Round 9 • The 5 graphics modes

OCA / Atlantis

// Introduction

In this article we are going to explore all 5 graphics modes of the Commodore 64. So far we only used one of them: the "hi-res" text mode.

// The 5 graphics modes

The 5 official graphics modes are:

Mode	Variant	Resolution	Colors
Text	HiRes	320×200 40 columns × 25 rows of 8×8 chars	1 global background color + 1 color per char
Text	Multicolor	160×200 40 columns × 25 rows of 8×8 chars made of 4×8 double-width pixels	3 global colors + 1 color per char (only chars whose color has bit 3 set are multicolor)
Text	Extended	320×200 40 columns × 25 rows of 8×8 chars	4 background colors, chosen per char; only 64 different chars
Bitmap	HiRes	320×200	2 colors per 8×8 cell
Bitmap	Multicolor	160×200 (double-width pixels)	1 global background color + 3 colors per 8×8 cell

To configure a mode, you have to choose between:
- Text or Bitmap, using bit 5 of register $D011
- HiRes or Multicolor, using bit 4 of register $D016
- When in HiRes text mode, Extended mode can be turned on using bit 6 of $D011.

Those 3 bits give 8 combinations, but only 5 of them are valid. Extended mode can't be combined with Bitmap or Multicolor: the other 3 combinations just give you a black screen.

> **Side note:** Only 5 graphics modes? What about FLI, NUFLI, Hyperscreen and the like?
>
> These are advanced tricks that make the VIC-II display things it normally couldn't: more colors per cell, graphics in the borders, and so on. We will cover a few of them in Part III.

// Modes and associated color registers

		$D021 background	**$D022** **$D023**	**$D024**	**$D800** color RAM	**Screen RAM** Chosen using $D018 / $DD00	**Bitmap RAM** Chosen using $D018 / $DD00
Text	**HiRes**	background	—	—	foreground	char index	—
	Multi-color	`%00` (color index 0)	`%01` ⟶ $D022 `%10` ⟶ $D023, (color indexes 1 and 2)	—	`%11` (color index 3) color + 8 (0–7 only, you cannot use "light red" = 10 here for instance.)	char index	—
	Extended	`%00` background	`%01` ⟶ $D022 `%10` ⟶ $D023	`%11` background	foreground	bits 7–6: background bits 5–0: char (64 chars only)	—
Bitmap	**HiRes**	—	—	—	—	1st hex digit: color for when bitmap bit = 1 2nd hex digit: color for when bitmap bit = 0	1 bit per pixel
	Multi-color	`%00`	—	—	`%11`	1st hex digit: color for when bitmap bits = `%01` 2nd hex digit: color for when bitmap bits = `%10`	2 bits per pixel

In the multicolor modes, pixels are twice as wide, and each one is made of 2 bits of the font (text) or of the bitmap. The value of those 2 bits (`%00`, `%01`, `%10` or `%11`) picks where the pixel's color comes from. In extended mode, the same values come from bits 7–6 of the char code in screen RAM, and pick the char's background color.

Example, hi-res bitmap: the screen RAM byte of an 8×8 cell is $21.

`%0010_0001`

2 = **RED**	1 = **WHITE**
color for when	color for when
bitmap bit = 1	bitmap bit = 0

So in that 8×8 cell, every bit set to 1 in the bitmap is drawn in red, and every bit set to 0 is drawn in white. The next cell can use two completely different colors.

// Changing individual bits of a register

Setting a graphics mode means setting or clearing individual bits of $D011 and $D016, something we haven't covered in detail yet. Let's see how.

$D011

bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0
bit 8 of the raster line (bits 0-7 are in $d012)	1 for Extended / 0 if not	1 for Bitmap / 0 for Text	DEN 1 = screen visible 0 = screen blanked (border color everywhere)	RSEL 1 for 25 rows 0 for 24 rows	YSCROLL vertical offset from 0 to 7 default is 3		
default value							
0	0	0	1	1	0	1	1

> **Side note: Bit 7's default value.** At power-on, the KERNAL writes $1B (%0001_1011) to $D011: text mode, screen visible, 25 rows, YSCROLL = 3, and bit 7 at 0.
>
> When you read $D011, however, bit 7 shows bit 8 of the raster line being drawn: 0 for lines 0–255, 1 from line 256 on. That's why `lda $d011` sometimes returns $1B and sometimes $9B.
>
> **Writing bit 7** only matters for raster interrupts, where it becomes bit 8 of the interrupt line (bits 0–7 are in $D012). We don't use interrupts yet, so you can ignore it for now.

$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

To change only some bits of a register, read it, change the bits, and write it back:

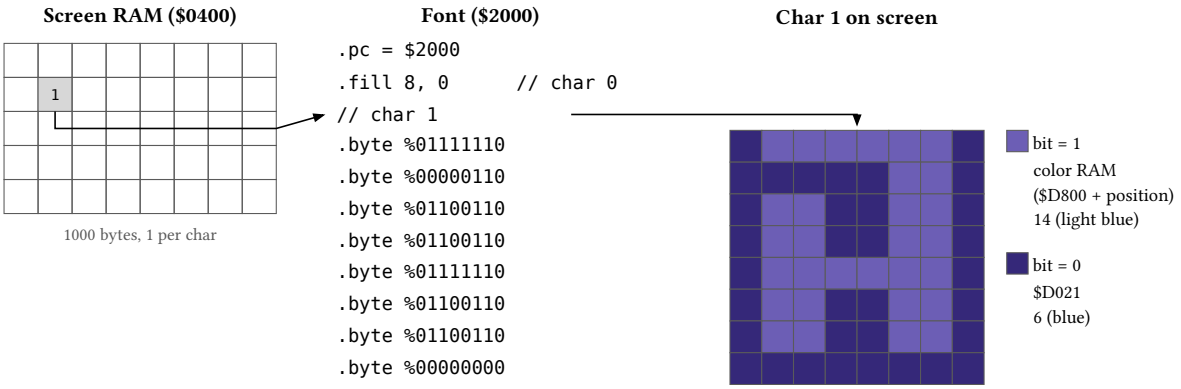
- the *and* instruction **clears** bits: every bit set to 0 in the value becomes 0, the others don't change. `and #%1110_1111` clears bit 4.
- the *ora* instruction **sets** bits: every bit set to 1 in the value becomes 1, the others don't change. `ora #%0001_0000` sets bit 4.

> **Side note:**
>
> There is no single instruction that sets some bits and clears others while leaving the rest untouched. When you need both, use `and` and `ora` one after the other, in either order.

```
lda $D011        // read value
ora #%0010_0000  // set bit 5
and #%1011_1111  // clear bit 6
sta $D011
```

// HiRes Text Mode

This is the “default” mode your C64 starts with when you power it on. That’s also the one we used all along so far.



$D011: bit 6 = 0, bit 5 = 0 **$D016**: bit 4 = 0 **$D018** = $18: screen at $0400, font at $2000.
Each screen RAM byte is a char index (0–255). The char’s 8 bytes are 8 rows of 8 pixels: each bit set to 1 is drawn in the char’s color, each bit set to 0 in the background color.

Figure 1: HiRes text mode

Listing for 01-text-hires.asm

```
BasicUpstart2(main)
main:
  lda $D011        // read current value
  and #%1001_1111  // clear bit 5 for Text,
                   // clear bit 6 to make sure Extended mode is off
  sta $D011

  lda $D016
  and #%1110_1111  // clear bit 4 for HiRes mode
  sta $D016

  lda #$18        // set the font to be $2000
  sta $D018

  // clear screen
  ldx #0
  lda #' '
  !:
  sta $0400,x
  sta $0500,x
  sta $0600,x
  sta $06e8,x
  inx
  bne !-

  lda #LIGHT_BLUE // just to be explicit, as this is already
                  // the default color
  sta $D800 + 41

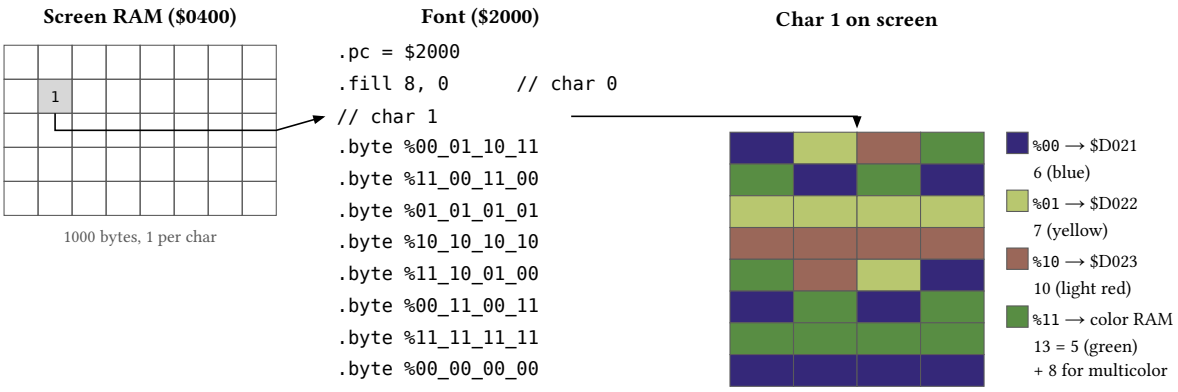
  // Put the A char on the screen
  lda #1
```

```
  sta $0400 + 41


forever:
  jmp forever

.pc = $2000 "Custom font"
.fill 8,0 // char 0 -> @
.byte %01111110 // char 1 -> a custom A
.byte %00000110
.byte %01100110
.byte %01100110
.byte %01111110
.byte %01100110
.byte %01100110
.byte %00000000
.fill (256-2)*8 , 0 // blank the rest of the letters
```

// Multicolor Text Mode



$D011: bit 6 = 0, bit 5 = 0 **$D016**: bit 4 = 1 **$D018** = $18.
Each pair of bits is one double-width pixel, so a char is 4×8 pixels. Pixels are only twice as wide, not twice as tall: each of the 8 bytes is still one row of the char. In the `.byte` lines, the _ between the pairs is only there for easy reading. $D021, $D022 and $D023 are shared by all the chars; the color RAM gives each char its own 4th color. Only chars whose color RAM value has bit 3 set (+ 8) are drawn in multicolor, so the 4th color can only be 0–7.

Figure 2: Multicolor text mode

Listing for 02-text-multicolor.asm

```
BasicUpstart2(main)
main:
  lda $D011        // read current value
  and #%1001_1111  // clear bit 5 for Text,
                   // clear bit 6 to make sure Extended mode is off,
  sta $D011

  lda $D016
  ora #%0001_0000  // set bit 4 for multicolor mode
  sta $D016

  // set the font to be $2000
```

```
  lda #$18   // check Round 8 if you are not
             // sure anymore why this is $18
  sta $D018

  // clear screen
  ldx #0
  lda #' '
  !:
  sta $0400,x
  sta $0500,x
  sta $0600,x
  sta $06e8,x
  inx
  bne !-

  // set the colors to use
  lda #BLUE
  sta $D021
  lda #YELLOW
  sta $D022
  lda #LIGHT_RED
  sta $D023

  lda #GREEN | 8 // | 8 sets bit 3. This is mandatory
                 // for the COLORAM in multicolor mode.
                 // That also means you can only use colors from 0 to 7
  sta $D800 + 41

  // put char 1 on the screen
  lda #1
  sta $0400 + 41

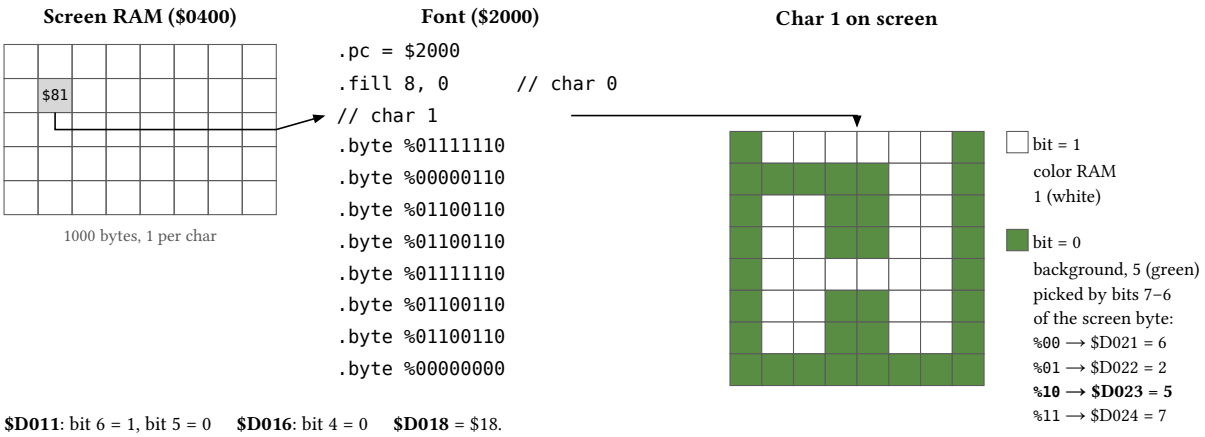
forever:
  jmp forever

.pc = $2000 "Custom font"
.fill 8,0 // char 0 -> @

// char 1 - illustrating the 4 colors
// using different combinations.
.byte %00_01_10_11
.byte %11_00_11_00
.byte %01_01_01_01
.byte %10_10_10_10
.byte %11_10_01_00
.byte %00_11_00_11
.byte %11_11_11_11
.byte %00_00_00_00

.fill (256-2)*8 , 0 // blank the rest of the letters
```

// Extended Text Mode



$D011: bit 6 = 1, bit 5 = 0 **$D016**: bit 4 = 0 **$D018** = $18.
The screen RAM byte $81 is `%10_000001`: bits 7–6 (`%10`) pick the background $D023, and bits 5–0 pick char 1. Only 6 bits are left for the char index, so only chars 0–63 can be used.

Figure 3: Extended text mode

Listing for 03-text-extended.asm

```
BasicUpstart2(main)
main:
  lda $D011        // read current value
  and #%1001_1111  // clear bit 5 for Text,
  ora #%0100_0000  // set bit 6 to enable Extended mode
  sta $D011

  lda $D016
  and #%1110_1111  // clear bit 4 for HiRes mode
  sta $D016

  lda #$18        // set the font to be $2000
  sta $D018

  // clear screen
  ldx #0
  lda #' '
  !:
  sta $0400,x
  sta $0500,x
  sta $0600,x
  sta $06e8,x
  inx
  bne !-

  // colors
  lda #BLUE
  sta $D021
  lda #RED
  sta $D022
  lda #GREEN
  sta $D023
  lda #YELLOW
  sta $D024
```

```
  // Put the A char on the screen
  // and set the color index to %10 -> $d023
  lda #%10_000001 // $81
  sta $0400 + 41

  lda #WHITE
  sta $D800 + 41

  // Put another A char on the screen
  // and set the color index to %01 -> $d022
  lda #%01_000001 // $41
  sta $0400 + 43

  lda #BLACK
  sta $D800 + 43

  // And yet another A char on the screen
  // and set the color index to %11 -> $d024
  lda #%11_000001 // $C1
  sta $0400 + 45

  lda #BROWN
  sta $D800 + 45


forever:
  jmp forever

.pc = $2000 "Custom font"
.fill 8,0 // char 0 -> @
.byte %01111110 // char 1 -> a custom A
.byte %00000110
.byte %01100110
.byte %01100110
.byte %01111110
.byte %01100110
.byte %01100110
.byte %00000000
.fill (256-2)*8 , 0 // blank the rest of the letters
```

// HiRes Bitmap Mode

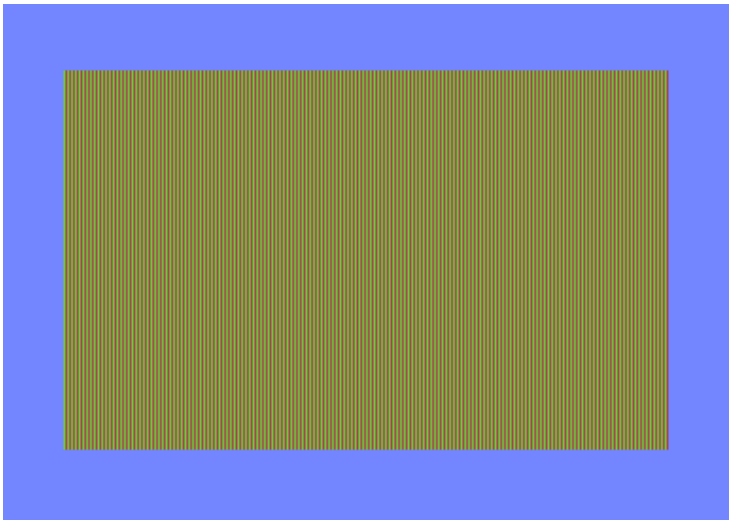


Figure 4: HiRes bitmap showing GREEN and RED stripes

Listing for 04-bitmap-hires.asm

```
BasicUpstart2(main)
main:
  lda $D011        // read current value
  ora #%0010_0000  // set bit 5 for Bitmap
  and #%1011_1111  // clear bit 6 to make sure Extended mode is off
  sta $D011

  lda $D016
  and #%1110_1111  // clear bit 4 for HiRes mode
  sta $D016

  lda #$18        // set the bitmap to be at $2000
  sta $D018       // and Screen RAM at $0400

  // set the color using the Screen RAM
  // yes, Bitmap HiRes use the Screen RAM for the
  // colors, not the Color RAM
  lda #RED << 4 | GREEN // or just $25
  ldx #0
  !:
    sta $0400,x
    sta $0500,x
    sta $0600,x
    sta $06e8,x
    inx
    bne !-

forever:
  jmp forever

.pc = $2000 "Bitmap"
// A bitmap takes 8000 (40 x 25 x 8) bytes of memory
.fill 8000, %0101_0101 // 0->GREEN, 1->RED
```

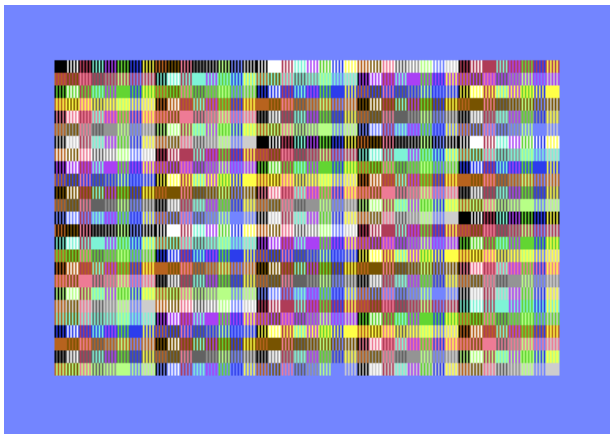


Figure 5: The same HiRes bitmap with more color combinations set using the screen RAM

```
...
  ldx #0
  !:
    txa // put x in a to get more color combinations
    // 00 -> BLACK / BLACK
    // 01 -> WHITE / BLACK
    // 02 -> RED / BLACK
    // 03 -> CYAN / BLACK
    // ...

    sta $0400,x
    sta $0500,x
    sta $0600,x
    sta $06e8,x
    inx
    bne !-
...
```

How the bitmap is organized

You might expect the bitmap to be stored line by line: 40 bytes for the first 320-pixel row, then 40 bytes for the next row, and so on. It isn't. The VIC-II reads the bitmap the same way it reads a font: one 8×8 cell at a time.

- Bytes 0–7 are the 8 rows of the first cell (top-left corner).
- Bytes 8–15 are the 8 rows of the cell to its right.
- …and so on until bytes 312–319, the last cell of the first cell row.
- Byte 320 starts the second row of cells, 8 pixels lower.

	column 0	**column 1**	**…**	**column 39**
cell row 0	0–7	8–15	…	312–319
cell row 1	320–327	328–335	…	632–639
…	…	…	…	…
cell row 24	7680–7687	7688–7695	…	7992–7999

So the bitmap is really a font of 1000 chars, displayed in order, where each char is used exactly once.

Displaying a single pixel

To find the byte and the bit of the pixel at (x, y), with x from 0 to 319 and y from 0 to 199, and given the bitmap is located at address $2000:

```
byte = $2000 + (y / 8) * 320 + (x / 8) * 8 + (y & 7)
bit  = 7 - (x & 7)
```

- `(y / 8) * 320` skips the full rows of cells above the pixel.
- `(x / 8) * 8` skips the cells to its left, 8 bytes each.
- `y & 7` picks the row inside the cell.
- `7 - (x & 7)` picks the bit: bit 7 is the leftmost pixel of the byte.

Example: the pixel at (10, 3) is in cell column 1 (`10 / 8`), cell row 0, pixel row 3 inside the cell. Its byte is `$2000 + 0 + 8 + 3 = $200B`, and its bit is `7 - 2 = 5`.

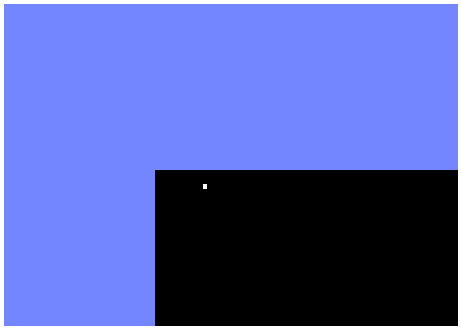


Figure 6: A pixel at (10, 3) in the top-left corner of the screen

Listing for 04b-bitmap-hires-pixel.asm

```
BasicUpstart2(main)
main:
  lda $D011
  ora #%0010_0000   // Bitmap
  and #%1011_1111 // no Extended mode
  sta $D011

  lda $D016
  and #%1110_1111   // HiRes
  sta $D016

  lda #$18          // bitmap at $2000, screen RAM at $0400
  sta $D018

  lda #WHITE << 4 | BLACK   // bit = 1 -> white, bit = 0 -> black
  ldx #0
  !:
    sta $0400,x
    sta $0500,x
    sta $0600,x
    sta $06e8,x
    inx
    bne !-

  lda #%0010_0000   // bit 5
  sta $2000 + 11    // byte $200B: the pixel at (10, 3)
```

```
forever:
  jmp forever

.pc = $2000 "Bitmap"
.fill 8000, 0       // empty bitmap
```

> **Side note:** In multicolor bitmap mode the layout is the same. Only the bits inside each byte change: with `x` from 0 to 159, the byte is `$2000 + (y / 8) * 320 + (x / 4) * 8 + (y & 7)`, and the pixel is the bit pair at position `3 - (x & 3)`, from the left of the byte.

// Multicolor Bitmap Mode

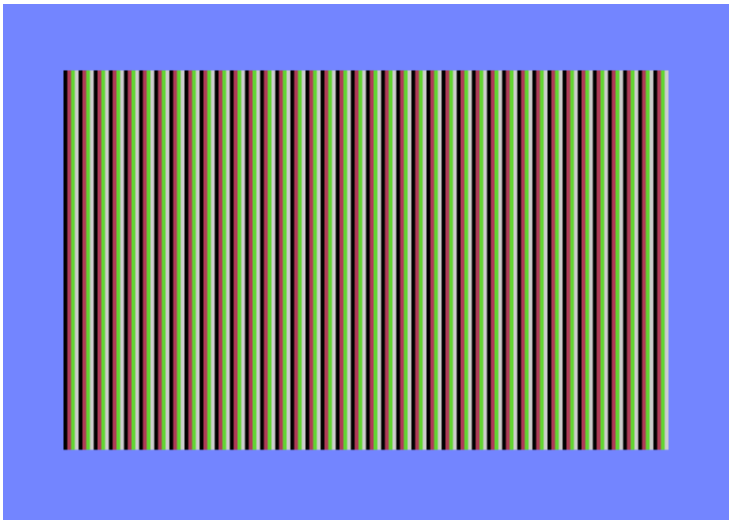


Figure 7: A multicolor bitmap (160×200)

Listing for 05-bitmap-mc.asm

```
BasicUpstart2(main)
main:
  lda $D011        // read current value
  ora #%0010_0000  // set bit 5 for Bitmap
  and #%1011_1111  // clear bit 6 to make sure Extended mode is off
  sta $D011

  lda $D016
  ora #%0001_0000  // set bit 4 for Multicolor mode
  sta $D016

  lda #$18        // set the bitmap to be at $2000
  sta $D018       // and Screen RAM at $0400

  // set two of the colors using the Screen RAM
  lda #RED << 4 | GREEN // or just $25
  ldx #0
  !:

    // Could be different colors for each 8×8 cell
```

```
    // (4×8 double-width pixels). You can try with txa below.
    // txa

    sta $0400,x
    sta $0500,x
    sta $0600,x
    sta $06E8,x
    inx
    bne !-

  // Background Color
  lda #BLACK
  sta $D021

  // Color RAM
  lda #LIGHT_GREY  // no need to set bit 3 in this case,
                   // you can use all 16 colors.
  ldx #0
  !:
    sta $D800,x
    sta $D900,x
    sta $DA00,x
    sta $DAE8,x
    inx
    bne !-

forever:
  jmp forever

.pc = $2000 "Bitmap"
// A bitmap takes 8000 (40 x 25 x 8) bytes of memory
.fill 8000, %00_01_10_11
```

Displaying a “Koala” image

Koala Painter is a famous painting program released in 1983. Many newer painting programs still support its file format, which became the standard for C64 multicolor bitmaps.



Figure 8: A beautiful multicolor bitmap by Mikael / Pretzel-Logic

Listing 06-koala.asm

```
.const KOALA_TEMPLATE = "C64FILE, Bitmap=$0000, ScreenRam=$1f40, ColorRam=$2328,
BackgroundColor = $2710"

// A beautiful image by Mikael / Pretzel-Logic,
// that was part of the demo "Breadbin of Horror" by The Pumpkins.
// Used with permission from the author.
.const picture = LoadBinary("image-by-mikael.koa", KOALA_TEMPLATE)

BasicUpstart2(main)
main:
  lda $D011        // read current value
  ora #%0010_0000  // set bit 5 for Bitmap
  and #%1011_1111  // clear bit 6 to make sure Extended mode is off
  sta $D011

  lda $D016
  ora #%0001_0000  // set bit 4 for multicolor mode
  sta $D016

  lda #$38        // set the bitmap to be at $2000
  sta $D018       // and Screen RAM at $0C00


  lda #BLACK      // Border color
  sta $D020
  lda #picture.getBackgroundColor()
  sta $D021

  // Copy the Color RAM Data to $D800
  ldx #0
  !:
    lda color_ram,x
    sta $D800,x
    lda color_ram +$0100,x
    sta $D800   +$0100,x
    lda color_ram +$0200,x
    sta $D800   +$0200,x
    lda color_ram +$02E8,x
    sta $D800   +$02E8,x
    inx
    bne !-

forever:
  jmp forever

.pc = $0C00 "Screen RAM"
.fill picture.getScreenRamSize(), picture.getScreenRam(i)

.pc = $1000 "Color RAM"
color_ram:  .fill picture.getColorRamSize(), picture.getColorRam(i)

.pc = $2000 "Bitmap"
.fill picture.getBitmapSize(), picture.getBitmap(i)
```

> **Side note: KickAssembler corner: templates.** A Koala picture file is just bytes in a fixed order: 2 bytes of load address, then the bitmap (8000 bytes), the screen RAM (1000 bytes), the color RAM (1000 bytes) and the background color (1 byte).
>
> `LoadBinary("image-by-mikael.koa", KOALA_TEMPLATE)` loads the file at compile time, and the template string describes that layout:
> - `C64FILE` skips the 2-byte load address at the start of the file, so the offsets that follow start right after it.
> - Each `Name=offset` entry gives a name to the block starting at that offset. A block ends where the next one starts.
>
> For each block in the template, KickAssembler adds two functions to the loaded file, named after the block: `get<Name>(i)` returns byte `i` of the block, and `get<Name>Size()` returns its size. So the `Bitmap` block gives you `picture.getBitmap(i)` and `picture.getBitmapSize()`, the `ScreenRam` block gives you `picture.getScreenRam(i)` and `picture.getScreenRamSize()`, and so on. Called without an index, as in `picture.getBackgroundColor()`, the function returns the first byte of the block, which is all we need for a 1-byte block.
>
> Loading the file doesn't put anything in your program. It only makes the bytes available at compile time. The `.fill` lines are what actually write them to the addresses you choose.

Warning: bank 0 is a tight fit for bitmaps

A bitmap takes 8000 bytes, and bit 3 of $D018 only lets you put it at the start ($0000) or in the middle ($2000) of the bank. In bank 0, $0000 doesn't work: it would cover the zero page, the stack, the default screen and your program, and the VIC-II sees the Char ROM at $1000–$1FFF anyway. That leaves only one place for the bitmap: $2000–$3F3F.

The screen RAM (1000 bytes) then has very few places left:

Screen RAM	Works?	Why
$0000	no	zero page and stack
$0400	yes	the default screen, but its data has to be copied there at run time (see below)
$0800	careful	your program starts at $0801
$0C00	yes	as long as your program doesn't reach it
$1000–$1C00	no	the VIC-II sees the Char ROM there
$2000–$3C00	no	taken by the bitmap

If you use $0400, don't put the picture's Screen RAM data there with `.pc = $0400`. Keep the data somewhere else, for example in $1000–$1FFF, and copy it to $0400 at run time, the same way we did in the example above to copy the color RAM. With $0C00, `.pc = $0C00` works fine: nothing touches that area between loading and running your program.

So in bank 0, your program has to fit between $0801 and the screen RAM, or go to $1000–$1FFF: the VIC-II sees the Char ROM there, but for the CPU it's plain RAM, which is fine for code, music or data. That's where this example keeps the color RAM data before copying it to $D800. For anything bigger, another bank gives you much more room. In bank 1, for example, the whole $4000–$7FFF range is free RAM: the bitmap can go at $4000 or $6000, and the screen RAM anywhere in the other half of the bank ($6000–$7C00 or $4000–$5C00).

// Splitting the screen between bitmap and text

Could we use both bitmap and text on the same screen? The answer is yes, that's something we could do with a basic naive implementation like this:

Listing for 07-bitmap-and-text-glitch.asm

```
.const SWITCH_LINE = 132
BasicUpstart2(main)
main:

  lda $D016
  and #%1110_1111   // HiRes
  sta $D016

  // init
  jsr setup_bitmap
  jsr setup_text

forever:
  jsr wait
  jsr switch_to_bitmap // switch to bitmap mode

  lda #SWITCH_LINE // wait for line to switch
  cmp $d012
  bne *-3

  jsr switch_to_text // switch to text mode

  jmp forever


setup_bitmap:
  // set the colors
  lda #WHITE << 4 | BLACK   // bit = 1 -> white, bit = 0 -> black
  ldx #0
  !:
    sta $0400,x
    sta $0500,x
    sta $0600,x
    sta $06E8,x
    inx
    bne !-

  // put a pixel
  lda #%0010_0000   // bit 5
  sta $2000 + 11    // byte $200B: the pixel at (10, 3)
  rts

setup_text:
  // fill with 'A'
  lda #'a'
  ldx #0
  !:
    sta $0C00,x
    sta $0D00,x
```

```
    sta $0E00,x
    sta $0EE8,x
    inx
    bne !-
  rts

switch_to_bitmap:
  lda #$18  // screen RAM at $0400,  bitmap at $2000
  sta $D018

  lda $D011
  ora #%0010_0000   // Bitmap
  sta $D011
  rts

switch_to_text:
  lda #$34  // screen RAM at $0C00, font at $1000 (where default ROM font is)
  sta $D018

  lda $D011
  and #%1101_1111   // Text
  sta $D011
  rts

wait:
  // wait for the raster beam to reach
  // the start of a new frame
  bit $d011
  bpl *-3
  bit $d011
  bmi *-3
  rts

.pc = $2000 "Bitmap"
.fill 8000, 0       // empty bitmap
```

Side note: Waiting for a raster line. $D012 holds the raster line the VIC-II is currently drawing (bits 0–7; bit 8 is bit 7 of $D011). By reading it in a loop, we can wait for a given line:

```
  lda #SWITCH_LINE  // the line we wait for
  cmp $d012         // are we there yet?
  bne *-3           // no: go back to the cmp
```

`*` is the address of the current instruction, so `*-3` jumps 3 bytes back, to the `cmp`. That saves us from creating a label for such a short loop.

When you run this code, you will notice that it somehow works, but it's a bit glitchy:

A) From time to time, the whole screen briefly turns black.
B) The border between the black and blue backgrounds jitters.
C) One row is filled with "P"s instead of "A"s, and some white dots show up at the split.

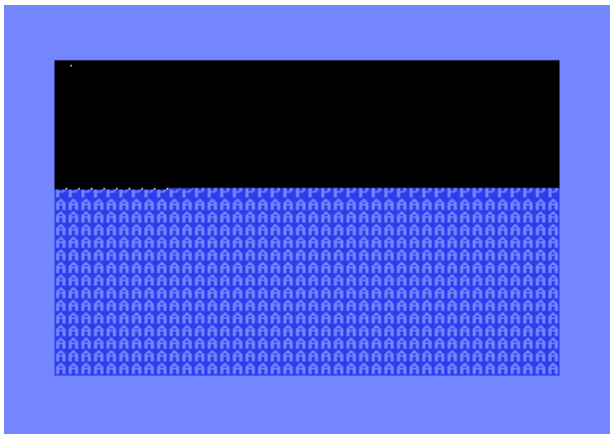


Figure 9: Splitting the screen between bitmap and text,
a first try that is a bit glitchy...

Each glitch has its own cause:

Fixing A - the black frame

The KERNAL interrupt runs 60 times per second (keyboard, cursor...). When it fires while we wait for `SWITCH_LINE`, we miss the line, and the whole frame stays in bitmap mode, which is all black here. `sei` turns it off.

```
BasicUpstart2(main)
main:
  sei  // Fix for A - disable interrupts to avoid glitches
  ...
```

Fixing B - the jitter

Our wait loop doesn't watch $D012 all the time. It checks it once per turn, so the CPU always notices the new line a little late, like someone who only glances at the clock now and then. The rest of the main loop takes a fixed number of cycles that doesn't match the loop's rhythm, so each frame the CPU glances at a slightly different moment, and the delay changes. Since every cycle is 8 pixels on screen, the switch moves around.

Interestingly, adding 5 `nop`s puts the main loop back in step with the wait loop, so the delay is the same every frame and the jitter disappears.

```
...
forever:
  nop
  nop
  nop
  nop
  nop
  jsr wait
  ...
```

But even with the jitter gone (the BLACK / BLUE transition occurs always at the exact same position on the screen), that change still occurs at the wrong position where the transition is visible on the screen.

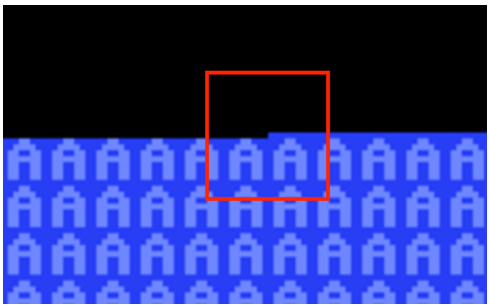


Figure 10: Jitter fixed, but transition still visible on the screen

This is also very brittle. Add one instruction and the timing with the 'nop' is off again.

The easiest fix is to move the transition a few cycles later, so that the transition occurs when not visible on the screen.

Let's remove the 'nop' and add a small delay using a short loop wasting a few cycles.

```
...

forever:
  jsr wait
  jsr switch_to_bitmap // switch to bitmap mode

  lda #SWITCH_LINE // wait for line to switch
  cmp $d012
  bne *-3

  ldx #7 // Fix for B -  waste a few cycles so that the jittering occurs later,
         // so that it's not visible on the screen.
         // Using a few 'nop' (Mnemonic for No OPeration) would do the same.
  !:
  dex
  bpl !-

  jsr switch_to_text // switch to text mode

  jmp forever
  ...
```

The transition should be seamless now.

Fixing C - the row of "P"s

The VIC-II reads the screen codes of a char row only once, on the row's first line (a "bad line", more on those in a later round). Line 132 is in the middle of a row, so the rest of that row keeps the codes read from the bitmap's screen RAM: $10, our color byte, which is the screen code for "P". Switching on line 130 makes the change before the next row starts.

```
.const SWITCH_LINE = 130 // choose a line so that we could have
                         // a cleaner split
BasicUpstart2(main)
...
```

Here are all the changes together

Listing for 08-bitmap-and-text-fixed.asm

```
.const SWITCH_LINE = 130 // choose a line so that we could have
                         // a cleaner split
BasicUpstart2(main)
main:

  sei  // disable interrupts to avoid glitches

  lda $D016
  and #%1110_1111   // HiRes
  sta $D016

  // init
  jsr setup_bitmap
  jsr setup_text

forever:
  jsr wait
  jsr switch_to_bitmap // switch to bitmap mode

  lda #SWITCH_LINE // wait for line to switch
  cmp $d012
  bne *-3

  ldx #7 // Fix for B -  waste a few cycles so that the jittering occurs later,
         // so that it's not visible on the screen.
         // Using a few 'nop' (mnemonic for No OPeration) would do the same.
  !:
  dex
  bpl !-

  jsr switch_to_text // switch to text mode

  jmp forever


setup_bitmap:
  // set the colors
  lda #WHITE << 4 | BLACK   // bit = 1 -> white, bit = 0 -> black
  ldx #0
  !:
    sta $0400,x
    sta $0500,x
    sta $0600,x
    sta $06E8,x
    inx
    bne !-

  // put a pixel
  lda #%0010_0000   // bit 5
  sta $2000 + 11    // byte $200B: the pixel at (10, 3)
  rts

setup_text:
```

```
  // fill with 'A'
  lda #'a'
  ldx #0
  !:
    sta $0C00,x
    sta $0D00,x
    sta $0E00,x
    sta $0EE8,x
    inx
    bne !-
  rts

switch_to_bitmap:
  lda #$18  // screen RAM at $0400,  bitmap at $2000
  sta $D018

  lda $D011
  ora #%0010_0000   // Bitmap
  sta $D011
  rts

switch_to_text:
  lda #$34  // screen RAM at $0C00, font at $1000 (where default ROM font is)
  sta $D018

  lda $D011
  and #%1101_1111   // Text
  sta $D011
  rts

wait:
  // wait for the raster beam to reach
  // the start of a new frame
  bit $d011
  bpl *-3
  bit $d011
  bmi *-3
  rts

.pc = $2000 "Bitmap"
.fill 8000, 0       // empty bitmap
```

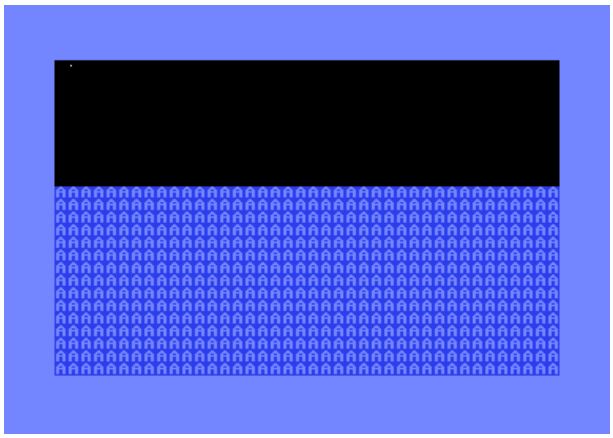


Figure 11: Splitting the screen between bitmap and text. Cleaner now.

How about a vertical split?

A vertical split is also possible, with some limitations, but it's far more complicated. We'll try it in a later round.

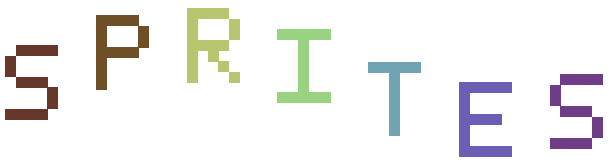
// Registers we saw so far

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
$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
$D016	VIC-II	Bit 4: Multicolor mode. Bit 3: 40 or 38 columns. Bits 0–2: horizontal scroll. Default: $C8.	9
$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
$D020	VIC-II	Border color.	8
$D021	VIC-II	Background color (color for `%00` in the multicolor 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
$D800–$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



Now that you know a bit more about the 5 graphics modes, the next round covers sprites: small 24×21 pixel images that you can place anywhere on the screen, in front of or behind the text or bitmap graphics.