The Ninja-Method

The basic idea has a pretty long history. I've heard of “using the CIA directly to compensate jitter” long ago. But the first code I saw that really does use this idea was a very optimized 2x2-fli routine by Wolfram Sang (Ninja / The Dreams). With a working routine on paper it's always a good start to write one's own routine. Now that I accomplished a 4×4-fli-routine based on this, I'll gladly explain the details.

Idea of a NMI-driven 4x4-routine

We'll set up CIA #2 to trigger a NMI every 8th rasterline (on PAL: a counter of 8*63-1 = 503 cycles). Then we set up CIA #1 Timer B to count down our Jitter, which means: CIA #1 Timer B must be started a little after the CIA #2 Timer. Why CIA #1, Timer B you might ask? Well, the CBM documentation about CIAs says: “Writing to a timer hi-value puts the written data into into the timer, if it is not running. Reading a timer returns the current count-down value.” So if we stopped a timer and write data to it in the correct lo/hi order, it will always read the last written 16bit value. Imagine we stop Timer A and put a value of $004c into it. The CPU will always read a Timer A value of $004c. If we were 1337 enough to execute $dc04, we'd execute

```
jmp $XX00
```

where XX == lo-value of Timer B counter in cycle 3 of the jmp command.

Now imagine we let the NMI execute this $dc04. The NMI takes 7 cycles to execute. The jmp takes 3 and we will have a jitter of 0-7 cycles. This means, our routines are executed between 10 to 17 cycles after the NMI-timer ran out. As it will jump to a certain routine for each jitter-value, we know exactly how many cycles to compensate in each routine.

For this amount of precision all we need is precision while setting it all up. Beware: the exact timing while setting all up is extremely crucial, because any $dc04-jmp to undefined memory will most likely crash. And as we use a NMI, every bad messing around with $dd0d might trigger a NMI and by that most likely crash. Also pressing RESTORE will most likely crash - but anybody who does that is out of his or her mind anyhow (as Oswald said)…

Caveats

$dd0d

To not mess around with $dd0d badly, you'll have to follow two simple rules: Activate the NMI by

```
lda #$81
bit $dd0d
sta $dd0d
```

Deactivate the NMI by

```
lda #$7f
sta $dd0d
```

Just do not `bit $dd0d` “just in case”, as it might create the crashing results I mentioned above.

6526 vs. 6526A

Another caveat is the difference between 6526 and 6526A. We know the 6526 triggers Interrupts one cycle after the timer ran out, but the 6526A triggers in the “right moment”. One could argue which is right, but it doesn't really matter. We just have to take the respective counter-measures. So all we need is a CIA-detection and a proper

```
if (6526) start NMI-timer one cycle earlier
```

As on both CIA-types the “normal counter operation” is the same, the jitter-timers need to be started at the same cycle for both types.

Actually JackAsser mentioned in the CSDB-forums, that we could go without any CIA-detection. But more on that in the “Advantages” section…

the C0DE (49374)

I'm sorry some labels have more or less German names. I translated all of my documentation, think that is sufficient.

```
;-----------------------------------------------------------------------------
----
;TODO: declare variables before !src'ing this file
;      (must be solvable even in the first pass!)
;
;NMI_base = address of the first NMI-routine in memory (ATTENTION:
theoretically
;           those might be located anywhere (except underneath the IO), if
the
;           CIA#1TimerB just counts down from a high enough value.  But most
;           demo coders like to make one of the timers reference-count
exactly
;           one rasterline, which would make $0100-$3700 the range for
NMI_base)
;zpreg    = zp-address for storing the accu during NMIs
;d018wert1= D018-value of the upper 4 pixel-rows
;d018wert2= D018-value of the lower 4 pixel-rows
;d011wert1= D011-value für D018wert1
;d011wert2= D011-value für D018wert2
;
;ATTENTION: this routine just does the syncing and starting of all timers.
```

```
You
;still need to create some raster-IRQs that start and stop the timer-NMIs.
;============================================================================
====
!ifdef NMI_base {

.wartung    inx        ;loop for waiting exactly 52 cycles
        ldy #7        ;(incl. jsr .wartung)
.check6     dey
        bne .check6
.check_6    nop
.rts        rts

init4x4
;FIRST UP: CIA-Detection and Initialization
        and #0
        sta .CIA_type
        sta $dd05
        sta $dc0e    ;stop all timers
        sta $dc0f
        sta $dd0e
        sta $dd0f
        ldy #$7f    ;disallow all Timer-Interrupts
        sty $dc0d
        cmp $dc0d
        sty $dd0d
        cmp $dd0d
        lda #4        ;prepare Detection (timer=4 cycles)
        sta $dd04
        bit $d011    ;wait for border, then start ...
        bpl *-3
        lda #<.CIA_detect_nmi
        sta $fffa
        lda #>.CIA_detect_nmi
        sta $fffb
        lda #$81
        ldx #%10011001
        stx $dd0e
        sta $dd0d
        bit $dd0d
        dec .CIA_type
.CIA_detect_nmi pla
        pla
        pla
        sty $dd0d    ;deactivate Timer-NMI
        cmp $dd0d

;ATTN: for mathematical purposes a line starts at cycle 0 and ends at cycle
62!

        ldx #$03    ;half variance delay:
```

```
.check0     cpx $d012       ;check is at cycle    0  1  2  3  4  5  6
        bne .check0     ;cycle in rasterline
                                ;ending this command: 2  3  4  5  6  7  8
.check_0    jsr .wartung    ;waste 54 cycles ... this just made 52 of them
        nop
        cpx $d012       ;now check in cycle: 60 61 62  0  1  2  3
.check1     beq .check_1
        cmp ($00),y     ;ending this command: 4  5  6  3  4  5  6
.check_1    jsr .wartung
        nop
.check2     cpx $d012       ;now check in cycle: 62  0  1 61 62  0  1
        beq .check_2
        bit $ea         ;ending this command: 4  3  4  3  4  3  4
.check_2    jsr .wartung
        bit $ea
        cpx $d012    ;now check in cycle:  0 62  0 62  0 62  0
.check3     bne .check_3    ;after this, we're at 2  2  2  2  2  2  2

;Calculation of timings:
;=======================
;- the first STA $d011 MUST end in cycle 13 (when starting to count at 0)
;- that means, it starts on cycle 9,
;- that means LDA #D018WERT starts at cycle 1
;- with 7 cycles jitter another sta zp (3 cycles) happens before
;means: 3 (save accu) + 7 (Jitter) + 3 (jmp) + 7 (NMI itself) = 20
;-> NMI must execute at cycle 44, so the (ForceLoad+Run) command has to
;   do its write in cycle 42 (as the nmi then happens after cycle 43)
.check_3            ;cycle-counting: (6526 / 6526A), starting
                                ;on cycle 3 ...
        lda .CIA_type    ;4
.check4     bpl .check_4    ;2/3

.check_4    lda #<8*63-1    ;2
        sta $dc06    ;4
        sta $dd04    ;4
        lda #>8*63-1    ;2
        sta $dc07    ;4
        sta $dd05    ;4
        lda #$4c    ;2
        sta $dc04    ;4
        lda #%10010001    ;2
        sta $dd0e    ;4 = 38/39 -> cycle 42 of this RL on 6526A

    .CIA_type = *+1
        ldx #0        ;2
.check5     bmi .check_5    ;3/2
.check_5            ;=5/4

;2nd calculation:
```

```
;================
;NMI = 7 cycles, jmp = 3 cycles, max.Jitter = 7 cycles
;means: 17+1 cycles later Timer B shall be started to "land at" $0000. This
also
;means that we have to start Timer B 17+1+hi(NMI_base) cycles later.

    !set .rest = (>NMI_base) + 14

    !do while .rest > 5 {
        nop
        !set .rest = .rest - 2
    }
    !if .rest = 4 {
        sta $dc0f
    } else {
        sta $dc0f,y
    }
        ;use CIA#1timerA as simple memory location for jmp $xx00
    !set .jmpval = (<NMI_base)*$100 + $4c
        +mv16im .jmpval,$dc04
    ;use that as NMI-vector
        +mv16im $dc04,$fffa
        rts
;IMPORTANT: check all the "checks", we do not want to lose any cycles
because a
;page boundary was crossed whithin a branch command.
!if ((>.check1 != >.check_1) OR (>.check2 != >.check_2) OR (>.check3 !=
>.check_3) OR (>.check4 != >.check_4) OR (>.check5 != >.check_5) OR
(>.check6 != >.check_6)) {
    !serious "Page boundary crossed where it was a bad thing to happen.
Relocate the Code!"
}
;============================================================================
====
!macro flinmi jitter, offset {
    * = offset + $0100*(8-jitter)
    !if (jitter & 1) = 0 {
        sta zpreg        ;3
    } else {
        sta .thisreg        ;4
    }
    !if jitter < 5 {
        bit $dd0d        ;4
        !if (jitter = 0) {
        nop            ;2
        nop            ;2
        }
    }
    !if ((jitter & 3) = 1) OR ((jitter & 3) = 2) {
        nop            ;2
    }
    ;Der eigentliche FLI-IRQ
        lda #d018wert2    ;damit muss dieser Befehl mit Zyklus 1 starten
        sta $d018
        lda #d011wert2
        sta $d011    ;letzter Zyklus muss Zyklus 14 der RL sein!
        lda #d011wert1
        sta $d011
        lda #d018wert1
        sta $d018
    !if jitter >= 5 {
        bit $dd0d
    }
    !if (jitter & 1) = 0 {
        lda zpreg
    } else {
    .thisreg = *+1
        lda #0
    }
        rti
}
;--------------------------------------------------------------------------
----
;create the NMI-Routines right on their spot
!set .oldaddr = *
!set antijitter = 0
!do {
    !set jitterVal = 8 - antijitter
    +flinmi jitterVal, NMI_base
    !set antijitter = antijitter+1
} while antijitter < 8
* = .oldaddr
;==========================================================================
====
} else {
    !serious "NMI_base not declared.  Assembly must fail!"
}
```

Advantages

no preparation for the next interrupt, no static starting cycle

If we create a “second” badline in the middle of a charline to switch to a different screen-ram, A Raster-IRQ would start at cycle #0 of the “desired to make bad” rasterline-1, while this routine's interrupt starts “just in time”, thus the raster-method loosing around 44 cycles (in this case of FLI), because we cannot tell VIC “do your raster-irq in cycle 17”. That time may be “abused” for jitter correction and preparing the next “condition”, like an “auxiliary timer method” routine does, but it's still wasted cycles.

Take the NMI instead: the CIA just reloads its timer - 0 cycles wasted, the only thing to do is create a starting and an ending condtion. As we reach one of our nmi-routines directly, if jitter == 7 we can start playing with VIC directly and after that acknowledge the NMI and do whatever. if jitter == 3 we could first acknowledge the nmi and then play with VIC. So basically we can abuse the unwanted jitter to save cycles and do some housekeeping - we just minimize all the “waiting for the right moment” and “wasting inaccuracies with nop”.

Advantage brought up by JackAsser

We could actually care less about the CIA-differences by adding another Jitter-Routine. So with a new CIA we'd only use the “0-7 cycles of jitter”-routines, with an old CIA (remember: it fires the Interrupt one cycle later) we'd use “1-8 cycles of jitter”-routines. Now it's up to you if you can waste or abuse (whatever makes you happy) another page for another routine. After all the NMI-routines are shorter than the CIA-detection (less than $20 bytes vs. a little more than $20 bytes), so it would save some space altogether, but the code distribution would be worse…

Disadvantages

Any Interrupt can jitter 0 to 7 cycles. This makes up the need to compensate with 8 different routines, which indeed means: we lose 8 pages, where our NMI-Routines are saved. But don't be a prick: a simple FLI does not need that much of RAM. One could f.e. mix the NMI-pages (let's call'em that) with data, or even locate other code around that…

To make the code-part work here you'll find a complete demo-part 😉 Please assemble test4x4fli.a with ACME 0.93. If you're missing any of the library-routines I use or get errors while compiling: these are my slightly changed library routines.

StOfF / Neoplasia