

Supper Guppy / Beluga cartridges

Introduction

Retro Gaming is very popular nowadays. Not only do people play old games for nostalgic reasons, there also a lot of software and hardware development and innovation being done on old computers.

One of the old computers that is still receives a lot of love in the 21th century is the Commodore 64. I am myself involved in the Commodore 64 community. The amount of people using a Commodore 64 today is still large enough to publish games on a commercial basis for the platform. Although the computer has a long and glorious history in games, over and over again programmers prove that the computer end of its capabilities has not been reached. In fact, during recent years the computer has received some of the most impressive and most polished games ever.

Some things have changed during the last 40 years...

Frantic Freddie (1983)



Size: 31KB

Soundtrack: 8 minutes

Graphics: Text mode with custom charset, sprites barely have animation

Storage medium: Cassette

A pig quest (2023)



Size: 1 MB

Soundtrack: 94 minutes

Graphics: Bitmap mode with background animations + fully animated sprites

1 bitmap = 9KB. Game contains 200 screens = $9 \times 200 = 1.8$ MB of bitmap data (compressed into cartridge) This is just the screen backgrounds.

Storage medium: EasyFlash cartridge

No graphics reuse, not between levels, not even

Graphics reused between levels:



between different screens:



Hardware: EasyFlash cartridge



During the last years several Commodore 64 titles have appeared that fill up an 1MB cartridge up to the very last byte that is available:

- Briley Witch Chronicles (2021) (homebrew)
- Lykia the Lost Island (2022) (commercial)
- Eye of the Beholder (2022) (homebrew)
- A pig quest (2023) (commercial)

More large games are in development and discussion with those developers learns that they are constrained by cartridge size in what they can do.

Conclusion:

We need to design even bigger cartridges!

A bigger flash ROM

Idea: Bigger parallel flash ROM

The most basic idea to get a bigger cartridge is to simply increase the size of the ROM. Most C64 cartridges currently use 512KB parallel ROMs and it is not difficult to see why: They are relatively low-cost, can run on 5V and the PLCC32 package is friendly for both hand-soldering as well as automated SMD assembly. What if we want a bigger parallel flash ROM?

Example: SST39VF6401

- 8MB parallel flash ROM
- cannot run on 5V and thus needs voltage conversion on a huge number of pins
- TSOP48 or even ball-grid-array not fun to solder by hand, okay for automated assembly
- Price: €8.18 excl. VAT

The price of the chip alone already shows that the big parallel flash ROM isn't going to help producing a game at a friendly price. We still need to add voltage regulators and voltage level converters to the cartridge. We also need extra bank switch registers to deal with the larger ROM size.

Idea: Big serial flash ROM

Example: W25Q64

- 8MB QPI flash ROM
- cannot run on 5V but we only need voltage conversion on a few pins
- Price: €0.41 excl. VAT

There is no escape from the economic reality: The price shows, that if we want to go for a bigger cartridge, serial flash ROMs are the way to go. If we only pay €0.41 for the flash, we have budget left for additional components and there is still opportunity to be able to produce the cartridge at a friendly price.

A serial ROM, that must be terribly slow, right? Well... please read on.

The problem is indeed that the bus of the Commodore 64 natively supports parallel flash ROMs while for serial flash ROMs interface circuitry will be needed. It is possible to design some very basic circuitry with 74xx logic chips that allows software to do bit-banging. This will work, but the flash memory cannot be directly accessed from the computer, bit-banging routines will have to retrieve the data. This makes flash access slow, but also removes the ability to execute code directly from cartridge memory, a technique regularly used by C64 programmers to save RAM.

There have been attempts before to construct cartridges that convert serial to parallel with the help of a small FPGA: The Gmod3 cartridge. The maximum 104MHz clock speed of the W25Q64 certainly allows for a lot of flash clock cycles within a Commodore 64 clock cycle, so the idea sounds feasible. However, the chip shortage has caused FPGA prices to skyrocket and it looks like the Gmod3 cartridge will not be economically feasible anytime soon.

My idea: QPI to OPI conversion

The Commodore 64 has an 8-bit data bus. A QPI flash ROM has a 4-bit bus to communicate with the outside world. What if we don't do a full parallel to serial conversion (where we use both the address and data busses), but just convert 4-bit to 8-bit and then connect the serial ROM just to the 8-bit data bus?

I'm calling it "octal peripheral interface", or OPI.

Suppose we map the flash ROM to a register in the IO1 memory region of the Commodore 64, for example \$DE00. The CS line of the flash is automatically activated on access to \$DE00 and will stay active. The register is also mirrored at memory address \$DE01, but any access to \$DE01 will make the flash CS line deactivate at the end of the clock cycle.

Suppose we want to read 256 bytes from flash memory at flash memory address \$012345. In that case the Commodore 64 can do this:

- Write "fast read quad I/O" command \$EB to \$DE00
- Write the high address byte \$01 to \$DE00
- Write the mid address byte \$23 to \$DE00
- Write the low address byte \$45 to \$DE00
- Write the M byte \$00 to \$DE00
- Read 255 times a byte from \$DE00
- Read a byte from \$DE01

Note that we can just repeatedly read bytes from \$DE00, without any need to increase a pointer, without worrying about page boundaries, without worrying about cartridge bank boundaries. These things would have to be worried about in a traditional bank switched parallel flash ROM.

This means that a serial flash ROM, using QPI to OPI conversion, can actually be accessed faster than a parallel flash ROM! This should help developers push the boundaries of the Commodore 64 even further.

Dummy cycles

QPI flash ROMs can run on high clock speeds often over 100MHz. To allow this, QPI flash ROMs require dummy cycles. Because the M byte counts as two dummy cycles (one for every nibble), the above example would work as described if the flash ROM uses two dummy cycles.

The amount of dummy cycles is configurable to allow the user to choose between latency and achievable clock speed. However, it turns out very few ROMs support two dummy cycles. Winbond is the only major manufacturer that supports two dummy cycles.

In the interest of avoiding vendor lock-in, we need to avoid the situation with two dummy cycles. For example, you can configure flash ROMs to 4 dummy cycles, which most flash ROMs appear to be able to support. This however, would require an extra write for each memory access.

The Beluga controller will assist with this: If \$DE02 is being written rather than \$DE00, in addition to the byte written, two dummy cycles to the flash ROMs are generated.

Therefore we modify the above recipe to:

- Write "fast read quad I/O" command \$EB to \$DE00
- Write the high address byte \$01 to \$DE00
- Write the mid address byte \$23 to \$DE00
- Write the low address byte \$45 to \$DE00
- Write the M byte \$00 to \$DE02
- Read 255 times a byte from \$DE00
- Read a byte from \$DE01

And we are able to access the flash memory with the minimum possible amount of 6502 instructions.

Sequential access memory

There is still a problem: The Commodore 64 won't do a cartridge boot with just a register in memory. The Commodore 64 detects how it needs to boot from a cartridge signature in the ROML memory region, if found it will boot from cartridge.

To solve this, we will also map the flash ROM register to the ROML memory region at \$8000..\$9FFF. The address bus is ignored, so it doesn't matter at which address you read, the flash ROM will always return the next byte from flash memory.

In other words, the flash ROM will become sequential access memory: Each access returns the next byte.

Mapping the register this way allows us to make the C64 detect the cartridge signature and it will do the cartridge boot. We can also execute code in the ROML region, the C64 CPU will increase the program counter and read the next instruction from flash. Just jumps will be problematic because the address is ignored.

This technique doesn't just allow us to boot, but is also useful for speedcode, a technique regularly used by C64 programmers to get more performance out of the CPU.

A challenge here is that the cartridge logic will have to send the initial command and address to the flash ROM on boot (for example \$EB \$00 \$00 \$00 \$00 to start execution from the beginning of flash memory).

A little bit of parallel memory

I believe that a cartridge with nothing more than sequential access memory will be too revolutionary and unconventional for people. I want to include a traditional parallel component, so programmers still can rely on what they have done for decades.

One possibility is to add a small parallel ROM to the cartridge. This also would solve the boot issue in a very traditional way.

However, it would be much more powerful to add a small SRAM to the cartridge. After all, when you have 8MB of flash memory, 16KB of additional ROM won't make big difference in capacity.

However, when adding RAM, programmers can fill of the SRAM with data from the serial flash ROM and use them as traditional cartridge ROM, for example to emulate traditional 16KB non-bank switched cartridges. In addition, they can use it as a RAM expansion. I'm sure C64 developers that try to push the limits of the machine will appreciate having 16KB of additional RAM available.

My current plan is do to both traditional ROM and SRAM: There will be a cartridge version with traditional parallel ROM and a version with SRAM.

DMA engine

The Commodore 64 cartridge port has DMA capability, which is very powerful: Cartridges can read/write C64 memory without involvement of the CPU. The 1764 RAM Expansion Unit uses DMA to allow the programmer to access memory of the REU.

Until now the DMA capabilities of the C64 have not been easily accessible to games. First of all, the REU occupies the cartridge port. That means that, if a game requires a REU, it cannot be released as a cartridge, while the cartridge format is the preferred format for C64 games in the 21st century. Second, the amount of C64 users who owns a REU, reduces the market potential for a game.

Implementing DMA capabilities on a game cartridge itself is complicated due to the amount of hardware involved: Suppose a DMA controller, like the REU, can access 16MB of RAM. This requires a 24-bit address bus and an 8-bit databus on the memory side. Then the C64 bus has an additional 16 address lines and another 8 data lines.

The MOS 8726 REU controller makes its life easier by using multiplexed DRAM address lines, which reduces the amount of address lines for the internal memory to 12. Still, the MOS 8726 is a big 68 pin integrated circuit.

If you want a DMA controller on a cartridge, you can use either:

- A big FPGA with sufficient pins to implement the DMA controller
- Many standard components to build a DMA controller.

In modern electronics, lots of pins means a big die, and big die means you have space for a lot of compute power, i.e. you need to buy a powerful FPGA, which is too expensive for a game cartridge.

If you want to implement a DMA controller for inexpensive standard components, you are quickly faced with the problem that you need so many of them that the design becomes too complex for a game cartridge.

Let's check what components we need for a DMA controller that can DMA from cartridge to C64 memory. We need:

1. A register for the source address that can count up
2. A register for the destination address that can count up
3. A register for the number of bytes to DMA, that counts down.
4. Control logic




If you read back a bit, you will notice that requirement (1) is implemented by the QPI ROM: You write the address where you want to read to the ROM, and then this address is automatically incremented every time you do a read. Thanks to its serialness, the QPI ROM has very few pins. Therefore the QPI ROM solves the issue of requiring big chips to a large extent as well.

Requirement (2) still needs to be implemented on a cartridge. This can be done with two TTL chips, for example 74x869.

Requirement (3) and (4) can be performed by an SLG46620 Greenpak in combination with a 74x597 shift register.

This means that adding DMA capability to the cartridge requires only 4 rather inexpensive chips! 3 TTL-chips and a GreenPAK. Although this will increase the total price of the cartridge a bit, the power of DMA is so enormous that this can be worth it.

Cartridge versions

Super Guppy cartridge	Beluga cartridge	Beluga XL cartridge
		
<p>8 megabytes serial flash 128 KB parallel flash (banks w) write-only register GAME/EXROM via register</p> <p>lowest amounts of components possible</p>	<p>16MB serial flash 16KB SRAM (direct map) read/write register GAME/EXROM via decoder sequential access memory</p>	<p>16MB serial flash 32KB SRAM (direct map) read/write register GAME/EXROM via decoder sequential access memory DMA engine</p>

The **Super Guppy cartridge** will be ideal for C64 games that do not necessarily want to push boundaries, but just a low cost storage medium. There will be no fancy decoder, SRAM or DMA engine, but it will be perfectly suited for games described at the start of this document and... because of the large amount of ROM and the faster sequential access, it will still allow more demanding games than existing cartridges.

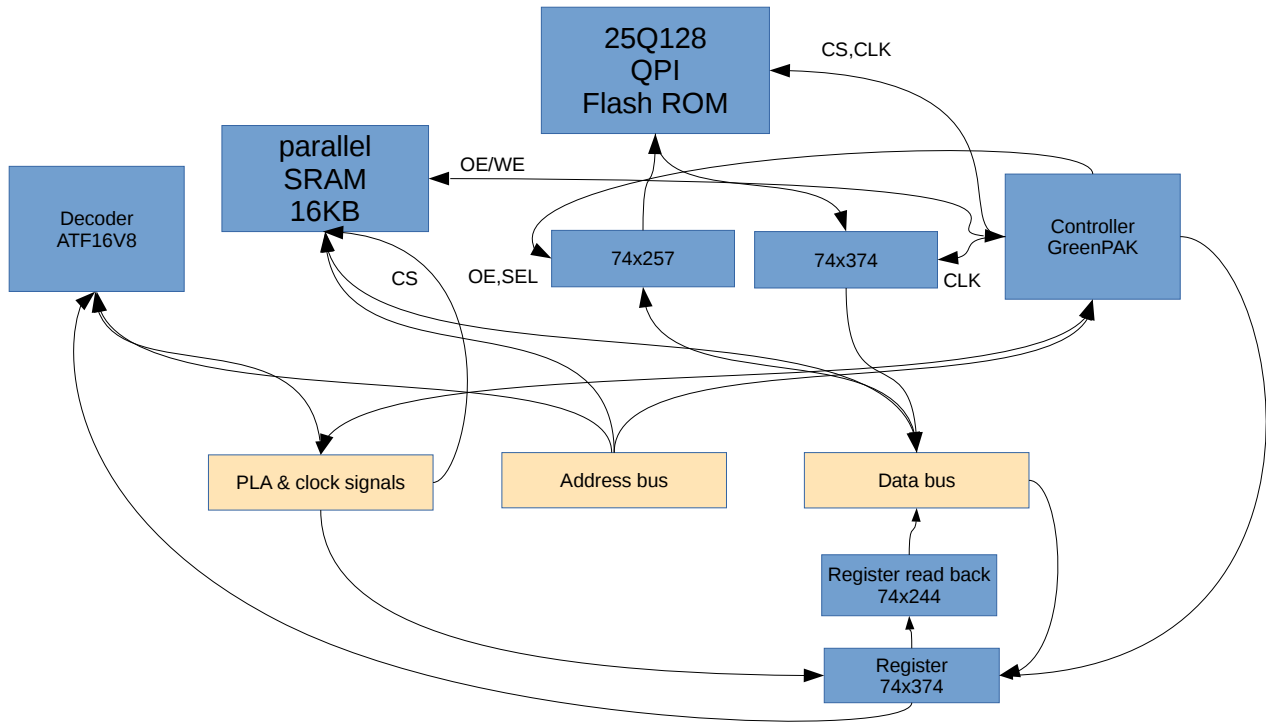
Releases by scene groups have slightly different requirements than new games being developed. The 256 bytes of RAM that EasyFlash offers is very important for the scene, because it allows addition of functionality to games that may already use all of the C64s memory. With its 16KB of SRAM, the **Beluga cartridge** not only provides such memory in the IO region, but also a significant amount of memory that can be banked in. The goal of the Beluga cartridge is to do at least anything that EasyFlash can do and should be perfectly suitable for this task, at slightly higher cost than the Super Guppy.

The **BelugaXL cartridge** would be ideal for software that wants to push the C64 way beyond its maximum. With 32KB of SRAM that can be mapped into the C64 address space, there is more RAM than every available, and the DMA engine allows acceleration of a lot of tasks.

The Beluga cartridge is the first cartridge that is being developed. Therefore, the rest of this document will talk about the design of the Beluga cartridge only.

Beluga cartridge design - schematic overview

The components of the cartridge are shown below:



The conversion between QPI and OPI is performed by a 74x257 (from C64 to flash) and 74x374 (from flash to C64). We will need a register to control memory mapping. The 74x374 and 74x273 are popular components to implement cartridge registers. To reduce the number of unique parts, we will use the 74x374 here, because the Beluga controller will initialize the register on reset. Therefore, we do not need the reset capability of the 74x273.

We will still need control signals for the 74x257 and 74x374. We need to generate the clock signal and chip select signal for the flash memory, we need to generate the chip select signal for the SRAM and we need to generate the Commodore 64 PLA control signals. This will be done inside a second chip, the Controller GreenPAK.

Readable register

In many game cartridges the configuration register is write-only. While this does reduce the amount of chips, this increases coding complexity, since in many cases, software has to keep a shadow copy of the register value in order to allow a routine to restore the situation before it modifies the register.

The need to update a shadow register reduces how fast you can change the cartridge configuration. Now, the goal of the Beluga cartridge is to push C64 gaming further. Considering that the Beluga cartridge isn't a minimalistic cartridge, I think we can afford one extra TTL chip to allow the register to be readable. This will allow software to be able to switch quicker between the memory configurations that the Beluga offers.

It is also possible to use a Greenpak to implement an R/W register. This would eliminate the extra chip. However, for the Beluga, I have chosen the TTL approach.

Memory mapping

We need to give the programmer control over how the cartridge presents itself in the memory of the Commodore 64. I am using 3 bits of the register to allow the programmer to select between 8 different modes. The decision to use 3 bits is because I want to be able to at least have a normal mode, 8K traditional cartridge mode (from SRAM) 16K traditional cartridge mode (from SRAM) and a writable SRAM mode. As I need modes where the Sequential Access Memory appears as well, 2 bits are not enough.

For the time being, these modes are as follows:

	Description	ROML REGION	BASIC/ROMH REGION	UPPER RAM REGION	I/O REGION	KERNAL REGION
Mode / Address		\$8000..\$9FFF	\$A000..\$BFFF	\$C000..\$CFFF	\$D000..\$DFFF	\$E000..\$FFFF
0	Normal mode, cartridge inactive	C64 RAM	BASIC	C64 RAM	Normal I/O	KERNAL
1	8KB cartridge SRAM	Cart SRAM readonly	BASIC	C64 RAM	Normal I/O	KERNAL
2	16KB cartridge SRAM	Cart SRAM readonly	Cart SRAM readonly	C64 RAM	Normal I/O	KERNAL
3	SRAM writable	Cart SRAM R/W	BASIC	Unallocated	Ultimax I/O	Cart SRAM R/W
4	SRAM ROMH	C64 RAM	SRAM readonly	C64 RAM	Normal I/O	KERNAL
5	SAM 8K	Cart SAM	BASIC	Unallocated	Ultimax I/O	KERNAL
6	SAM 16K	Cart SAM	Cart SAM	Unallocated	Ultimax I/O	KERNAL
7	SRAM/SAM mix	SRAM R/W	Cart SAM	Unallocated	Ultimax I/O	KERNAL

- Mode 0 allows the programmer to switch of the cartridge mapping in memory. Modes 1 and 2 implement traditional cartridge modes based on SRAM (which is read-only). Mode 3 makes the SRAM writable by the programmer.
- Mode 4 makes SRAM appear in the ROMH region only. This is normally the location of the BASIC interpreter. This will allow people to design cartridges that replace the BASIC interpreter in the C64.
- Mode 5 allows sequential access memory in the ROML region. This mode will likely be used to boot the cartridge.
- Mode 6 is a mode with 16K sequential access memory. I'm not sure wether this is really needed, but I have a mode available. It could be changed if I later think of something more useful.
- Mode 7 has both SRAM and sequential access memory visible in the memory map. This mode might be useful for situations where the cartridge is normally hidden (mode 0), but has some system vectors hooken, which dynamically activate the cartridge (into this mode 7). Having the sequential access memory also visible allows these routines to access the huge flash memory as well.

The Commodore 64 PLA has 2 inputs on the cartridge port, called EXROM and GAME to select the cartridge mapping mode:

Mode	EXROM	GAME
Normal	1	1
8K cartridge	0	1
Ultimax	1	0
16K cartridge	0	0

(The upper RAM region is sometimes unallocated. The reason is that the Commodore 128 detects the presence of a Commodore 64 cartridge in the I/O region. By activating Ultimix mode in the I/O region, we make the Commodore 128 start in C64 mode rather than its native mode. However an undesired side effect is that the upper RAM region is deallocated (Ultimix mode deactivates internal RAM). This would be solvable by an extra address bus bit, but combinatorial logic will be already complex enough with the current situation.)

In order to implement the above memory map, we need to drive EXROM and GAME with the right value depending on the address bus and/or ROML/ROMH signals. Using ROML/ROMH signals where it is possible has preference, since it gives the programmer more control over the memory map via the processor port register at \$0001 in the C64 memory.

Converting the above table results in:

	Description	ROML REGION	BASIC/ROMH REGION	UPPER RAM REGION	I/O REGION	KERNAL REGION
Mode / Address		\$8000..\$9FFF	\$A000..\$BFFF	\$C000..\$CFFF	\$D000..\$DFFF	\$E000..\$FFFF
0	Normal mode, cartridge inactive	Normal	Normal	Normal	Normal I/O	Normal
1	8KB cartridge SRAM	8K crt	8K crt	8K crt	8K crt	8K crt
2	16KB cartridge SRAM	16k crt	16k crt	16k crt	16k crt	16k crt
3	SRAM writable	Ultimix	Normal	Ultimix	Ultimix	Ultimix
4	SRAM ROMH	Normal	16k crt	Normal	Normal	Normal
5	SAM 8K	8K crt	8k crt	Ultimix	Ultimix	Normal
6	SAM 16K	16k crt	16k crt	Ultimix	Ultimix	Normal
7	SRAM/SAM mix	Ultimix	16k crt	Ultimix	Ultimix	Normal

The Controller Greenpak will have to drive the EXROM and GAME lines according to the above table.

To give the programmer even more control over the memory map, the SRAM or SAM can also appear in the IO2 region at \$DF00..\$DFFF. Bit 3 of the register controls whether SRAM or SAM is visible.

Why only 16KB of RAM?

You might ask, if the Supper Guppy cartridge will get 128KB of parallel ROM, why restrict the Beluga cartridge to 16KB of RAM?

On a game cartridge a very careful balance needs to be kept between keeping the hardware as simple as possible and features. I foresee that a major use of the parallel memory will be to run game code from cartridge. For the Super Guppy cartridge, the consideration is that 16KB of parallel ROM won't be enough to allow complex software to be run directly from cartridge. Therefore, adding a bank switch mechanism makes quite a bit of sense. 128KB in addition to the QPI ROM, should be enough for very complex games.

For the Beluga cartridge, the consideration, is that we do not add the parallel memory because we want more capacity. There is a huge amount of ROM available in the QPI ROM. 16KB of SRAM can be overwritten at any time at relatively high speed from the QPI ROM to provide similar functionality as bank switching would allow.

The Beluga cartridge has a decoder to allow its powerful memory configurations, which needs a few bits in the onboard configuration register to select the right config. This means that these bits cannot be used for banks switching the parallel memory. To support bank switching of the SRAM, we would need to add a second register.

As a result, the design trade-off pleads against a bank-switchable SRAM: Due to the fact that the memory is RAM, bank switching a lot of RAM isn't all that necessary, while extra hardware would be required to make it possible.

Why no EEPROM for save games?

The popular Gmod2 cartridge has 512KB of flash and 2KB of EEPROM for save games. Flash is not expected to be written by user software. The popular EasyFlash cartridge has 1MB of flash ROM, it has no EEPROM, but user software can write to the flash in order to implement save functionality.

Implementing game-saving on EasyFlash sucks compared to Gmod2. On Gmod2, the EEPROM is word-addressable, i.e. each individual 2 bytes can be written independently from the rest of the memory. Thanks to the EEPROM, the flash does not need to be writeable by user software, so Gmod2 can get away with an extremely minimal write circuit.

On EasyFlash, memory needs to be erased before it can be written again, and the EasyFlash API imposes a 64KB sector size. 64KB is too much data to backup in C64 memory during an erase. There many EasyFlash games implement a very space-wasteful algorithm: They allocate a 64KB slot per savegame, and if a game is saved in a certain sector, the previous save is invalidated by setting bytes to 0, and the new game is written into not yet written space of the sector. Once a sector is full, it is erased. An EasyFlash cartridge needs a good write circuit in order to support all C64 mainboards.

The Beluga cartridge will not have an EEPROM, because:

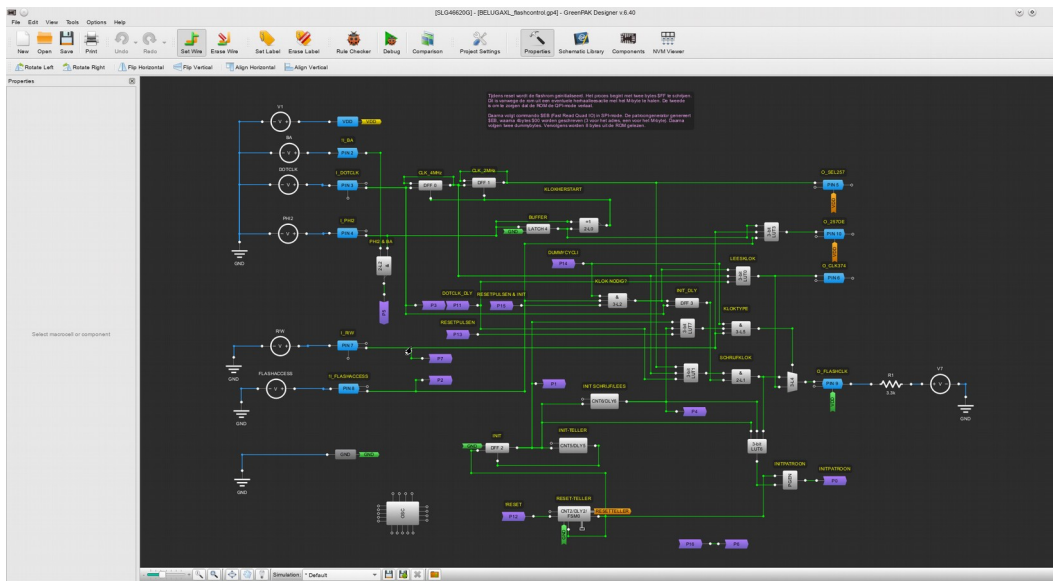
- The write circuit is there, one cannot use a QPI flash ROM without writing commands to it.
- Even though saving on EasyFlash sucks, the community doesn't seem to be terribly affected by it, since a huge number of EasyFlash releases support saving on cartridge.
- The QPI flash ROM has a 4KB sector size. This allows games to be a lot less wasteful in terms of flash memory usage, for example, use a 4KB sector per slot. Also, 4KB is easy to backup and restore in C64 memory, for example for garbage-collecting a sector in case a developer decides to store multiple save slots in a sector.
- Beluga is a huge cartridge. While 2KB seems to work for Gmod2, I believe Beluga should not be limited by a small EEPROM. 2KB is the maximum for Microwire EEPROM, an I2C or SPI EEPROM could be considered for more capacity, but then the question is how big the EEPROM would have to be. By intending saves in flash memory this question is avoided. Flash memory enough...

Controller Greenpak

The Beluga controller is implemented in an SLG46620 Greenpak, and needs to:

- generate the control signals for the 74x257, 74x374 chips
- initialize the flash chip during reset so it is ready to serve sequential access memory
- Generate OE and WE pulses for the parallel memory (flash or SRAM)

Generating the control signals for the 74x257, 74x374 chips is done in matrix 0 of the SLG46620. Inputs are the DOTCLOCK, PHI2 clock, R/W line and (temporary) flash access signal, which indicates the C64 wants to read from sequential access memory



The top 2/3 of the above schematic deals with these control signals.

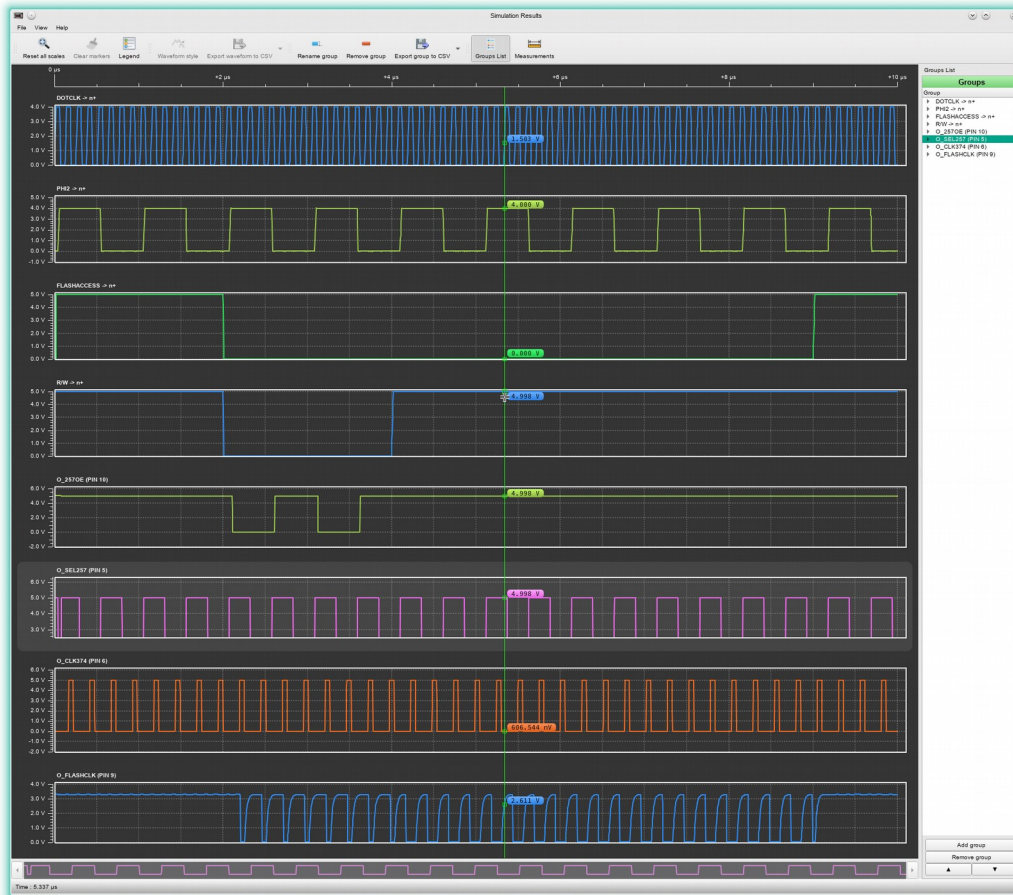
The bottom of the schematic deals with the initialization of the flash chip during reset, with obviously reset as input. The clock signals for initialization are generated by the one-shot counters of the SLG46620 and digital comparators (the digital comparators are in matrix 1, coming up) and the data signal for the flash chip is generated by the pattern generator inside the SLG46620.

Almost all logic present in matrix 0 is in use, only one LUT, one flip-flop, one counter, a pipe delay and the analog components remain available.

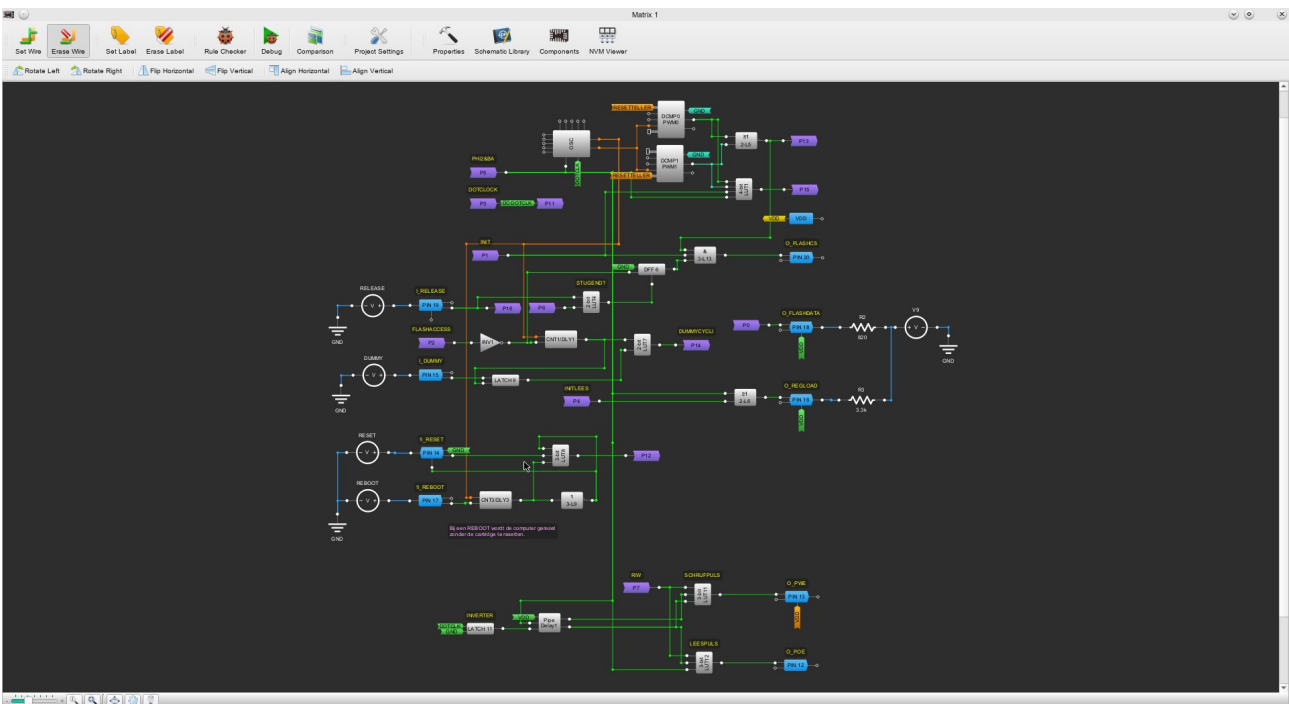
The Greenpak appears to generate the right signals. The simulation below simulates 5 consecutive flash access clock cycles. The first two are writes to the flash memory, the last three are reads from the flash memory.

You can see that:

- OE for the 74x257 is only generated during writes to the flash memory
- The flash clock is only generated during flash access
- Two flash clocks are generated per half C64 cycle. So for example when PHI2 is high (this is when the C64 CPU has the bus) there are two flash clocks while PHI2 is high, in order to send both D0..D3 and D4..D7 of the data bus to the flash memory in sequence.
- The timing for the flash memory clock is different for read and write cycles



Let's look at matrix 1.

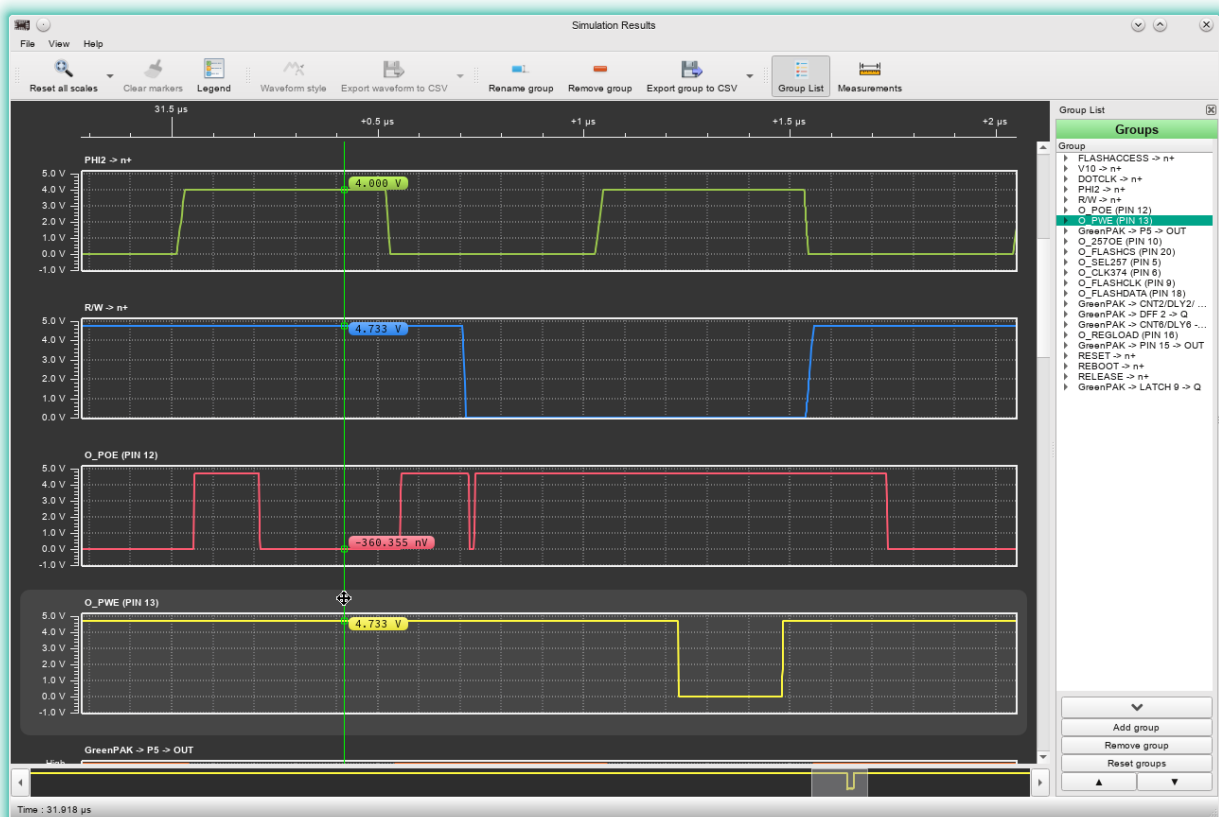


In matrix 1, the digital comparators of the SLG4620 are used to help generating clock signals during the reset sequence. You can also call them abused, since they are quite powerful devices that can generate PWM modulated signals when combined with counters, which we do not need for our purpose. This is how it works in a GreenPAK, the components are available, so better use them.

Matrix 1 also contains the reboot logic, which the programmer can use to reset the computer without resetting the cartridge. This will allow programmers, for example, to reset a Commodore 128 back into C128 mode when in C64 mode.

Lastly, in the bottom of the schematic, the OE and WE signals for the parallel memory are being generated. This is being done with the pipe delay component. The PHI2 signal is input into the pipe delay with the inverted dot clock as clock signal. PHI delayed by 2 and 4 inverted dot clocks are output from the pipe delay.

Then OE is generated by a LUT out of PHI2, the 2-dot clock delayed PHI2 and R/W. WE is generated by a LUT out of the 2 and 4 dot-clock delayed PHI and R/W. The result is that the WE and OE pulses start about 180ns after PHI2. The OE pulse ends when PHI2 indicates cycle end (plus propagation delay).



The WE pulse ends about 50ns before end-of-cycle. At that time the data bus is fully stable, and doing it this avoids a race at the end of the clock cycle to end the write pulse before the address bus starts to change (as is the case with EasyFlash). There is all the time in the world, and this will make the cartridge a lot less sensitive to SRAM speed and the speed of other components than the EasyFlash cartridge is.

OE pulses are generated both when PHI2 is low and when it is high (so 2MHz mode on the C128 is supported). WE pulses are only generated when PHI2 is high.

Decoder

A task that cannot be performed by a Greenpak very well, is decoding the PLA signals and register contents, and based on this generate the GAME and EXROM signals to the C64, and the SRAM and flash select signals to the cartridge. This will therefore be implemented by an ATF16V8.

The inputs are:

- C64mode2..0: The C64 mode bits from the Register GreenPAK
- IO2mode: The IO2 mode bit from the Register GreenPAK
- a15..a13: The upper 3 bits of the C64 address bus
- a3,a0: Another 2 bits of the C64 address bus
- IO1, IO2, ROML, ROMH: Commodore 64 PLA outputs

The outputs to be generated are:

- flashaccess: Access from the C64 to flash memory
- flashcs: Chip select signal to flash memory
- exrom, game: Discussed before
- sramcs: Chip select signal for the SRAM memory

The logic equations are:

```
FIELD c64mode = [i_c64mode2..0];
FIELD addr = [i_a15..12];

/* 8K or 16K CRT*/
o_exrom = !(c64mode:1 & i_phi2 & !addr:[D000..DFFF] #
c64mode:2 & i_phi2 & !addr:[D000..DFFF] #
c64mode:4 & i_phi2 & addr:[A000..BFFF] #
c64mode:5 & i_phi2 & !addr:[D000..DFFF] #
c64mode:6 & i_phi2 & !addr:[D000..DFFF] #
c64mode:7 & i_phi2 & addr:[A000..BFFF] #
addr:[D000..DFFF] & i_phi2 & !i_boot128);

/*16K CRT or Ultimax*/
o_game = !(c64mode:2 & i_phi2 #
c64mode:3 & i_phi2 & ( addr:[A000..BFFF] # addr:[E000..FFFF] ) #
c64mode:4 & i_phi2 & addr:[A000..BFFF] #
c64mode:6 & i_phi2 #
c64mode:7 & i_phi2 & addr:[8000..BFFF]);

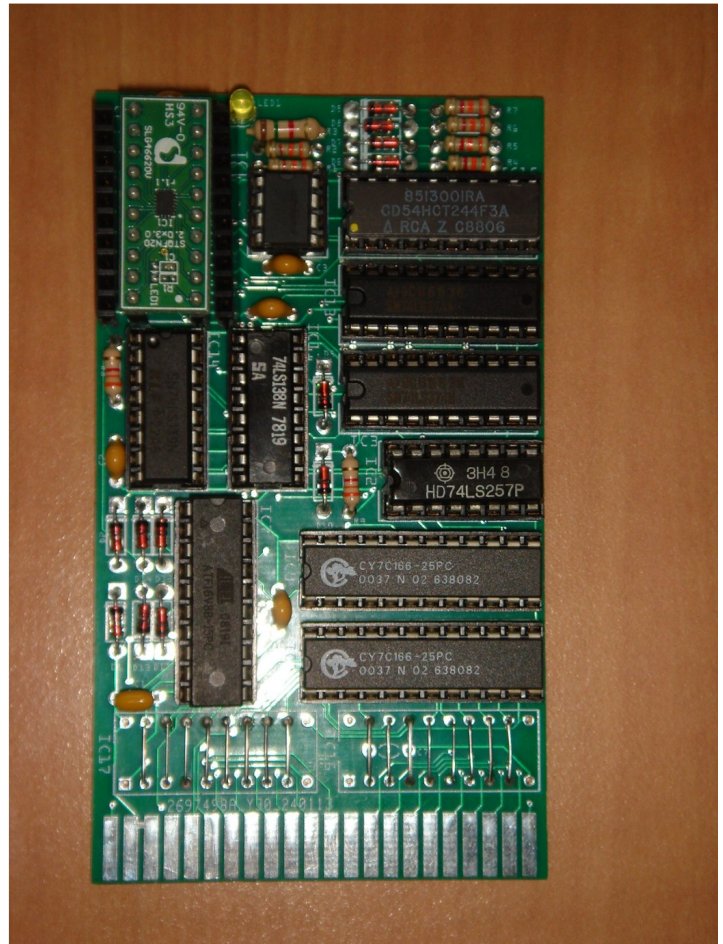
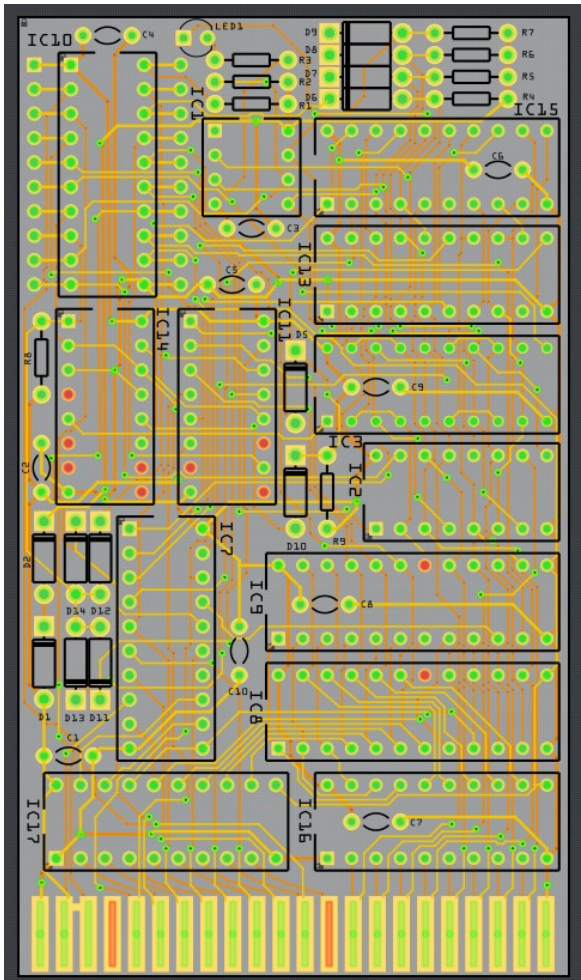
samaccess = c64mode:5 & !i_roml #
c64mode:6 & !i_roml #
c64mode:6 & !i_romh #
c64mode:7 & !i_romh #
i_io2mode & !i_io2;

o_flashsel = !(samaccess);

o_sramcs = !(c64mode:1 & !i_roml #
c64mode:2 & !i_roml #
c64mode:3 & !i_roml #
c64mode:3 & !i_romh #
c64mode:7 & !i_roml #
!i_io2mode & !i_io2);
```

Beluga cartridge PCB prototype

A prototype of the cartridge has been built, and the debug process is in progress:



As the debugging is in progress, there is not a lot to show right now. Nevertheless, this photo was taken on the battlefield right after the cartridge came alive: What the monitor is showing is X, Y and A contain values \$55 \$AA \$55. There values come from cartridge ROM. In other words, the cartridge is able to execute a ROM read command from the C64 and data can be read. While the debugging is not finished yet, all important components are working at this time.



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 automatically initialize the cartridge configuration register from flash memory. The first thing that happens is that the Beluga controller will send two times a \$FF command in order to reset the flash memory to SPI mode. Then it sends the \$EB command serially and \$00 \$00 \$00 \$00 using all 4 IO lines to start reading flash memory at position \$000000. It will read the first 8 bytes of flash memory and write them to the configuration register, i.e. 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.)

Because the \$EB command is a “quad I/O” command, the actual data will be output using all four QPI lines, so the cartridge can convert this to 8 parallel data bits for the C64 data bus.

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. To 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 $de01 ; 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.