# Round 11 • Horizontal Scrollers

*OCA / Atlantis*

## // Introduction

At the end of Round 7, we left with a half-cooked smooth 1px horizontal scroller. In this article, we are going to continue from there, and make it work. We will start by exploring the XSCROLL part of the $D016 register.

After that, we are going to have a look at two variants for doing a scroller using sprites. One where the sprites are moved to do the scrolling, and one with fixed sprite position, where the scrolling effect is achieved by updating the data.

Everything in this article assumes a PAL C64: 312 raster lines, 63 CPU cycles per line. On an NTSC machine, some of the numbers (raster lines, timings, sprite X range) would be different.

## // XSCROLL part of the $D016 register

As already seen during Round 9, the $D016 register looks like this.

**$D016**

| bit7 | bit6 | bit5 | bit4 | bit3 | bit2 | bit1 | bit0 |
|---|---|---|---|---|---|---|---|
| unused | unused | unused (keep at 0) | 1 for Multicolor / 0 for HiRes | CSEL 1 for 40 columns 0 for 38 columns | XSCROLL horizontal offset from 0 to 7 default is 0 | | |
| default value | | | | | | | |
| 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 |

In this article, we are mainly going to work with bits 0, 1 and 2 that form together what is called the XSCROLL position of the screen. We will also need bit 3 to switch to 38 columns mode. In 38 columns mode, the border covers a bit more of the screen: 7 pixels more on the left, 9 on the right. That's exactly where the letters of a scroller enter and leave the screen, so the border hides them instead of letting them pop in and out.

But first let's try a few POKEs directly from the Commodore 64 BASIC screen. $D016 is 53270 in decimal.

```
POKE 53270,0
```

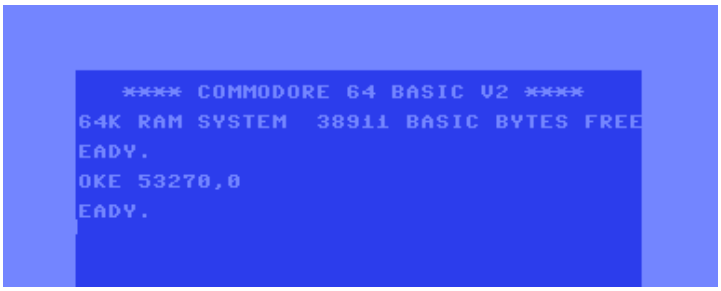



Figure 1: POKE 53270,0

Putting 0 into $D016 (53270) like this affects all bits. Bit 3 becomes 0 too, turning the C64 into 38 columns mode.

Let's not touch bit 3 and try the value 8, that is %00001000 in binary: all bits at 0 except bit 3.

```
POKE 53270,8
```

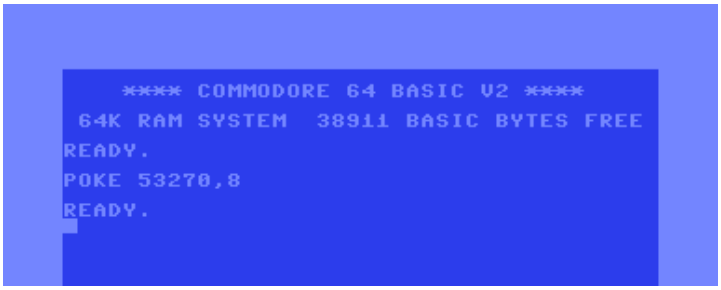



Figure 2: POKE 53270,8

We are back to 40 columns mode, with default value for XSCROLL (0 -> no offset). Let's now try a few values that should change XSCROLL, still keeping bit 3 at 1 to stay in 40 columns mode (so 8 + offset).

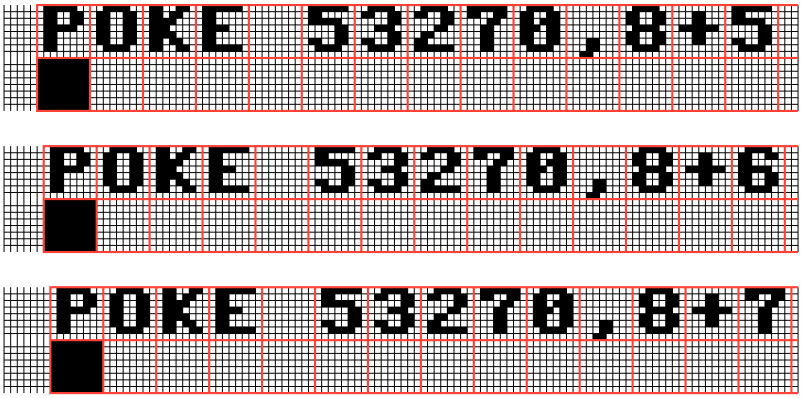
```
POKE 53270,8+0
```

```
POKE 53270,8+1
```

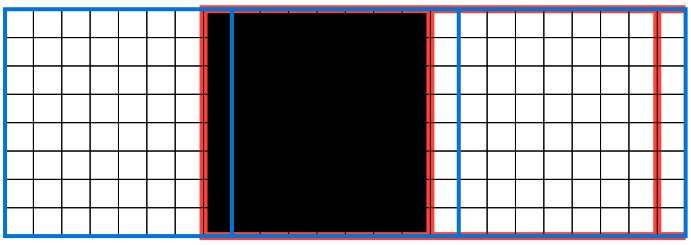
```
POKE 53270,8+2
```

```
POKE 53270,8+3
```

```
POKE 53270,8+4
```




Having a closer look at POKE 53270, 8+7, we can see that this moved the screen 7px to the right, so almost 1 full char.



Trying to go one step further to the right with POKE 53270, 8+8 doesn't work. You will end up with this.

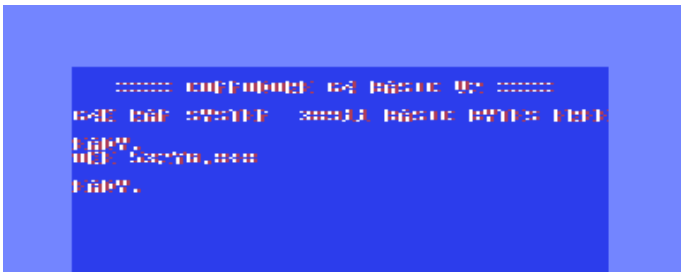
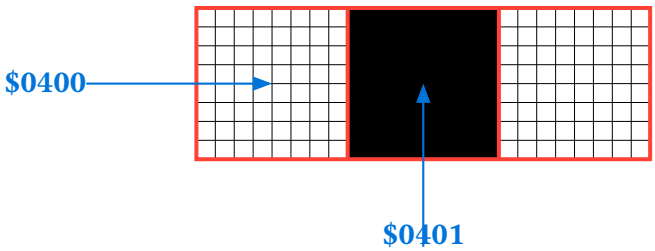


Figure 3: POKE 53270, 8+8

POKE 53270, 8+8 results in putting 16 into $D016, %00010000 in binary, and setting bit 4 activates the multicolor mode.

In order to move one pixel more to the right, we have to:

- reset XSCROLL to 0 (POKE 53270, 8+0)
- copy the char itself to the next screen position (for instance from $400 to $401 if the char is in the top left corner).




This is really all about XSCROLL in $D016. Let's now continue that rough char by char scrolling we did in Round 7 where we left with something like this.

## // X Scroller using $D016

At the end of Round 7, the code looked like this.

```
.const SCREEN = $0400
.const LINE = 12
.const LINE_ADR = SCREEN + LINE * 40
.pc = $02 "Zero Page" virtual
text_ptr: .word 0


BasicUpstart2(main)
main:
    jsr init_scroll
    jsr clear_scroll_line
loop:
    jsr delay
    jsr do_scroll
    jmp loop

init_scroll:
    lda #<scroll_text
    sta text_ptr
    lda #>scroll_text
    sta text_ptr+1
    rts

clear_scroll_line:
    ldx #39
    lda #' '
    !:
    sta LINE_ADR,x
    dex
    bpl !-
    rts

delay:
    ldy #8
    !:
    jsr wait
    dey
    bne !-
    rts

wait:
    bit $d011
    bpl *-3
    bit $d011
    bmi *-3
    rts


do_scroll:
    jsr shift_line_left
    jsr put_letter_on_right_side
    jsr next_letter
    jsr reset_scroll_if_needed
    rts

shift_line_left:
    ldy #0
    !:
    lda LINE_ADR + 1,y
    sta LINE_ADR,y
    iny
    cpy #39
    bne !-
    rts

put_letter_on_right_side:
    ldy #0
    lda (text_ptr),y
    sta LINE_ADR + 39
    rts

next_letter:
    inc text_ptr
    bne !+
    inc text_ptr+1
    !:
    rts

reset_scroll_if_needed:
    ldy #0
    lda (text_ptr),y
    bne !+
    jsr init_scroll
    !:
    rts

.pc = * "SCROLL TEXT DATA"
scroll_text:
  .text "test 123 - "
  .byte 0
```

Listing 1: 01_text-scroll-char-by-char.asm

$D016 was introduced at the very end of that article, but the whole screen was moving, do you remember?

```
loop:
    jsr wait
    jsr do_d016_scroll
    jmp loop

do_d016_scroll:
    dex
    bpl !+
    ldx #7
    jsr do_scroll
    !:
    stx $d016
    rts
```

If you tried some of the POKEs we saw above, the problem should now be obvious. $D016 acts on the whole screen, not just on the line we want to scroll.

So in order to fix that we could do the following:

- Wait for the raster to reach a line just before the text scroller
- Update $D016
- Apply other scroller routines
- Wait for a line after the text scroller
- Restore (or reset) $D016 to its default value

So, how do we wait for the raster beam to reach a given line? One way is to use an interrupt, but we haven't covered interrupts yet, so that's for the next round. The other way is just to read the current raster line number in a loop, and to exit that loop when the expected line is reached. Waiting for line 100 would look like this for instance:

```
BasicUpstart2(main)
main:
  lda #100 // the line to reach
loop:
  cmp $d012
  bne loop

  // line 100 reached
  lda #RED
  sta $d020
  sta $d021

  jmp *
```

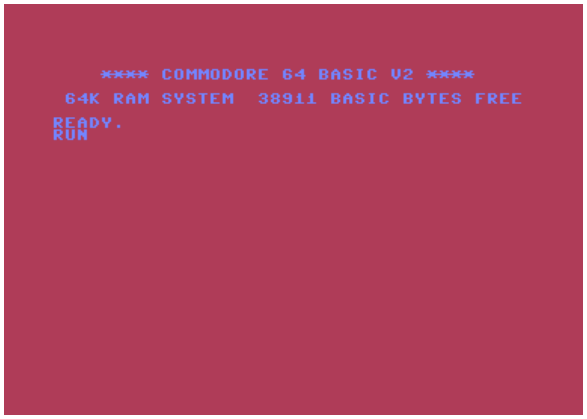



Figure 4: This produces a RED screen.

Does that really work? Well, yes and no. When you run this prg you will see that the screen is completely RED. There is no bug however: when line 100 is reached we change the color to RED, but then we never reset it, so it stays RED forever! What we should do instead is:

- Wait for line 100, change to RED
- Wait for line 150, change to GREEN
- do that forever

We could also use a shorter loop without using the ‘loop’ label.

```
BasicUpstart2(main)
main:

  lda #100 // Wait for line 100
  cmp $d012
  bne *-3

  // line 100 reached
  lda #RED
  sta $d020
  sta $d021

  lda #150 // Wait for line 150
  cmp $d012
  bne *-3

  // line 150 reached
  lda #GREEN
  sta $d020
  sta $d021

  jmp main // restart from the
beginning
```

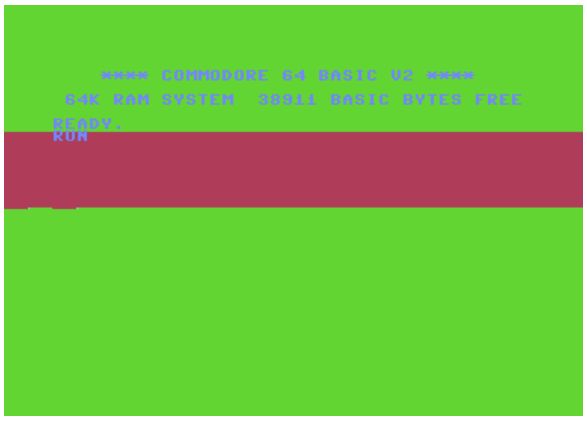



Figure 5: Works, but running this prg, you would see a bit of glitching and flickering.

**Side note:** Does that work for any line?

For lines 100 and 150, yes: no other raster line reads as 100 or 150 in $D012, so the first match is always the right one. And in the scroller below, each frame starts with `jsr wait`, which returns when the raster beam is back at the top of the frame (line 0). From there, the lines we wait for come in order down the screen.

But be aware that $D012 only holds the low 8 bits of the raster line, and a PAL C64 has 312 lines (0 to 311). The 9th bit is bit 7 of $D011, the same bit our `wait` routine uses. Lines 0 to 55 and lines 256 to 311 share the same $D012 values: line 300 reads as 44, just like line 44.

- Waiting for line 300 with `cmp $d012` alone stops at line 44 first.
- Without `jsr wait` before, waiting for line 30 may stop at line 286.

In those cases, check the 9th bit too:

```
    lda #<300
!:  cmp $d012
    bne !-
    bit $d011   // bit 7 = bit 8 of the line
    bpl !-      // that was line 44: wait more
```

For a line below 56, use `bmi` instead of `bpl`.

Getting rid of the glitching is easy: just add `sei` right below `BasicUpstart2(main)`. This prevents interrupts from occurring. Without it, the KERNAL interrupt sometimes runs while we are waiting, and we notice the line we wait for too late.

We will see what to do about flickering later. For now we just move on and apply the same to our scroller.

Let's review the plan again:

- Wait for the raster to reach a line just before the text scroller
- Update $D016
- Apply other scroller routines
- Wait for a line after the text scroller
- Restore (or reset) $D016 to its default value

That means the following change in our code.

from this:

```
loop:
    jsr wait
    jsr do_d016_scroll
    jmp loop
```

to this:

```
loop:
    jsr wait

    // wait for raster line 130
    lda #130
    cmp $d012
    bne *-3

    jsr do_d016_scroll

    // wait for raster line 160
    lda #160
    cmp $d012
    bne *-3

    // reset d016 to its
    // default value
    lda #8    // XSCROLL = 0, 8 is
              // to set it back to
              // 40 column mode
    sta $d016

    jmp loop
```

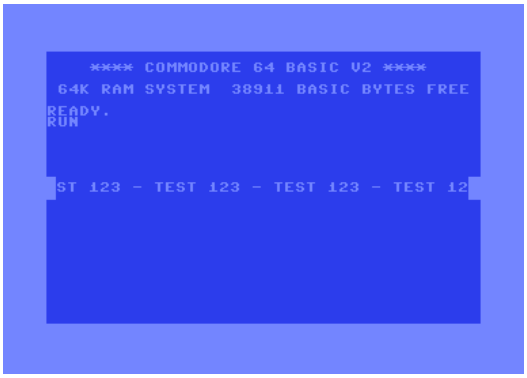



Figure 6: src/01_d016-scroll-try-1.asm

The top and bottom parts are not moving anymore. This is already way better than where we left off at the end of Round 7. That's the right direction, but:

- There still seems to be some flickering.
- Lines 130 and 160 seem to be a bit far away from the text scroller itself. Let's try to fine-tune the numbers here.

The scroller is on row 12, and the first row of characters starts at raster line 51, so that should be like this:

```
...
// one line before ROW 12
lda #12*8+51 -1
cmp $d012
bne *-3

jsr do_d016_scroll

lda #(12+1)*8+51  // ROW 13
cmp $d012
bne *-3

lda #8 // reset to d016 default
sta $d016
...
```

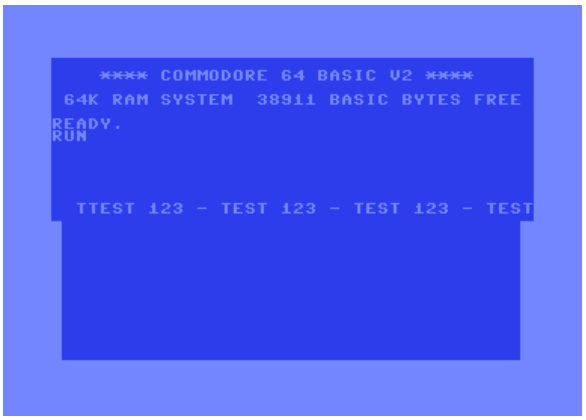



Figure 7: src/02_d016-scroll-try-2.asm

Ouch, that's really bad. You should try this one for real to see how glitchy it is. But why?

The reason this is not working well is that the time it takes to compute the scrolling conflicts with our raster timing.

The computation is done in `do_scroll`. Let's trace the raster time it takes by changing the color of the border during the execution of that part of the code.

```
do_scroll:
    // a very common way of tracing
    // the time a routine takes.
    inc $d020

    jsr shift_line_left
    jsr put_letter_on_right_side
    jsr next_letter
    jsr reset_scroll_if_needed

    dec $d020 // end of tracing
    rts
```

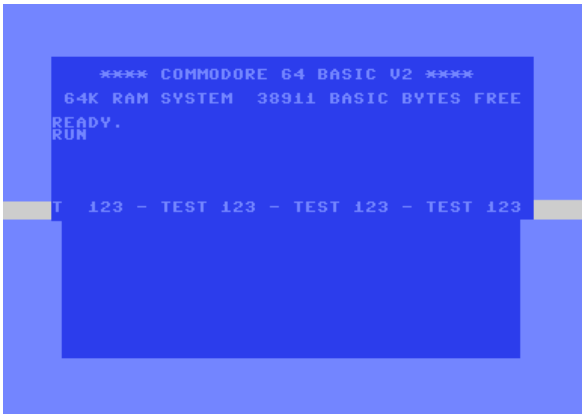



Figure 8: Tracing the raster time a routine takes by coloring the border

The problem is not the time it takes: there are plenty of cycles to do a proper scrolling on the C64. The problem is that the computation occurs during the scrolling itself. The grey color overlaps with the scroll text, which means both happen at the same time.

Why is that an issue? Because we do time-critical stuff like this:

```
    jsr do_d016_scroll // this one calls do_scroll

    // wait for ROW 13...
    // but that's already too late...
    // we may already be past this line when we arrive here...
    // and then we are going to wait for that line on the next frame
    // making the whole effect glitchy...

    lda #(12+1)*8+51
    cmp $d012
    bne *-3
```

Let's say you add an offset (here 10 raster lines) to ensure all the computation is done before we reach that line:

```
    .const COMPENSATION = 10
    ...

    lda #(12+1)*8+51 + COMPENSATION
    cmp $d012
    bne *-3

    lda #8 // reset to d016 default
    sta $d016
    ...
```

Meh. That looks slightly better, but still not perfect, and it looks more like a hack, right? So let's fix that properly, by moving the computation below the scrolling region.

First, let's put some chars on the screen so that we can check that our scroller is only moving the screen region it should.

```
.const LINE = 12
.const LINE_ADR = $400 + LINE*40
BasicUpstart2(main)
main:
    lda #BLACK
    sta $d020

    lda #$a0
    sta LINE_ADR -39
    sta LINE_ADR -20
    sta LINE_ADR -2

    sta LINE_ADR +41
    sta LINE_ADR +60
    sta LINE_ADR +78

    jmp *
```




Figure 9: Some blocks on the screen to ensure only the scroll region is updated

Now do the computation after the critical raster timing part.

```
.const LINE = 12
.const LINE_ADR = $400 + LINE*40
.pc = $02 "Zero Page" virtual
text_ptr: .word 0

BasicUpstart2(main)
main:
    lda #BLACK
    sta $d020
    sei
    jsr init_scroll

    // put some blocks on the screen
    // to ensure only the scroll region
    // is updated

    lda #$a0
    sta LINE_ADR -39
    sta LINE_ADR -20
    sta LINE_ADR -2

    sta LINE_ADR +41
    sta LINE_ADR +60
    sta LINE_ADR +78

    ldx #0
loop:
    jsr wait
    jsr raster_scroll
    jmp loop

init_scroll:
    lda #<text
    sta text_ptr
    lda #>text
    sta text_ptr+1
    rts

raster_scroll:

    lda #LINE*8 +51 -1
    cmp $d012
    bne *-3

    stx $d016

    lda #(LINE+1)*8 +51
    cmp $d012
    bne *-3

    lda #$08
    sta $d016

    // most of the computation occurs now after
    // the scroll region
    jsr do_d016_scroll
    rts
```

```
do_d016_scroll:
    dex
    bpl !+
    ldx #7
    jsr do_scroll
    !:
    rts

do_scroll:
    inc $d020 // colored border to trace
              // where the execution occurs.

    jsr put_letter
    jsr shift_line_left
    jsr next_letter
    jsr reset_scroll_if_needed

    dec $d020
    rts

reset_scroll_if_needed:
    ldy #0
    lda (text_ptr),y
    bne !+
    jsr init_scroll
    !:
    rts

next_letter:
    inc text_ptr
    bne !+
    inc text_ptr+1
    !:
    rts

put_letter:
    ldy #0
    lda (text_ptr),y
    sta LINE_ADR + 39
    rts

shift_line_left:
    .for (var col = 1; col < 40; col ++) {
        lda LINE_ADR + col
        sta LINE_ADR + col - 1
    }
    rts

wait:
    bit $d011
    bpl *-3
    bit $d011
    bmi *-3
    rts

text:
    .text "test 123 "
    .byte $a0, ' '
    .byte 0
```

Listing 2: 03_d016-almost-working.asm

A few things to notice in this listing:

- `raster_scroll` does the time-critical part (waiting for the lines and writing $D016), then calls `do_d016_scroll` for the rest.
- `stx $d016` writes a value between 0 and 7, so bit 3 is 0: the scroll row is in 38 columns mode. After the row, `lda #$08` / `sta $d016` puts the rest of the screen back in 40 columns mode.
- `put_letter` is now called before `shift_line_left`. The new letter goes into column 39 and is copied into column 38 right away. Column 39 is never visible in 38 columns mode, and when XSCROLL is 7, column 38 is still behind the border too. The letter then slides in pixel by pixel.

We are really close now: the computation (the white zone in the border) occurs after the scroll region.




But the scroll region is not pixel perfect yet.




The easiest trick for now is to add a few NOPs before writing $D016. They delay the write to the end of the raster line, when the beam is in the border, so the change no longer happens in the middle of a visible line.

```
raster_scroll:
    lda #LINE*8 +50
    cmp $d012
    bne *-3

    .for (var i=0;i<26;i++) {
        nop
    }

    stx $d016

    lda #(LINE+1)*8 +50
    cmp $d012
    bne *-3

    .for (var i=0;i<24;i++) {
        nop
    }

    lda #$08
    sta $d016
    jsr do_d016_scroll
    rts
```

**And here we are: a simple, yet beautiful, smooth scroller.**

Whole listing available in src/04_d016-working-scroller.asm

## // Scrollers using sprites

Using sprites is another very common way of doing a text scroller. We are going to explore two different ways of doing it. The first one where we are going to move the sprites in x, and a second one where sprites stay at a fixed position, and where we do the scrolling by updating the data.

### Scroller with moving sprites

It's always good to think about the layout of the screen before jumping into the code.

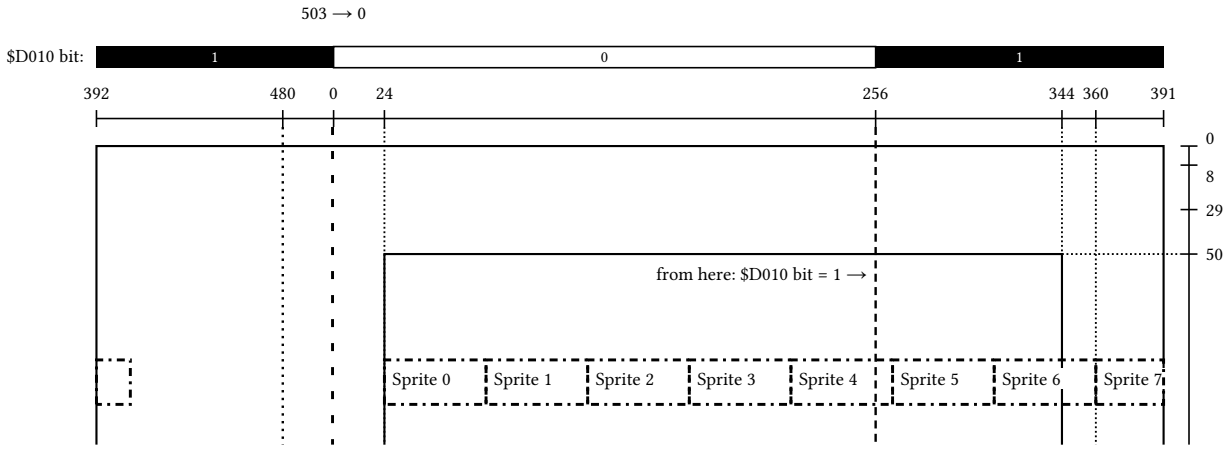



Figure 10: Starting position

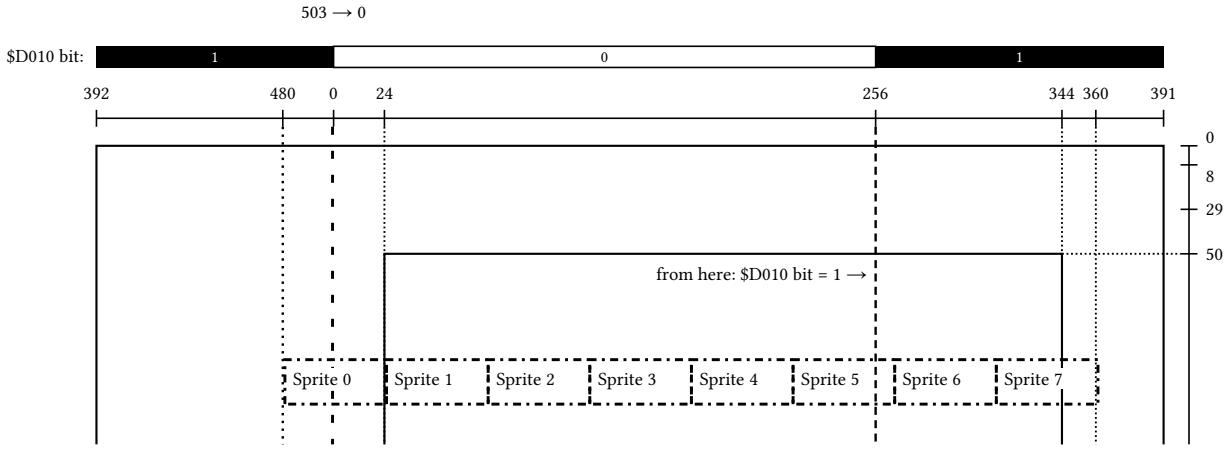



Figure 11: frame 47

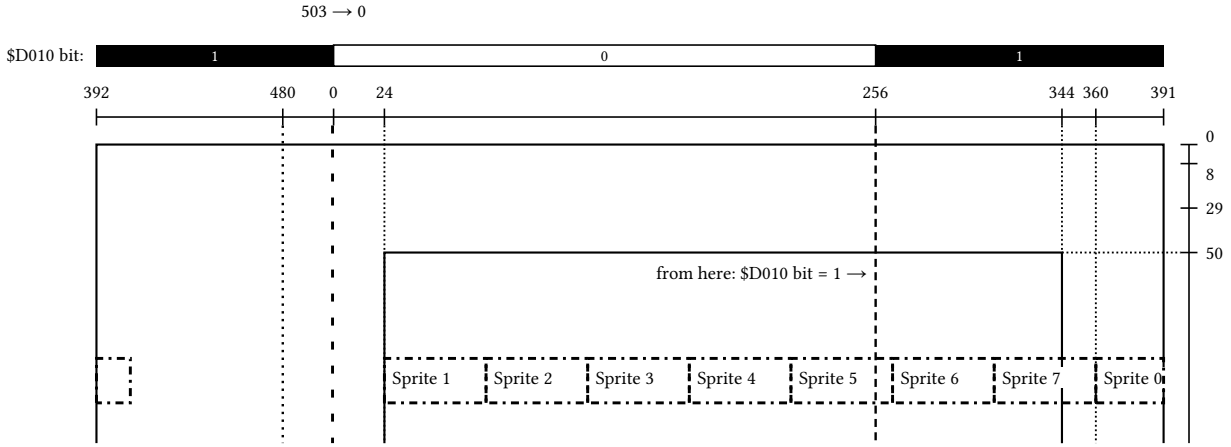



Figure 12: frame 48 - When a sprite reaches X = 480 (here sprite 0), it goes back to 360

Let's start with just this, without thinking about the text for now.

```
.const SCROLL_Y_POS = 100
.const SCREEN = $0400
.const NUM = 8

BasicUpstart2(main)
main:
    lda #BLACK
    sta $d021
    sei
    jsr init_sprites
loop:
    jsr wait
    jsr scroll
    jmp loop

// each sprite moves left by
// 1 pixel per frame
scroll:
    .for (var i = 0; i < NUM; i++) {
        // 16-bit decrement
        sec
        lda sprites_pos.lo+i
        sbc #1
        sta sprites_pos.lo+i
        lda sprites_pos.hi+i
        sbc #0
        bpl !ok+
            lda #<503  // 0 - 1 -> 503
            sta sprites_pos.lo+i
            lda #>503
    !ok:
        sta sprites_pos.hi+i

        // fully hidden on the
        // left (480)? -> move it
        // behind the last sprite
        cmp #>480
        bne !next+
        lda sprites_pos.lo+i
        cmp #<480
        bne !next+
            lda #<360
            sta sprites_pos.lo+i
            lda #>360
            sta sprites_pos.hi+i

    !next:
    }
    jsr update_x
    rts
```

```
init_sprites:
    lda #SCROLL_Y_POS
    .for (var i = 0; i < NUM; i++) {
        sta $d001 + i*2
    }
    .for (var i = 0; i < NUM; i++) {
        // just makes all sprites
        // pointing to the same data
        lda #sprites_data/64
        sta SCREEN + $3f8 + i
    }
    lda #0
    sta $d010
    jsr update_x
    lda #$ff
    sta $d01d   // expand X
    sta $d015   // all sprites on
    rts

// copies sprites_pos
// into $D000-$D00F
// and the 9th bit into $D010
update_x:
    ldx #0
    ldy #0
!loop:
    lda sprites_pos.lo,x
    sta $d000,y
    lda sprites_pos.hi,x
    beq !+
        lda mask,x
        ora $d010
        jmp !c+
    !:
        lda inv_mask,x
        and $d010
!c:
    sta $d010
    iny
    iny
    inx
    cpx #NUM
    bne !loop-
    rts


wait:
    bit $d011
    bpl *-3
    bit $d011
    bmi *-3
    rts

sprites_pos:
    .lohifill NUM, 24 + i*48
mask:
    .fill 8, 1 << i
inv_mask:
    .fill 8, (1 << i) ^ $ff

* = $2000 "Sprites"
sprites_data:
    .fill 3, $ff
    .fill 19,[128,0,1]
    .fill 3, $ff
    .byte 0
```

Listing 3: 05_sprite-x-move.asm

You should see something like this, with the sprites moving left.




Let's now fill those sprites with our scroller's text.

### Reading the default font and putting a letter into a sprite

Again, let's proceed step by step. Here is a simple snippet filling a single sprite with a letter coming from the default C64 charset.

```
.const LETTER = 1          // screen code: 1 = "A" (0 = @, 2 = B, ...)

BasicUpstart2(main)
main:
    sei                    // no IRQ while I/O is switched off
    lda #$33               // char ROM visible at $D000 (instead of I/O)
    sta $01

    ldx #0                 // X: row in the char (0..7)
    ldy #0                 // Y: offset in the sprite (0, 3, 6 ... 21)
!:
    lda $d000 + LETTER*8,x // 8 bytes per char -> the char starts at LETTER*8
    sta sprite_data,y
    iny                    // a sprite row is 3 bytes wide
    iny
    iny
    inx
    cpx #8
    bne !-

    lda #$37               // back to normal: I/O visible again
    sta $01

    // show sprite 0
    lda #sprite_data/64
    sta $07f8              // sprite 0 pointer
    lda #100
    sta $d000              // X
    sta $d001              // Y
    lda #WHITE
    sta $d027
    lda #1
    sta $d01d              // expand X
    sta $d015              // sprite 0 on

    jmp *

* = $2000 "Sprite"
sprite_data:
    .fill 64, 0
```

Listing 4: 06_sprite_single_letter.asm




> **Side note:** $D000 vs $D000?
> $D000 is both the address at which we can read the data for the default font AND the X position register of sprite 0: two very different things. The value that is put into register $01 (or $0001, that's an address like any other one) defines how $D000 is going to behave. That's something we already saw in Round 8.

Good, but we need 3 letters per sprite, not just one. Let's do this now.

```
.pc = $02 "Zero Page" virtual
text_ptr: .word 0
src:      .word 0  // address of the char in the
font
dst:      .word 0  // current column in the sprite

.label font = $d000

BasicUpstart2(main)
main:
    sei

    jsr init_scroll
    // dst -> first column of sprite 0
    lda #<sprite_data
    sta dst
    lda #>sprite_data
    sta dst+1

    lda #$33         // char ROM visible at $D000
    sta $01
    jsr put_letter   // column 0
    jsr next_letter
    jsr put_letter   // column 1
    jsr next_letter
    jsr put_letter   // column 2
    lda #$37         // I/O visible again
    sta $01

    // show sprite 0
    lda #sprite_data/64
    sta $07f8
    lda #100
    sta $d000
    sta $d001
    lda #WHITE
    sta $d027
    lda #1
    sta $d01d        // expand X
    sta $d015
    jmp *
```

```
init_scroll:
    lda #<text
    sta text_ptr
    lda #>text
    sta text_ptr+1
    rts

next_letter:
    inc text_ptr
    bne !+
    inc text_ptr+1
    !:
    rts

// copy the 8 bytes of the char into one column
// of the sprite
put_letter:
    ldy #0
    lda (text_ptr),y  // screen code of the letter
    sta src           // src = font + code * 8
    sty src+1
    .for (var s = 0; s < 3; s++) {
        asl src
        rol src+1
    }
    clc
    lda src
    adc #<font
    sta src
    lda src+1
    adc #>font
    sta src+1
    .for (var r = 0; r < 8; r++) {
        ldy #r
        lda (src),y  // row r of the char
        ldy #r*3     // row r of the sprite
                     // (3 bytes per row)
        sta (dst),y
    }
    inc dst          // next column for the next
call
    rts

text:
    .text "test 123 "
    .byte $a0, ' '
    .byte 0

* = $2000 "Sprite"
sprite_data:
    .fill 64, 0
```

Listing 5: 07_sprite_3_letters.asm

That should display something like this.




## Putting everything together

Now we have to put all that together. We already have all the pieces: the moving sprites from `05_sprite-x-move.asm`, and `put_letter` from `07_sprite_3_letters.asm`. Here is what changes when we merge them.




### Each sprite gets its own data

In `05_sprite-x-move.asm`, all the sprites pointed to the same data. Now each sprite needs its own 64 bytes, and a table gives the address of each one:

```
init_sprites:
    ...
    .for (var i = 0; i < NUM; i++) {
        lda #sprites_data/64 + i    // sprite i -> its own block
        sta SCREEN + $3f8 + i
    }
    ...

spr_addr:
    .lohifill NUM, sprites_data + i*64

* = $2000 "Sprites"
sprites_data:
    .fill NUM*64, 0
```

### New letters when a sprite goes back to the right

The only change in `scroll` is in the part that moves a sprite from 480 back to 360. That sprite is now hidden behind the right border, so it's the right moment to fill it with the next 3 letters:

```
            lda #<360
            sta sprites_pos.lo+i
            lda #>360
            sta sprites_pos.hi+i
            lda #i            // which sprite to fill
            jsr do_scroll
    !next:
```

`do_scroll` points `dst` to the data of that sprite, then calls `put_letter` 3 times, once per column:

```
do_scroll:
    tax
    lda spr_addr.lo,x
    sta dst
    lda spr_addr.hi,x
    sta dst+1
    ldx #3
!:
    jsr put_letter      // writes 1 column, then inc dst
    jsr next_letter
    jsr reset_scroll_if_needed
    dex
    bne !-
    rts
```

### A copy of the font

In `07_sprite_3_letters.asm`, we switched `$01` to read the char ROM directly. Here, so that `put_letter` doesn't have to care about `$01`, `copy_font` copies the 2 KB of the char ROM to $1000 once, at startup, and `put_letter` reads from there (`font` = $1000). This way, I/O stays visible all the time, and we can't forget to switch it back before writing to the sprite registers.

Why $1000? The VIC-II can't use $1000–$1FFF for sprites or fonts (it sees the char ROM there), but for the CPU it's normal RAM. That's a perfect place for data that only the CPU reads.

```
copy_font:
    lda $01
    pha
    lda #$33          // char ROM visible at $D000
    sta $01
    ldx #0
!:
    .for (var p = 0; p < 8; p++) {
        lda $d000 + p*$100,x
        sta font  + p*$100,x
    }
    inx
    bne !-
    pla
    sta $01
    rts
```

Everything else (`update_x`, `wait`, `init_scroll`, `next_letter`, `reset_scroll_if_needed`) is unchanged. The whole listing is available in `src/08_sprite-scroll.asm`.

## Scroller with sprites at fixed positions

This time, let's see the whole code first, and then we will talk about the differences compared to the previous version.

```
.const SCROLL_Y_POS = 100
.const SCREEN = $0400
.const NUM = 7
.const COLUMNS = NUM * 3        // sprite bytes per row, side by
side

.pc = $02 "Zero Page" virtual
text_ptr: .word 0
src:      .word 0               // address of the char in the font

// address of row r, column c (0..20, left to right)
// in the sprite data
.function sprite_byte(r, c) {
    .return sprites_data + floor(c/3)*64 + mod(c, 3) + r*3
}

BasicUpstart2(main)
main:
    lda #BLACK
    sta $d020
    sta $d021
    sei
    jsr init_scroll
    jsr copy_font
    jsr init_sprites
    ldx #0
loop:
    jsr wait
    jsr scroll
    jmp loop

init_scroll:
    lda #<text
    sta text_ptr
    lda #>text
    sta text_ptr+1
    rts

do_scroll:
    jsr put_letter
    jsr next_letter
    jsr reset_scroll_if_needed
    rts

reset_scroll_if_needed:
    ldy #0
    lda (text_ptr),y
    bne !+
    jsr init_scroll
    !:
    rts

next_letter:
    inc text_ptr
    bne !+
    inc text_ptr+1
    !:
    rts

wait:
    bit $d011
    bpl *-3
    bit $d011
    bmi *-3
    rts

scroll:
    dex
    bpl !+
    ldx #7
    jsr do_scroll
    !:
    jsr shift_pixels_left
    rts

shift_pixels_left:
    .for (var r = 0; r < 8; r++) {
        asl sprite_byte(r, COLUMNS - 1)
        .for (var c = COLUMNS - 2; c >= 0; c--) {
            rol sprite_byte(r, c)
        }
    }
    rts
```

```
put_letter:
    ldy #0
    lda (text_ptr),y
    sta src     // src = font + code * 8
    sty src+1
    .for (var s = 0; s < 3; s++) {
        asl src
        rol src+1
    }
    clc
    lda src
    adc #<font
    sta src
    lda src+1
    adc #>font
    sta src+1
    .for (var r = 0; r < 8; r++) {
        ldy #r
        lda (src),y
        sta sprite_byte(r, COLUMNS - 1)  // column 20
    }
    rts

copy_font:
    lda $01
    pha
    lda #$33   // char ROM visible at $D000
    sta $01
    ldx #0
!:
    .for (var p = 0; p < 8; p++) {
        lda $d000 + p*$100,x
        sta font  + p*$100,x
    }
    inx
    bne !-
    pla
    sta $01
    rts

init_sprites:
    lda #SCROLL_Y_POS
    .for (var i = 0; i < NUM; i++) {
        sta $d001 + i*2
    }
    .for (var i = 0; i < NUM; i++) {
        lda #sprites_data/64 + i
        sta SCREEN + $3f8 + i
    }

    .var msb = 0
    .for (var i = 0; i < NUM; i++) {
        .var x = 24 + i*48
        .if (x > 255) .eval msb = msb | (1 << i)
        lda #<x
        sta $d000 + i*2
    }
    lda #msb
    sta $d010
    lda #$ff
    sta $d01d   // expand X
    sta $d015
    rts

text:
    .text "test 123 "
    .byte $a0, ' '
    .byte 0

* = $1000 "Font"
font:
    .fill 2048, 0

* = $2000 "Sprites"
sprites_data:
    .fill NUM*64, 0
```

Listing 6: 09_sprite-scroll-rol.asm

### What's different in this version

The first thing is that this version with fixed sprites "only" needs 7 sprites. The initial layout is similar to the previous version, except that we are not using sprite 7.

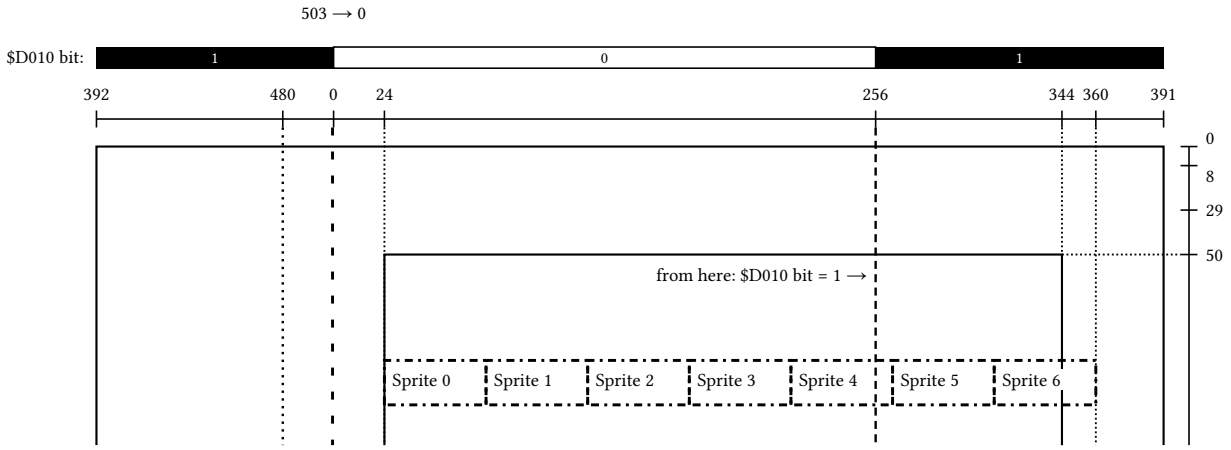



Figure 13: 7 expanded sprites, from X = 24 to X = 360. They never move.

7 sprites of 48 pixels give 336 pixels, from 24 to 360. That's enough to cover the 320 pixels of the screen (24 to 344). In the previous version we needed an 8th sprite, because there is always one sprite on its way out on the left while its replacement is already entering on the right. Here nothing moves, so sprite 7 is free for something else.

### The sprites never move

Since the positions never change, there is no `sprites_pos` table, no `mask` / `inv_mask` tables and no `update_x` anymore. The X positions, and the value for $D010, are computed once by KickAssembler, at assembly time:

```
    .var msb = 0
    .for (var i = 0; i < NUM; i++) {
        .var x = 24 + i*48
        .if (x > 255) .eval msb = msb | (1 << i)
        lda #<x
        sta $d000 + i*2
    }
    lda #msb
    sta $d010
```

### Back to the structure of the text scroller

Have a look at `scroll`. Does it ring a bell?

```
// text version (d016)
do_d016_scroll:
    dex
    bpl !+
    ldx #7
    jsr do_scroll
    !:
    rts
```

```
// fixed sprites version
scroll:
    dex
    bpl !+
    ldx #7
    jsr do_scroll
    !:
    jsr shift_pixels_left
    rts
```

It's the same counter: every 8 frames, `do_scroll` brings in a new letter. The difference is how the pixels move. With $D016, the VIC-II does it for us, and every 8 frames we have to move the whole line by one char ourselves (`shift_line_left`). With the sprites, there is no XSCROLL register: we move the pixels ourselves, one sprite pixel per frame, in `shift_pixels_left`. And because the pixels are really moved, there is no "move by one char" step anymore: `do_scroll` only has to bring the next letter in.

Careful: the sprites are expanded in X, so one sprite pixel is 2 pixels wide on the screen. This scroller moves 2 screen pixels per frame, twice as fast as the $D016 version and the moving sprites version. (In both sprite versions, the letters are 16 pixels wide on the screen, instead of 8 in the $D016 version.)

| | **Text ($D016)** | **Fixed sprites** |
|---|---|---|
| each frame | 1 screen pixel: `stx $d016` (the VIC-II does it) | 1 sprite pixel = 2 screen pixels: `shift_pixels_left` (the CPU does it) |
| every 8 frames | `shift_line_left`, then `put_letter` | `put_letter` only |
| new letter goes to | column 39 (hidden) | column 20 (hidden) |
| raster timing | critical: wait for the line, NOPs | none, just `wait` |

### Column 20: our hidden column 39

In the text version, `put_letter` writes the new letter in column 39, at the right edge of the screen, hidden by the 38 columns mode, and XSCROLL makes it appear pixel by pixel.

Here, we use the same trick with the border. Each sprite row is 3 bytes, so the 7 sprites give 21 columns of bytes, numbered 0 to 20 from left to right. Each column is 8 sprite pixels, so 16 screen pixels once expanded:

| **sprite** | Sprite 0 | | | Sprite 1 | | | Sprite 2 | | | Sprite 3 | | | Sprite 4 | | | Sprite 5 | | | Sprite 6 | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **X** | 24–72 | | | 72–120 | | | 120–168 | | | 168–216 | | | 216–264 | | | 264–312 | | | 312–360 | | |
| **column** | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 |
| **byte** | L | M | R | L | M | R | L | M | R | L | M | R | L | M | R | L | M | R | L | M | R |
| **screen** | visible: X = 24 ⟶ 344 | | | | | | | | | | | | | | | | | | | | border |

Figure 14: The 21 columns of bytes. Columns 0 to 19 cover the 320 visible pixels. Column 20 (X = 344 to 360) is behind the right border.

Sprite 6 goes from 312 to 360, so its last byte, column 20, is between 344 and 360, behind the border. `put_letter` copies the 8 bytes of the letter straight into column 20, one byte per row. Each frame, one pixel of each row leaves column 20 and enters column 19, which is visible. After 8 frames column 20 is empty, and it's time for the next letter.

### Shifting the pixels with `rol`

Each row of the text is made of 21 bytes, 3 per sprite, side by side. To move a row one pixel to the left, every bit must move one place to the left, including the bit that crosses from one byte to the next. That's exactly what the carry is for as we already saw in Part I:

- `asl` moves all bits one place to the left, puts a 0 in bit 0, and bit 7 goes into the carry.
- `rol` does the same, but bit 0 receives the old carry instead of a 0.

So we start with `asl` on the rightmost byte (column 20). Then we go from right to left with `rol`, and the carry brings the leftmost pixel of each byte into the byte on its left:

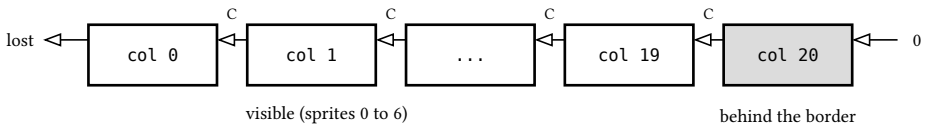



Figure 15: One row, shifted 1 sprite pixel to the left. `asl` first, then `rol` from right to left.

The order matters: going from left to right would put the carry into the wrong byte. That's why the inner loop goes from `COLUMNS - 2` (column 19) down to 0, after the `asl` on column 20:

```
shift_pixels_left:
    .for (var r = 0; r < 8; r++) {
        asl sprite_byte(r, COLUMNS - 1)
        .for (var c = COLUMNS - 2; c >= 0; c--) {
            rol sprite_byte(r, c)
        }
    }
    rts
```

**From "row r, column c" to an address: `sprite_byte`**

In `shift_pixels_left`, we think of the text as a simple grid: 8 rows (one per row of the char) and 21 columns of bytes, from column 0 on the left to column 20 on the right. But in memory, the bytes are not stored like that. They are spread over 7 sprites. `sprite_byte(r, c)` does the translation.

**Inside one sprite.** A sprite row is 24 pixels, so 3 bytes: left, middle and right. The rows are stored one after the other, 3 bytes each. Row r starts at offset `r*3`:

| | left | middle | right |
|---|---|---|---|
| **row 0** | 0 | 1 | 2 |
| **row 1** | 3 | 4 | 5 |
| **row 2** | 6 | 7 | 8 |
| **...** | ... | ... | ... |
| **row 7** | 21 | 22 | 23 |

**From one sprite to the next.** The sprites are stored one after the other, every 64 bytes: sprite 0 at $2000, sprite 1 at $2040, sprite 2 at $2080, and so on.

**Column c.** Every sprite holds 3 columns, so:

- column c is in sprite `floor(c/3)`: columns 0 to 2 are in sprite 0, 3 to 5 in sprite 1, and so on,
- inside that sprite row, it is the byte `mod(c, 3)`: 0 for left, 1 for middle, 2 for right.

Putting the three parts together:

```
.function sprite_byte(r, c) {
    .return sprites_data + floor(c/3)*64 + mod(c, 3) + r*3
    //                     which sprite    which byte  which row
}
```

For example, row 2, column 19: column 19 is in sprite 6 (19/3 = 6), at the middle byte (19 mod 3 = 1). The address is $2000 + 6×64 + 1 + 2×3 = $2000 + 384 + 7 = $2187.

Here are the addresses of row 0 for some columns. Note the jump of 64 bytes ($40) every 3 columns:

| **column c** | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | … | 18 | 19 | 20 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **sprite** | 0 | 0 | 0 | 1 | 1 | 1 | 2 | 2 | 2 | … | 6 | 6 | 6 |
| **byte** | L | M | R | L | M | R | L | M | R | … | L | M | R |
| **row 0** | $2000 | $2001 | $2002 | $2040 | $2041 | $2042 | $2080 | $2081 | $2082 | … | $2180 | $2181 | $2182 |

`sprite_byte` is a KickAssembler function: it runs at assembly time, not on the C64. Each `rol sprite_byte(r, c)` becomes a `rol` with a fixed address, like `rol $2187`. No pointer and no index register are needed, which is why `shift_pixels_left` is so simple.

Note that only the first 8 rows of each sprite are used, one per row of the char. The other 13 rows stay empty.

### Which one to choose?

Both versions copy letters into the sprites from time to time: 3 letters every 48 frames for the moving sprites, 1 letter every 8 frames for the fixed sprites. The real difference is what happens every frame. The moving sprites version only updates 8 X positions. The fixed sprites version runs 168 asl / rol instructions (8 rows × 21 bytes), which costs a lot more. In exchange, nothing moves, sprite 7 is free, and since we are changing the pixels ourselves anyway, it's a good starting point for effects on the text itself.

## // Comments?

Something is wrong or missing? Feel free to send me a PM (OCA/Atlantis) on CSDb.

## // What's coming next

In the next article, we are going to play with interrupts and explore the kernel a bit.

## // Registers we saw so far

| Address | Chip | What it does | Round |
|---|---|---|---|
| $01 | 6510 (CPU) | Bits 0–2 (LORAM, HIRAM, CHAREN) choose what the CPU sees at $A000–$BFFF, $D000–$DFFF and $E000–$FFFF. Default: $37. | 8 |
| screen + $3F8–$3FF | VIC-II | Sprite pointers. Data of sprite n starts at bank start + value × 64. $07F8–$07FF with the default screen. | 10 |
| $D000–$D00F | VIC-II | Sprite positions: $D000 + 2×n is the X (low 8 bits), $D001 + 2×n the Y of sprite n. | 10 |
| $D010 | VIC-II | Bit n: 9th bit of the X position of sprite n. | 10 |
| $D011 | VIC-II | Bit 6: Extended mode. Bit 5: Bitmap mode. Bit 4: screen visible. Bit 3: 25 or 24 rows. Bits 0–2: vertical scroll. Bit 7: bit 8 of the raster line. Default: $1B. | 9 |
| $D012 | VIC-II | Raster line currently being drawn (bits 0–7). Bit 8 is bit 7 of $D011. | 9 |
| $D015 | VIC-II | Bit n: sprite n on (1) or off (0). | 10 |
| $D016 | VIC-II | Bit 4: Multicolor mode. Bit 3: 40 or 38 columns. Bits 0–2: horizontal scroll. Default: $C8. | 9 |
| $D017 | VIC-II | Bit n: sprite n expanded in Y. | 10 |
| $D018 | VIC-II | First digit: screen position (steps of $0400). Second digit: font position (steps of $0800, bit 0 unused). Both inside the current bank. | 8 |
| $D01B | VIC-II | Bit n: sprite n behind (1) or in front of (0) the foreground graphics. | 10 |
| $D01C | VIC-II | Bit n: sprite n is multicolor. | 10 |
| $D01D | VIC-II | Bit n: sprite n expanded in X. | 10 |
| $D01E | VIC-II | Bit n: sprite n hit another sprite. Reading clears it. | 10 |
| $D01F | VIC-II | Bit n: sprite n hit foreground graphics. Reading clears it. | 10 |
| $D020 | VIC-II | Border color. | 8 |
| $D021 | VIC-II | Background color (color for `%00` in the multicolor text and bitmap modes). | 8 |
| $D022–$D023 | VIC-II | Colors for `%01` and `%10` in multicolor text, and extra background colors in Extended mode. | 9 |
| $D024 | VIC-II | 4th background color in Extended mode. | 9 |
| $D025–$D026 | VIC-II | Shared colors of multicolor sprites: `%01` and `%11`. | 10 |
| $D027–$D02E | VIC-II | Color of sprites 0 to 7 (`%10` in multicolor). | 10 |
| $D800–$DBE7 | Color RAM | Color of each character on the screen. Always at $D800, wherever the screen is. Only the low 4 bits are stored. | 8 |
| $DD00 | CIA 2 | Bits 0–1 choose the 16 KB bank the VIC-II sees. Inverted: 3 = bank 0, 2 = bank 1, 1 = bank 2, 0 = bank 3. | 8 |