Making stable raster routines (C64 and VIC-20) by Marko Makela (Marko.Makela@HUT.FI)

Preface

Too many graphical effects, also called raster effects, have been coded in a very sloppy way. For instance, if there are any color bars on the screen in a game or demo, the colors often jitter a bit, e.g. they are not stable. And also, it is far too easy to make virtually any demo crash by hitting the Restore key, or at least cause visual distortions on the screen.

As late as a year ago I still hadn't coded a stable raster interrupt routine myself. But then I had to do it, since I was researching the video chip timing details together with my German friend Andreas Boose. It was ashaming that we had the same level of knowledge when it came to the hardware, but he was the only of us who had written a stable raster routine. Well, finally I made me to start coding. I used the same double-interrupt idea as Andreas used in his routine.

After a couple of errors my routine worked, and I understood how it works exactly. (This is something that separates us normal coders from demo people: They often code by instinct; by patching the routine until it works, without knowing exactly what is happening. That's why demos often rely on weird things, like crash if the memory is not initialized properly.)

In this article, I document two methods of creating stable raster routines on Commodore computers. The principles apply for most 8-bit computers, not only Commodores, but raster effects are very rarely seen on other computers.

Background

What are raster effects? They are effects, where you change the screen appearance while it is being drawn. For instance, you can set the screen color to white in the top of the screen, and to black in the middle of the screen. In that way, you will get a picture whose top half is white and bottom half black. Normally such effects are implemented with interrupt routines that are executed synchronized with the screen refresh.

The video chip on the Commodore 64 and many other videochips have a special interrupt feature called the Raster interrupt. It will generate an IRQ in the beginning of a specified raster line. On other computers, like the VIC-20, there is no Raster interrupt, but you can generate the interrupts with a timer, provided that the timer and the

videochip are clocked from the same source.

Even if the processor gets an interrupt signal at the same position on each video frame, it won't always be executing the first instruction of the interrupt routine at the same screen position. The NMOS 6502 machine instructions can take 2 to 9 machine cycles to execute, and if the main program contains instructions of very varying lengths, the beginning position of the interrupt can jump between 7 different positions. This is why you need to synchronize the raster routine when doing serious effects.

Also, executing the interrupt sequence will take 7 additional cycles, and the interrupt sequence will only start after the current instruction if the interrupt arrived at least two cycles before the end of the current instruction. It is even possible that an interrupt arrives while interrupts are disabled and the processor is just starting to execute a CLI instruction. Alas, the processor will not jump to the interrupt right after the CLI, but it will execute the next instruction before jumping to it. This is natural, since the CLI takes only two cycles. But anyway, this is only a constant in our equation, and actually out of the scope of this article.

How to synchronize a raster interrupt routine? The only way is to check the current screen position and delay appropriately many cycles. There are several ways of doing this, some of which are very awful and inefficient. The ugliest ways of doing this on the Commodore 64 I know are busy-waiting several raster lines and polling the raster line value, or using the Light pen feature, which will fail if the user presses the fire button on Joystick port 1. Here I will present two ways, both very elegant in my opinion.

Using an auxiliary timer

On the VIC-20, there is no Raster interrupt feature in the video chip. All you can do is to use a timer for generating raster interrupts. And if you use two timers running at a constant phase difference, you can get full synchronization. The first timer generates the raster interrupt, and the second timer, the auxiliary timer, tells the raster routine where it is running. Actually you could even use the first timer also for the checking, but the code will look nicer in the way I will be presenting now. Besides, you can use the auxiliary timer idea even when real raster interrupts are available.

The major drawback of using an auxiliary timer is initializing it. The initialization routine must synchronize with the screen, that is, wait for the beginning of the wanted raster line. To accomplish this, the routine must first wait for a raster line that occurs a bit earlier. About the only way to do this is with a loop like

```
        LDA #value
loop    CMP raster
        BNE loop
```

One round of this loop will take 4+3=7 cycles to execute, assuming that absolute addressing is being used. The loop will be finished if the raster register contains the wanted value while the processor reads it on the last cycle of the CMP instruction. The raster register can actually have changed already on the first cycle of the BNE instruction on the previous run of the loop, that is 7 cycles earlier!

Because of this, the routine must poll the raster register for several raster lines, always consuming one cycle more if the raster register changed too early. As the synchronization can be off at most by 7 cycles, a loop of 7 raster register value changes would do, but I made the loop a bit longer in my VIC-20 routine. (Well, I have to admit it, I was too lazy to make it work only with 7 rounds.)

After the initialization routine is fully synchronized the screen, it can set up the timer(s) and interrupts and exit. The auxiliary timer in my VIC-20 demo routine is several dozens of cycles after the primary timer, see the source code for comments. It is arranged so that the auxiliary timer will be at least 0 when it is being read in the raster routine. The raster routine will wait as many extra cycles as the auxiliary timer reads, however at most 15 cycles.

Using double raster interrupt

On the Commodore 64, I have never seen the auxiliary timer scheme being used. Actually I haven't seen it being used anywhere, I was probably the first one who made a stable raster interrupt routine on the VIC-20. Instead, the double interrupt method is becoming the standard on the C64 side.

The double interrupt method is based entirely on the Raster interrupt feature of the video chip. In the first raster interrupt routine, the program sets up another raster interrupt on a further line, changes the interrupt vector and enables interrupts.

In the place where the second raster interrupt will occur, there will be 2-byte instructions in the first interrupt routine. In this way, the beginning of the next raster interrupt will be off at most by one cycle. Some coders might not care about this one cycle, but if you can do it right, why wouldn't you do it right until the end?

At the beginning of the second raster interrupt routine, you will read the raster line counter register at the point where it is about to change. When the raster routine is being executed, there are two

possibilities: Either the raster counter has just changed, or it will change on the next cycle. So, you just need to compare if the register changed one cycle too early or not, and delay a cycle when needed. This is easily accomplished with a branch to the next address.

Of course, somewhere in your second raster interrupt routine you must restore the original raster interrupt position and set the interrupt vector to point to the first interrupt routine.

Applying in practice

I almost forgot my complaints about demos crashing when you actively hit the Restore key. On the VIC-20, you can disable NMI interrupts generated by the Restore key, and on the C64, you can generate an NMI interrupt with the CIA2 timer and leave the NMI-line low, so that no further high-to-low transitions will be recognized on the line. The example programs demonstrate how to do this.

So far, this article has been pretty theoretical. To apply these results in practice, you must definitely know how many CPU clock cycles the video chip consumes while drawing a scan line. This is fairly easy to measure with a timer interrupt, if you patch the interrupt handler so that it changes the screen color on each run. Set the timer interval to LINES*COLUMNS cycles, where LINES is the amount of raster lines and COLUMNS is your guess for the amount of clock cycles spent in a raster line.

If your guess is right, the color will always be changed in the same screen position (neglecting the 7-cycle jitter). When adjusting the timer, remember that the timers on the 6522 VIA require 2 cycles for re-loading, and the ones on the 6526 CIA need one extra cycle. Keep trying different timer values until you the screen color changes at one fixed position.

Commodore used several different values for LINES and COLUMNS on its videochips. They never managed to make the screen refresh rate exactly 50 or 60 Hertz, but they didn't hesitate to claim that their computers comply with the PAL-B or NTSC-M standards. In the following tables I have gathered some information of some Commodore video chips.

NTSC-M systems:

Host	Chip ID	Crystal freq/Hz	Dot clock/Hz	Processor clock/Hz	Cycles/ line	Lines/ 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:

Host	Chip ID	Crystal freq/Hz	Dot clock/Hz	Processor clock/Hz	Cycles/ line	Lines/ frame
VIC-20	6561-101	4433618	4433618	1108405	71	312
C64	6569	17734472	7881988	985248	63	312

On the PAL-B VIC-20, the crystal frequency is simultaneously the dot clock, which is BTW a 4th of the crystal frequency used on the C64. On the C64, the crystal frequency is divided by 18 to generate the processor clock, which in turn is multiplied by 8 to generate the dot clock.

The basic timings are the same on all 6569 revisions, and also on any later C64 and C128 video chips. If I remember correctly, these values were the same on the C16 videochip TED as well.

Note that the dot clock is 4 times the processor clock on the VIC-20, and 8 times that on the C64. That is, one processor cycle is half a character wide on the VIC-20, and a full character on a C64. I don't have exact measurements of the VIC-20 timing, but it seems that while the VIC-20 videochips draw the characters on the screen, it first reads the character code, and then, on the following video cycle, the appearance on the current character line. There are no bad lines, like on the C64, where the character codes (and colors) are fetched on every 8th raster line.

Those ones who got upset when I said that Commodore has never managed to make a fully PAL-B or NTSC-M compliant 8-bit computer should take a closer look at the "Lines/frame" columns. If that does not convince you, calculate the raster line rate and the screen refresh rate from the values in the table and see that they don't comply with the standards. To calculate the line rate, divide the processor clock frequency by the amount of cycles per line. To get the screen refresh rate, divide that frequency by the amount of raster lines.

The Code

OK, enough theory and background. Here are the two example programs, one for the VIC-20 and one for the C64. In order to fully understand them, you need to know the exact execution times of NMOS 6502 instructions. (All 8-bit Commodore computers use the NMOS 6502 processor core, except the C65 prototype, which used a inferior CMOS

version with all nice poorly-documented features removed.) You should check the 64doc document, available on my WWW pages at http://www.hut.fi/~msmakela/cbm/emul/x64/64doc.html, or via FTP at ftp.funet.fi:/pub/cbm/documents/64doc. I can also e-mail it to you on request.

Also, I have written a complete description of the video timing on the 6567R56A, 6567R8 and 6569 video chips, which could maybe be turned into another C=Hacking article. The document is currently partially in English and partially in German. The English part is available from ftp.funet.fi as /pub/cbm/documents/pal.timing, and I can send copies of the German part (screen resolution, sprite disturbance measurements, and more precise timing information) via e-mail.

The code is written for the DASM assembler, or more precisely for a extended ANSI C port of it made by Olaf Seibert. This excellent cross-assembler is available at ftp.funet.fi in /pub/cbm/programming.

First the raster demo for the VIC-20. Note that on the VIC-20, the $9004 register contains the upper 8 bits of the raster counter. So, this register changes only on every second line. I have tested the program on my 6561-101-based VIC-20, but not on an NTSC-M system.

It was hard to get in contact with NTSC-M VIC-20 owners. Daniel Dallmann, who has a NTSC-M VIC-20, although he lives in Germany, ran my test to determine the amount of cycles per line and lines per frame on the 6560-101. Unfortunately, the second VIA of his VIC-20 is partially broken, and because of this, this program did not work on his computer. Craig Bruce ran the program once, and he reported that it almost worked. I corrected a little bug in the code, so that now the display should be stable on an NTSC-M system, too. But the actual raster effect, six 16*16-pixel boxes centered at the top border, are very likely to be off their position.

```
  processor 6502

NTSC    = 1
PAL     = 2

;SYSTEM = NTSC  ; 6560-101: 65 cycles per raster line, 261 lines
SYSTEM  = PAL   ; 6561-101: 71 cycles per raster line, 312 lines

#if SYSTEM & PAL
LINES = 312
CYCLES_PER_LINE = 71
#endif
#if SYSTEM & NTSC
LINES = 261
CYCLES_PER_LINE = 65
```

```
#endif
TIMER_VALUE = LINES * CYCLES_PER_LINE - 2

  .org $1001    ; for the unexpanded Vic-20

; The BASIC line

basic:
  .word 0$      ; link to next line
  .word 1995    ; line number
  .byte $9E     ; SYS token

; SYS digits

  .if (* + 8) / 10000
  .byte $30 + (* + 8) / 10000
  .endif
  .if (* + 7) / 1000
  .byte $30 + (* + 7) % 10000 / 1000
  .endif
  .if (* + 6) / 100
  .byte $30 + (* + 6) % 1000 / 100
  .endif
  .if (* + 5) / 10
  .byte $30 + (* + 5) % 100 / 10
  .endif
  .byte $30 + (* + 4) % 10
0$:
  .byte 0,0,0   ; end of BASIC program

start:
  lda #$7f
  sta $912e     ; disable and acknowledge interrupts
  sta $912d
  sta $911e     ; disable NMIs (Restore key)

;synchronize with the screen
sync:
  ldx #28       ; wait for this raster line (times 2)
0$:
  cpx $9004
  bne 0$        ; at this stage, the inaccuracy is 7 clock cycles
                ; the processor is in this place 2 to 9 cycles
                ; after $9004 has changed
  ldy #9
  bit $24
1$:
  ldx $9004
  txa
  bit $24
```

```
#if SYSTEM & PAL
  ldx #24
#endif
#if SYSTEM & NTSC
  bit $24
  ldx #21
#endif
  dex
  bne *-1       ; first spend some time (so that the whole
  cmp $9004     ; loop will be 2 raster lines)
  bcs *+2       ; save one cycle if $9004 changed too late
  dey
  bne 1$
                ; now it is fully synchronized
                ; 6 cycles have passed since last $9004 change
                ; and we are on line 2(28+9)=74

;initialize the timers
timers:
  lda #$40      ; enable Timer A free run of both VIAs
  sta $911b
  sta $912b

  lda #<TIMER_VALUE
  ldx #>TIMER_VALUE
  sta $9116     ; load the timer low byte latches
  sta $9126

#if SYSTEM & PAL
  ldy #7        ; make a little delay to get the raster effect to the
  dey           ; right place
  bne *-1
  nop
  nop
#endif
#if SYSTEM & NTSC
  ldy #6
  dey
  bne *-1
  bit $24
#endif

  stx $9125     ; start the IRQ timer A
                ; 6560-101: 65 cycles from $9004 change
                ; 6561-101: 77 cycles from $9004 change
  ldy #10       ; spend some time (1+5*9+4=55 cycles)
  dey           ; before starting the reference timer
  bne *-1
  stx $9115     ; start the reference timer
```

```
pointers:
  lda #<irq     ; set the raster IRQ routine pointer
  sta $314
  lda #>irq
  sta $315
  lda #$c0
  sta $912e     ; enable Timer A underflow interrupts
  rts           ; return

irq:
; irq (event)   ; > 7 + at least 2 cycles of last instruction (9 to 16 total)
; pha           ; 3
; txa           ; 2
; pha           ; 3
; tya           ; 2
; pha           ; 3
; tsx           ; 2
; lda $0104,x   ; 4
; and #xx       ; 2
; beq           ; 3
; jmp ($314)    ; 5
                ; ---
                ; 38 to 45 cycles delay at this stage

  lda $9114     ; get the NMI timer A value
                ; (42 to 49 cycles delay at this stage)
; sta $1e00     ; uncomment these if you want to monitor
; ldy $9115     ; the reference timer on the screen
; sty $1e01
  cmp #8        ; are we more than 7 cycles ahead of time?
  bcc 0$
  pha           ; yes, spend 8 extra cycles
  pla
  and #7        ; and reset the high bit
0$:
  cmp #4
  bcc 1$
  bit $24       ; waste 4 cycles
  and #3
1$:
  cmp #2        ; spend the rest of the cycles
  bcs *+2
  bcs *+2
  lsr
  bcs *+2       ; now it has taken 82 cycles from the beginning of the IRQ

effect:
  ldy #16       ; perform amazing video effect
  lda $900f
  tax
```

```
  eor #$f7
0$:
  sta $900f
  stx $900f
  sta $900f
  stx $900f
  sta $900f
  stx $900f
  sta $900f
  stx $900f
  sta $900f
  stx $900f
  sta $900f
  stx $900f
  pha
  pla
#if SYSTEM & PAL
  pha
  pla
  nop
#endif
#if SYSTEM & NTSC
  bit $24
#endif
  nop
  dey
  bne 0$        ; end of amazing video effect

  jmp $eabf     ; return to normal IRQ
```

And after you have recovered from the schock of seeing a VIC-20 program, here is an example for the C64. It does also something noteworthy; it removes the side borders on a normal screen while displaying all eight sprites. Well, it cannot remove the borders on bad lines, and the bad lines look pretty bad. But I could use the program for what I wanted: I measured the sprite distortions on all videochip types I had at hand. (FYI: the sprites 0-2 get distorted at the very right of the screen, and the sprites 6 and 7 are invisible at the very left of the screen. You will need a monitor with horizontal size controls to witness these effects.)

This program is really robust, it installs itself nicely to the interrupt routine chain. It even has an entry point for deinstalling itself. But in its robustness it uses self-modifying code to store the original interrupt routine address. :-)

The code also relies on the page boundaries in being where they are. The cycles are counted so that the branches "irqloop" must take 4 cycles. If the "irqloop" comes to the same CPU page with the branch

instructions, you must add one cycle to the loop in a way or another. When coding the routine, I noticed again how stupid assembly coding can be, especially conditional assembling. In a machine language monitor you have far better control on page boundaries. BTW, you might wonder why I disable the Restore key in a subroutine at the end and not in the beginning of the program. Well, the routine was so long that it would have affected the "irqloop" page boundaries. And I didn't want to risk the modified programs working on all three different videochip types on the first try.

In the code, there are some comments that document the video timing, like this one:

```
;3s4s5s6s7srrrrrgggggggggggggggggggggggggggggggggggggggg--||0s1s2s Phi-1 VIC-II
;sssssssss                                                 ||ssssss Phi-2 VIC-II
;==========xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx||XXX====== Phi-2 6510
;          ^ now we are here
```

The two vertical bars "|" denote optional cycles. On PAL-B systems (63 cycles per line), they are not present. On 6567R56A, which has 64 cycles per line, there is one additional cycle on this position, and the 6567R8 has two additional cycles there.

The numbers 0 through 7 are sprite pointer fetches (from the end of the character matrix, e.g. the text screen), the "s" characters denote sprite image fetches, the "r"s are memory refresh, and the "g" are graphics fetches. The two idle video chip cycles are marked with "-". On the processor timing line, the "=" signs show halted CPU, "x" means free bus, and "X" means that the processor will be halted at once, unless it is performing write cycles.

```
  processor 6502

; Select the video timing (processor clock cycles per raster line)
CYCLES = 65     ; 6567R8 and above, NTSC-M
;CYCLES = 64    ; 6567R5 6A, NTSC-M
;CYCLES = 63    ; 6569 (all revisions), PAL-B

cinv = $314
cnmi = $318
raster = 52     ; start of raster interrupt
m = $fb         ; zero page variable

  .org $801
basic:
  .word 0$      ; link to next line
  .word 1995    ; line number
  .byte $9E     ; SYS token
```

```
; SYS digits

  .if (* + 8) / 10000
  .byte $30 + (* + 8) / 10000
  .endif
  .if (* + 7) / 1000
  .byte $30 + (* + 7) % 10000 / 1000
  .endif
  .if (* + 6) / 100
  .byte $30 + (* + 6) % 1000 / 100
  .endif
  .if (* + 5) / 10
  .byte $30 + (* + 5) % 100 / 10
  .endif
  .byte $30 + (* + 4) % 10

0$:
  .byte 0,0,0   ; end of BASIC program

start:
  jmp install
  jmp deinstall

install:        ; install the raster routine
  jsr restore   ; Disable the Restore key (disable NMI interrupts)
checkirq:
  lda cinv      ; check the original IRQ vector
  ldx cinv+1    ; (to avoid multiple installation)
  cmp #<irq1
  bne irqinit
  cpx #>irq1
  beq skipinit
irqinit:
  sei
  sta oldirq    ; store the old IRQ vector
  stx oldirq+1
  lda #<irq1
  ldx #>irq1
  sta cinv      ; set the new interrupt vector
  stx cinv+1
skipinit:
  lda #$1b
  sta $d011     ; set the raster interrupt location
  lda #raster
  sta $d012
  ldx #$e
  clc
  adc #3
  tay
  lda #0
```

```
  sta m
0$:
  lda m
  sta $d000,x   ; set the sprite X
  adc #24
  sta m
  tya
  sta $d001,x   ; and Y coordinates
  dex
  dex
  bpl 0$
  lda #$7f
  sta $dc0d     ; disable timer interrupts
  sta $dd0d
  ldx #1
  stx $d01a     ; enable raster interrupt
  lda $dc0d     ; acknowledge CIA interrupts
  lsr $d019     ; and video interrupts
  ldy #$ff
  sty $d015     ; turn on all sprites
  cli
  rts

deinstall:
  sei           ; disable interrupts
  lda #$1b
  sta $d011     ; restore text screen mode
  lda #$81
  sta $dc0d     ; enable Timer A interrupts on CIA 1
  lda #0
  sta $d01a     ; disable video interrupts
  lda oldirq
  sta cinv      ; restore old IRQ vector
  lda oldirq+1
  sta cinv+1
  bit $dd0d     ; re-enable NMI interrupts
  cli
  rts

; Auxiliary raster interrupt (for syncronization)
irq1:
; irq (event)   ; > 7 + at least 2 cycles of last instruction (9 to 16 total)
; pha           ; 3
; txa           ; 2
; pha           ; 3
; tya           ; 2
; pha           ; 3
; tsx           ; 2
; lda $0104,x   ; 4
; and #xx       ; 2
```

```
; beq           ; 3
; jmp ($314)    ; 5
                ; ---
                ; 38 to 45 cycles delay at this stage
  lda #<irq2
  sta cinv
  lda #>irq2
  sta cinv+1
  nop           ; waste at least 12 cycles
  nop           ; (up to 64 cycles delay allowed here)
  nop
  nop
  nop
  nop
  inc $d012     ; At this stage, $d012 has already been incremented by one.
  lda #1
  sta $d019     ; acknowledge the first raster interrupt
  cli           ; enable interrupts (the second interrupt can now occur)
  ldy #9
  dey
  bne *-1       ; delay
  nop           ; The second interrupt will occur while executing these
  nop           ; two-cycle instructions.
  nop
  nop
  nop
oldirq = * + 1  ; Placeholder for self-modifying code
  jmp *         ; Return to the original interrupt

; Main raster interrupt
irq2:
; irq (event)   ; 7 + 2 or 3 cycles of last instruction (9 or 10 total)
; pha           ; 3
; txa           ; 2
; pha           ; 3
; tya           ; 2
; pha           ; 3
; tsx           ; 2
; lda $0104,x   ; 4
; and #xx       ; 2
; beq           ; 3
; jmp (cinv)    ; 5
                ; ---
                ; 38 or 39 cycles delay at this stage
  lda #<irq1
  sta cinv
  lda #>irq1
  sta cinv+1
  ldx $d012
  nop
```

```
#if CYCLES - 63
#if CYCLES - 64
  nop           ; 6567R8, 65 cycles/line
  bit $24
#else
  nop           ; 6567R56A, 64 cycles/line
  nop
#endif
#else
  bit $24       ; 6569, 63 cycles/line
#endif
  cpx $d012     ; The comparison cycle is executed CYCLES or CYCLES+1 cycles
                ; after the interrupt has occurred.
  beq *+2       ; Delay by one cycle if $d012 hadn't changed.
                ; Now exactly CYCLES+3 cycles have passed since the interrupt.
  dex
  dex
  stx $d012     ; restore original raster interrupt position
  ldx #1
  stx $d019     ; acknowledge the raster interrupt
  ldx #2
  dex
  bne *-1
  nop
  nop
  lda #20       ; set the amount of raster lines-1 for the loop
  sta m
  ldx #$c8
irqloop:
  ldy #2
  dey
  bne *-1       ; delay
  dec $d016     ; narrow the screen (exact timing required)
;3s4s5s6s7srrrrrgggggggggggggggggggggggggggggggggggggggg--||0s1s2s Phi-1 VIC-II
;sssssssss                                                 ||ssssss Phi-2 VIC-II
;==========xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx||XXX====== Phi-2 6510
;          ^ now we are here
  stx $d016     ; expand the screen
#if CYCLES - 63
#if CYCLES - 64
  bit $24       ; 6567R8
#else
  nop           ; 6567R56A
#endif
#else
  nop           ; 6569
#endif
  dec m
  bmi endirq
  clc
```

```
  lda $d011
  sbc $d012
  and #7
  bne irqloop   ; This instruction takes 4 cycles instead of 3,
                ; because the page boundary is crossed.
badline:
  dec m
  nop
  nop
  nop
  nop
  dec $d016
;3s4s5s6s7srrrrrgggggggggggggggggggggggggggggggggggggggg--||0s1s2s Phi-1 VIC-II
;sssssssssss    ccccccccccccccccccccccccccccccccccccccc   ||ssssss Phi-2 VIC-II
;==========xXXX=========================================||***====== Phi-2 6510
;          ^ we are here
  stx $d016
;3s4s5s6s7srrrrrgggggggggggggggggggggggggggggggggggggggg--||0s1s2s Phi-1 VIC-II
;sssssssssss                                               ||ssssss Phi-2 VIC-II
;==========xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx||XXX====== Phi-2 6510
;          ^ ^^- we are here (6569)
;          | \- or here (6567R56A)
;          \- or here (6567R8)
  ldy #2
  dey
  bne *-1
  nop
  nop
#if CYCLES - 63
#if CYCLES - 64
  nop           ; 6567R8, 65 cycles/line
  nop
  nop
#else
  bit $24       ; 6567R56A, 64 cycles/line
#endif
#else
  nop           ; 6569, 63 cycles/line
#endif
  dec m
  bpl irqloop   ; This is a 4-cycle branch (page boundary crossed)
endirq:
  jmp $ea81     ; return to the auxiliary raster interrupt

restore:        ; disable the Restore key
  lda cnmi
  ldy cnmi+1
  pha
  lda #<nmi     ; Set the NMI vector
  sta cnmi
```

```
  lda #>nmi
  sta cnmi+1
  ldx #$81
  stx $dd0d     ; Enable CIA 2 Timer A interrupt
  ldx #0
  stx $dd05
  inx
  stx $dd04     ; Prepare Timer A to count from 1 to 0.
  ldx #$dd
  stx $dd0e     ; Cause an interrupt.
nmi = * + 1
  lda #$40      ; RTI placeholder
  pla
  sta cnmi
  sty cnmi+1    ; restore original NMI vector (although it won't be used)
  rts
```