Round 8 • $D018, $DD00 and $01

OCA / Atlantis

// Introduction

In Round1, we put characters on the screen by writing to $0400, and it just worked. Nice and simple. But sooner or later, you'll want more: move the screen somewhere else, design your own font, make room for sprites or music…

That's where the C64 starts setting traps. You change one value, and suddenly the screen is full of garbage. You write your own font at a perfectly reasonable address, and the VIC-II keeps showing the default one or sometimes the CPU and the VIC-II don't seem to agree on what's in memory.

None of this is random. Once you know the few rules behind it, it all makes sense, and that's what this article is about. We'll start from the screen you already know, play with the $D018 register, and fall into the traps one at a time.

At some point, you'll get to choose:
- the **blue pill**: go straight to a table telling you which addresses work and which don't;
- the **red pill**: dig into how the hardware really works, and why the CPU and the VIC-II don't always see the same thing.

Both are fine. You can always come back for the other one later.

All the examples use KickAssembler, like in Round1. You can run them in an emulator such as VICE.

One last note… We will start our exploration with $d018… that's for sure not the easiest one to grasp as a beginner… I could have started with $D020 & $D021 like this…

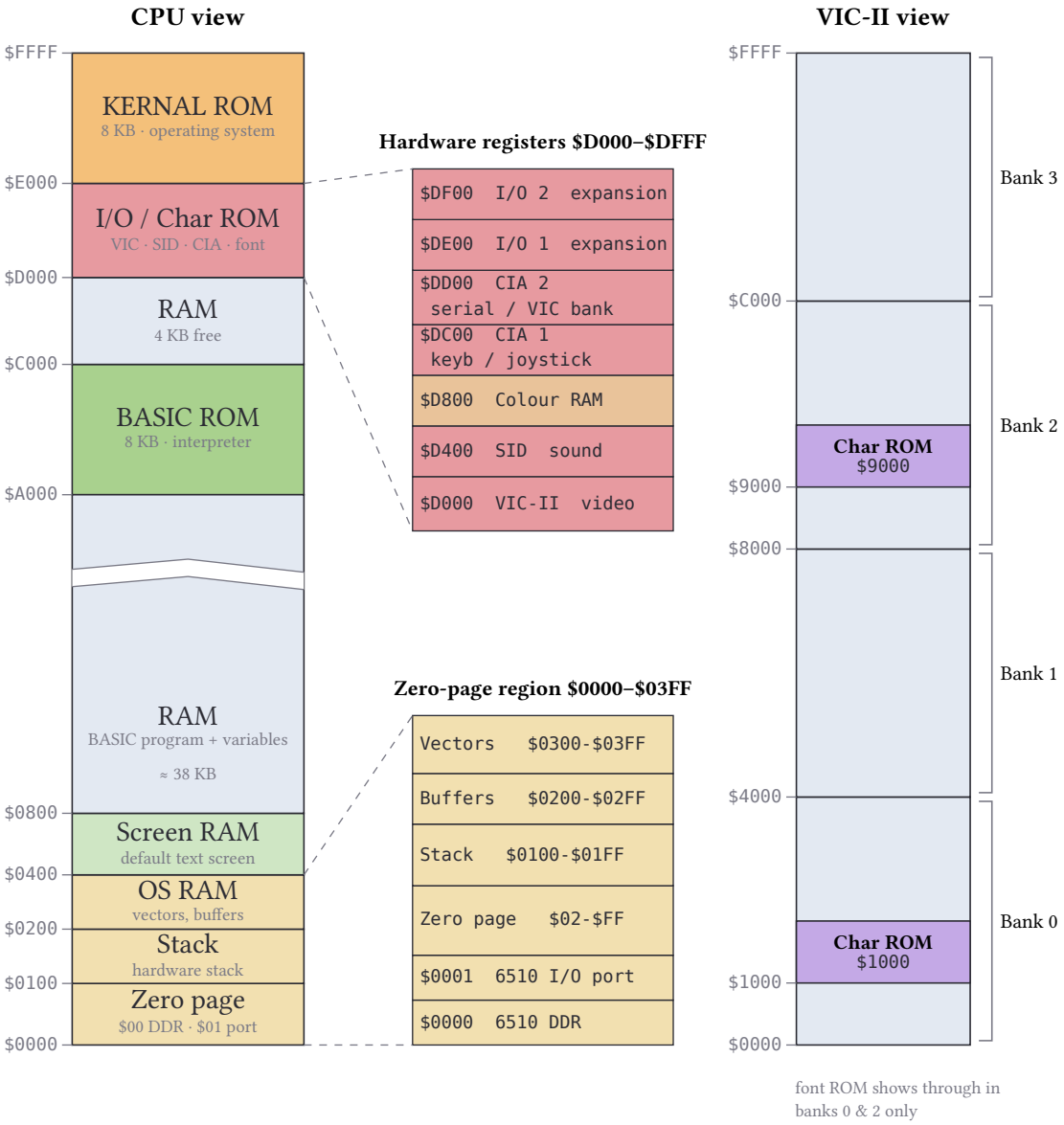
Hey … you probably already know `POKE 53281,0`, this assembly code does the same.

```
lda #0
sta $D021
```

Easy, end of the article. but where is the fun right ;)

So… let's start right away with a scary diagram…

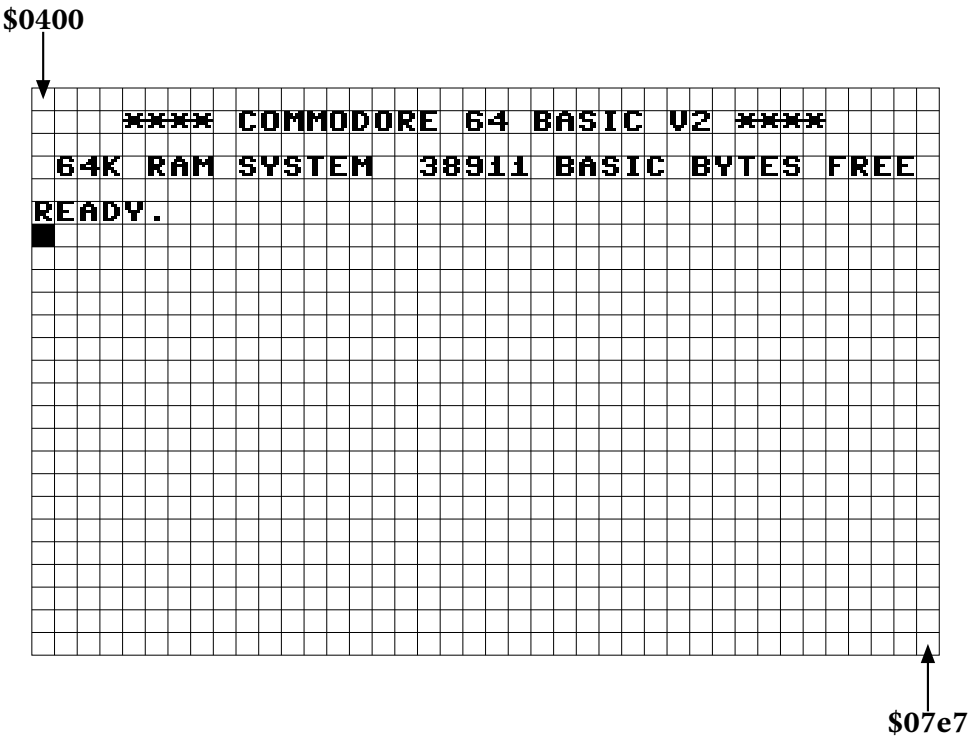
// The Commodore 64 memory map



I don't know about you, but when I first saw that first, it was pretty intimidating… For now, let's ignore this diagram, and let's explore things one at a time.

// A new take on the text screen

Do you remember one of the first things we did in Round1? We saw the address we could use to put characters on the screen.



We saw that $0400 is the top/left corner of the screen, so the following code puts an 'A' in the top/left corner.

```
BasicUpstart2(main)
  main:
    lda #1 // 'A'
    sta $0400
  forever:
      jmp forever
```

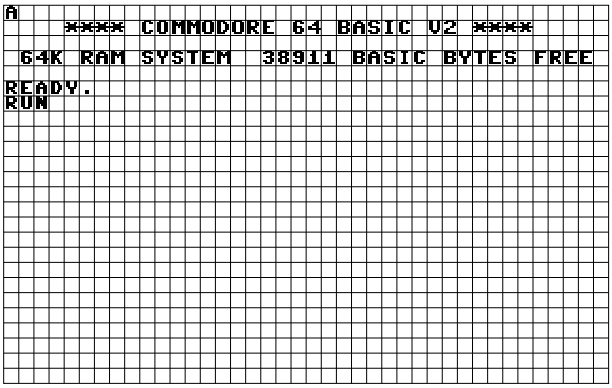


Figure 1: 'A' in the top/left corner

Cool, but...

- Why $0400? And is that something we could change?
- Can I change what the 'A' character looks like? Where does the image for that letter come from?
- What if I want to draw individual pixels instead of 8x8 characters?

TLDR: yes, you can do all that... let's see how.

// The screen base address

What if you want your screen base address to be somewhere else in memory than $0400? You can tell the VIC-II (the video controller chip) to get its data from somewhere else by changing the value of the $D018 register.

Let's first have a look at what the memory layout of our screen will look like with a base address at $0C00.

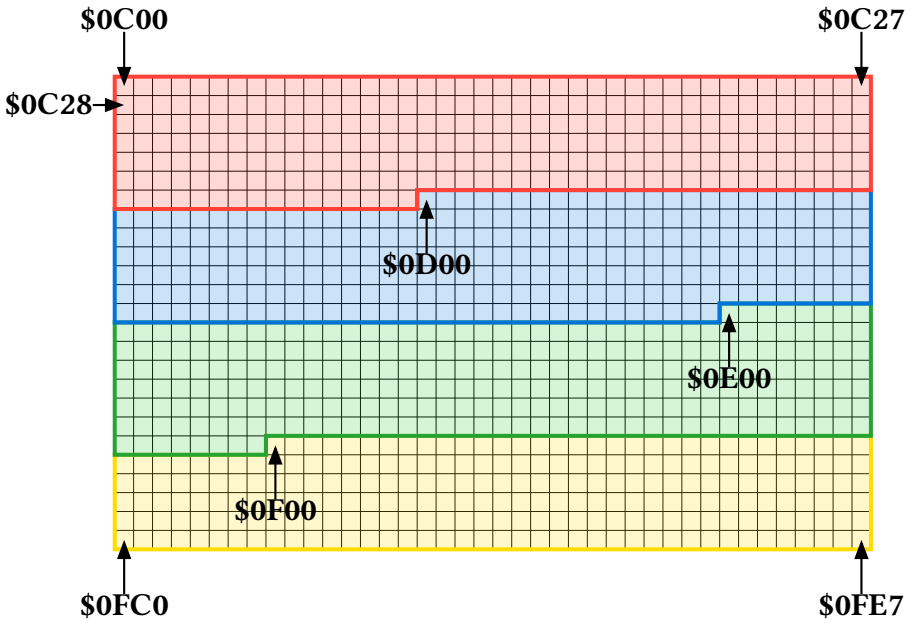


Figure 2: Screen memory layout for a base address at $0C00

> **Side note:** But why to change the screen address in the first place?
>
> For now, all we did was putting a few characters on the screen, for that, the default screen location is just plain fine but later we are going to do more interesting stuff, and choosing wisely how you are going to use your memory is going to be a crucial thing.

```
BasicUpstart2(main)
main:
    // Change the screen base address to be
    // $0C00 instead of $0400
    lda #$34  // We will explain why $34 next
    sta $D018

    // Put an 'A' in all four corners
    lda #1 // 'A'
    sta $0C00 // top/left corner
    sta $0C27 // top/right corner
    sta $0FC0 // bottom/left corner
    sta $0FE7 // bottom/right corner

    // Let's change the color RAM ($D800)
    // to display the four 'A' chars in yellow.
    lda #YELLOW // or #7
    sta $D800
    sta $D827
    sta $DBC0
    sta $DBE7

forever:
    jmp forever
```

Listing 1: round8-ex1.asm

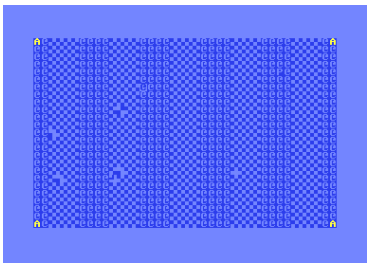


Figure 3: Yellow ‘A’ in each of the corners, with some garbage on the screen.

You should see a yellow ‘A’ in each of the corners of the screen. The rest of the screen is filled with other chars, as we didn’t clear the whole screen memory range ($0C00 to $0FE7) in our program. So let’s do this first, before we dive into how $D018 works.

```
BasicUpstart2(main)
main:

  lda #' '
  jsr fill_screen



   // Same code here...

forever:
    jmp forever

fill_screen:
    // Clear the screen
    ldx #0
    !:
    sta $0C00,x
    sta $0D00,x
    sta $0E00,x
    sta $0EE8,x // Remember why $0EE8 and not $0F00?
    dex
    bne !-
    rts
```

Listing 2: round8-ex2.asm

Good, you should now have a cleared screen, with yellow ‘A’ in each corner.

// $D018 Register

> **Side note:** For now we will focus only on the text mode, we will keep the bitmap modes for later.

Ok… why did we put the value $34 into $D018 to set the screen base address to $0C00? Or in other words, what is the relationship between $34 and $0C00?

The first thing we need to know is that $D018 serves two purposes:

- Setting the screen base address (for instance to $0C00, as we did). This is the first digit, the 3 in $34 in this case.
- Setting where (at which address) the font for the characters is located. This is the second digit, the 4 in $34 in this case.

  To better understand the relationship between '3' and $0C00, let's have a look at this value in binary. Converting from hex ($) to binary (%) is relatively easy when you get used to it.

- $3 is %0011
- $4 is %0100
- $34 is %0011_0100 (_ is just a separator to ease reading)

To set the screen base address, we only care about the first digit, '3'. This narrows the question down to: how does '3' relates to $0C00.

The key thing is that the screen cannot be anywhere in memory: the address for the screen has to start on a multiple of $0400. $0C00 is 3 × $0400 (block number 3, counting from 0), that's why the first digit is $3, as we can see in the following table.

Screen base address	**$D018 first digit**
$0000	$0
$0400	$1
$0800	$2
$0C00	$3
$1000	$4
$1400	$5
$1800	$6
$1C00	$7
$2000	$8
$2400	$9
$2800	$A
$2C00	$B
$3000	$C
$3400	$D
$3800	$E
$3C00	$F

So let's say we would like the screen to start at address $2800, then we would do it like this.

```
BasicUpstart2(main)
main:
  lda #$A4 // we will talk about the second digit '4' next.
  sta $d018

  lda #1 // 'A'
  sta $2800 // put an 'A' in the top/left corner

  lda #YELLOW
  sta $D800 // color RAM is always at $D800

forever: jmp forever

.pc = $2800
.fill 1000, ' ' // A very lazy way of filling the screen with spaces.
                // This is wasting memory, but it's ok when you just
                // want to do a quick test.
```

Listing 3: round8-ex3.asm

Good, now try with $44 to put the screen at $1000.

```
BasicUpstart2(main)
main:
  lda #$44   // This is not working... garbage everywhere...
  sta $d018

  lda #1 // 'A'
  sta $1000 // put an 'A' in the top/left corner

  lda #YELLOW
  sta $D800 // color RAM is always at $D800

forever: jmp forever

.pc = $1000
.fill 1000, ' '
```

Listing 4: round8-ex4.asm

Oops... not working at all... garbage everywhere on the screen...



Figure 4: $A4 in $D018, screen base address at $2800, working fine.

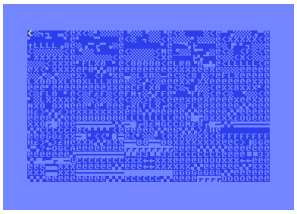


Figure 5: $44 in $D018, screen base address at $1000, garbage everywhere!

Now, let's try to put our screen at $0800, using the value $24 for $D018.

```
BasicUpstart2(main)
main:
  lda #$24
  sta $d018

  lda #1 // 'A'
  sta $0800 // put an 'A' in the top/left corner

  lda #YELLOW
  sta $D800 // color RAM is always at $D800

forever:
  jmp forever

.pc = $0800
.fill 1000, ' '
```

Boom... this doesn't even compile. You should get this error:

```
 Got 2 errors and 0 warnings while executing:
  Error: In segment 'Default' the memoryblock 'Unnamed' ($0800-$0be7) overlaps
'Basic' ($0801-$080c)
  Error: In segment 'Default' the memoryblock 'Unnamed' ($0800-$0be7) overlaps 'Basic
End' ($080e-$081f)

Error: In segment 'Default' the memoryblock 'Unnamed' ($0800-$0be7) overlaps
'Basic' ($0801-$080c)
```

Finally let's say we would like to put our screen "after" $3C00 (which is the last entry of our table above). For $3C00 that would be $F4, the last value in the table above:

```
lda #$F4   // set screen base address to $3C00
sta $d018
```

What about $6000 or even $FC00 ??? After all, 64 KB of memory goes up to $FFFF...

And what is that second digit for, in the end? We used '4' all the time so far, like in $34, $A4, $24...

That's a lot of legitimate questions, let's maybe try to answer the last one first. The second digit works a bit like the first one, but is about setting the base address of the font (or of the bitmap but that's for a later round.): The second digit tells the VIC-II where the data for the characters starts.

The difference is the size of the blocks. A screen takes 1000 bytes, so it fits in blocks of $0400 (1024 bytes - the last ones are used for something else than the screen data, again something we are going to talk about later.). A font takes 256 characters × 8 bytes = 2048 bytes, so it needs blocks of $0800 (2 KB). That's why the second digit counts twice as fast: 2 means "block 1", 4 means "block 2", and so on. Bit 0 of $D018 is unused: the VIC-II ignores it. That's why $2 and $3 (%0010 and %0011) select the same font, and why the table lists two values for each address.

Font start address	$D018 second digit
$0000	$0 or $1
$0800	$2 or $3
$1000	$4 or $5
$1800	$6 or $7
$2000	$8 or $9
$2800	$A or $B
$3000	$C or $D
$3800	$E or $F

So in $34, the '4' means: the font starts at $1000.

Now, if you try changing the data at $1000 to make your own font, that's gonna be an EPIC fail… Seems we are in a dead end here. Answering questions brings even more questions. There are two paths from here. As the Grail Knight said, "Choose wisely." And for those preferring Matrix to Indiana Jones:



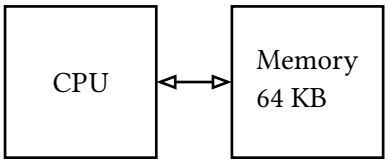
- Take the blue pill, and go directly to the end of this article, where there is a table with what works well and what doesn't…
- Take the red pill, and dive into the reality of how the hardware really works.

// The red pill - How the hardware works

At this point, this is no longer "only" about learning assembly. This is about understanding how the hardware works.

A first guess at the memory model

Maybe you see the memory as a big "flat" array that the CPU can read from and write to. Write the value 54 to any given memory slot, then read that exact same slot, and get your value back...



That would make things really easy, but **this is not always as simple as that.**

Let's make a few experiments first...

```
                    BasicUpstart2(main)
                    main:

                    lda #1
                    sta $6000 // put 1 in memory at $6000

                    lda $6000 // read it back from $6000
                    sta $0400 // and put it on the screen

                    jmp *
```

Listing 5: round8-ex5.asm

The round trip works fine, we got an 'A' in the top left corner.

Let's use a different memory location now. $d020 instead of $6000 (I guess you see me coming here...)

```
                    BasicUpstart2(main)
                    main:

                    lda #1
                    sta $D020 // put 1 in memory at $D020

                    lda $D020 // read it back from $D020
                    sta $0400 // and put it on the screen

                    jmp *
```

Listing 6: round8-ex6.asm

Ok, we get something in the top left corner, but not an 'A'... and the border turned white... Now do this, which is almost the same... we are still using $d021 as memory location, but this time, that's going to work like it did for $6000...

```
        BasicUpstart2(main)
        main:

        lda #$34 // to use the I/O range ($D000 / $DFFF) as 'regular' memory
        sta $01

        lda #1
        sta $D020 // put 1 in memory at $D020

        lda $D020 // read it back from $D020
        sta $0400 // and put it on the screen

        jmp *
```

Listing 7: round8-ex7.asm

Border color doesn't change anymore, and 'A' is back in the top left corner. We are going to talk about all this next.

This is not only about the CPU

The CPU is not the only chip, there are others like:

- the SID (for the music)
- the VIC-II (the video controller)
- two CIA chips (Complex Interface Adapter)
- the PLA (Programmable Logic Array)

and many more.

RAM vs ROM

- RAM (Random Access Memory) is memory you can read from and write to.
- ROM (Read Only Memory) is memory you can only read from.

RAM and ROM use different chips. I mean, even the RAM itself is spread among different chips:

- A dedicated chip for the KERNAL ROM
- A dedicated chip for the BASIC ROM (at least on early models; it was later put in the same chip as the KERNAL)
- A dedicated chip for the CHAR ROM (the default C64 font)
- 8 chips for the "regular" RAM
- A dedicated chip for the COLOR RAM (the memory you access within the $D800–$DBE7 range, to change the color of a character in text mode for instance)

The key idea

When you do "lda $d000" or "sta $d000", $d000 is not the direct "slot" in which you read or write data in a "flat memory array". There is a *mapping* between the address ($d000 in this case) and the physical memory location that comes into play.

In the case of $d000 for instance, that could either:

- be the x position of sprite 0,
- be regular memory to read/write general purpose data,
- be the base address of your screen (for the VIC-II, in bank 3),
- or even the location where the CPU reads the font data from.

> **Side note:** (The font data at $D000? Weren't we talking about $1000 a few lines above, with that '4' for the second digit of $D018?
>
> I've been there - and that confused me a lot at first: the CPU can access the font data at $D000 under certain conditions, and not at $1000, which is the default location where the VIC-II sees the font data.) This is part of that *mapping* we will be talking about next. The CPU and the VIC-II don't necessarily see the same thing at the same place...

Let's see how this *mapping* of the addresses works now.

A memory model closer to reality

To exchange data, all those chips are connected with electrical wires. Some of those wires work together and form what is called a *bus*.

To understand how $D018 works, we are mainly interested in the relationship between a few of those chips (we don't really care about the SID, for instance).

> **Side note:** The goal here is not to draw the exact electrical schematic of the C64. This is a simple block diagram, just meant to get the overall idea.

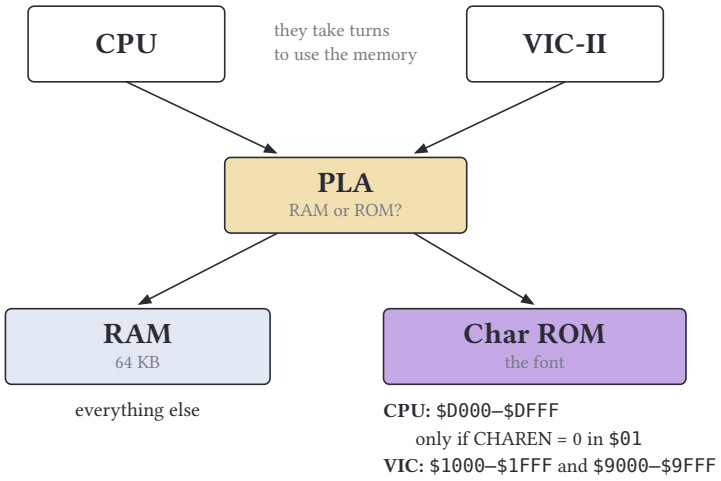


Figure 6: The PLA sends each read either to the RAM or to the Char ROM, with a different rule for the CPU and for the VIC-II.

Something even closer to reality

> **Side note:** Maybe I should not put that diagram here… as we are definitely going too far for a beginner tutorial.
> **It's perfectly fine if you don't understand all the details here.** There are many thing we didn't talk about yet.

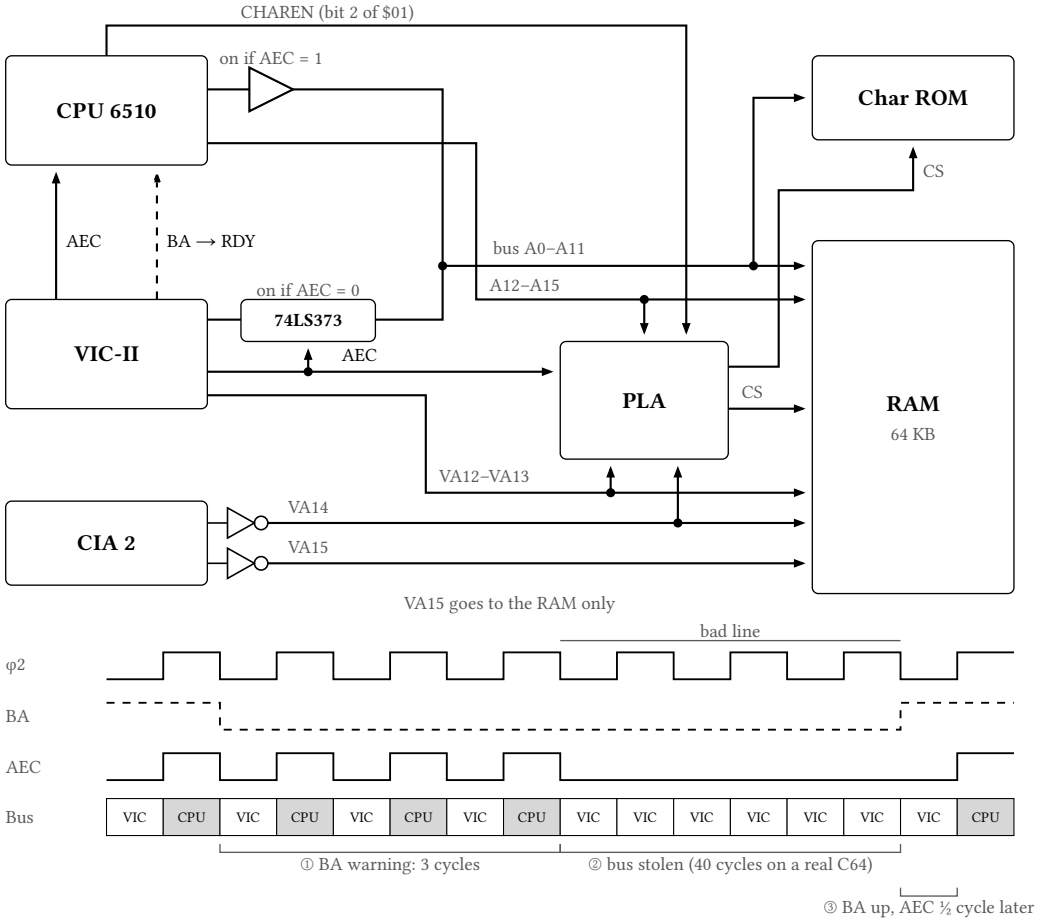


Figure 7: The CPU and the VIC-II share the address bus: AEC decides who drives it, and the PLA decides whether the RAM or the Char ROM answers.

Reading the diagram
The top part shows how the chips are wired. The bottom part shows, cycle by cycle, who is allowed to use the bus.

Address bus A0–A11 the low address lines, shared by the CPU and the VIC-II. They go to both the RAM and the Char ROM, and pick one byte inside the chip.
A12–A15 / VA12–VA15 the high address lines. The CPU drives A12–A15 itself. The VIC-II only produces VA12–VA13; VA14 and VA15 come from the CIA 2 (bits 0 and 1 of `$DD00`), which is how the 16 KB bank is selected.
AEC generated by the VIC-II. At 1, the CPU drives the bus; at 0, the VIC-II does. The small triangle after the CPU and the 74LS373 latch are "tri-state" outputs: when it is not their turn, they disconnect from the bus.
BA → RDY also generated by the VIC-II. When it drops to 0, it warns the CPU that the VIC-II is about to take the bus for several cycles in a row.
CHAREN bit 2 of `$01`, set by your program. The PLA only uses it for CPU accesses.
PLA for every access it looks at the high address lines, AEC and CHAREN, and sends its chip select (CS) either to the RAM or to the Char ROM.

One bad line, step by step
Most of the time the two chips simply alternate: the VIC-II uses the bus while φ2 is low, the CPU while φ2 is high, and AEC follows φ2. Neither notices the other.

On a bad line the VIC-II needs more than its half:

1. **BA warning.** BA drops to 0, three cycles before the VIC-II needs the bus. AEC keeps alternating, so the CPU can finish any pending write; as soon as it reaches a read, RDY makes it stop and wait.
2. **Bus stolen.** AEC now stays at 0 even while φ2 is high: the VIC-II uses the bus on both phases, and the CPU is frozen. On a real C64 this lasts 40 cycles, one per character of the text line.
3. **Return.** BA goes back to 1 as soon as the VIC-II is done. AEC follows half a cycle later, on the next high phase of φ2, because the low phase always belongs to the VIC-II anyway. The CPU then resumes exactly where it stopped.

Coming back to the questions we put aside earlier

> **Question 1.** Why does the screen show garbage when its base address is `$1000`?
>
> **Answer.** In bank 0 (the default one), the VIC-II does not see the RAM at `$1000`: it sees the Char ROM. It reads the font bytes as screen codes. There is nothing you can do to change this.

> **Question 2.** Why does putting the screen at `$0800` fail to compile?
>
> **Answer.** `BasicUpstart2` puts the BASIC `SYS` line at `$0801`, and the `.fill` overlaps with it. This is not a hardware issue: two sections of your program are trying to use overlapping address ranges.

Question 3. How to put the base address after $3C00?

Answer. Unlike the CPU, which can address the whole 64 KB, the VIC-II can only address 16 KB at once. The 64 KB are therefore split into 4 banks, and you choose which one the VIC-II uses with the two lower bits of register $DD00, a register of the CIA 2 chip.

There is a trap here: the bits are inverted.

- 3 is bank 0 ($0000–$3FFF)
- 2 is bank 1 ($4000–$7FFF)
- 1 is bank 2 ($8000–$BFFF)
- 0 is bank 3 ($C000–$FFFF)

The other bits of $DD00 talk to the disk drive, so the clean way is to change only bits 0 and 1:

```
lda $dd00
and #%11111100   // clear bits 0-1
ora #2           // bank 1
sta $dd00
```

That said, for a quick testing prg, this will work just fine:

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

This is fine as long as nothing is using the disk drive at that moment: no `LOAD` in progress and no fast loader running in the background. That's the case in most small programs, but keep the first version in mind.

$D018 then works **inside** the selected bank: in bank 1, a screen at "$0400" is really at $4400. Beware: in banks 1 and 3 the VIC-II does not see the font ROM, so you need to copy a font into RAM yourself.

That means that this **is not** going to display an 'A' in top/left corner as the default c64 font is not seen by the VIC in bank1 (and bank3)

```
BasicUpstart2(main)
main:
  lda #2 // bank 1
  sta $DD00

  lda #$14    // screen at $4000 + $0400, font at $4000 + $1000

  sta $D018

  lda #1 // 'A'
  sta $4400 // default top/left corner address for bank1

  jmp *
```

Will probably just display some horizontal stripes...

Key takeaways

- The CPU and the VIC-II don't always see the same thing at the same address.
- The VIC-II only sees 16 KB at a time. You choose the bank with $DD00, and $D018 positions the screen and the font **inside** that bank.
- The VIC-II always sees the Char ROM at $1000–$1FFF and $9000–$9FFF, so don't put your screen, sprites or custom font there. The CPU still sees RAM there, which is fine for code or data. $1000 is a good place for music: many SID tunes are made to load there.
- The CPU sees the font at $D000 only when CHAREN (bit 2 of $01) is 0. More on that just below.
- Some RAM is hidden behind the BASIC ROM, the KERNAL ROM and the I/O area. To read it, you first need to switch those out with $01.

But what is that CHAREN and that $01 stuff?

The $01 register

$01 is not a regular memory location: it's a small port built into the CPU itself. Its three lower bits tell the PLA what the CPU should see in the areas where a ROM or the I/O registers can hide the RAM.

Bit	Name	When the bit is 1…
0	LORAM	the CPU sees the BASIC ROM at $A000–$BFFF
1	HIRAM	the CPU sees the KERNAL ROM at $E000–$FFFF
2	CHAREN	the CPU sees the I/O registers at $D000–$DFFF (when 0: the Char ROM)

The other bits are used for the tape recorder, so only change these three. A few useful values:

$01	$A000–$BFFF	$D000–$DFFF	$E000–$FFFF
$37 (default)	BASIC ROM	I/O	KERNAL ROM
$36	RAM	I/O	KERNAL ROM
$35	RAM	I/O	RAM
$34	RAM	RAM	RAM
$33	BASIC ROM	Char ROM	KERNAL ROM

Three things to keep in mind:

- $01 only changes what the CPU **reads**. A write to a ROM area always goes to the RAM underneath. A write to $D000–$DFFF goes to the I/O registers when they're visible, and to the RAM otherwise.
- $01 has no effect on the VIC-II: it never sees the BASIC ROM, the KERNAL ROM or the I/O registers, only the RAM and the Char ROM.

A few more experimentations

The CPU sees RAM at $1000

```
BasicUpstart2(main)
main:
    lda #1          // 'A'
    sta $1000       // the CPU writes into the RAM at $1000
    lda $1000       // ...reads it back
    sta $0400       // ...and shows it in the top/left corner
forever:
    jmp forever
```

An ‘A’ appears in the top/left corner: for the CPU, $1000 is plain RAM.

The VIC-II sees ROM at $1000

Same program, but this time we ask the VIC-II to use $1000 as the screen:

```
BasicUpstart2(main)
main:
    lda #1          // 'A'
    sta $1000
    lda #$44        // screen at $1000
    sta $d018
forever:
    jmp forever
```

The top/left corner doesn’t show our ‘A’ but a ‘<’, and the rest of the screen is garbage. The VIC-II reads the Char ROM at $1000, and the first byte of the font ($3C) happens to be the screen code of ‘<’.

Writing under a ROM, reading from it

```
        BasicUpstart2(main)
        main:
            lda #1          // 'A'
            sta $a000       // the write goes to the RAM under the BASIC ROM
            lda $a000       // but the read comes from the BASIC ROM
            sta $0400

            lda #$36        // switch the BASIC ROM out
            sta $01
            lda $a000       // now the read comes from the RAM
            sta $0401
        forever:
            jmp forever
```

Listing 8: round8-ex9.asm

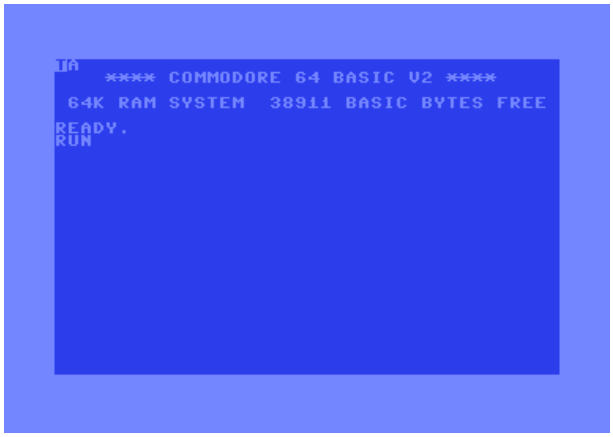


Figure 8: The first character is a reversed ‘T’ (the BASIC ROM’s first byte, $94). The second one is our ‘A’: it was in the RAM all along, hidden behind the ROM.

// The blue pill - Pragmatic tables per bank of what works well, and not so well, for your screen and font

Pick a bank, then pick a screen and a font **in that bank**. $D018 is simply the two digits side by side: for example, a screen digit of 8 and a font digit of C give `lda #$8C / sta $d018`.

One more rule: the screen and the font must not overlap. A font takes 2 KB, the size of two screen blocks, so they overlap when both digits are in the same pair: 0–1, 2–3, 4–5, 6–7, 8–9, A–B, C–D or E–F. For example, $88 or $89 puts the screen right in the middle of the font: every character you print changes the shape of other characters. $8C is fine: the screen is at $2000 and the font at $3000.

The same goes for anything else you put in the bank: your program, your sprites, your music. The tables only tell you what the VIC-II can see, not what you already put there.

Bank 0 ($0000–$3FFF, $DD00 = 3, the default)

Screen	$D018 first digit	Works?	Why
$0000–$03E7	0	no	zero page and stack
$0400–$07E7	1	yes	the default screen
$0800–$0BE7	2	careful	your program starts at $0801
$0C00–$0FE7	3	yes	as long as your program doesn't reach it
$1000–$13E7	4	no	the VIC-II sees the Char ROM there
$1400–$17E7	5	no	the VIC-II sees the Char ROM there
$1800–$1BE7	6	no	the VIC-II sees the Char ROM there
$1C00–$1FE7	7	no	the VIC-II sees the Char ROM there
$2000–$23E7	8	yes	free RAM
$2400–$27E7	9	yes	free RAM
$2800–$2BE7	A	yes	free RAM
$2C00–$2FE7	B	yes	free RAM
$3000–$33E7	C	yes	free RAM
$3400–$37E7	D	yes	free RAM
$3800–$3BE7	E	yes	free RAM
$3C00–$3FE7	F	yes	free RAM

Font	$D018 second digit	Works?	Why
$0000–$07FF	0 or 1	no	zero page, stack and default screen
$0800–$0FFF	2 or 3	careful	your program starts at $0801
$1000–$17FF	4 or 5	yes	ROM font: uppercase and graphics (default)
$1800–$1FFF	6 or 7	yes	ROM font: lowercase
$2000–$27FF	8 or 9	yes	your own font, in RAM
$2800–$2FFF	A or B	yes	your own font, in RAM
$3000–$37FF	C or D	yes	your own font, in RAM
$3800–$3FFF	E or F	yes	your own font, in RAM

Bank 1 ($4000–$7FFF, $DD00 = 2)

All RAM, nothing in the way. But the VIC-II can't see the ROM font here, so you have to copy a font into RAM first.

Screen	$D018 first digit	Works?	Why
$4000–$43E7	0	yes	free RAM
$4400–$47E7	1	yes	free RAM
$4800–$4BE7	2	yes	free RAM
$4C00–$4FE7	3	yes	free RAM
$5000–$53E7	4	yes	free RAM
$5400–$57E7	5	yes	free RAM
$5800–$5BE7	6	yes	free RAM
$5C00–$5FE7	7	yes	free RAM
$6000–$63E7	8	yes	free RAM
$6400–$67E7	9	yes	free RAM
$6800–$6BE7	A	yes	free RAM
$6C00–$6FE7	B	yes	free RAM
$7000–$73E7	C	yes	free RAM
$7400–$77E7	D	yes	free RAM
$7800–$7BE7	E	yes	free RAM
$7C00–$7FE7	F	yes	free RAM

Font	$D018 second digit	Works?	Why
$4000–$47FF	0 or 1	yes	your own font
$4800–$4FFF	2 or 3	yes	your own font
$5000–$57FF	4 or 5	yes	your own font
$5800–$5FFF	6 or 7	yes	your own font
$6000–$67FF	8 or 9	yes	your own font
$6800–$6FFF	A or B	yes	your own font
$7000–$77FF	C or D	yes	your own font
$7800–$7FFF	E or F	yes	your own font

Bank 2 ($8000–$BFFF, $DD00 = 1)

Same idea as bank 0: the VIC-II sees the ROM font at $9000. From $A000, the RAM is under the BASIC ROM: the VIC-II sees the RAM, and your writes go through, so you can use it like any other RAM.

Screen	$D018 first digit	Works?	Why
$8000–$83E7	0	yes	free RAM
$8400–$87E7	1	yes	free RAM
$8800–$8BE7	2	yes	free RAM
$8C00–$8FE7	3	yes	free RAM
$9000–$93E7	4	no	the VIC-II sees the Char ROM there
$9400–$97E7	5	no	the VIC-II sees the Char ROM there
$9800–$9BE7	6	no	the VIC-II sees the Char ROM there
$9C00–$9FE7	7	no	the VIC-II sees the Char ROM there
$A000–$A3E7	8	yes	RAM under the BASIC ROM
$A400–$A7E7	9	yes	RAM under the BASIC ROM
$A800–$ABE7	A	yes	RAM under the BASIC ROM
$AC00–$AFE7	B	yes	RAM under the BASIC ROM
$B000–$B3E7	C	yes	RAM under the BASIC ROM
$B400–$B7E7	D	yes	RAM under the BASIC ROM
$B800–$BBE7	E	yes	RAM under the BASIC ROM
$BC00–$BFE7	F	yes	RAM under the BASIC ROM

Font	$D018 second digit	Works?	Why
$8000–$87FF	0 or 1	yes	your own font, in RAM
$8800–$8FFF	2 or 3	yes	your own font, in RAM
$9000–$97FF	4 or 5	yes	ROM font: uppercase and graphics
$9800–$9FFF	6 or 7	yes	ROM font: lowercase
$A000–$A7FF	8 or 9	yes	your own font, under the BASIC ROM
$A800–$AFFF	A or B	yes	your own font, under the BASIC ROM
$B000–$B7FF	C or D	yes	your own font, under the BASIC ROM
$B800–$BFFF	E or F	yes	your own font, under the BASIC ROM

Bank 3 ($C000–$FFFF, $DD00 = 0)

No ROM font here either, but $C000–$CFFF is 4 KB of free RAM that neither BASIC nor the KERNAL uses, which makes this bank a popular choice.

The RAM under the KERNAL ROM (from $E000) works like the RAM under the BASIC ROM in bank 2. The RAM under the I/O area ($D000–$DFFF) is trickier: the VIC-II sees it fine, but to write there the CPU must first hide the I/O registers with $01.

Screen	$D018 first digit	Works?	Why
$C000–$C3E7	0	yes	free RAM
$C400–$C7E7	1	yes	free RAM
$C800–$CBE7	2	yes	free RAM
$CC00–$CFE7	3	yes	free RAM
$D000–$D3E7	4	careful	RAM under the I/O area
$D400–$D7E7	5	careful	RAM under the I/O area
$D800–$DBE7	6	careful	RAM under the I/O area
$DC00–$DFE7	7	careful	RAM under the I/O area
$E000–$E3E7	8	yes	RAM under the KERNAL ROM
$E400–$E7E7	9	yes	RAM under the KERNAL ROM
$E800–$EBE7	A	yes	RAM under the KERNAL ROM
$EC00–$EFE7	B	yes	RAM under the KERNAL ROM
$F000–$F3E7	C	yes	RAM under the KERNAL ROM
$F400–$F7E7	D	yes	RAM under the KERNAL ROM
$F800–$FBE7	E	yes	RAM under the KERNAL ROM
$FC00–$FFE7	F	yes	RAM under the KERNAL ROM

Font	$D018 second digit	Works?	Why
$C000–$C7FF	0 or 1	yes	your own font, in RAM
$C800–$CFFF	2 or 3	yes	your own font, in RAM
$D000–$D7FF	4 or 5	careful	your own font, under the I/O area
$D800–$DFFF	6 or 7	careful	your own font, under the I/O area
$E000–$E7FF	8 or 9	yes	your own font, under the KERNAL ROM
$E800–$EFFF	A or B	yes	your own font, under the KERNAL ROM
$F000–$F7FF	C or D	yes	your own font, under the KERNAL ROM
$F800–$FFFF	E or F	yes	your own font, under the KERNAL ROM

// Explore by yourself

The best way to understand all this is to test it yourself. Here is a short snippet for your own exploration: change `SCREEN` and `FONT`, and the rest is computed for you. (The code is also available in src/round8-explore.asm)

```
// Try an unusual location: under the I/O registers
.const SCREEN = $D000   // must be a multiple of $0400
.const FONT   = $D800   // must be a multiple of $0800, in the same bank

// ---- everything below is computed from SCREEN and FONT ----
.var bank       = floor(SCREEN / $4000)
.var screenDigit = floor((SCREEN & $3fff) / $0400)
.var fontBlock   = floor((FONT & $3fff) / $0800)

.errorif (SCREEN & $03ff) != 0, "SCREEN must be a multiple of $0400"
.errorif (FONT & $07ff) != 0, "FONT must be a multiple of $0800"
.errorif floor(FONT / $4000) != bank, "SCREEN and FONT must be in the same bank"
.errorif floor(screenDigit / 2) == fontBlock, "SCREEN and FONT overlap"

// The VIC-II sees the ROM font in banks 0 and 2, at font blocks 2 and 3.
.var useRomFont = (bank == 0 || bank == 2) && (fontBlock == 2 || fontBlock == 3)
// Something to write under the I/O registers?
.var underIO    = (SCREEN >= $d000 && SCREEN < $e000) || (FONT >= $d000 && FONT <
$e000)
// The copy reads the Char ROM, so it needs the I/O registers hidden too.
.var hideIO     = !useRomFont || underIO

.print "ROM font used directly: " + useRomFont
.print "I/O registers hidden:   " + hideIO

BasicUpstart2(main)
main:
    .if (hideIO) {
        sei
        lda #$33    // the CPU reads the Char ROM at $D000,
        sta $01   // and all writes go to the RAM
    }

    // 1. copy the ROM font to FONT (only when the VIC-II can't see it)
    .if (!useRomFont) {
        ldx #0
    copy:
        .for (var page = 0; page < 8; page++) {
            lda $d000 + page * $100, x
            sta FONT + page * $100, x
        }
        inx
        bne copy
    }

    // 2. clear the screen
    lda #' '
    ldx #0
clear:
```

```
    sta SCREEN, x
    sta SCREEN + $100, x
    sta SCREEN + $200, x
    sta SCREEN + $2e8, x
    dex
    bne clear

    // 3. print a message on the screen
    ldx #0
print:
    lda message, x
    beq done
    sta SCREEN, x
    inx
    bne print

done:
    .if (hideIO) {
        lda #$37    // back to normal: step 4 needs the I/O registers
        sta $01
        cli
    }

    // 4. tell the VIC-II where to look
    lda #3 - bank   // $DD00: the bits are inverted
    sta $dd00
    lda #(screenDigit << 4) | (fontBlock << 1)
    sta $d018

forever:
    jmp forever

message:
    .text "abcd 1234"
    .byte 0
```

Side note: KickAssembler corner. This snippet uses a few features that only exist at compile time. None of them generate code on their own: they compute values or decide which instructions get assembled.

- `.const` and `.var` define values. `.const` can't change, `.var` can.
- `floor()`, `&`, `<<` and `|` compute values from them, like in any programming language. That's how `lda #(screenDigit << 4) | (fontBlock << 1)` ends up as a plain `lda #$xx` in the final program.
- `.errorif` stops the compilation with a message when a condition is true.
- `.print` displays a message in the compiler's output.
- `.if` assembles the instructions inside its braces only when the condition is true. Otherwise they simply don't exist in the program.
- `.for` repeats the instructions inside its braces, here to write the 8 `lda`/`sta` pairs of the copy.

A few things you may want to try:

- `SCREEN = $1000` and `FONT = $2000`: garbage, the VIC-II sees the Char ROM where the screen should be.
- `SCREEN = $0400` and `FONT = $1800`: the message shows up in lowercase. No copy needed: the VIC-II reads the lowercase half of the Char ROM directly.
- `SCREEN = $C000` and `FONT = $C800`: bank 3 works fine, thanks to the copy.

- `SCREEN = $D000` and `FONT = $D800`: the screen and the font sit in the RAM under the I/O registers. It works because every write happens while $01 hides the I/O area.
- Use bank 1 and replace `!useRomFont` by `false` in the copy's `.if`: no more font, only garbage.
- `SCREEN = $2000` and `FONT = $2000`: KickAssembler refuses to compile.

// Recap of the registers used 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
$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
$D800–$DBE7	Colour RAM	Colour 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

We will talk about the 5 text/graphic modes.

BTW, the shaking scroller from the end of Part I hasn't been forgotten: $D016 is still on the list.