrupts
   JSR TAPE_Set_IRQ_Vector
   BNE TAWI_50       ; restore registers and exit interrupt, branch always

TAPE_Write_Leader
   LDA #$78          ; set time constant low byte for bit = leader
   JSR TAPE_Write_Timer
   BNE TAWI_50       ; if tape bit high restore registers and exit interrupt
   DEC INBIT         ; decrement cycle count
   BNE TAWI_50       ; if not all done restore registers and exit interrupt
   JSR TAPE_New_Byte_Setup
   DEC RIPRTY        ; decrement cassette leader count
   BPL TAWI_50       ; if not all done restore registers and exit interrupt
   LDX #$0A          ; set index for tape write vector
   JSR TAPE_Set_IRQ_Vector
   CLI               ; enable interrupts
   INC RIPRTY        ; clear cassette leader counter, was $FF
   LDA FSBLK         ; get cassette block count
   BEQ Reset_TAPE_IRQ      ; if all done restore everything for STOP and exit interrupt
   JSR Set_IO_Start
   LDX #$09          ; set nine synchronisation bytes
   STX CNTDN         ; save cassette synchronization byte count
#if C64
   STX RODATA
#endif
   BNE TAWI_25      ; go do next tape byte, branch always

; ********************
  Restoring_After_STOP
; ********************

   PHP               ; save status
   SEI               ; disable interrupts
#if C64
   LDA VIC_CONTROL_1
   ORA #$10          ; set DEN bit
   STA VIC_CONTROL_1
#endif
   JSR TAPE_Stop_Motor
   LDA #$7F          ; disable all interrupts
#if C64
   STA CIA1_ICR      ; CIA1 Interrupt Control Register
#endif
#if VIC
   STA $912E
#endif
#if VIC
   LDA #$F7          ; set keyboard column 3 active
   STA KEYB_COL      ; set VIA 2 DRB, keyboard column
   LDA #$40          ; set T1 free run, T2 clock �2,
   STA VIA2_ACR      ; set VIA 2 ACR
#endif
   JSR Program_Timer_A
   LDA IRQTMP+1      ; get saved IRQ vector high byte
   BEQ RAS_10        ; branch if null
   STA CINV+1        ; restore IRQ vector high byte
   LDA IRQTMP        ; get saved IRQ vector low byte
   STA CINV          ; restore IRQ vector low byte

RAS_10
```

```
   PLP               ; restore status
   RTS

; **************
  Reset_TAPE_IRQ
; **************

   JSR Restoring_After_STOP
   BEQ TAWI_50      ; restore registers and exit interrupt, branch always

; *******************
  TAPE_Set_IRQ_Vector
; *******************

   LDA TAPE_IRQ_Vectors-8,X  ; get tape IRQ vector low byte
   STA CINV          ; set IRQ vector low byte
   LDA TAPE_IRQ_Vectors-7,X  ; get tape IRQ vector high byte
   STA CINV+1        ; set IRQ vector high byte
   RTS

; ***************
  TAPE_Stop_Motor
; ***************

#if C64
   LDA R6510
   ORA #$20
   STA R6510
#endif
#if VIC
   LDA VIA1_PCR      ; get VIA 1 PCR
   ORA #$0E          ; set CA2 high, cassette motor off
   STA VIA1_PCR      ; set VIA 1 PCR
#endif
   RTS
#endif               ; JIFFY

; ************
  Check_IO_End
; ************

   SEC
   LDA SAL           ; get buffer address low byte
   SBC EAL           ; subtract buffer end low byte
   LDA SAL+1         ; get buffer address high byte
   SBC EAL+1         ; subtract buffer end high byte
   RTS

; ************
  Inc_SAL_Word
; ************

   INC SAL
   BNE ISW_Ret
   INC SAL+1
ISW_Ret
   RTS

; ***********
  Entry_RESET        ; hardware reset
; ***********
```

```
   LDX #$FF          ; set X for stack
   SEI               ; disable interrupts
   TXS               ; clear stack
   CLD               ; clear decimal mode
   JSR Scan_Autostart_Signature
   BNE HARE_10       ; if not there continue Vic startup
   JMP (OPTION_ROM)

HARE_10
#if C64
   STX VIC_CONTROL_2
   JSR Initialise_IO
   JSR Init_RAM
   JSR Kernal_RESTOR
   JSR Init_Editor
#endif
#if VIC
   JSR Init_RAM
   JSR Kernal_RESTOR
   JSR Initialise_IO
   JSR Initialise_Hardware
#endif
   CLI               ; enable interrupts
   JMP (BASIC_ROM)   ; start BASIC

; ************************
  Scan_Autostart_Signature
; ************************

   LDX #5            ; five characters to test

SAAS_Loop
   LDA ROM_SIG-1,X   ; get test character
   CMP OPTION_ROM+3,X; compare with byte in block A000
   BNE SAAS_Exit     ; exit if no match
   DEX               ; decrement index
   BNE SAAS_Loop

SAAS_Exit
   RTS

#if C64
ROM_SIG .byte $C3,$C2,$CD,"80" ; CBM80
#endif
#if VIC
ROM_SIG .byte "A0",$C3,$C2,$CD ; A0CBM
#endif

; *************
  Kernal_RESTOR
; *************

   LDX #<Kernal_Vectors    ; pointer to vector table low byte
   LDY #>Kernal_Vectors    ; pointer to vector table high byte
   CLC               ; flag set vectors

; *************
  Kernal_VECTOR
; *************

   STXY(MEMUSS)      ; save pointer
   LDY #$1F          ; set byte count
```

```
KeVE_10
#if JIFFY & VIC
   LDA (MEMUSS),Y    ; bugfix for the original code, that writes
   BCC KeVe_30       ; into the RAM bank at address of the
   LDA CINV,Y        ; Kernal vectors in ROM
#else
   LDA CINV,Y        ; read vector byte from vectors
   BCS KeVE_20       ; if read vectors skip the read from XY
   LDA (MEMUSS),Y    ; read vector byte from (XY)
#endif

KeVE_20
   STA (MEMUSS),Y    ; save byte to (XY) [may be ROM address]

KeVe_30
   STA CINV,Y        ; save byte to vector
   DEY               ; decrement index
   BPL KeVE_10       ; loop if more to do
   RTS

; **************
  Kernal_Vectors
; **************

   .word   Default_IRQ   ; CINV   IRQ vector
   .word   Default_BRK   ; CBINV  BRK vector
   .word   Default_NMI   ; NMINV  NMI vector
   .word   Kernal_OPEN   ; IOPEN  Open a logical file
   .word   Kernal_CLOSE  ; ICLOSE close a logical file
   .word   Kernal_ICHKIN ; ICHKIN open channel for input
   .word   Kernal_CHKOUT ; ICKOUT open channel for output
   .word   Kernal_CLRCHN ; ICLRCH clear I/O channels
   .word   Kernal_CHRIN  ; IBASIN get a character from the input channel
   .word   Kernal_CHROUT ; IBSOUT output a character
   .word   Kernal_STOP   ; ISTOP  check if stop key is pressed
   .word   Kernal_GETIN  ; IGETIN get character from keyboard queue
   .word   Kernal_CLALL  ; ICLALL close all channels and files
   .word   Default_BRK   ; USRCMD user function

; Vector to user defined command, currently points to BRK.

; This appears to be a holdover from PET days, when the built-in machine language monitor
; would jump through the USRCMD vector when it encountered a command that it did not
; understand, allowing the user to add new commands to the monitor.

; Although this vector is initialized to point to the routine called by STOP/RESTORE and
; the BRK interrupt, and is updated by the kernal vector routine at $FD57, it no longer
; has any function.

   .word   Default_LOAD; ILOAD      load
   .word   Default_SAVE; ISAVE      save

; initialise and test RAM, the RAM from $000 to $03FF is never tested and is just assumed
; to work. first a search is done from $0401 for the start of memory and this is saved, if
; this start is at or beyond $1100 then the routine dead ends. once the start of memory is
; found the routine looks for the end of memory, if this end is before $2000 the routine
; again dead ends. lastly, if the end of memory is at $2000 then the screen is set to
; $1E00, but if the memory extends to or beyond $2100 then the screen is moved to $1000

Init_RAM
   LDA #0
```

```
#if C64
   TAY               ; clear index

InRA_10
   STA $0002,Y       ; clear page 0
   STA $0200,Y       ; clear page 2
   STA $0300,Y       ; clear page 3
   INY
#endif
#if VIC
   TAX               ; clear index

InRA_10
   STA $0000,X       ; clear page 0
   STA $0200,X       ; clear page 2
   STA $0300,X       ; clear page 3
   INX               ; increment index
#endif
   BNE InRA_10       ; loop if more to do
   LDX #<TBUFFR      ; set cassette buffer pointer low byte
   LDY #>TBUFFR      ; set cassette buffer pointer high byte
   STX TAPE1         ; save tape buffer start pointer low byte
   STY TAPE1+1       ; save tape buffer start pointer high byte

#if C64
   TAY
   LDA #3
   STA STAL+1

InRA_20
   INC STAL+1

InRA_30
   LDA (STAL),Y
   TAX
   LDA #$55 ; 'U'
   STA (STAL),Y
   CMP (STAL),Y
   BNE InRA_40
   ROL A
   STA (STAL),Y
   CMP (STAL),Y
   BNE InRA_40
   TXA
   STA (STAL),Y
   INY
   BNE InRA_30
   BEQ InRA_20

InRA_40
   TYA
   TAX
   LDY STAL+1
   CLC
   JSR Set_memtop
   LDA #>BASIC_RAM_START
   STA OSSTAR+1
   LDA #>BASIC_SCREEN
   STA SCNMPG
   RTS
#endif
```

```
#if VIC
   STA STAL           ; clear RAM test pointer low byte
   STA TEMPX          ; clear looking for end flag
   STA OSSTAR         ; clear OS start of memory low byte
   TAY                ; clear Y
   LDA #$04           ; set RAM test pointer high byte
   STA STAL+1         ; save RAM test pointer high byte

InRA_50
   INC STAL           ; increment RAM test pointer low byte
   BNE InRA_55        ; if no rollover skip the high byte increment
   INC STAL+1         ; increment RAM test pointer high byte

InRA_55
   JSR Test_RAM_byte ; test RAM byte, return Cb=0 if failed
   LDA TEMPX          ; test looking for end flag
   BEQ InRA_70        ; branch if not looking for end
   BCS InRA_50        ; loop if byte test passed
   LDY STAL+1         ; get test address high byte
   LDX STAL           ; get test address low byte
   CPY #$20           ; compare with $2000, RAM should always end at or after
   BCC InRA_75        ; if end address < $2000 go do dead end loop
   CPY #$21           ; compare with $2100
   BCS InRA_65        ; branch if >= $2100
   LDY #$1E           ; set screen memory page to $1E00
   STY SCNMPG         ; save screen memory page

InRA_60
   JMP Set_memtop       ; set the top of memory and return

InRA_65
   LDA #$12           ; set OS start of memory high byte
   STA OSSTAR+1       ; save OS start of memory high byte
   LDA #$10           ; set screen memory page to $1000
   STA SCNMPG         ; save screen memory page
   BNE InRA_60        ; set the top of memory and return, branch always

InRA_70
   BCC InRA_50        ; loop if byte test failed, not found start yet
   LDA STAL+1         ; get test address high byte
   STA OSSTAR+1       ; save OS start of memory high byte
   STA TEMPX          ; set looking for end flag
   CMP #$11           ; compare start with $1100, RAM should always start before
   BCC InRA_50        ; go find end of RAM, branch always

InRA_75
   JSR Init_VIC_Chip
   JMP InRA_75        ; loop forever
#endif

TAPE_IRQ_Vectors
#if JIFFY
#if C64
   .word   $fc6a             ; $08   write tape leader IRQ routine
   .word   $fbcd             ; $0A   tape write IRQ routine
   .word   $ea31             ; $0C   normal IRQ vector
   .word   $f92c             ; $0E   read tape bits IRQ routine
#endif
#if VIC
   .word   $fca8             ; $08   write tape leader IRQ routine
   .word   $fc0b             ; $0A   tape write IRQ routine
   .word   Default_IRQ       ; $0C   normal IRQ vector
   .word   $f98e             ; $0E   read tape bits IRQ routine
#endif
#else
   .word   TAPE_Write_Leader ; $08   write tape leader IRQ routine
   .word   TAPE_Write_IRQ    ; $0A   tape write IRQ routine
   .word   Default_IRQ       ; $0C   normal IRQ vector
   .word   TAPE_Read_IRQ     ; $0E   read tape bits IRQ routine
#endif

; *************
  Initialise_IO
; *************

#if C64
   LDA #$7f
   STA CIA1_ICR    ; CIA1 Interrupt Control Register
   STA CIA2_ICR
   STA KEYB_COL
   LDA #8
   STA CIA1_CRA
   STA CIA2_CRA
   STA CIA1_CRB    ; CIA1 Control Register B
   STA CIA2_CRB
   LDX #0
   STX $DC03
   STX $DD03
   STX $D418
   DEX
   STX $DC02
   LDA #7
   STA $DD00
   LDA #$3f ; '?'
   STA $DD02
   LDA #$e7
   STA R6510
   LDA #$2f ; '/'
   STA D6510
#endif
#if VIC
   LDA #$7F          ; disable all interrupts
   STA RS2_IRQ_REG   ; on VIA 1 IER ..
   STA VIA2_IER      ; .. and VIA 2 IER
   LDA #$40          ; set T1 free run, T2 clock �2,
                     ; SR disabled, latches disabled
   STA VIA2_ACR      ; set VIA 2 ACR
   LDA #$40          ; set T1 free run, T2 clock �2,
                     ; SR disabled, latches disabled
   STA VIA1_ACR      ; set VIA 1 ACR
   LDA #$FE          ; CB2 high, RS232 Tx
                     ; CB1 positive edge,
                     ; CA2 high, tape motor off
                     ; CA1 negative edge
   STA VIA1_PCR      ; set VIA 1 PCR
   LDA #$DE          ; CB2 low, serial data out high
                     ; CB1 positive edge,
                     ; CA2 high, serial clock out low
                     ; CA1 negative edge
   STA IEC_PCR       ; set VIA 2 PCR
   LDX #$00          ; all inputs, RS232 interface or parallel user port
   STX VIA1_DDRB     ; set VIA 1 DDRB
   LDX #$FF          ; all outputs, keyboard column
   STX VIA2_DDRB     ; set VIA 2 DDRB
   LDX #$00           ; all inputs, keyboard row
   STX VIA2_DDRA      ; set VIA 2 DDRA
   LDX #$80           ; OIII IIII, ATN out, light pen, joystick, serial data
                      ; in, serial clk in
   STX VIA1_DDRA      ; set VIA 1 DDRA
   LDX #$00           ; ATN out low, set ATN high
   STX IEC_DRAN       ; set VIA 1 DRA, no handshake
   JSR CLR_IEC_CLK    ; set serial clock high
   LDA #$82           ; enable CA1 interrupt, [RESTORE] key
   STA RS2_IRQ_REG    ; set VIA 1 IER
   JSR SET_IEC_CLK    ; set serial clock low
#endif

; ***************
  Program_Timer_A
; ***************

#if C64
   LDA TVSFLG
   BEQ TASF_10
   LDA #<$4025        ; 16421
   STA CIA1_TALO
   LDA #>$4025
   JMP TASF_20
TASF_10
   LDA #<$4295        ; 17045
   STA CIA1_TALO
   LDA #>$4295
TASF_20
   STA CIA1_TAHI
   JMP PTA_10
#endif
#if VIC
   LDA #$C0           ; enable T1 interrupt
   STA VIA2_IER       ; set VIA 2 IER
#if PAL
   LDA #$26           ; set timer constant low byte [PAL]
#else
   LDA #$89           ; set timer constant low byte [NTSC]
#endif
   STA VIA2_T1CL      ; set VIA 2 T1C_l
#if PAL
   LDA #$48           ; set timer constant high byte [PAL]
#else
   LDA #$42           ; set timer constant high byte [NTSC]
#endif
   STA VIA2_T1CH      ; set VIA 2 T1C_h
   RTS
#endif

; *************
  Kernal_SETNAM
; *************

   STA FNLEN          ; set file name length
   STX FNADR          ; set file name pointer low byte
   STY FNADR+1        ; set file name pointer high byte
   RTS

; *************
  Kernal_SETLFS
; *************
```

```
   STA LA            ; set logical file
   STX FA            ; set device number
   STY SA            ; set secondary address or command
   RTS

; *************
  Kernal_READST
; *************

   LDA FA            ; get device number
   CMP #$02          ; compare device with RS232 device
   BNE Get_Status    ; branch if not RS232 device
   LDA RSSTAT        ; read RS232 status word
#if C64
   PHA
#endif
   LDA #0
   STA RSSTAT        ; clear RS232 status
#if C64
   PLA
#endif

; the above code is wrong. the RS232 status is in A but A is cleared and that is used
; to clear the RS232 status byte. so whatever the status the result is always $00 and
; the status byte is always cleared. A solution is to use X to clear the status after
; it is read instead of the above like this ..
;
;   LDX   #$00         ; clear X
;   STX   RSSTAT       ; clear RS232 status ##
   RTS

; *************
  Kernal_SETMSG
; *************

   STA MSGFLG        ; set message mode flag

Get_Status
   LDA STATUS        ; read serial status byte

; **********
  Ora_Status
; **********

   ORA STATUS        ; OR with serial status byte
   STA STATUS        ; save serial status byte
   RTS

; *************
  Kernal_SETTMO
; *************

   STA STIMOT        ; save serial bus timeout flag
   RTS

; *************
  Kernal_MEMTOP
; *************

   BCC Set_memtop      ; if Cb clear go set the top of memory
```

```
; ***********
  Read_Memtop
; ***********

   LDX OSTOP         ; get memory top low byte
   LDY OSTOP+1       ; get memory top high byte

; **********
  Set_memtop
; **********

   STX OSTOP         ; set memory top low byte
   STY OSTOP+1       ; set memory top high byte
   RTS

; *************
  Kernal_MEMBOT
; *************

   BCC MEM_10        ; if Cb clear go set the bottom of memory

; read the bottom of memory

   LDX OSSTAR        ; read OS start of memory low byte
   LDY OSSTAR+1      ; read OS start of memory high byte

; set the bottom of memory

MEM_10
   STXY(OSSTAR)      ; set OS start of memory
   RTS

#if VIC
; *************
  Test_RAM_byte      ; return (C=1) on failure
; *************

   LDA (STAL),Y      ; get existing RAM byte
   TAX               ; copy to X
   LDA #$55          ; set first test byte
   STA (STAL),Y      ; save to RAM
   CMP (STAL),Y      ; compare with saved
   BNE Exit_No_RAM
   ROR A             ; make byte $AA, carry is set here
   STA (STAL),Y      ; save to RAM
   CMP (STAL),Y      ; compare with saved
   BNE Exit_No_RAM
   .byte   $A9       ; makes next line LDA #$18

Exit_No_RAM
   CLC               ; flag test failed
   TXA               ; get original byte back
   STA (STAL),Y      ; restore original byte
   RTS
#endif

; *********
  Entry_NMI
; *********

   SEI               ; disable interrupts
   JMP (NMINV)       ; next statement by default
```

```
; ***********
  Default_NMI
; ***********

   PHA               ; save A
   TXA               ; copy X
   PHA               ; save X
   TYA               ; copy Y
   PHA               ; save Y
#if C64
   LDA #$7F
   STA CIA2_ICR
   LDY CIA2_ICR
   BMI RS232_NMI
#endif
#if VIC
   LDA VIA1_IFR      ; get interrupt flag register
   BPL JMP_Exit_IRQ  ; if no interrupt restore registers and exit
   AND RS2_IRQ_REG   ; AND with interrupt enable register
   TAX               ; copy to X
   AND #2            ; mask [RESTORE] key
   BEQ RS232_NMI     ; if not [RESTORE] key continue with RS232
#endif
   JSR Scan_Autostart_Signature
   BNE NMI_10        ; branch if no autostart ROM
   JMP (OPTION_ROM+2); else do autostart ROM break entry

NMI_10
#if VIC
   BIT VIA1_DATA     ; test VIA 1 DRA
#endif
#if C64
   JSR Look_For_Special_Keys
#endif
#if VIC
   JSR Kernal_UDTIM  ; Update the system clock
#endif
   JSR STOP          ; Check if stop key is pressed
#if C64
   BNE RS232_NMI
#endif
#if VIC
   BNE JMP_Exit_IRQ  ; if not [STOP] exit interrupt
#endif

; ***********
  Default_BRK
; ***********

   JSR Kernal_RESTOR
   JSR Initialise_IO
   JSR Initialise_Hardware
   JMP (BASIC_BRK)

; *********
  RS232_NMI
; *********

#if C64
Bfe72
   TYA
   AND ENABL
   TAX
   AND #1
   BEQ NMI_20
   LDA $DD00
   AND #$fb
   ORA NXTBIT
   STA $DD00
   LDA ENABL
   STA CIA2_ICR
   TXA
   AND #$12
   BEQ Mfe9d
   AND #2
   BEQ Bfe9a
   JSR RS232_In
   JMP Mfe9d
Bfe9a
   JSR RS232_Out
#endif
#if VIC
   LDA RS2_IRQ_REG   ; interrupt enable register
   ORA #$80          ; set enable bit
   PHA               ; save to re-enable interrupts
   LDA #$7F          ; disable all interrupts
   STA RS2_IRQ_REG   ; interrupt enable register
   TXA               ; get active interrupts back
   AND #$40          ; mask T1 interrupt
   BEQ NMI_20        ; branch if not T1 interrupt
   LDA #$CE          ; CB2 low, CB1 negative edge, CA2 high, CA1 negative edge
   ORA NXTBIT        ; OR RS232 next bit to send, sets CB2 high if set
   STA VIA1_PCR      ; set VIA 1 PCR
   LDA RS2_TIM_LOW   ; get VIA 1 T1C_l
   PLA               ; restore interrupt enable mask
   STA RS2_IRQ_REG   ; interrupt enable register
#endif
Mfe9d
   JSR RS232_NMI_Transmit

JMP_Exit_IRQ
#if C64
   JMP Bfeb6
#endif
#if VIC
   JMP Exit_IRQ      ; restore registers and exit interrupt
#endif

NMI_20
   TXA               ; get active interrupts back
#if C64
   AND #2
#endif
#if VIC
   AND #$20          ; mask T2 interrupt
#endif
   BEQ NMI_30        ; branch if not T2 interrupt
#if C64
   JSR RS232_In
#endif
#if VIC
   LDA RS2_DSR_CTS   ; get VIA 1 DRB
   AND #$01          ; mask RS232 data in
   STA INBIT         ; save receiver input bit temp storage
   LDA VIA1_T2CL     ; get VIA 1 T2C_l
   SBC #$16          ;.
   ADC BAUDOF        ; add baud rate bit time low byte
   STA VIA1_T2CL     ; set VIA 1 T2C_l
   LDA VIA1_T2CH     ; get VIA 1 T2C_h
   ADC BAUDOF+1      ; add baud rate bit time high byte
   STA VIA1_T2CH     ; set VIA 1 T2C_h
   PLA               ; restore interrupt enable mask
   STA RS2_IRQ_REG   ; set VIA 1 IER, restore interrupts
   JSR RS232_NMI_Receive
#endif
#if C64
   JMP Bfeb6
#endif
#if VIC
   JMP Exit_IRQ      ; restore registers and exit interrupt
#endif

NMI_30
   TXA               ; get active interrupts back
   AND #$10          ; mask CB1 interrupt, Rx data bit transition
   BEQ Bfeb6         ; if no bit restore registers and exit interrupt
#if C64
   JSR RS232_Out
#endif
#if VIC
   LDA M51CTR        ; get pseudo 6551 control register
   AND #$0F          ; clear non baud bits
   BNE NMI_40        ; quirk

NMI_40
   ASL A             ; 2 bytes per baud index
   TAX               ; copy to index
   LDA Baudrate-2,X  ; get baud count low byte
   STA VIA1_T2CL     ; set VIA 1 T2C_l
   LDA Baudrate-1,X  ; get baud count high byte
   STA VIA1_T2CH     ; set VIA 1 T2C_h
   LDA RS2_DSR_CTS   ; read VIA 1 DRB, clear interrupt flag
   PLA               ; restore interrupt enable mask
   ORA #$20          ; enable T2 interrupt
   AND #$EF          ; disable CB1 interrupt
   STA RS2_IRQ_REG   ; set VIA 1 IER
   LDX BITNUM        ; get number of bits to be sent/received
   STX BITCI         ; save receiver bit count in
#endif

Bfeb6

#if C64
   LDA ENABL
   STA CIA2_ICR
#endif

; ********
  Exit_IRQ
; ********

   PLA               ; pull Y
   TAY               ; restore Y
   PLA               ; pull X
   TAX               ; restore X
   PLA               ; restore A
   RTI

; ********
  Baudrate
; ********

#if C64
; --------------------------------
   .word   $27c1     ;   50   baud
   .word   $1a3e     ;   75   baud
   .word   $11c5     ;  110   baud
   .word   $0e74     ;  134.5 baud
   .word   $0ced     ;  150   baud
   .word   $0645     ;  300   baud
   .word   $02f0     ;  600   baud
   .word   $0146     ; 1200   baud
   .word   $00b8     ; 1800   baud
   .word   $0071     ; 2400   baud
#endif
#if VIC
#if PAL
; PAL Value = 1108404 Hz / baudrate
; --------------------------------
   .word   $2AE6     ;   50   baud
   .word   $1C78     ;   75   baud
   .word   $1349     ;  110   baud
   .word   $0FB1     ;  134.5 baud
   .word   $0E0A     ;  150   baud
   .word   $06D3     ;  300   baud
   .word   $0338     ;  600   baud
   .word   $016A     ; 1200   baud
   .word   $00D0     ; 1800   baud
   .word   $0083     ; 2400   baud
   .word   $0036     ; 3600   baud
#else
; NTSC Value = 1022727 Hz / baudrate
; --------------------------------
   .word   $2792     ;   50   baud
   .word   $1A40     ;   75   baud
   .word   $11C6     ;  110   baud
   .word   $0E74     ;  134.5 baud
   .word   $0CEE     ;  150   baud
   .word   $0645     ;  300   baud
   .word   $02F1     ;  600   baud
   .word   $0146     ; 1200   baud
   .word   $00B8     ; 1800   baud
   .word   $0071     ; 2400   baud
   .word   $002A     ; 3600   baud
#endif
#endif

#if C64
; ********
  RS232_In
; ********

   LDA CIA2_PRB
   AND #1
   STA INBIT
   LDA CIA2_TBLO
   SBC #$1c
```

```
    ADC BAUDOF
    STA CIA2_TBLO
    LDA CIA2_TBHI
    ADC BAUDOF+1
    STA CIA2_TBHI
    LDA #$11
    STA CIA2_CRB
    LDA ENABL
    STA CIA2_ICR
    LDA #$ff
    STA CIA2_TBLO
    STA CIA2_TBHI
    JMP RS232_NMI_Receive

; *********
  RS232_Out
; *********

    LDA M51AJB
    STA CIA2_TBLO
    LDA M51AJB+1
    STA CIA2_TBHI
    LDA #$11
    STA CIA2_CRB
    LDA #$12
    EOR ENABL
    STA ENABL
    LDA #$ff
    STA CIA2_TBLO
    STA CIA2_TBHI
    LDX BITNUM
    STX BITCI
    RTS

; *************
  Set_Baud_Rate
; *************

    TAX
    LDA M51AJB+1
    ROL A
    TAY
    TXA
    ADC #$c8
    STA BAUDOF
    TYA
    ADC #0
    STA BAUDOF+1
    RTS

    .BYTE $ea,$ea     ; 2 NOP's

; ****************
  Clear_BREAK_Flag
; ****************

    PHP
    PLA
    AND #$ef
    PHA

#endif
```

```
; *********
  Entry_IRQ
; *********

   PHA                ; save A
   TXA                ; copy X
   PHA                ; save X
   TYA                ; copy Y
   PHA                ; save Y
   TSX                ; copy stack pointer
   LDA STACK+4,X      ; get the stacked status register
   AND #$10           ; mask the BRK flag bit
   BEQ BFF82          ; if not BRK go do the hardware IRQ vector
   JMP (CBINV)        ; else do the BRK vector
BFF82
   JMP (CINV)         ; do IRQ vector

#if C64
; ***********
  Init_Editor
; ***********

   JSR Initialise_Hardware

InEd_10
   LDA $D012
   BNE InEd_10
   LDA $D019
   AND #1
   STA TVSFLG
   JMP Program_Timer_A

; ======
  PTA_10              ; continue Program_Timer_A
; ======

   LDA #$81           ; Enable timer A interrupt
   STA CIA1_ICR       ; Interrupt Control Register
   LDA CIA1_CRA       ; Read Conrol Register A
   AND #$80           ; Clear all values except frequency (50/60 Hz)
   ORA #$11           ; Start time A in single shot mode
   STA CIA1_CRA       ; run timer A
   JMP SET_IEC_CLK

; **************
  Kernal_Version
; **************

   .BYTE $03

    JMP Init_Editor
    JMP Initialise_IO
    JMP Init_RAM

#endif

#if VIC
   .fill 5 (-1)       ; unused
#endif

; ******
  RESTOR              ; Restore default system and interrupt vectors
```

```
; ******

; Call address: $FF8A
; Preparatory routines: None
; Error returns: None
; Stack requirements: 2
; Registers affected: A, X, Y

; This routine restores the default values of all system vectors used in
; KERNAL and BASIC routines and interrupts. (See appendix D for the
; default vector contents). The KERNAL VECTOR routine is used to read
; and alter individual system vectors.

; 1) Call this routine.

; JSR RESTOR

   JMP Kernal_RESTOR

; ******
  VECTOR             ; Manage RAM vectors
; ******

; Call address: $FF8D
; Communication registers: X, Y
; Preparatory routines: None
; Error returns: None
; Stack requirements: 2
; Registers affected: A, X, Y

; This routine manages all system vector jump addresses stored in RAM.
; Calling this routine with the accumulator carry ait set wili store the
; current contents of the RAM vectors in a list pointed to by the X and
; Y registers. When this routine is called with the carry clear, the
; user list pointed to by the X and Y registers is transferred to the
; system RAM vectors. NOTE: This routine requires caution in its use.
; The best way to use it is to first read the entire vector contents
; into the user area, alter the desired vectors, and then copy the
; contents back to the system vectors.

; READ THE SYSTEM RAM VECTORS
; 1) Set the carry.
; 2) Set the X and Y registers to the address to put the vectors.
; 3) Call this routine.

; LOAD THE SYSTEM RAM VECTORS
; 1) Clear the carry bit.
; 2) Set the X and Y registers to the address of the vector list in RAM
;    that must be loaded
; 3) Call this routine.

; CHANGE THE INPUT ROUTINES TO NEW SYSTEM
; LDX #<USER
; LDY #>USER
; SEC
; JSR VECTOR   ;read old vectors
; LDA #<MYINP  ;change input
; STA USER+10
; LDA #>MYINP
; STA USER+11
; LDX #<USER
; LDY #>USER
; CLC
; JSR VECTOR  ;alter system
; USER * = * + 26

   JMP Kernal_VECTOR

; ******
  SETMSG             ; Control system message output
; ******

; Call address: $FF90
; Communication registers: A
; Preparatory routines: None
; Error returns: None
; Stack requirements: 2
; Registers affected: A

; This routine controls the printing of error and control messages by
; the KERNAL. Either print error messages or print control messages can
; be selected by setting the accumulator when the routine is called.
; FILE NOT FOUND is an example of an error message. PRESS PLAY ON
; CASSETTE is an example of a control message. Bits 6 and 7 of this
; value determine where the message will come from, ft bit 7 is 1, one
; of the error messages from the KERNAL will be printed. If bit 6 is
; set, a control message wifl be printed.

; 1) Set accumulator to desired value.
; 2) Call this routine.

; LDA #S40
; JSR SETMSG       ; TURN ON CONTROL MESSAGES
; LDA #SB0
; JSR SETMSG       ; TURN ON ERROR MESSAGES
; LDA #0
; JSR SETMSG       ; TURN OFF ALL KERNAL MESSAGES

   JMP Kernal_SETMSG

; ******
  SECOND             ; Send secondary address for LISTEN
; ******

; Call address: $FF93
; Communication registers: A
; Preparatory routines: LISTEN
; Error returns: See READST
; Stack requirements: one
; Registers affected: .A

; This routine is used to send a secondary address to an I/O device
; after a call to the LISTEN routine is made, and the device commanded
; to LISTEN. The routine cannot be used to send a secondary address
; after a call to the TALK routine. A secondary address is usually used
; to give set-up information to a device before I;O operations begin.
; When a secondary address is to be sent to a device on the serial bus,
; the address must first be ORed with $60.

; 1) Load the accumulator with the secondary address to be sent.
; 2) Call this routine.

; ADDRESS DEVICE #8 WITH COMMAND (SECONDARY ADDRESS) #15
; LDA #8
; JSR LISTEN
; LDA #15
; ORA #60
; JSR SECOND

   JMP Kernal_SECOND

; ****
  TKSA  ; Send a secondary address to a device commanded to TALK
; ****

; Call address: $FF96
; Communication registers: A
; Preparatory routines: TALK
; Error returns: See READST
; Slack requirements: None
; Registers affected: A

; This routine transmits a secondary address on the serial bus for a
; TALK device. This routine must be called with a number between 4 and
; 31 in the accumulator. The routine will send this number as a
; secondary address command over the serial bus. This routine can only
; be called after a call to the TALK routine, it will not work after a
; LISTEN.

; 0) Use the TALK routine.
; 1) Load the accumulator with the secondary address.
; 2) Call this routine.

; ;TELL DEVICE #4 TO TALK WITH COMMAND #7
; LDA #4
; JSR TALK
; LDA #7
; JSR TKSA

   JMP Kernal_TKSA

; ******
  MEMTOP             ; Read or set the top of RAM
; ******

; Call address: $FF99
; Communication registers: X, Y
; Preparatory routines: None
; Error returns: None
; Stack requirements: 2
; Registers affected: X, Y

; This routine is used to set the top of RAM. When this routine is
; called with the carry bit of the accumulator set, the pointer to the
; top of RAM will be loaded into the .X and. Y registers. When this
; routine is called with the accumulator carry bit clear, the contents
; of the X and Y registers will be loaded in the top of memory pointer,
; changing the top of memory.

; DEALLOCATE THE RS-232 BUFFER
; SEC
; JSR MEMTOP ;READ TOP OF MEMORY
; DEX
; CLC
; JSR MEMTOP ;SET NEW TOP OF MEMORY
```

```
   JMP Kernal_MEMTOP

; ******
  MEMBOT             ; Read or set bottom of memory
; ******

; Call address: $FF9C
; Communication registers: X, Y
; Preparatory routines: None
; Error returns: None
; Stack requirements: None
; Registers affected: X, Y

; This routine is used to set the bottom of the memory. If the
; accumulator carry bit is set when this routine is called, a pointer to
; the lowest byte of RAM will be returned in the X and Y registers. On
; the unexpancted VIC the initial value of this pointer is $1000. If the
; accumulator carry bit is clear (0) when this routine is called, the
; values of the X and Y registers will be transferred to the low and
; high bytes respectively of the pointer to the beginning of RAM.

; TO READ THE BOTTOM OF RAM
; 1) Set the carry.
; 2) Call this routine.

; TO SET THE BOTTOM OF MEMORY
; 1) Clear the carry.
; 2) Call this routine.

; MOVE BOTTOM OF MEMORY UP 1 PAGE
; SEC      ;READ MEMORY BOTTOM
; JSR MEMBOT
; INY
; CLC      ;SET MEMORY BOTTOM TO NEW VALUE
; JSR MEMBOT

   JMP Kernal_MEMBOT

; ******
  SCNKEY             ; Scan the keyboard
; ******

; Call address: $FF9F
; Com muni cation registers: None
; Preparatory routines: None
; Error returns: None
; Stack requirements: None
; Registers affected: A, X, Y

; This routine will scan the VIC keyboard and check for pressed keys. It
; is the same routine called by the interrupt handler. If a key is down,
; its ASCII value is placed in the keyboard queue.

; 1) Call this routine

; GET JSR SCNKEY ;SCAN KEYBOARD
;     JSR GETIN  ;GET CHARACTER
;     CMP #0     ;IS IT NULL?
;     BEQ GET    ;YES, SCAN AGAIN
;     JSR CHROUT ;PRINT IT

   JMP Kernal_SCNKEY
```

```
; ******
  SETTMO             ; Set serial bus timeout flag
; ******

; Call address: $FFA2
; Communication registers: A
; Preparatory routines: None
; Error returns: None
; Stack requirements: 2
; Registers affected: None

; This routine sets the timeout flag for the serial bus. When the
; timeout flag is set, the VIC will wait for a device on the serial port
; for 64 milliseconds. If the device does not respond to the VIC's DAV
; signal within that time the VIC will recognize an error condition and
; leave the handshake sequence. When this routine is called when the
; accumulator contains a 0 in bit 7, timeouts are enabled, A 1 in bit 7
; will disable the timeouts. NOTE: The VIC uses the timeout feature to
; communicate that adisk file is not found on an attempt to OPEN a file.

; TO SET THE TIMEOUT FLAG
; 1) Set bit 7 of the accumulator to 0,
; 2) Call this routine.

; TO RESET THE TIMEOUT FLAG
; 1) Set bit 7 of the accumulator to 1.
; 2) Call this routine,

; DISABLE TIMEOUT
; LDA #0
; JSR SETTMO

   JMP Kernal_SETTMO

; *****
  ACPTR              ; Get data from the serial bus
; *****

; Call address: $FFA5
; Communicaiion registers: A
; Preparatory routines: TALK, TKSA
; Error returns: See READST
; Stack requirements: 13
; Registers affected: A X

; This is the routine to use to get information from a device on the
; serial bus (like the disk). This routine gets a byte of data off the
; serial bus using full handshaking. The data is returned in the
; accumulator. To prepare for this routine the TALK routine must have
; been called first to command the device on the serial bus to send data
; on the bus. If the input device needs a secondary command, it must be
; sent by using the TKSA KERNAL routine before calling this routine.
; Errors are returned in the status word. The READST routine is used to
; read the status word.

; 0) Command a device on the serial bus to prepare to send data to the
;    VIC. (Use the TALK and TKSA kernal routines).
; 1) Call this routine (using JSR)
; 2) Store or otherwise use the data.

; Get a byte from the bus
; JSR ACPTR
; STA DATA

#if JIFFY
   JMP Jiffy_ACPTR
#else
   JMP Kernal_ACPTR
#endif

; *****
  CIOUT              ; Transmit a byte over the serial bus
; *****

; Call address: $FFA8
; Communication registers: A
; Preparatory routines: LISTEN, [SECOND]
; Error returns: See READST
; Stack requirements: None
; Registers affected: A

; This routine is used to send information to devices on the serial bus.
; A call to this routine wilt put a data byte onto the serial bus using
; full serial handshaking. Before this routine is called, the LISTEN
; KERNAL routine must be used to command a device on the serial bus to
; get ready to receive data. (If a device needs a secondary address, it
; must also be sent by using the SECOND KERNAL routine.) The accumulator
; is loaded with a byte to handshake as data on the serial bus. A device
; must be listening or the status word will return a timeout. This
; routine always buffers one character. (The routine holds the previous
; character to be sent back,) So when a call to the KERNAL UNLSN routine
; is made to end the data transmission, the buffered character is sent
; with EOl set. Then the UNLSN command is sent to the device.

; 0) Use the LISTEN KERNAL routine (and the SECOND routine if needed).
; 1) Load the accumulator with a byte of data.
; 2) Call this routine to send the data byte.

; Send an X to the serial bus
; LDA #'X'
; JSR CIOUT

   JMP Kernal_CIOUT

; *****
  UNTLK              ; Send an UNTALK command
; *****

; Call address; $FFAB
; Communication registers: None
; Preparatory routines: None
; Error returns: See READST
; Stack requirements: None
; Registers affected: A

; This routine will transmit an UNTALK command on the serial bus. All
; devices previously set to TALK will stop sending data when this
; command is received.

; 1) Call this routine.

; JSR UNTALK
```

```
   JMP Kernal_UNTLK

; *****
  UNLSN              ; Send an UNLISTEN command
; *****

; Call address: $FFAE
; Communication registers: None
; Preparatory routines: None
; Error returns: See READST
; Stack requirements: None
; Registers affected: A

; This routine commands all devices on the serial bus to stop receiving
; data from the VIC. (i.e., UNLISTEN}. Calling this routine results in
; an UNLISTEN command being transmitted on the serial bus. Only devices
; previously commanded to listen wilt be affected. This routine is
; normally used after the VIC is finished sending data to external
; devices. Sending the UNLISTEN will command the listening devices to
; get off the serial bus so it can be used for other purposes.

; 1) Call this routine.

; JSR UNLSN

   JMP Kernal_UNLSN

; ******
  LISTEN             ; Command a device to LISTEN
; ******

; Call Address: $FFB1
; Communication registers: A
; Preparatory routines: None
; Error returns: See READST
; Stack requirements: None
; Registers affected: A

; This routine will command a device on the serial bus to receive data.
; The accumulator must be loaded with a device number between 4 and 31
; before calling the routine. LISTEN will OR the number bit by bit to
; convert to a listen address, then transmit this data as a command on
; the serial bus. The specified device will then go into listen mode,
; and be ready to accept information.

; 1) Load the accumulator with the number of the device to command to
;    LISTEN.
; 2) Call this routine using the JSR instruction.

; COMMAND DEVICE #8 TO LISTEN
; LDA #8
; JSR LISTEN

   JMP Kernal_LISTEN

; ****
  TALK               ; Command a device on the serial bus to TALK
; ****

; Call address: $FFB4
; Communication registers: A
; Preparatory routines: None
; Error returns: See READST
; Stack requirements: None
; Registers affected: A

; To use this routine the accumulator must first be loaded with a device
; number between 4 and 30. When called, this routine then ORs bit by
; bits to convert this device number to a talk address. Then this data
; is transmitted as a command on the Serial bus.

; 1) Load the accumulator with the device number.
; 2) Call this routine.

; COMMAND DEVICE #4 TO TALK
; LDA #4
; JSR TALK

   JMP Kernal_TALK

; ******
  READST             ; Read status word
; ******

; Call address: $FFB7
; Communication registers: A
; Preparatory routines: None
; Error returns: None
; Stack requirements: 2
; Registers affected: A

; This routine returns the current status of the I/O devices in the
; accumulator. The routine is usually called after new communication to
; an I/O device. The routine will give information about device status,
; or errors that have occurred during the I/O operation. The bits
; returned in the accumulator contain the following information
; (see table befow):

; How to use:
; 1) Call this routine.
; 2) Decode the information in the .A register as it refers to your
;    program.

; CHECK FOR END OF FILE DURING READ
; JSR READST
; AND #64 ;check eof bit
; BNE EOF ;branch on eof

; Bit Val Comment
----------------------------------------
; 0    1  timeout writing to   IEC bus
; 1    2  timeout reading from IEC bus
; 2    4  short block    during tape I/O
; 3    8  long  block    during tape I/O
; 4   16  read error     during tape I/O
; 5   32  checksum error during tape I/O
; 6   64  EOF (End-Of-File) on IEC bus
; 7 -128  device not present

   JMP Kernal_READST

; ******
  SETLFS             ; Set up a logical file
; ******
; Call address: $FFBA
; Communication registers: A, X, Y
; Preparatory routines: None
; Error returns: None
; Stack requirements: 2
; Registers affected: None

; This routine will set the logica! file number, device address, and
; secondary address (command number) for other KERNAL routines. The
; logical fife number is used by the system as a key to the file table
; created by the OPEN file routine. Device addresses can range from 0 to
; 30. The following codes are used for the following CBM devices.

; Addr. DEVICE
; ------------
;    0  Keyboard
;    1  Cassette #1
;    2  RS-232C device
;    3  CRT display
;  4-5  Serial Bus printer
;  6-7  Serial Bus plotter
;  8-x  Serial bus disk drive

; Device numbers 4 or greater automatically refer to devices on the
; serial bus. A command to the device is sent as a secondary address on
; the serial bus after the device number is sent during the serial
; attention handshaking sequence. It no secondary address is to be sent
; the Y index register should be set to 255.

; 1) Load the accumulator with the logical file number.
; 2) Load the X index register with the device number.
; 3) Load the Y index register with the command.

; For logical file 32, device #4, and no command:
; LDA #32
; LDX #4
; LDY #255
; JSR SETLFS

   JMP Kernal_SETLFS

; ******
  SETNAM             ; Set up file name
; ******

; Call address: $FFBD
; Communication registers: A, X, Y
; Preparatory routines: None
; Stack requirements: None
; Registers affected: None

; This routine is used to set up the file name for the OPEN, SAVE, or
; LOAD routines. The accumulator must be loaded with the length of the
; file name. The X and Y registers must be loaded with the address of
; the file name, in standard 6502 low byte, high byte format.
; The address can be any valid memory address in the system where a
; string of characters for the file name is stored. If no file name is
; desired, the accumulator must be set to 0, representing a zero file
; length. The X and Y registers may be set to any memory address in that
; case.
; 1) Load the accumulator with the length of the file name.
; 2) Load the X index register with the low order address of the file
;     name.
; 3) Load the Y index register with the high order address.
; 4) Call this routine.

; LDA #NAME2-NAME ;LOAD LENGTH OF FILE NAME
; LDX #<NAME
; LDY #>NAME
; JSR SETNAM

   JMP Kernal_SETNAM

; ****
  OPEN               ; Open a logical file
; ****

; Call address: $FFC0
; Communication registers: None
; Preparatory routines: SETLFS, SETNAM
; Error returns: 1,2,4,5,6
; Stack requirements: None
; Registers affected: A, X, Y

; This routine is used to open a logical file. Once the logical file is
; set up, it can be used for input/output operations. Most of the I/O
; KERNAL routines call on this routine to create the logical files to
; operate on. No arguments need to be set up to use this routine, but
; both the SETLFS and SETNAM KERNAL routines must be called before using
; this routine.

; 0) Use the SETLFS routine.
; 1) Use the SETNAM routine.
; 2) Call this routine.

; This is an implementation of the BASIC statement: OPEN 15,8,15,"I0"
; LDA #NAME2-NAME ;LENGTH OF FILE NAME FOR SETLFS
; LDY #>NAME
; JSR SETNAW
; LDA #15
; LDX #8
; LDY #15
; JSR SETLFS
; JSR OPEN
; NAME  .BYTE "I0"
; NAME2

   JMP (IOPEN)

; *****
  CLOSE              ; Close a logical file
; *****

; Call address: $FFC3
; Communication registers: A
; Preparatory routines: None
; Error returns: None
; Stack requirements: None
; Registers affected: A, X

; This routine is used to close a logical file after all I/O operations
; have been completed on that file. This routine is called after the
; accumulator is loaded with the logical file number to be closed (the
; same number used when the file was opened using the OPEN routine).

; How to use:
; 1) Load the accumulator with the number of the logical file to be
;    closed.
; 2) Call this routine

; CLOSE 15
; LDA #15
; JSR CLOSE

   JMP (ICLOSE)

; *****
  CHKIN              ; Open a channel for input
; *****

; Call address: $FFC6
; Communication registers: X
; Preparatory routines: (OPEN)
; Error returns: 3,5,6
; Stack requirements: None
; Registers affected: A, X

; Any logical file that has already been opened by the KERNAL OPEN
; routine can be defined as an input channel by this routine. Naturally,
; the device on the channel must be an input device. Otherwise, an error
; will occur, and the routine will abort. If you are getting data from
; anywhere other than the keyboard, this routine must be called before
; using either the CHRIN or the GETIN KERNAL routines for data input. If
; input from the keyboard is desired, and no other input channels are
; opened, then the calls to this routine, and to the OPEN routine, are
; not needed. When this routine is used with a device on the serial bus,
; this routine automatically sends the talk address (and the secondary
; address if one was specified by the OPEN routine) over the bus.

; To use this routine:
; 0) OPEN the logical file (if necessary; see description above).
; 1) Load the .X register with number of the logical file to be used.
; 2) Call this routine (using a JSR command}.

; Possible errors are:
; #3: File not open
; #5: Device not present
; #6: File not an input file

; PREPARE FOR INPUT FROM LOGICAL FILE 2
; LDX #2
; JSR CHKIN

   JMP (ICHKIN)

; ******
  CHKOUT             ; Open a channel for output
; ******

; Call address: $FFC9
; Communication registers: X
; Preparatory routines: (OPEN)
; Error returns: 3,5,7
; Stack requirements: None
; Registers Affected: A, X

; Any logical file number which has been created by the KERNAL routine
; OPEN can be defined as an output channel. Of course, the device you
; intend opening a channel to must be an output device. Otherwise, an
; error will occur, and the routine will be aborted. This routine must
; be called before any data is sent to any output device unless you want
; to use the VIC screen as your output device, If screen output is
; desired, and there are no other output channels already defined, then
; the calls to this routine, and to the OPEN routine are not needed.
; When used to open a channel to a device on the serial bus, this
; routine will automatically send the LISTEN address specified by the
; OPEN routine (and a secondary address if there was one).

; How to use: (This routine is NOT NEEDED to send data to the screen)
; 0) Use the KERNAL OPEN routine to specify a logical file number, a
;    LISTEN address, and a secondary address (if needed).
; 1) Load the X register with the logical file number used in the open
;    statement,
; 2) Call this routine (by using the JSR instruction).

; Possible error returns:
; 3: File not open
; 5: Device not present
; 7: Not an output file

   JMP (ICKOUT)      ; do open for output vector

; ******
  CLRCHN             ; Clear I/O channels
; ******

; Call address: $FFCC
; Communication registers: None
; Preparatory routines: None
; Error routines: None
; Stack requirements: 9
; Registers affected: A, X

; This routine is called to clear all open channels and restore the I/O
; channels to their original default values. It is usually called after
; opening other I/O channels (like to the disk or tape drive) and using
; them for input-output operations. The default input device is 0
; (keyboard). The default output device is 3 (the VIC screen). If one of
; the channels to be closed is to the serial port, an UNTALK signal is
; sent first to clear the input channel or an UNLISTEN is sent to clear
; the output channel. By not calling this routine (and leaving
; listeners active on the serial bus) several devices can receive the
; same data from the VIC at the same time. One way to take advantage of
; this would be to command the printer to LISTEN and the disk to TALK.
; This would allow direct printing of a disk file.

; JSR CLRCHN

   JMP (ICLRCH)

; *****
  CHRIN              ; Get a character from the input channel
; *****

; Call address: $FFCF
; Communication registers: A
; Preparatory routines: (OPEN, CHKIN)
; Error returns: See READST
; Stack requirements: None
; Registers affected: A, X

; This routine wtll get a byte of data from the channel already set up
; as the input channel by the KERNAL routine CHKIN. If the CHKIN has
; not been used to define another input channel, data is expected from
; the keyboard. The data byte is returned in the accumulator. The
; channel remains open after the call. Input from the keyboard is
; handled in a special way. First, the cursor is turned on, and will
; blink until a carriage return is typed on the keyboard. All characters
; on the line (up to 88 characters) will be stored in the BASIC input
; buffer. Then the characters can be retrieved one at a time by calling
; this routine once for each character. When the carriage return is
; retrieved, the entire line has been processed. The next time this
; routine is called, the whole process begins again, i.e., by flashing
; the cursor.

; FROM THE KEYBOARD
; 1) Call this routine (using the JSR instruction).
; 2) Retrieve a byte of data by calling this routine,
; 3) Store the data byte.
; 4) Check if it is the last data byte (a CR ?). If not, goto step 2.

; RD          ; label
; LDX #0        ; store 0 in the X register
; JSR CHRIN
; STA DATA,X    ; store data byte in the Xth location in the data area.
; CMP #CR       ; is it a carriage return?
; BNE RD        ; no, get another data byte

; FROM KEYBOARD
; JSR CHRIN
; STA DATA

; FROM OTHER DEVICES
; 0) Use the KERNAL OPEN and CHKIN routines,
; 1) Call this routine (using a JSR instruction)-
; 2) Store the data.

; JSR CHRIN
; STA DATA

   JMP (IBASIN)

; ******
  CHROUT             ; Output a character
; ******

; Call address: $FFD2
; Communication registers: A
; Preparatory routines: (CHKOUT, OPEN)
; Error returns: See READST
; Stack requirements: None
; Registers affected: None

; This routine will output a character to an already opened channel. Use
; the KERNAL OPEN and CHKOUT routines to set up the output channel
; before calling this routine. If this call is omitted, data will be
; sent to the default output device (number 3, on the screen). The data
; byte to be output is loaded into the accumulator, and this routine is
; called. The data is then sent to the specified output device. The
; channel is left open after the call.
; NOTE: Care must be taken when using this routine to send data to a
; serial device since data will be sent to all open output channels on
; the bus. Unless this Is desired, all open output channels on the
; serial bus other than the actually intended destination channel must
; be closed by a call to the KERNAL close channel routine.

; How to use:
; 0) Use the CHKOUT KERNAL routine if needed (see description above).
; 1) Load the data to be output into the accumulator.
; 2) Call this routine.

; Duplicale the BASIC instruction CMD 4,"A";
; LDX #4     ;LOGICAL FILE #4
; JSR CHKOUT ;OPEN CHANNEL OUT
; LDA #'A'
; JSR CHROUT ;SEND CHARACTER

   JMP (IBSOUT)

; ****
  LOAD               ; Load RAM from device
; ****

; Call address: $FFD5
; Communication registers: A, X, Y
; Preparatory routines: SETLFS, SETNAM
; Error returns: 0,4,5,8,9
; Stack requirements: None
; Registers affected: A, X, Y

; This routine will load data bytes from any input device directly into
; the memory of the VIC. It can also be used for a verily operation,
; comparing data from a device with the data already in memory, leaving
; the data stored in RAM unchanged. The accumulator (A) must be set to 0
; for a load operation, or 1 for a verify. If the input device was
; OPENed with a secondary address (SA) of 0 the header information from
; device will be ignored. In this case, the X and Y registers must
; contain the starting address for the load. If the device was addressed
; with a secondary address of 0, 1, or 2 the data will load into memory
; starting at the location specified by the header, This routine returns
; the address of the highest RAM location which was loaded. Before this
; routine can be called, the KERNAL SETLFS, and SETNAM routines must be
; called.

; How to use
; 0) Call the SETLFS, and SETNAM routines. If a relocated load is
;    desired, use the SETLFS routine to send a secondary address of 3.
; 1) Set the A register to 0 for load, 1 for verify.
; 2) If a relocated load is desired, the X and Y registers must be set
;    to the start address for the load.
; 3) Call the routine using the JSR instruction.

   JMP Kernal_LOAD

; ****
  SAVE               ; Save memory to a device
; ****

; Call address: $FFD8
; Communication registers: A, X, Y
; Preparatory routines: SETLFS, SETNAM
; Error returns: 5,8,9
; Stack requirements: None
; Registers affected: A, X, Y

; This routine saves a section of memory. Memory is saved from an
; indirect address on page 0 specified by the accumulator to the address
; stored in the X and Y registers to a logical file (an input/output
; device). The SETLFS and SETNAM routines must be used before calling
; this routine. However, a file name is not required to SAVE to device 1
; (the cassette tape recorder). Any attempt to save to other devices
; without using a file name results in an error.
; NOTE: Device 0 (the keyboard) and device 3 (the screen) cannot be
; SAVEd to. If the attempt is made, an error will occur, and the SAVE
; stopped.

; How to use;
; 0) Use the SETLFS routine and the SETNAM routine (unless a SAVE with
;    no file name is desired on a save Jo the tape recorder).
; 1) Load two consecutive locations on page 0 with a pointer to the
;    start ol your save (in standard 6502 low byte first, high byte next
;    format).
; 2) Load the accumulator with the single byte page zero offset to the
;    pointer.
; 3) Load the X and Y registers with the low byte and high byte
;    respectively of the location of the end of the save.
; 4) Call this routine.

   JMP Kernal_SAVE

; ******
  SETTIM             ; Set the system clock
; ******

; Call address: $FFDB
; Communication registers: A, X, Y
; Preparatory routines: None
; Error returns: None
; Stack requirements: 2
; Registers affected: None

; A system clock is maintained by an interrupt routine that updates the
; clock every 1/60th of a second (one 'jiffy'). The clock is three bytes
; long, which gives it the capability to count up to 5,184,000 jiffies
; (24 hours). At that point the clock resets to zero. Before calling
; this routine to set the clock, the accumulator must contain the most
; significant byte, the X index registerthe next most significant byte,
; and the Y index register the least significant byte of the initial
; time setting (in jiffies).

; How to use:
; 1) Load the accumulator with the MSB of the 3 byte number.
; 2) Load the X register with the next byte.
; 3) Load the Y register with the LSB.
; 4) Call this routine.

   JMP Kernal_SETTIM

; *****
  RDTIM              ; Read system clock
; *****
; Call address: $FFDE
; Communication registers: A, X, Y
; Preparatory routines: None
; Error returns: None
; Stack requirements: 2
; Registers affected: A, X, Y

; This routine is used to read the system clock. The clock's resolution
; is a 60th of a second. Three bytes are returned by the routine. The
; accumulator contains the most significant byte, the X index register
; contains the next most significant byter and the Y index register
; contains the least significant byte.

   JMP Kernal_RDTIM

; ****
  STOP               ; Check if stop key is pressed
; ****

; Call address: $FFE1
; Communication registers: A
; Preparatory routines: None
; Error returns: None
; Stack requirements: None
; Registers affected: A, X

; If the STOP key on the keyboard is pressed when this routine is
; called, the Z flag will be set. All other flags remain unchanged. If
; the STOP key is not pressed then the accumulator will contain a byte
; representing the last row of the keyboard scan.
; The user can also check for certain other keys this way.

; How to use this routine:
; 1) Call this routine.
; 2) Test for the zero flag.

#if JIFFY & VIC
   JMP Jiffy_STOP
#else
   JMP (ISTOP)
#endif

; *****
  GETIN              ; Get a character from the keyboard buffer
; *****

; Call address: $FFE4
; Communication registers: A
; Preparatory routines: None
; Error returns: None
; Stack requirements: None
; Registers affected: A, X

; This subroutine removes one character from the keyboard queue and
; returns it as an ASCII value in the accumulator. If the queue is
; empty, the value returned in the accumulator will be zero. Characters
; are put into the queue automatically by an interrupt driven keyboard
; scan routine which calls the SCNKEY routine. The keyboard buffer can
; hold up to ten characters. After the buffer is filled, additional
; characters are ignored until at least one character has been removed
; from the queue.
; How to use:
; 1) Call this routine using a JSR instruction
; 2) Check for a zero in the accumulator (empty buffer)
; 3) Process the data

   JMP (IGETIN)

; *****
  CLALL              ; Close all files
; *****

; Call address: SFFE7
; Communication registers: None
; Preparatory routines: None
; Error returns: None
; Stack requirements: 11
; Registers affected: A, X

; This routine closes all open files. When this routine is called, the
; pointers into the open file table are reset, closing all files. Also,
; the routine automatically resets the I/O channels.

   JMP (ICLALL)

; *****
  UDTIM              ; Update the system clock
; *****

; Call address: $FFEA
; Communication registers: None
; Preparatory routines: None
; Error returns: None
; Stack requirements: 2
; Registers affected: A, X

; This routine updates the system clock. Normally this routine is called
; by the normal KERNAL interrupt routine every 1 /60th of a second. If
; the user program processes its own interrupts this routine must be
; called to update the time. Also, the STOP key routine must be called,
; if the stop key is to remain functional.

   JMP Kernal_UDTIM

; ******
  SCREEN             ; Return screen format
; ******

; Call address: $FFED
; Communication registers: X, Y
; Preparatory routines: None
; Stack requirements: 2
; Registers affected: X, Y

; This routine returns the format of the screen, e.g., 22 columns in X
; and 23 lines in Y. This routine can be used to determine what machine
; a program is running on, and has been implemented on the VIC to help
; upward compatibility in programs.

   JMP Kernal_SCREEN

; ****
  PLOT               ; Read or set cursor location
```

```
; ****

; Call address: $FFF0
; Communication registers: A, X, Y
; Preparatory routines: None
; Error returns: None
; Stack requirements: 2
; Registers affected: A, X, Y

; A call to this routine, with the accumulator carry flag set, loads the
; current position of the cursor on the screen (in X,Y coordinates) into
; the X and Y registers. X is the column number of the cursor location
; (0-21), and Y is the row number of the location of the cursor (0-22).
; A call with the carry bit clear moves the cursor to X,Y as determined
; by the X and Y registers.

; READING CURSOR LOCATION
; 1) Set the carry flag.
; 2) Call this routine.
; 3) Get the X and Y position from the X and Y registers respectively.

; SETTING CURSOR LOCATION
; 1) Clear carry flag.
; 2) Set the X and Y registers to the desired cursor location.
; 3) Call this routine.

   JMP Kernal_PLOT

; ******
  IOBASE             ; Define I/O memory page
; ******

; Call address: $FFF3
; Communication registers: X, Y
; Preparatory routines: None
; Error returns: None ,
; Stack requirements: 2
; Registers affected: X, Y

; This routine will set the X and Y registers to the address of the
; memory section where the memory mapped I/O devices are located. This
; address can then be used with an offset to access the memory mapped
; I/O devices in the VIC. The offset will be the number of locations
; from the beginning of the page that the desired I/O register is
; located. The X register will contain the low order address byte,
; while the Y register will contain the high order address byte.
; This routine exists to provide compatibility between the VIC 20 and
; future models of the VIC. IF the I/O locations for a machine language
; program are set by a call to this routine, they should still remain
; compatible with future versions of the VIC, the KERNAL and BASIC.

; How to use:
; 1) Call this routine by using the JSR instruction.
; 2) Store the X and the Y registers in consecutive locations.
; 3) Load the Y register with the offset.
; 4) Access that I/O location.

   JMP Kernal_IOBASE

#if C64
   .byte "RRBY"
#endif
```

```
#if VIC
   .FILL   4 (-1)    ; unused
#endif

   .word   Entry_NMI ; Non Maskable Interrupt vector
   .word   Entry_RESET; Reset vector
   .word   Entry_IRQ ; Interrupt Request vector
```