The Beluga boot sector

When the Beluga cartridge starts up, the computer is in a much more dire situation than when a normal cartridge starts up. The first 4K sector of the QPI flash ROM is the boot sector. During reset, the Beluga controller will initialize the cartridge the configuration register from flash memory. The Beluga controller will write the first 8 bytes of flash memory to the configuration register, therefore the register is actually written 8 times.

(This is for future proofing: Should 8 bit of configuration state not be enough for a possible future cartridge, then it is easy to add an external counter to the register to initialize multiple registers in turn, without changing the controller. Because GreenPAKs can be bought pre-programmed on a reel of 4000 pieces, it might be a good idea to use the same controller for multiple cartridges.)

The first 8 bytes of flash therefore contain the initial configuration:

```
boot:
        .byte $85,$85,$85,$85,$85,$85,$85,$85
```

The value $85 means that the Beluga cartridge boots with the led powered on in mode 5, in other words, there is 8KB of sequential access memory mapped at $8000 and the IO2 region at $DF00 contains SRAM.

Very quickly in the boot process, the C64 kernal will start to check the CBM80 signature at $8004. This is done with a decreasing loop counter so the signature is checked in reverse. Because the sequential access has no address bus, this means that the CBM80 signature will need to be in the boot sector in reverse in order for it to be verified correctly:

```
CBM80:
        .byte $30,$38,$CD,$C2,$C3
```

When the CBM80 signature has been found, the KERNAL will do a jmp ($8000) to execute the coldstart vector. Therefore next, we will need to present the coldstart vector:

```
coldstart:
        .word $8000
```

We just start execution at $8000, something not possible with a traditional ROM, but makes sense for sequential access memory. Now the C64 starts to execute code at $8000.

Since sequential access memory is only suitable for executing speedcode, we can only make the C64 run speedcode at cartridge boot. We want to get out of this situation as soon as possible. Speedcode in SAM is a bit different from normal speedcode because of 6502 phantom reads. When a 6502 executes an instruction like LDA #$00, there is no phantom read. This is because the instruction needs 2 bytes in memory and 2 bytes to execute, i.e. it does a read from memory in each cycle.

However, an instruction like PHA takes 3 cycles. One cycle is needed to read the instruction from memory, one is needed to write the value to the stack, but one cycle is used internally by the 6502 (I assume to increase the stack pointer). A 6502 bus can never be idle, each cycle has to be either a read or write. Therefore, the 6502 does a read cycle and ignores what is read, a phantom read. For

some instructions the 6502 puts the program counter on the address bus during a phantom read, for other instructions with a 16-operand it can put the operand on the address bus.

A phantom read is a dummy operation in case of a traditional parallel memory: The same value will be read twice. For sequential access memory, it has a side effect: It advances to the next byte, i.e. two different bytes will be read. Therefore for SAM speed code, we will need to add dummy bytes for these phantom reads into the code. The keep the code readable and compatible with normal 6502 code, we will insert a NOP byte.

The next code in the boot sector is therefore:

```
boot_stage_1:
    lda #$01
    sta $d020     ; Border colour turns white
    ldx #$0d
    txs
    nop           ; phantom
    lda #$04
    pha
    nop           ; phantom
    lda #$00
    pha
    nop           ; phantom
    lda #$4c
    pha
    nop           ; phantom
    lda #$f7
    pha
    nop           ; phantom
    lda #$d0
    pha
    nop           ; phantom
    lda #$e8
    pha
    nop           ; phantom
    lda #$04
    pha
    nop           ; phantom
    lda #$00
    pha
    nop           ; phantom
    lda #$9d
    pha
    nop           ; phantom
    lda #$de
    pha
    nop           ; phantom
    lda #$00
    pha
    nop           ; phantom
    lda #$ad
    pha
    nop           ; phantom
    lda #$00
    pha
```

```
    nop           ; phantom
    lda #$a2
    pha
    nop           ; phantom
    inc $d020     ; Border colour turns red
    jmp $0100
```

The effect of this speedcode is that a small routine is installed at $0100. Then we jump to $0100. Should the C64 crash, we can see how far it got via the border colour. The stack wraps around and the stack pointer is $ff when we jump. At $0100, we are in normal RAM and can execute normal code with branches.

The code written to $0100 is:

```
boot_stage_2:
    ldx #$00
:   lda $de00
    sta $0400,x
    inx
    bne :-
    jmp $0400
```

In other words, the routine reads 256 bytes from cartridge memory and writes these in RAM at $0400, this time from the permanently visible register at $de00. We could also have used $8000, since sequential access memory is still visible there.

Note that the above stage 2 boot is not present in this form inside the boot sector, since it is written to memory by the stage 1 boot. Next in the boot sector however, is the stage 3 boot. The stage 3 boot is 256 bytes long which should be sufficient for more extensive initialization. It currently reads:

```
boot_stage_3:
    ldx $de01     ; Deselect flash
    ; Now tell the flash to enter QPI mode by sending command $38
    ; The flash is in SPI mode and accepts two bits at a time
    ; $38 = %00 11 10 00… send $00 $ff $f0 $00
    lda #$00
    sta $de00
    ldx #$ff
    stx $de00
    ldx #$f0
    stx $de00
    sta $de01     ; Write and deselect flash

    ; The flash is now in QPI mode, so we now can send commands to flash memory
    ; without encoding them two bits at a time, we now can send them the way
    ; they are.

    ; We now configure four dummy cycles for flash reads. Send command $C0 (set
    ; read parameters) and $10 as parameter.
    lda #$c0
    sta $de00
    lda #$10
    sta $d001     ; Write and deselect flash

    inc $d020     ; Border colour becomes cyan
```

```
; We now switch the C64 to normal mode (to disable SAM) from memory map.
; This also turns off cartridge led.
lda #$00
sta $de03

; As we have no nice software on the cartridge yet, let’s just continue
; normal startup of the C64.
jmp $fcef
```

The stage 3 boot can be further enhanced to initialize the SRAM memory from cartridge. In such case the Beluga cartridge can emulate standard 8K/16K cartridges. The boot code could also copy a main program to C64 memory.