Round 10 • Sprites

OCA / Atlantis

// Introduction

In this article we are going to play with sprites. A sprite is a graphical object that can be placed anywhere on the screen, on top of or below the text or bitmap.

There are many, many tricks involving sprites... but in this article, we are going to stay with the basic operations. We will explore a few sprite tricks later in Part-III.

We will start playing with things that work and then fall into a few traps where sprites are corrupted or not shown at all. Finally we will list the different registers related to sprites.

Along the way, we will also explore **8.8 fractional numbers**. They are not strictly related to sprites, but they come in very handy when moving them around.

// The happy path

Displaying a sprite on the screen

As you can see in the following example, there are 3 things that are mandatory for the sprite to appear on the screen.

- Setting where the sprite data are located in memory
- Setting x and y positions
- Telling the VIC-II which sprites should be on or off.

```
BasicUpstart2(main)
main:
  // memory location for data
  lda #sprite_data/64
  sta $07F8

  // set the position on the screen
  lda #100
  sta $D000 // X position
  sta $D001 // Y position

  // decide which sprites to display
  lda #%0000_0001
  sta $D015

forever: jmp forever

.align 64
sprite_data:
  // vertical lines
  .fill 63, %10101010 // same as $aa
  .byte 0
```

Listing 1: 01-displaying-sprite.asm



Figure 1: A single sprite at position (100,100)

The Sprite data

```
  ...

  lda #sprite_data/64
  sta $07F8

  ...

.align 64
sprite_data:
  // vertical lines
  .fill 63, %10101010 // same as $aa
  .byte 0
```

First, let's try to understand why "sta $07F8". We have to tell the VIC-II chip where the data is for each of our sprites. We do this by putting a value into a very specific address, in this case $07F8.

We saw in Round 8 that the screen configuration in $D018 is done using $0400 chunks of memory. $0400 is 1024 in decimal. Our screen is 40x25 chars, so a total of 1000 bytes for the chars. That means the last 24 bytes are not used to display chars on the screen.

The very last 8 bytes of that 1024 "chunk" are the ones that hold a pointer to the data for each of the 8 sprites.

- Sprite 0 pointer is at $0400 + $3F8 + 0 = $07F8
- Sprite 1 pointer is at $0400 + $3F8 + 1 = $07F9
- Sprite 2 pointer is at $0400 + $3F8 + 2 = $07FA
- Sprite 3 pointer is at $0400 + $3F8 + 3 = $07FB
- Sprite 4 pointer is at $0400 + $3F8 + 4 = $07FC
- Sprite 5 pointer is at $0400 + $3F8 + 5 = $07FD
- Sprite 6 pointer is at $0400 + $3F8 + 6 = $07FE
- Sprite 7 pointer is at $0400 + $3F8 + 7 = $07FF

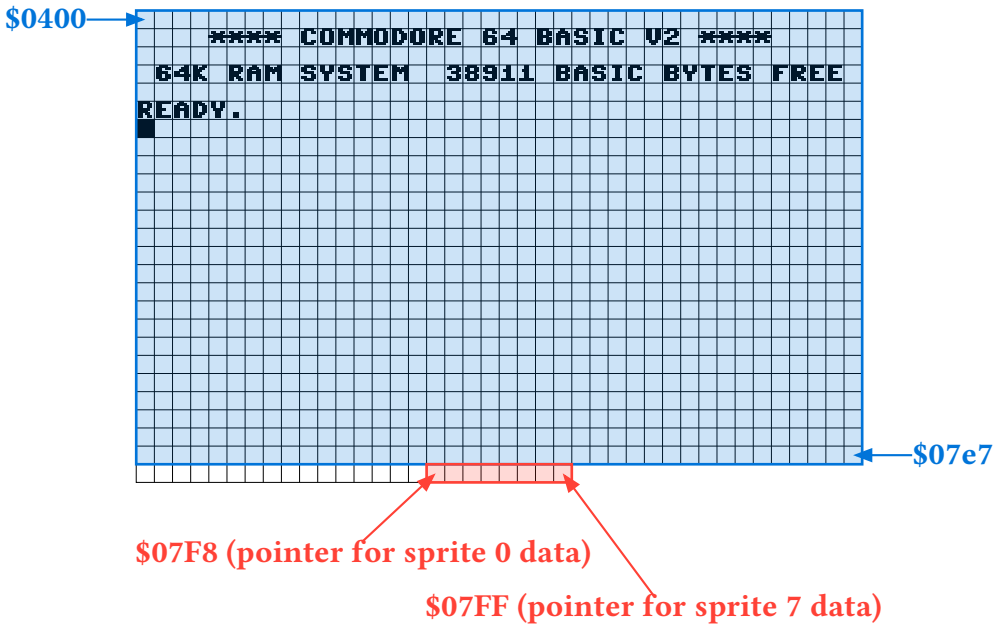


Figure 2: A chunk of 1024 bytes, 1000 of them (in blue) are for the chars, the last 8 of them (in red) are the 8 pointers for each sprite data

Good. Now what about "`lda #sprite_data/64`" to set the value of that pointer?

The key thing is that the sprite's data cannot be located just anywhere in memory.

A few rules

- Data has to be in the same bank as your screen / bitmap. For instance if you are using bank 0, the data has to be within the $0000 - $3FFF range.
- Data must start at an address that is a multiple of 64. So you cannot put your sprite data at, let's say, $0903... That's why we have `.align 64` before the label `sprite_data`. `.align 64` does some padding with 0 to ensure the next address will be aligned with 64 in this case.
- Data cannot be within the $1000-$1FFF range, and same for $9000-$9FFF. The reason is that the VIC-II chip sees the ROM font at those addresses.
- Unless you want to do something special, it is most probably not a good idea to put your sprite data at the same place as your SCREEN or BITMAP memory...
- And as a general rule, you should remember what we discussed about memory in Round 8, but we will come back to this later in this document when talking about what could go wrong.

> **Side note:** Putting your sprite data in the I/O range ($D000-$DFFF) is perfectly fine as long as you are using bank 3. The VIC-II chip always sees RAM when reading the ($D000-$DFFF) memory range.

How to interpret that pointer value?

Maybe you are confused about "`lda #sprite_data/64`", as this is an 8-bit value between 0 and 255... Shouldn't a pointer be 16 bits, as all memory addresses are 16 bits after all?

Using an 8-bit value works because of two things.
- Sprite data are restricted to the current BANK
- Sprite data are aligned on 64-byte blocks.

A value of 0 means your sprite data would start at address:
- $0000 if you are using bank 0
- $4000 if you are using bank 1
- $8000 if you are using bank 2
- $C000 if you are using bank 3

But let's check that...

```
BasicUpstart2(main)
main:

  lda #1 // 1 = bank 2
  sta $dd00

  lda #$14 // SCREEN at 1-> $400 (+ $8000) / font at 4-> $1000 (+$8000)
  sta $d018

  //fill the screen with 'a'
  lda #'a'
  jsr clear_screen

  // memory location for data
  lda #0 // first block of 64 bytes of the current bank
```

```
        // #1 would mean the data should start at $8000 + 64 so $8040
  sta $8000 + $0400 + $3F8

  // set the position on the screen
  lda #100
  sta $D000 // X position
  sta $D001 // Y position

  // decide which sprites to display
  lda #%0000_0001
  sta $D015

forever: jmp forever

clear_screen:
  ldx #0
  !:
    sta $8000+$400,x
    sta $8000+$500,x
    sta $8000+$600,x
    sta $8000+$6e8,x
    inx
    bne !-
  rts

.pc = $8000
sprite_data:
  // vertical lines
  .fill 63, %10101010 // same as $aa
  .byte 0
```

You should see your sprite on top of a screen filled with 'A'.

Summary

A bank is 16384 ($4000) bytes. Sprite data are aligned on 64 bytes.

16384 / 64 = 256 => it works, a single byte is enough for our pointer to point to any valid memory address for the data.

What is the data format for sprites?
Sprite data uses 24x21 pixels, 3 bytes per row × 21 rows = 63 bytes (and not 64). Let's update our code so that we display a border instead of vertical lines.

```
sprite_data:
  // a filled top line of 3 bytes (3x8 = 24)
  .byte $ff, $ff, $ff

  // 19 lines of 3 bytes, left border / empty / right border
  .for (var i= 0; i< 19; i++) {
    .byte %10000000, %00000000, %00000001
  }

  // a filled bottom line of 3 bytes
  .byte $ff, $ff, $ff

  .byte 0 // to pad to 64 so that the next sprite data can start right after.
```

Or even shorter, without the for loop, using .fill with a range [ ]

```
sprite_data:
  // a filled top line of 3 bytes (3x8 = 24)
  .byte $ff, $ff, $ff

  // 19 lines of 3 bytes, left border / empty / right border
  .fill 19, [%10000000, %00000000, %00000001]

  // a filled bottom line of 3 bytes
  .byte $ff, $ff, $ff

  .byte 0 // to pad to 64 so that the next sprite data can start right after.
```



Figure 3: Drawing borders using sprite data

> **Side note:** As you can see, this is not the same for sprites as for fonts or bitmaps. For sprites we fill the data line by line, while for fonts or bitmaps, it is char by char.

Sprite Colors

By default, sprites are monocolor. Each sprite has a corresponding register to change its color.

- $D027 for sprite 0
- $D028 for sprite 1
- $D029 for sprite 2
- $D02A for sprite 3
- $D02B for sprite 4
- $D02C for sprite 5
- $D02D for sprite 6
- $D02E for sprite 7

Each sprite can be configured individually to use multicolor, using register $D01C. Unlike $D000, which is dedicated to one sprite only (sprite 0), $D01C affects all the sprites at once. Each bit of that register corresponds to one of the sprites. For each bit, a value of 0 means monocolor, and a value of 1 means multicolor.

```
  // sprite index:    7 6 5 4 3 2 1 0
  lda               #%0_0_0_0_0_1_0_0 // turn sprite 2 to be multicolor.
  sta $D01C
```

Multicolor sprites can use up to 4 colors. The data uses pairs of bits to specify the color index. Horizontal resolution is then divided by two (still 3 bytes, but representing 12 “pixels” instead of 24 in monocolor).

```
lda #%0000_0001
sta $D01C // sprite 0 is multicolor

lda #GREEN
sta $D025 // shared color 0

lda #YELLOW
sta $D026 // shared color 1

lda #RED
sta $D027 // sprite 0 own color

...

.align 64
sprite_data:
  .fill 63, %00_01_10_11 // test color index 0, 1, 2 and 3
  .byte 0
```

- color 0 (00): Transparent
- color 1 (01): $D025 - shared color 0
- color 2 (10): $D027 - sprite own color
- color 3 (11): $D026 - shared color 1

Shared colors 0 and 1 are the same for all multicolor sprites.



Figure 4: A multicolor sprite

Changing the Y position

Basic motion

Each of the sprites uses its own register to set the Y position.

- sprite 0 is using $D001
- sprite 1 is using $D003

... and so on ...

- sprite 7 is using $D00F

Here is how you can move sprite 0 along the Y axis. This is mainly about `inc $D001`, nothing that special here.

```
BasicUpstart2(main)
main:
  lda #sprite_data/64
  sta $07F8

  // initial position
  lda #100
  sta $D000 // X position
  sta $D001 // Y position

  // display sprite 0
  lda #%0000_0001
  sta $d015

forever:
  // slow the animation down a bit
  // by waiting for the raster beam to
  // reach the top of the screen
  jsr wait

  jsr move_y
  jmp forever

move_y:
  inc $D001 // increment sprite 0 Y position
  rts

wait:
  bit $D011
  bpl *-3
  bit $D011
  bmi *-3
  rts

.align 64
sprite_data:
  // filled borders
  .fill 3, $ff
  .fill 19, [$80,$0,$1]
  .fill 3, $ff
  .byte 0
```

Good, now let's say you want this to be twice as fast, what do you do?

Probably this

```
move_y:
  inc $D001 // increment sprite 0 Y position
  inc $D001 // twice
  rts
```

or this

```
.const SPEED_Y = 2
...
move_y:
  lda $D001
  clc
  adc #SPEED_Y
  sta $D001

  rts
```

Nice... now let's say you want to move half as fast, what do you do?

You would probably just do `inc $D001` once every two frames, right? But how?

```
forever:
  jsr wait
  inc frame_ctr
  jsr move_y
  jmp forever

move_y:
  lda frame_ctr
  and #01      // once every two frames
  bne !+
  inc $D001
  !:
  rts

...

.pc = * "VARS"
frame_ctr: .byte 0
```

That and is a nice way of doing that, but it works only for once every 2, 4, 8, 16 … frames.

What if we want something in between?

```
move_y: {
  lda move_y_ctr
  cmp #5
  bcc _skip  // smaller than 5 -> just skip
  inc $D001
  lda #0     // reset counter
  sta move_y_ctr
_skip:
  inc move_y_ctr
  rts

  move_y_ctr: .byte 0
}
```

I renamed `frame_ctr` to `move_y_ctr` because I don't like the idea of resetting `frame_ctr`: it would not be a frame counter anymore. I also usually use a 16-bit value for `frame_ctr`.

The whole logic for the counter is also now encapsulated in the `move_y` routine.

I often use _ as a prefix for labels that are not exposed outside of the routine, but that's just a naming convention I decided to use, nothing mandatory.

Now, what if 2 is too fast and 3 is too slow? Could we do something a bit smoother using fractional numbers? And how do we represent them?

This is what we are going to explore in the next section.

Fractional values (8.8)

An "8.8" number consists of 2 bytes.

- one byte for the integer part (0-255)
- one byte for the fractional part, counted in 1/256ths (so between 0 and 0.996)

The value is simply: `integer + fraction / 256`.

8.8 value	bytes (hi.lo)	decimal
$0100	$01.$00	1.0
$0080	$00.$80	0.5
$0040	$00.$40	0.25
$00C0	$00.$C0	0.75
$0180	$01.$80	1.5

Another way to see it: an 8.8 number is just a 16-bit number counting in 1/256ths of a pixel. The VIC-II only cares about the high byte, which is the integer part. That's the one we copy to $D001. The low byte is invisible on the screen, but it accumulates until it overflows into the high byte.

So our Y position becomes a 16-bit variable, and our speed too.

```
.const SPEED_Y = round(0.75 * 256) // = $00C0

move_y: {
  clc
  lda pos_y_lo
  adc #<SPEED_Y   // add fractional parts
  sta pos_y_lo
  lda pos_y_hi
  adc #>SPEED_Y   // add integer parts + the carry from the fractional part
  sta pos_y_hi
  sta $D001       // only the integer part goes to the VIC-II
  rts

pos_y_lo: .byte 0
pos_y_hi: .byte 100
}
```

The magic is in the carry. When the fractional part goes over 255, the carry is set, and the second `adc` adds that extra 1 to the integer part.

Let's follow what happens with a speed of 0.75, starting at 100:

frame	pos_y (hi.lo)	$D001
0	$64.$00	100
1	$64.$C0	100
2	$65.$80	101
3	$66.$40	102
4	$67.$00	103

The sprite moved 3 pixels in 4 frames, so 0.75 pixel per frame, as expected.

Now let's try a speed that doesn't divide nicely: 0.3. `round(0.3 * 256)` gives 77, so $4D.

frame	pos_y (hi.lo)	$D001
0	$64.$00	100
1	$64.$4D	100
2	$64.$9A	100
3	$64.$E7	100
4	$65.$34	101
5	$65.$81	101
6	$65.$CE	101
7	$66.$1B	102
8	$66.$68	102
9	$66.$B5	102
10	$67.$02	103

The sprite moves after 4 frames, then 3, then 3… 3 pixels in 10 frames, so 0.3 pixel per frame. The steps are not all the same length, but the leftover in the low byte is never lost, so on average the speed is right. This is something our `cmp #5` counter could not do.

> **Side note:** 0.3 cannot be stored exactly in 8.8: $4D is 77/256 = 0.30078… The error is tiny, after 256 frames the sprite has moved 77 pixels instead of 76.8. For moving sprites, this is plenty precise.

And with the same routine, you can now use 1.5, 2.25, 0.1… just by changing `SPEED_Y`. No more counters, no more "every 5 frames".

If you look closely, this is the same idea as our `cmp #5` counter: something accumulates every frame, and the sprite moves one pixel when it overflows. The difference is that we now add the speed itself instead of 1, and we let the byte overflow at 256 instead of comparing it ourselves: the carry flag does the `cmp` for free, and nothing is lost when it wraps.

Going up?
To move up, we could write a second routine using `sbc`… but there is a nicer trick. A negative speed in 16 bits (two's complement) works with the very same `adc` code. For instance −0.5 is $10000 - $0080 = $FF80.

```
.const SPEED_Y = $10000 - round(0.5 * 256) // -0.5 = $FF80
```

Adding $FF80 is the same as subtracting $0080, because the result wraps around 16 bits.

> **Side note:** A very common trick on the Commodore 64 is to "open" the top and bottom borders. Managing the Y position properly when the top/bottom borders are open is a bit more tricky. For some of the values, your sprite will be shown both on top and bottom at the same time. While opening the top/bottom borders is very easy to achieve, we will still keep it for Part-III, which is all about C64 tricks.

That's almost it for the Y position. Let's just have a very quick look at multiplexing before we move on to the X axis.

Basic multiplexing

The Commodore 64 has "only" 8 sprites, but you can multiplex them on different lines so that you actually see more of them on the screen.

Multiplexing is a broad topic and it deserves its own article, but here is a very basic example of multiplexing sprite 0 so that it appears twice on the screen.



Figure 5: Basic multiplexing for sprite 0

```
.const POS_Y_0 = 60

// ! cannot overlap with the first one
.const POS_Y_1 = 100

BasicUpstart2(main)
main:
  lda #sprite_data/64
  sta $07F8

  lda #100
  sta $D000 // X position

  lda #%0000_0001
  sta $d015

  sei // disable interrupts to avoid glitches

forever:
  jsr wait

  lda #POS_Y_0
  sta $D001 // sprite 0 Y position

  lda #POS_Y_0 + 1 // wait for the raster beam to be
  cmp $D012        // at least at POS_Y_0 + 1
  bne *-3          // before setting the second Y position

  lda #POS_Y_1
  sta $D001 // another Y position for sprite 0

  jmp forever

wait:
  bit $D011
  bpl *-3
  bit $D011
  bmi *-3
  rts

.align 64
sprite_data:
  .fill 3, $ff
  .fill 19, [$80,$0,$1]
  .fill 3, $ff
  .byte 0
```

Why does it work? The VIC-II only looks at the Y register to decide when to **start** drawing a sprite. Once it has started, it draws all 21 lines, whatever we write to $D001 in the meantime. That's also why the second position cannot overlap with the first one: it must be at least 21 lines below it.

Changing the X position

This is very similar to the Y position, but setting the X position involves 9 registers this time. Eight of them are specific to a given sprite, and one is shared among all of them ($D010).

The main difference is that one byte is no longer enough to represent all positions along the X-axis. We have some positions - like 300 for instance - that are bigger than the maximum value of a single byte (255).

To deal with this, the shared register $D010 acts as a ninth bit for each of the sprites.

$D010 works the same way as $D01C and $D015 that we saw already.

```
  // sprite index:    7 6 5 4 3 2 1 0
  lda              #%0_0_1_0_0_1_0_0 // decide which sprites have
                                     // their x position > 255 (sprite 2 and 5)
  sta $D010
```

To set the exact position of sprite 0, you should set $D000 and bit 0 of $D010.

To set the exact position of sprite 7, you should set $D00E and bit 7 of $D010.

With this in mind, let's make sprite 3 cross the whole screen on the x-axis.

Before writing code, let's describe the logic.

- draw sprite 3 on the left side, hidden behind the left border.
- increment x position
- if position > 255 set bit 3 of $D010
- reset the values when sprite 3 reaches its final position (fully hidden behind the right side border)

That sounds like a plan... let's start with the first step "draw sprite 3 on the left side, hidden behind the left border."... What value for $D006 (sprite 3 X register) should we use? 0? Let's try...

```
.const SPRITE_INDEX = 3 // 0 to 7
BasicUpstart2(main)
main:
  lda #sprite_data/64
  sta $07F8 + SPRITE_INDEX
  lda #0
  sta $D000 + 2*SPRITE_INDEX // X position
  lda #100
  sta $D001 + 2*SPRITE_INDEX // Y position
  lda #1 << SPRITE_INDEX
  sta $d015

forever:
  jsr wait
  jsr move_x
  jmp forever

move_x:
  inc $D000 + 2*SPRITE_INDEX
  rts

wait:
  bit $D011
  bpl *-3
  bit $D011
  bmi *-3
  rts

.align 64
sprite_data:
  .fill 3, $ff
  .fill 19, [$80,$0,$1]
  .fill 3, $ff
  .byte 0
```

Ok, seems to work so far... the sprite starts at the proper position and moves in the right direction. Let's tackle that $D010 register now.

```
move_x: {
  inc $D000 + 2*SPRITE_INDEX
  bne _skip // not 0 -> no wrap from 255 to 0, so we skip setting $D010
  lda #%0000_1000 // bit 3
  sta $D010
_skip:
  rts
}
```

At first, it looks like we are done... **but no, it's not fully working**...

There are at least two issues.

- $D006 and $D010 are not reset when reaching the right side.
- `lda #%0000_1000 // bit 3` works only for sprite 3. What if we change the value of SPRITE_INDEX? And what if we had other sprites somewhere on the screen? We cannot do it simply that way.

Let's fix both.

```
.const MASK = 1 << SPRITE_INDEX
.const RIGHT_EDGE = 344 // first X where the sprite is fully out of the screen (on the right)

move_x: {
  inc $D000 + 2*SPRITE_INDEX
  bne _check_right
  lda $D010           // 255 -> 0: set the 9th bit
  ora #MASK
  sta $D010
  rts

_check_right:
  lda $D010
  and #MASK
  beq _done           // still below 256
  lda $D000 + 2*SPRITE_INDEX
  cmp #<RIGHT_EDGE    // 344 - 256 = 88
  bne _done
  lda #0              // reset X to 0 ...
  sta $D000 + 2*SPRITE_INDEX
  lda $D010           // ... and clear the 9th bit
  and #$ff ^ MASK
  sta $D010
_done:
  rts
}
```

"1 << SPRITE_INDEX" is a bit operation that shifts the value 1 to the left, SPRITE_INDEX times.

SPRITE_INDEX	1 << SPRITE_INDEX
0	%00000001
1	%00000010
2	%00000100
3	%00001000
4	%00010000
5	%00100000
6	%01000000
7	%10000000

Expanded size

Each sprite can be expanded in X and/or in Y. Expanding a sprite doubles its size in X and/or Y. The resolution of the sprite doesn't change however. (A monocolor sprite expanded in X and Y is still 24x21 pixels, but appears twice as big on the screen).

- Expand in X : $D01D
- Expand in Y : $D017

There are quite a few tricks related to $D017, but again that's for Part-III.

Let's expand sprite 3 in X

```
lda #%0000_1000 // expand sprite 3
sta $D01D
```

And now let's try to hide this expanded sprite 3 behind the left side border like we did before.

```
lda #0
sta $D006 // position X for sprite 3
sta $D010 // no sprite with X position > 255.
```



Figure 6: Try 1 at hiding an expanded sprite in X on the left side

Ok not really... still half of the sprite is visible... As the sprite is doubled in X, we should move it 24 px more on the left... but we are already using 0... so that should be... −24?

```
lda #0 - 24 // let's try...
sta $D006 // position X for sprite 3
sta $D010 // no sprite with X position > 255.
```



Figure 7: Try 2 at hiding an expanded sprite in X on the left side

Uh... not any better...

Oh... we should probably set the corresponding bit in $D010, right?

```
lda #0 - 24
sta $D006 // position X for sprite 3
lda #%0000_1000
sta $D010
```



Figure 8: Try 3 at hiding an expanded sprite in X on the left side

Still not... the proper value to use is 224...

```
lda #224
sta $D006 // position X for sprite 3
lda #%0000_1000
sta $D010
```

This one works. The sprite is fully hidden on the left.



Figure 9: Try 4 at hiding an expanded sprite in X on the left side, this one is working fine.

But how to be sure? Let's move it one px right to display only the right border of the sprite using value 224 + 1.

```
lda #224 + 1
sta $D006 // position X for sprite 3
lda #%0000_1000
sta $D010
```



Figure 10: Try 5 showing only the right border of the expanded sprite.

Where does 224 come from?

The X axis is not a straight line going from 0 to 511, it's a circle. On a PAL machine, the VIC-II uses X positions from 0 to 503, and the position right after 503 is… 0 again. So moving left from 0 brings you to 503, then 502, and so on.

An expanded sprite is 48 pixels wide. To be fully hidden on the left, its last pixel must be just before X = 24, the first visible column. So it must start 48 pixels earlier. Going back 24 pixels from 24 brings us to 0, and going back 24 more wraps around to 480 (503, 502, … 480).

X position	bit 3 of $D010	$D006 (low byte)
480 = $1E0	1	$E0 = 224

So 224 is not magic at all. It is the low byte of 480, the position 48 pixels before X = 24.

For a normal sprite (24 pixels wide), going back 24 pixels from 24 stops at 0, which is why X = 0 worked perfectly before we expanded it.

Now our failed tries make sense:

Try	$D006 / $D010 bit	real X	what we see
1	0 / 0	0	The sprite covers 0 to 47. Pixels 24 to 47 are visible: half of the sprite.
2	`0 - 24` = 232 / 0	232	In a byte, 0 − 24 is 232, not −24. The sprite is simply drawn at 232, on the right.
3	232 / 1	488	488 is 8 pixels after 480, so 8 pixels of the sprite reach the visible area.
4	224 / 1	480	The sprite covers 480 … 503 and 0 … 23: fully hidden.
5	225 / 1	481	Only the last column reaches X = 24: the right border of the sprite.

Instead of remembering 224, we can let the assembler do the math:

```
.const SPRITE_INDEX = 3 // 0 to 7
.const MASK = 1 << SPRITE_INDEX

.const X_WRAP = 504            // PAL: X goes from 0 to 503
.const HIDDEN_X = X_WRAP - 24  // 48 px before 24, wrapping around: 480

  lda #<HIDDEN_X               // 224
  sta $D000 + 2*SPRITE_INDEX
  lda $D010
  ora #MASK                    // 480 > 255 -> set the 9th bit
  sta $D010
```

> **Side note:** The value 504 is for PAL machines. NTSC machines have a different number of X positions, so the value to hide a sprite on the left is different. Always test on the system you target.

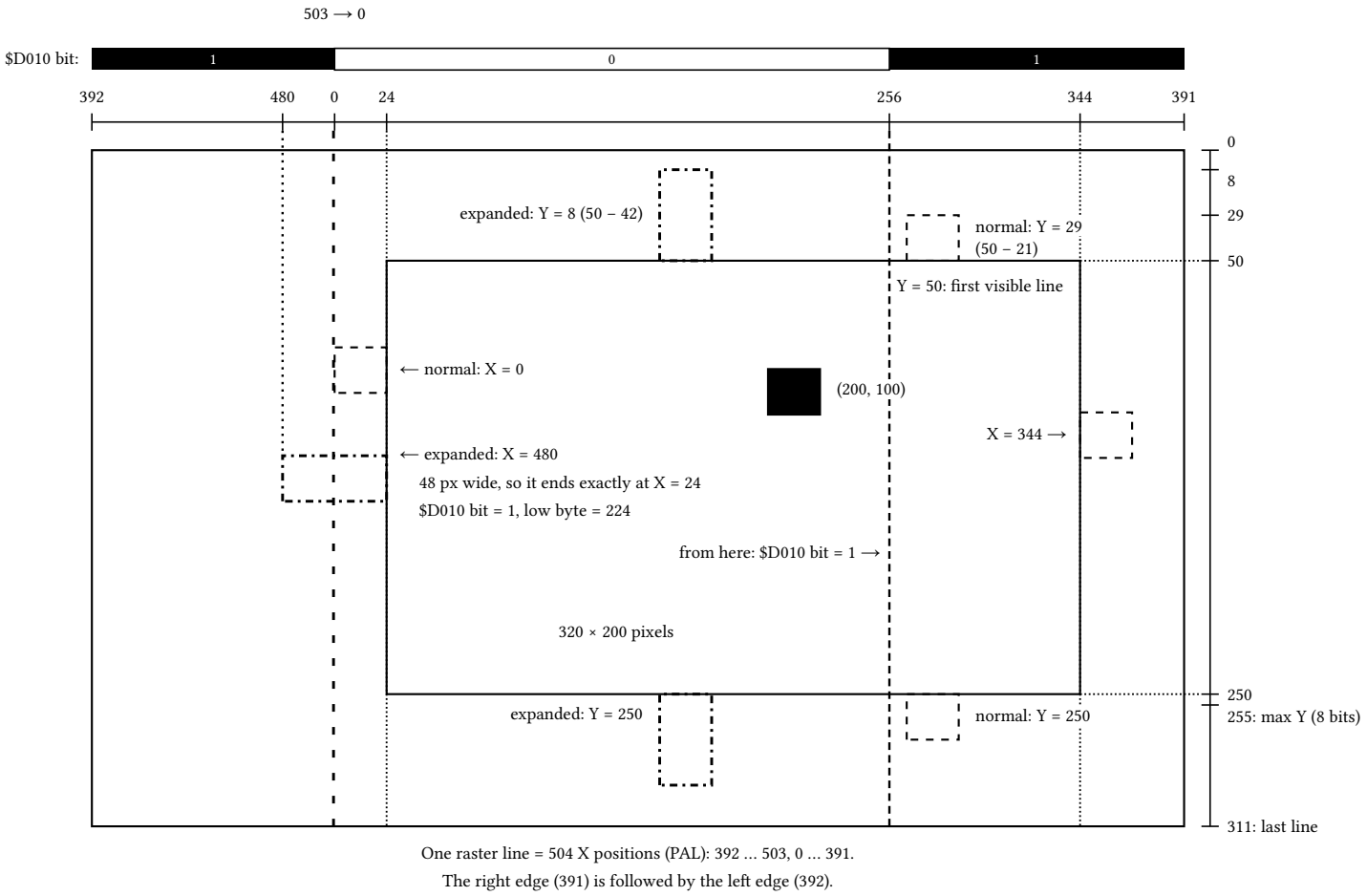


Figure 11: Important sprite X and Y values (PAL, VICE debug borders 504×312). Dashed sprites are fully hidden behind the border.

Priority

Over sprites themselves

When two sprites overlap, the one with the lowest number is drawn on top. Sprite 0 is always in front, sprite 7 is always behind. There is no register to change that.

```
BasicUpstart2(main)
main:
  lda #sprite_data/64
  sta $07F8           // sprite 0
  sta $07F9           // sprite 1, same data

  lda #100
  sta $D000           // sprite 0 X
  sta $D001           // sprite 0 Y
  lda #110
  sta $D002           // sprite 1 X
  sta $D003           // sprite 1 Y

  lda #WHITE
  sta $D027           // sprite 0 color
  lda #RED
  sta $D028           // sprite 1 color

  lda #%0000_0011     // sprites 0 and 1
  sta $D015

forever: jmp forever

.align 64
sprite_data:
  .fill 63, $ff       // a filled square
  .byte 0
```

The white sprite 0 is drawn over the red sprite 1, even though sprite 1 was set up last.

Only the pixels that are set in a sprite hide the sprites behind it. Transparent pixels (0 in monocolor, `%00` in multicolor) let the sprites behind show through.

Over the background

By default, sprites are drawn in front of the text or bitmap. Register $D01B changes that, one bit per sprite, like $D015 or $D01C:

- 0: the sprite is in front of the background (default)
- 1: the sprite is behind the background

```
  // sprite index:    7 6 5 4 3 2 1 0
  lda               #%0_0_0_0_0_0_0_1  // sprite 0 behind the text
  sta $D01B
```

Move your sprite over the “READY.” text with this bit set, and the letters appear on top of the sprite.

But “behind the background” does not mean behind **everything**. The VIC-II splits the screen pixels into two groups:

- background pixels: always behind the sprites, whatever $D01B says

- foreground pixels: in front of the sprite when its $D01B bit is 1

mode	background pixels	foreground pixels
hires text / bitmap	bit = 0	bit = 1
multicolor text / bitmap	`%00` and **`%01`**	`%10` and `%11`

Watch out for the multicolor case: `%01` counts as background. A sprite behind the background will still be drawn **over** the `%01` pixels. If you design multicolor graphics that a sprite should go behind (trees, columns, a foreground wall…), don't use `%01` for the parts that should cover it.

And the border? It is always in front of the sprites, which is exactly what we used to hide them. As already mentioned, the borders can still be "opened", but that's something for later.

Side note: When sprites with different $D01B bits overlap, the VIC-II first decides which sprite wins (lowest number), and only then checks the background. So if sprite 0 is behind the background and sprite 1 is in front, where they overlap on foreground pixels, the background hides sprite 0 **and** the part of sprite 1 under it. This looks like a bug, but it is a well-known trick to mask parts of a sprite.

```
BasicUpstart2(main)
main:

 // disable interrupts to prevent cursor blinking over our block
  sei

  // a block of reverse spaces ($A0: all pixels set = foreground)
  // on rows 4 to 12, columns 0 to 10
  lda #$a0
  ldx #10
  !:
  .for (var row = 4; row < 13; row++) {
    sta $0400 + row*40, x
  }
  dex
  bpl !-

  lda #sprite_data/64
  sta $07F8           // sprite 0
  sta $07F9           // sprite 1

  lda #104
  sta $D000           // sprite 0 X
  lda #110
  sta $D001           // sprite 0 Y
  lda #100
  sta $D002           // sprite 1 X
  sta $D003           // sprite 1 Y

  lda #WHITE
  sta $D027           // sprite 0: white
  lda #RED
  sta $D028           // sprite 1: red

  lda #%0000_0001     // sprite 0 behind the background
  sta $D01B           // sprite 1 in front of it

  lda #%0000_0011
  sta $D015

forever:
    jmp forever

.align 64
sprite_data:
  .fill 63, $ff
  .byte 0
```

What you should see:

- the red sprite 1 is in front of the block, as expected,
- the white sprite 0 is hidden by the block, and only visible on its right part,
- where they overlap **on the block**, there is a hole in the red sprite: the block shows through, even though sprite 1 is in front of the background.

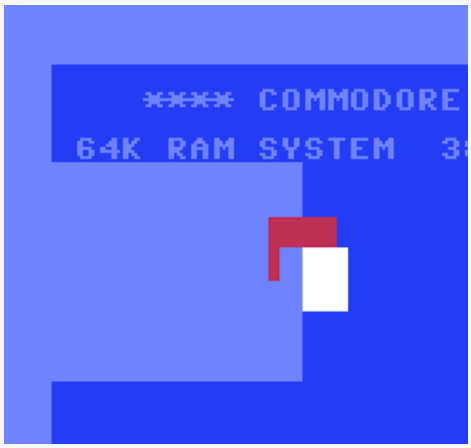


Figure 12: Sprite 0 wins against sprite 1, then loses against the background... and takes the overlapping part of sprite 1 with it.

Collision detection

The VIC-II can tell you when sprites touch something. There are two registers, one bit per sprite, like $D015:

- $D01E: sprite-sprite collisions
- $D01F: sprite-background collisions

The detection is pixel perfect: a collision only happens where pixels that are **set** overlap. Transparent pixels (0 in monocolor, `%00` in multicolor) never collide.

Sprite - sprite

Let's move sprite 0 to the right, until it hits sprite 1, and change the border color when it happens.

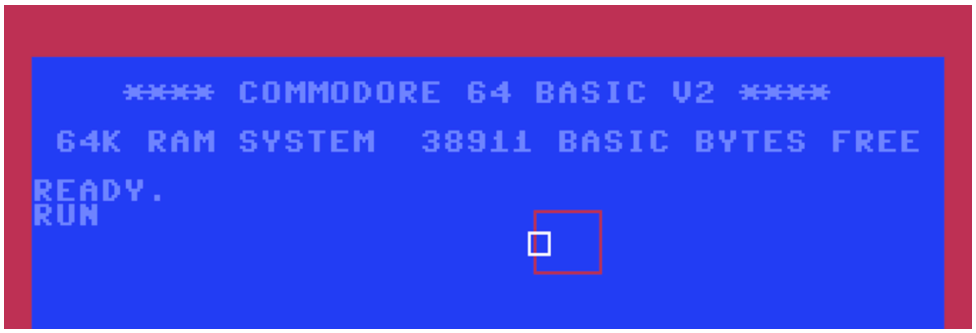


Figure 13: A collision between two sprites

```
BasicUpstart2(main)
main:
  lda #sprite0/64
  sta $07F8
  lda #sprite1/64
  sta $07F9

  lda #0
  sta $D000           // sprite 0 X, moving
  lda #200
  sta $D002           // sprite 1 X, not moving
  lda #100
  sta $D001
  sta $D003           // same Y for both

  lda #%0000_0011
  sta $D015

forever:
  jsr wait
  inc $D000           // move sprite 0

  lda $D01E           // read the collisions... and clear them
  and #%0000_0001     // is sprite 0 involved?
  beq !+
  lda #RED            // yes: red border
  .byte $2c           // a trick to skip the next instruction
                      // $2c -> opcode for bit
!:
  lda #LIGHT_BLUE     // no: default border
  sta $D020
  jmp forever

wait:
  bit $D011
  bpl *-3
  bit $D011
  bmi *-3
  rts

.align 64
.pc = * "sprite_data"
sprite0: // a small square
  .fill 7, [0,0,0]
  .fill 1, [0,$ff,0]
  .fill 6, [$00,%1000_0001,$00]
  .fill 1, [0,$ff,0]
  .fill 6, [0,0,0]
  .byte 0
sprite1: // a bigger rectangle
  .fill 3, $ff
  .fill 19, [$80,$0,$1]
  .fill 3, $ff
  .byte 0
```

> **Side note:** `.byte $2c` is the opcode of `bit` absolute. It "eats" the two bytes of the next `lda #LIGHT_BLUE`, so that instruction is skipped. That's a trick to save a `jmp`.

The border turns red when the two squares touch… and it goes back to light blue when sprite 0 is fully **inside** sprite 1, even though they still overlap! Both sprites are outlines: only the edges are set, the inside is transparent. As long as the outlines don't cross, there is no collision.



Figure 14: No collision here, even if the sprites are overlapping.

A few things to know about $D01E:

- **Reading it clears it.** Once a collision happened, the bits stay set until you read the register. So read it once per frame, store the value, and work with your copy.
- **It does not say who hit whom.** If bits 0, 1 and 2 are set, it could be 0 with 1 and 2, or 0 with 1 while 2 hits… well, someone. If you need to know, you still have to compare the positions.
- **It is updated while the frame is drawn.** The collision is detected when the raster beam draws the overlapping pixels. Reading after the frame (like our `wait` does) gives you the whole frame.

Sprite - background

$D01F works the same way, but for collisions between a sprite and the foreground pixels of the text or bitmap. Remember the table from the Priority section: background pixels never collide, and in multicolor mode that includes `%01`.

```
lda $D01F       // read the collisions with the background... and clear them
and #%0000_0001 // sprite 0 touching some text?
```

Move a sprite over the "READY." text and you will see it trigger on the letters, not on the spaces between them.

A collision with the background tells you nothing about **what** was hit. To know if it's a wall, a coin or a ladder, you still have to compute which char is under the sprite, from its X and Y position. And that's exactly the values we put in the figure earlier: X − 24 and Y − 50 give you the position in pixels inside the screen, divide by 8 and you get the column and the row.

> **Side note:** Collisions can also trigger an interrupt, using $D019 and $D01A. We will see that when talking about interrupts in more detail.

// When things go wrong

Here are a few code snippets where the sprites are corrupted or not showing. Can you fix them?

Problem 1

Sprite data are corrupted

```
BasicUpstart2(main)
main:
  lda #data/64
  sta $7F8
  lda #100
  sta $D000
  sta $D001
  lda #$01
  sta $D015
  forever: jmp forever
data:
  .fill 63, $ff // filled sprite
  .byte 0
```

Problem 2

Sprite data are corrupted

```
BasicUpstart2(main)
main:
  lda #data/64
  sta $7F8
  lda #100
  sta $D000
  sta $D001
  lda #$01
  sta $D015

  jsr clear_screen
  forever: jmp forever

clear_screen:
  lda #' '
  ldx #0
  !:
    sta $0400,x
    sta $0400 + $100,x
    sta $0400 + $200,x
    sta $0400 + $300,x
    inx
    bne !-

  rts

.align 64
data:
  .fill 63, $ff // filled sprite
  .byte 0
```

Problem 3

Sprite data are corrupted

```
BasicUpstart2(main)

main:
  lda #data/64
  sta $7F8
  lda #100
  sta $D000
  sta $D001
  lda #$01
  sta $D015
  jsr clear_screen
  forever: jmp forever

clear_screen:
  lda #' '
  ldx #0
  !:
    sta $0400,x
    sta $0400 + $100,x
    sta $0400 + $200,x
    sta $0400 + $2E8,x
    inx
    bne !-
  rts

// Some data here
.pc = * "DATA"
scroll_text:
.fill 2048, 0 // just using 0 to keep the sample short.

.align 64
data:
  .fill 63, $ff // filled sprite
  .byte 0
```

Problem 4

Sprite data are corrupted

```
BasicUpstart2(main)
main:

  lda #$34 // screen at $0C00, font at $1000
  sta $D018

  jsr init_sprite
  jsr clear_screen

  forever: jmp forever

clear_screen:
  lda #' '
  ldx #0
  !:
    sta $0C00,x
    sta $0C00 + $100,x
    sta $0C00 + $200,x
    sta $0C00 + $2E8,x
    inx
    bne !-

  rts

init_sprite:
  lda #data/64
  sta $7F8
  lda #$01
  sta $D015
  lda #100
  sta $D000
  sta $D001
  rts

.align 64
data:
  .fill 63, $ff // filled sprite
  .byte 0
```

Problem 5

Sprite is not showing at all

```
BasicUpstart2(main)
main:

  lda #$34 // screen at $0C00, font at $1000
  sta $D018

  jsr init_sprite
  jsr clear_screen

  forever: jmp forever

clear_screen:
  lda #' '
  ldx #0
  !:
    sta $0C00,x
    sta $0C00 + $100,x
    sta $0C00 + $200,x
    sta $0C00 + $2E8,x
    inx
    bne !-
  rts

init_sprite:
  lda #data/64
  sta $C00 +$3F8
  lda #$01
  sta $D015
  lda x_pos
  sta $D000
  lda y_pos
  sta $D001
  rts

x_pos: .byte 0
y_pos: .byte 0

.align 64
data:
  .fill 63, $ff // filled sprite
  .byte 0
```

Problem 6

Sprite data are corrupted

```
.const BANK = $4000
.const FONT = BANK
.const SCREEN = $2000

BasicUpstart2(main)
main:

  lda #2
  sta $dd00 // using bank 1

  lda #$80  // screen at $2000 (+BANK), font at BANK
  sta $D018

  jsr init_sprite
  jsr clear_screen

  forever: jmp forever

clear_screen:
  lda #' '
  ldx #0
  !:
    sta BANK+ SCREEN,x
    sta BANK+ SCREEN + $100,x
    sta BANK+ SCREEN + $200,x
    sta BANK+ SCREEN + $2E8,x
    inx
    bne !-
  rts

init_sprite:
  lda #data/64
  sta BANK+ SCREEN +$3F8
  lda #$01
  sta $D015
  lda x_pos
  sta $D000
  lda y_pos
  sta $D001
  rts

x_pos: .byte 100
y_pos: .byte 100

.align 64
data:
  .fill 63, $ff // filled sprite
  .byte 0

.pc = FONT
font:
  .fill 8*256, 0 // just an empty font
```

Page left blank intentionally

Answers

Answer to problem 1

Sprite data are located at an address that is not a multiple of 64.

This is easily fixed by using `.align 64`

```
.align 64
data:
  .fill 63, $ff // filled sprite
  .byte 0
```

Answer to problem 2

Sprite pointers are cleared by the clear_screen routine

```
sta $0400 + $300,x   should be    sta $0400 + $2E8,x
```

Answer to problem 3

Sprite data fall into $1000 - $1FFF range where the VIC-II chip sees the ROM font. The fix is to move your data somewhere else.

Answer to problem 4

```
sta $7F8
```

is using the wrong address as we are now using $0C00 for the screen. The fix is to use the proper address for sprite pointer.

```
 sta $0C00 + $3F8
```

Answer to problem 5

the problem is the initial values for x and y.

```
x_pos: .byte 0
y_pos: .byte 0
```

The sprite is drawn at a position that is not visible on the screen.

Answer to problem 6

The fix is to put the data in bank 1, for instance by putting them after the font like this:

```
.pc = FONT
font:
  .fill 8*256, 0 // just an empty font

.align 64
data:
  .fill 63, $ff // filled sprite
  .byte 0
```

// 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

// Comments?

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

// What's coming next

Text scroller will be back!

We will come back to the text scroller we put on hold at the end of Part I - round 7, And also do a scroller using sprites.