BELUGA
SUPER GUPPY

PROGRAMMING MANUAL

Table of contents

Introducing Beluga and Super Guppy

Why do we need new cartridges?

Retro gaming is very popular nowadays. Despite having celebrated its 40th birthday, the Commodore 64 is still very popular in the year 2024, and dozes of new games are created for it each year. Many of these games are sophisticated. Because the hardware of the Commodore 64 is very well documented, modern software development tools, and especially bank switched cartridges, game developers are nowadays capable of designing games at higher technical standards than were possible in the 80s.

Cartridge hardware that has been very successful as a carrier for modern Commodore 64 games include Magic Desk, Gmod2 and EasyFlash. These cartridges allow access to significantly larges amounts of data compared to diskettes, and all this data is almost immediately accessible without any loading. Further, the Commodore 64 can execute code directly from cartridge, which allows freeing up precious main memory for more graphics and dynamic game data.

The Beluga and Super Guppy cartridges attempt to remove several limitations of current Commodore 64 cartridges, while at the same time, they attempt to keep the cost of a cartridge low. Cost is very important, the main reason why many games are released in the mentioned cartridge format is the low cost of these cartridges.

The first limitation that needs to be discussed is size. The Magic Desk and EasyFlash formats allow games up to 1MB in size. The Gmod2 cartridges allows games up to 512KB in size. Although this is large compared to historic C64 games, 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. Therefore there is a need for an even larger cartridge.

A second limitation with current cartridges is the Commodore 64 KERNAL. The Commodore 64 allows you to disable all ROMs to have access to the full 64KB of RAM. If you do this, you can also gain access to the CPU interrupt vector table at $FFFA..$FFFF to handle interrupts with very low latency, allowing advanced game effects.

However... if you try to make cartridge ROM visible, you will discover that this is impossible without making the KERNAL visible, greatly disturbing your interrupts. This also complicates running code directly from cartridge, because in that case, the KERNAL needs to handle the interrupts.

Another limitation that the Beluga and Super Guppy cartridges address is that, if you make cartridge ROM visisble, there is suddenly 24KB of ROM visible in the C64's address space. This greatly restricts your freedom in how you use the C64's memory. Although it is not so easy to see to what extends this limits game developers, there is no doubt this limits what can be done on the C64 and certainly makes developing for the C64 more difficult.

A BIGGER FLASH ROM

As discussed above, current C64 games are hitting the limits of current cartridges, and one of the goals of the Beluga and Super Guppy cartridges is simply more space. Space alone will allow for more graphics and music and other content and thus allow for more sophisticated Commodore 64 games.

Idea: Bigger parallel flash ROM

The most basic idea to get a bigger cartridge is to simply increase the size of the ROM. However, there is a good reason why the current cartridges have the sizes they have. 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: €7.99 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: €0.33 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 €0.33 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 here 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 CPLD: 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 CPLD prices to skyrocket and it looks like the Gmod3 cartridge will not be economically feasible anytime soon.

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 pheripheral 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. There is a need 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. The Super Guppy cartridge combines 8MB of QPI flash memory with 128KB of parallel flash memory.

However, since we can boot the C64 from serial flash memory, we do not need parallel ROM. 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.

The Beluga Cartridge combines 16MB of flash memory with 16KB of parallel SRAM.

The parallel memory on both cartridges is not strictly necessary, in fact, if you unplug the parallel memory they will still work. However, the parallel memory makes them more powerful.

Back to the economic side of the equation

After the discussion about the technical feature, let's back to the cost question. Because the cost of the storage medium puts a lot of pressure on the feasibility to release a C64 game in physical form. How low can we get the price of a cartridge?

I will focus on the material costs of the cartridges. Labour also is a factor in the price of a cartridge, but the cost of labour is highly dependent on production volume and will decrease over time. You can expect that the labour costs will initially be significant, but as cartridges will be produced in larger volumes and for a longer time, labour costs will drop.

An important factor for the cost of materials of a cartridge is the source of parts: Obtaining parts from official channels is usually most expensive. This is especially true for traditional cartridges, which use parallel bus components, which are legacy components for manufacturers. The costs can drop by buying parts from electronic component traders. Lowest costs can be achieved by obtaining parts from AliExpress and other non-professional sources.

When buying components from official sources, the traditional cartridges become rather expensive. EasyFlash 1 is by far the most expensive cartridge, because it needs two parallel ROMs and a parallel SRAM. Magic Desk and Gmod2 are less expensive to produce. The cost of the ROM may weight on Magic Desk in its largest 1MB size, but in smaller sizes the price of Magic Desk is difficult to beat. The cost of a Gmod2 is difficult to beat as well, because it needs a single parallel ROM.

Parallel ROMs are quite easy to obtain from professional IC traders at significantly lower prices than the official sources. In such case the costs of traditional cartridges drop a lot. A problem with IC traders is that if you try to mass produce a cartridge, the factory normally will want to do the purchasing, and it prefers to purchase through official channels to make some invisible margin.

The Super Guppy and Beluga cartridges are made of relatively inexpensive components. Serial flash memory is not a legacy product and made by many vendors, so competition keeps the price down, even though official channels. The Beluga controller, which is the heart of these cartridges is an inexpensive GreenPAK. The cartridges have more 74 series TTL chips than traditional cartridges, but TTL chips are very inexpensive.

Therefore, if components have to be obtained through official channels, this benefits the Super Guppy and Beluga cartridges in comparison to traditional cartridges.

Both cartridges are definitely less expensive to produce than an EasyFlash cartridge. The difference is significant when buying components from official sources, but even at IC trader prices, the Beluga still beats the EasyFlash and the Super Guppy is even less expensive.

As discussed, Magic Desk and Gmod2 are such simple designs, that it is difficult to beat them, but when buying from official sources, a Super Guppy might actually be able to beat them, because it doesn't need a big parallel ROM. Magic Desk and Gmod2 will be less expensive to produce if their components are sourced from IC traders.

When looking at realistic prices for components, partially from official sources, partially from traders, the materials cost of a Super Guppy can be below €4 incl. VAT, thus parts and PCB. Labour, transport costs, customs costs are not included. A Beluga can be below €5,50.

When the absolutely lowest possible cost needs to be achieved, a Super Guppy without the parallel ROM can be considered. The parallel ROM on the Super Guppy, even though it is just 128KB, is still the most expensive component on the cartridge. A Super Guppy without parallel ROM should have a materials cost below €3 and may actually beat the Magic Desk and Gmod2 cartridges.

Now that we have discussed the background behind the cartridges, let's get into some coding!

Serial flash memory

The core technology of the Super Guppy and Beluga cartridges is the QPI flash memory. Conventional cartridges occupy an 8KB or 16KB memory window starting at memory location $8000. The serial memory of the Super Guppy and Beluga cartridges is quite different from this: You will interact with the serial memory with just 3 registers. These registers are located at $DE00, $DE01 and $DE02 in the IO1 window.

Selecting and deselecting

Every operation with a serial flash memory start with selecting the flash memory. This is a physical pin on the flash memory that goes from 1 to 0. After the flash memory has been selected, it expects a command to sent. The command may need several more bytes to be sent and then the flash memory may send a response.

On the Super Guppy and Beluga cartridges, you do not need to select the flash memory manually. The flash memory will automatically become selected once you perform a write to $DE00, $DE01 or $DE02. A read to these memory locations will not cause the flash memory to become selected, if you read from these memory locations while the flash memory is not selected, you will read $FF.

If you read or write to $DE00, you will read or write a byte to the flash memory normally. On the other hand, a read or write to $DE01 will also read or write a byte to the flash memory, but after the byte has been read/written, the cartridge will deselect the flash memory.

For example, the flash memory has a NOP (no operation) command $00. To send the NOP command to the flash memory, you can use the following instructions:

```
LDA #$00
STA $DE01
```

The flash memory will become automatically selected, and the command code $00 is written to the flash memory. Because $DE01 was used, the flash memory will become automatically deselected.

Reading from flash memory

The flash memory has two commands that are suitable for reading from flash memory: $0B and $EB. The main difference between these commands is the M byte, which allows for repeated reads, which we will discuss. Let's say we would like to read 100 bytes starting at location $AABBCC in the flash memory and store these at $C000 onwards in C64 memory. When using the $0B command this would work as follows:

```
    LDA #$0B     ; Command
    STA $DE00
    LDA #$AA     ; High byte of address
    STA #$DE00
    LDA #$BB     ; Mid byte of address
    STA $DE00
    LDA #$CC     ; Low byte of address
    STA $DE00
    LDA $DE02    ; Dummy cycles
    LDX #0
@1: LDA $DE00    ; Read a byte
    STA $C000,X  ; Store the byte
    INX
    CPX #99
    BNE @1
    LDA $DE01    ; Read the last byte
    STA $C000,X  ; Store the byte
```

Ignore the dummy cycles for now, we will discuss them in a bit. As you can see the above loop can just repeatedly read data from the flash memory without increasing a pointer. Therefore a loop like this to read data from a cartridge will be faster than on a conventional C64 cartridge.

If we would use the $EB command, the only thing that chances for now, is the command itself:

```
    LDA #$EB     ; Command
    STA $DE00
    LDA #$AA     ; High byte of address
    STA #$DE00
    LDA #$BB     ; Mid byte of address
    STA $DE00
    LDA #$CC     ; Low byte of address
    STA $DE00
    LDA $DE02    ; Dummy cycles
    LDX #0
@1: LDA $DE00    ; Read a byte
    STA $C000,X  ; Store the byte
    INX
    CPX #99
    BNE @1
    LDA $DE01    ; Read the last byte
    STA $C000,X  ; Store the byte
```

However, the $EB command allows for a trick: The M byte. The M byte is part of the dummy cycles, and allows for continuous read. What does this mean? First we have to tell the flash memory that we would like to do a continuous read:

```
    LDA #$EB     ; Command
    STA $DE00
    LDA #$AA     ; High byte of address
    STA #$DE00
    LDA #$BB     ; Mid byte of address
    STA $DE00
    LDA #$CC     ; Low byte of address
    STA $DE00
    LDA #$20     ; Continuous read!
    STA $DE02    ; Dummy cycles
    LDX #0
@1: LDA $DE00    ; Read a byte
    STA $C000,X  ; Store the byte
    INX
    CPX #99
    BNE @1
    LDA $DE01    ; Read the last byte
    STA $C000,X  ; Store the byte
```

By writing the value $20, we indicate the flash memory that we would like to do a continuous read. The rest of the read continues as planned. But suppose we would like to do another read after this one, we would like to read 2 bytes at $BBCCDD in flash memory. We can now skip the command as part of our read!

```
    LDA #$BB     ; High byte of address
    STA #$DE00
    LDA #$CC     ; Mid byte of address
    STA $DE00
    LDA #$DD     ; Low byte of address
    STA $DE00
    LDA #$20     ; Continuous read!
    STA $DE02    ; Dummy cycles
    LDX #0
    LDA $DE00    ; Read first byte
    LDX $DE01    ; Read second byte an deselect
```

After this, we can do another read without sending a command. However, if we are done with our reads, instead of writing $20, we can do a read again (LDA $DE02) to indicate that this is our final read.

Most of the time, games will just read from cartridge, and only in specific situations, such as saving a game, they will write to flash memory. Therefore the use of continuous reads is highly recommended! By using continuous reads, you can minimize the overhead of reading data from flash memory and achieve the highest possible performance.

Dummy cycles... at this point you must be very curious what these are... A serial flash memory can run at very high clock speeds well over 100 MHz. But in order to support those clock speeds, the flash memory requires some dummy reads or writes in order to introduce some latency. The number of dummy cycles is configurable, but for the Super Guppy and Beluga cartridges, the number of dummy cycles that we use is 4 flash cycles, or 2 C64 cycles.

This means that after writing the address bytes to flash memory, we would need to perform two reads or writes to $DE00 in order to execute those dummy cycles. You can either read or write in order to perform a dummy cycle, therefore:

```
LDA $DE00
LDA $DE00
```

... would perform the required dummy cycles. Now, two instructions to perform 4 dummy cycles is overhead, so why don't we configure the flash memory for 2 dummy cycles, so we would need only one LDA? The troubles is, only a few flash memory manufacturers support 2 dummy cycles, and we do not want a vendor lock-in. Instead of putting pressure on the programmer whether to support all types of flash memories, the cartridges standardize on 4 dummy cycles and solves the matter in hardware.

This is where register $DE02 comes in. If you read/write $DE02, after completing your read/write, the hardware of the cartridge performs 2 dummy flash cycles in the background. Therefore, the above two LDAs can be replaced with:

```
LDA $DE02
```

... as was done in the examples.

So what you need to remember, is that a single read or write to $DE02 will send the required amount of dummy cycles.

The M byte is a special dummy cycle: The flash memory counts it as a dummy cycle, but you also tell the memory that you want to do a continuous read.

To conclude, both the $0B and $EB commands can be used to read from flash memory, but $EB is more powerful. If you do not use a continuous read, the $EB command takes the same amount of time as the $0B command, however, thanks to the M byte, the $EB command allows you to do continuous reads and therefore reduce the overhead to read from the flash memory.

WRITING TO FLASH MEMORY

In a flash memory, an unprogrammed bit returns a 1, while a programmed bit returns a 0. By writing to flash memory, an unprogrammed bit can become a programmed bit, but a programmed bit cannot become an unprogrammed bit by a simple write.

In order to make programmed bits unprogrammed, the flash memory needs to be erased. A flash memory is erased one sector at a time. The size of a sector in the Beluga and Super Guppy cartridges is 4KB. This is much lower than the 64KB used by the popular EasyFlash cartridges, and makes working with flash memory easier. For example, it is possible to backup and restore a flash sector in C64 memory to change part of it.

In the Beluga and Super Guppy cartridges, a sector is further divided in pages with a size of 256 bytes. Therefore a sector contains 16 pages.

Any command that writes to flash memory is ignored unless it is preceded by a write enable command $06. This command must be issued again before every write, it is not acceptable to send it just once.

A write command is executed in the background. Once you issue a write command, the flash memory is busy and will only allow you to read the status bytes while the operation is in progress. You can check if a write is in progress by issuing command $05 and then you can repeatedly read the value of status register 1. Bit 0 indicates if a write is in progress.

A 4KB sector can be erased by issuing a $20 command. This command is followed by a sector address in flash memory. You will write a full 24-bit address, but only the first 12 bits indicate the sector to be erased.

We are ready for an example: Suppose we want to erase a sector in flash memory starting at address $ABC000. This means we will erase flash memory from $ABC000 to $ABCFFF. The code needed to perform this operation is:

```
    LDA #$06    ; Enable write command
    STA $DE01   ; This is a one byte command, so deselect immedeately
    LDA #$20    ; Erase a sector command
    STA $DE00
    LDA #$AB    ; Send the high byte of the sector address
    STA $DE00
    LDA #$C0    ; Send the mid byte of the sector address
    STA $DE00
    LDA #$00    ; Send the low byte of the sector address
    STA $DE01   ; And deselect
    LDA #$05    ; Send the read Status byte command
    STA $DE00
    LDA #$01    ; Check bit 0
@1: BIT $DE00
    BNE @1
    BIT $DE01  ; Deselect the flash memory
```

After executing this code, all bytes from $ABC000 to $ABCFFF in the flash memory will read $FF.

An erasing is a slow operation, it make take up to 0.4s to get a sector erased.

After flash memory has been erased, it can be rewritten. Writing to flash memory is done page per page. You can start a write at any random place in flash memory, but a single write command cannot cross the boundaries of a page. This means that you can program a maximum of 256 bytes with a single write command.

Command $02 is used to perform writes within a page. After sending the command you write the address where you would like to write in the next 3 bytes. After that, you write up to 256 bytes. If you exceed a page boundary, the address will wrap around. For example, if you write a byte to flash address $ABCDFF and then another byte, this byte will be written to address $ABCD00, not $ABCE000.

In the following example, we will write 256 bytes located in C64 memory at $C000 to the flash memory starting at address $ABCD00.

```
    LDA #$06    ; Enable write command
    STA $DE01   ; This is a one byte command, so deselect immedeately

    LDA #$02    ; Erase a sector command
    STA $DE00
    LDA #$AB    ; Send the high byte of the sector address
    STA $DE00
    LDA #$CD    ; Send the mid byte of the sector address
    STA $DE00
    LDA #$00    ; Send the low byte of the sector address
    STA $DE00
    LDX #$00
@1: LDA $C000,X
    STA $DE00
    INX
    CPX #$FF
    BNE @1
    LDX $C0FF   ; Last byte
    STA $DE01   ; Make sure flash gets deselected

    LDA #$05    ; Send the read Status byte command
    STA $DE00
    LDA #$01    ; Check bit 0
@2: BIT $DE00
    BNE @2
    BIT $DE01  ; Deselect the flash memory
```

Note that a 0 bit means a bit in the flash memory should be programmed, while and 1 means a bit should keep its current value. Therefore the result is that the value of a byte in flash memory is always a logical AND of the old value and the value that you write.

Note that there are no dummy cycles when writing to flash memory, because the programming is done in the background. A flash write can take up to 150us for a single byte and up to 50ms for a full page.

6502 instructions to avoid on serial ROM registers

Some 6502 instructions generate extra cycles. This means an instruction can for example do two reads, while you might expect a single read. With the serial ROM doing two reads may have undesired effects. The following 6502 instructions should not be used on the serial ROM registers $DE00..$DE02:

```
STA absolute,x
STA absolute,y
STA (indirect,x)
STA (indirect),y
```

While you might expect these instructions just do a single write these instructions perform a read before doing the write that you expect.

```
DEC absolute
DEC absolute,X
INC absolute
INC absolute,X
```

While you might expect these instructions will do a read, then a write, they actually perform a read and then two writes.

Things to know

- The POKE instruction in BASIC cannot be used to write a value to the serial flash memory registers, because it uses the STA absolute,y instruction. PEEK can be used to read bytes from the serial flash memory.
- If you read from serial flash memory registers if the memory is not selected, you will read $FF
- If you read from serial flash memory registers when the memory expects a write, you will write $FF and the read will return $FF as well.
- You should not write to the serial flash memory registers when the memory expects a read.
- You should not use the serial flash memory in interrupt handlers, because a command might be in progress. Or alternatively, you should only use the serial flash memory in interrupt handlers.

QPI and SPI mode

Until now, all examples that were shown, have assumed that the flash memory is in QPI mode. However,. the serial flash memory has actually two modes: QPI and SPI. The Beluga and Super Guppy cartridges have been designed for QPI mode and therefore I do recommend that application software always uses the flash memory in QPI mode, however, after a reset, the serial flash memory is actually in SPI mode. Therefore we need to discuss SPI mode a bit.

In QPI mode, the serial flash memory accepts 4 bits at a time, while in SPI mode, it accepts 1 bit at a time. Because the cartridges perform 2 clock cycles of flash in 1 Commodore 64 clock cycle, in QPI mode you can send 8 bits per write to the flash memory, while in SPI mode, you can send 2 bits at a time.

In SPI mode, if you write data to the flash, bit 0 and bit 4 contain the data, the other bits are ignored. If you read data from the flash, the data is returned in bit 1 and bit 5, the other bits will read 1.

The flash memory can be switched to QPI mode by sending command $38, and it can be switched to SPI mode by sending command $FF.

Suppose the flash memory is in SPI mode, and we want to switch to QPI mode, then we need to convert the command into groups of 2 bits. $38 = %00 11 10 00. Because of bit 0/4 we can literally take these groups of 2 bits, and use them as nibbles:

```
LDA #$00
STA $DE00
LDX #$01
STX $de00
LDX #$10
STX $DE00
LDA #$00
STA $DE01
```

After this, the flash is in QPI mode, so we can now send commands as a single byte. In case we would like to return to SPI mode, we can use:

```
LDA #$FF
STA $DE01
```

After this, the flash is in SPI mode.

Commands in SPI mode are always 1 bit at a time (2 bits seen from the C64). However, depending on the actual command, additional bytes that you either write or read from the flash memory might still be expected as 4 bits at a time (thus 8 bits seen from the C64). This depends on the exact command you use, but this is outside the scope of this manual, if you want to know more information about this, you will have to read the datasheet of the serial flash memory.

Flash commands not to be used by application software

The flash commands that are documented in this manual should be what you need to make your creative dreams come true. However, if you read the datasheets of the serial flash memory, you will see that they support quite a bit more commands. Some of these additional commands can be used, but there are also commands that I do not want you to use in your game or program. These commands may cause the cartridge to malfunction, may make it unnecessarily complex to emulate the cartridge, or create incompatibilities between different flash memory vendors.

Your game/program shall not use the following commands:

Category	Commands	Explanation
Dual SPI commands	$3B $92 $BB	Dual SPI commands to read/write 2 bits at a time don't make any sense for our cartridges and make emulation more difficult.
Write to status register	$01 $31	Writing to the status registers is unnecessary during normal operation and can cause malfunction of the cartridge hardware.
Block erase	$52 $D8	Block erase is not supported by all flash vendors
Secured OTP memory	$B1 $C1 $42 $48	Secured OTP memory might be used by flash programs in the future to store metadata. Therefore application software should not use the secured OTP memory for itself. This memory is implemented in a different way between flash memory vendors.
Security registers	$2B $2F	Touching the security registers makes no sense for application software. Security registers are implemented in a different way between flash memory vendors.
Serial flash discovery parameters	$5A	JEDEC SFDP provides software information about the current flash memory and allows software to support flash memories from different vendors. However, the information inside the SFDP tables is not of any use for Beluga / Super Guppy application software.
Octal word read quad I/O	$E3	This command is not supported by all flash vendors.
Sector lock/unlock	$36 $39 $3D $7E $98	Locking sectors is not the task of application software, it could be done by flashing software. Furthermore, these commands are different between various flash vendors.

Configuration register and parallel ROM (Super Guppy)

In addition to the 8MB serial ROM, the Super Guppy cartridge also has a traditional 128KB parallel ROM. This ROM is optional and if you do not need it, it can be removed for maximum cost savings, making the Super Guppy a cartridge with an extremely low material cost. However, the cartridge is quite a bit more powerful if the parallel ROM is installed.

The configuration register

The parallel ROM works exactly as on other bank switched cartridges: There is a configuration register that allows you to set the current bank, and the register also has bits to set the state of the GAME and EXROM lines. The configuration register is located at memory location $DE04 (56836). The meaning of the bits is as follows:

Bit #	Description
0-2	**Parallel ROM bank switch**
3	**SAM/Parallel** 0 = $8000..$9FFF is SAM from serial flash 1 = $8000..$9FFF is parallel flash
4	**GAME** 0 = GAME is pulled low 1 = GAME is not being pulled
5	**EXROM** 0 = EXROM is pulled low 1 = EXROM is not being pulled
6	**KERNAL replacement** 0 = No KERNAL replacement 1 = Bank 7 of parallel flash rom visible at $E000..$FFFF
7	**Cartridge LED** 0 = Led is off 1 = Led is on

Because the parallel ROM is 128KB in size, the cartridge has 8 16KB banks. Bits 0-2 set the currently visible bank number and speak for themselves.

Bit 3 controls whether the serial ROM or the parallel flash ROM is visible in the ROML memory region from $8000..$9FFF. Mapping the serial flash ROM into ROM is useful for booting and SAM speedcode, which we will discuss further on in this manual. Setting bit 3 to 1 makes the parallel flash ROM visible in the ROML memory region (if active via GAME and EXROM).

The parallel flash ROM is always visible in the ROMH memory region (if active via GAME/EXROM). Address $1F00..$1FFF of the current bank is also visible in the IO2 memory region from $DF00..$DFFF (if IO is visible via $01).

Bit 4 and 5 control the state of the GAME and EXROM lines. They work exactly like on other cartridges:

GAME	EXROM	Description
0	0	16KB cartridge mode
0	1	Ultimax mode
1	0	8KB cartridge mode
1	1	Normal mode (cartridge invisible except through IO1/IO2)

Bit 6 activates an unique feature of the Super Guppy cartridge: If you set this bit to 1, the upper 8KB of bank 7 becomes visible at $E000..$FFFF. This replaces the KERNAL ROM in memory.

Bit 7 controls the led of the cartridge.

KERNAL REPLACEMENT

Suppose you have hidden all ROMs and now want to make the cartridge ROM visible. This is not possible without making the KERNAL visible as well. However, if you make the KERNAL visible, the interrupt vector table at $FFFA..$FFFF also suddenly points to the KERNAL interrupt handlers. If you have timing senstive raster interrupts, this can ruin your day.

The Super Guppy cartridge solves this problem with its KERNAL replacement feature. If you set bit 6 of the configuration register, the upper 8KB of bank 7 becomes visible at $E000..$FFFF independent of the value of the processor port at $01. This is for read operations only, write operations to $E000..$FFFF still go to RAM under the KERNAL.

Cartridge ROM at $E000..$FFFF is useful for interrupt handlers, but a great place for trampolines: Routines that switch to the correct bank and then jump to the routine inside the actual bank. This allows you to do call your routines from RAM without worrying which bank is currently visible, or to do far calls from one ROM bank to another.

READABLE CONFIGURATION REGISTER

An unique feature of the Super Guppy is that you can't just write to the configuration register, but also read from it.

FAR CALL EXAMPLE

Suppose you are executing code from cartridge ROM in bank 1 and want to call a subroutine in bank 2. With the Super Guppy you can do this like this:

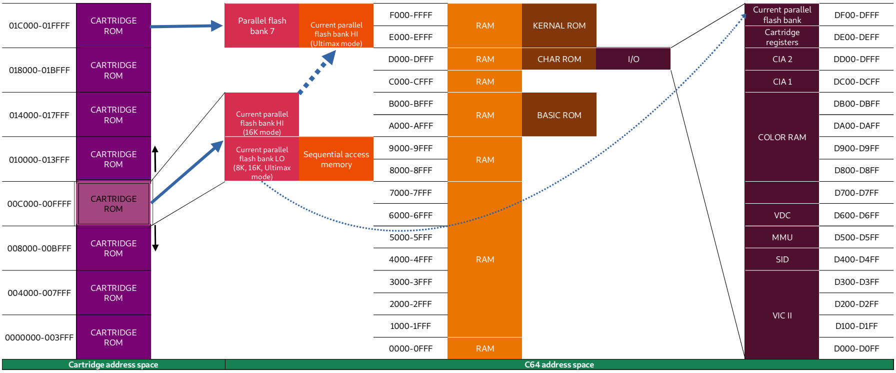
```
; Bank 1
;
 .ORG $8000

    ....
    blahblah
    JSR routine_far
    blahblah

; Bank 2
;
.ORG $8000

routine_near:
     blahblahblah
     rts

; Trampoline
;
.ORG $E000

routine_far:
    STA zeropage_temp
    LDA $DE04
    PHA
    LDA #2|8
    STA $DE04
    LDA zeropage_temp
    JSR routine_near
    PLA
    STA $DE04
    RTS
```

The trampoline routine_far takes advantage of the readable configuration register to push the current bank on the stack, call the routine in bank 2, and the pop the bank again before returning.

This way routines that have a far call trampoline can be called from anywhere in the cartridge. The memory at $E000..$FFFF allows for a lot of trampolines, making the Super Guppy very suitable to execute code directly from cartridge.

Super Guppy memory Map

A schematic drawing of the Commodore 64's memory map with an inserted Supper Guppy:



Writing to parallel flash

The Super Guppy generates write cycles to the parallel flash ROM in Ultimax mode. Therefore the parallel ROM can be reprogrammed from the C64. This should be done according to the algorithm documented in the datasheet of the parallel ROM. The Super Guppy does not have a circuit to generate 12V to support 12V flash ROMs, but has a header so 12V can be applied externally in order to use a 12V flash ROM.

I expect that developers of application software will want to use the serial flash ROM for things like save-on-cartridge and high scores, and the parallel ROM will be written mainly by flasher software, not application software.

Therefore I am not defining an API such as the EasyFlash EAPI at this time and not documenting this in more detail. Just know that if you write a byte to the parallel flash memory when you are in Ultimax mode, the cartridge will deliver that byte to the parallel ROM.

Configuration register and parallel SRAM (Beluga)

In addition to the 16MB serial ROM, the Beluga cartridge also has a traditional 16KB parallel SRAM. While the Beluga can in theory work without this SRAM just like the Super Guppy can work without parallel ROM, expect anyone who doesn't need parallel memory will use a Super Guppy without parallel ROM, so in practise a Beluga will always have the parallel SRAM installed.

A key feature of the Beluga is that it doesn't have traditional GAME, EXROM control bits, but has a decode that will operate GAME and EXROM dynamically. Therefore, instead of GAME/EXROM control bits, the Beluga has modes.

Configuration register

The configuration register of the Beluga is located at $DE03 (56835) and the bits have the following meaning:

Bit #	Description
0-2	**C64 memory configuration mode.** Selects a Commodore 64 memory configuration 0 = Normal mode, cartridge is invisible except via registers at $DExx 1 = 8KB SRAM read-only at $8000..$9FFF 2 = 16KB SRAM read-only at $8000..$BFFF 3 = 8KB SRAM writable at $8000..$9FFF, 8KB SRAM writable at $E000..$FFFF 4 = 8KB SRAM read-only at $A000..$BFFF 5 = 8KB SAM at $8000..$9FFF 6 = 16KB SAM at $8000..$BFFF 7 = 8KB SRAM writable at $8000..$9FFF, 8KB SAM at $A000..$BFFF
3	**IO2 mode** 0 = $DFxx is SRAM 1 = $DFxx is SAM
4	**Commodore 128 startup mode** 0 = Commodore 128 starts in Commodore 64 mode 1 = Commodore 128 starts in Commodore 128 mode
5	Unused
6	Unused
7	**Cartridge LED** 0 = Led is off 1 = Led is on

Bits 0-2 set the active memory configuration of the Beluga cartridge. We will discuss these in detail shortly. Bit 3 controls whether the IO2 region at $DF00 will see SRAM (in which case it is a mirror of $9F00..$9FFF if SRAM visible at $8000), or whether it will see sequential access memory.

Bit 4 allows you to control what the cartridge should do on a Commodore 128: Should it start in C64 mode or Commodore 128 mode. Bit 7 allows you to switch the cartridge led on or off.

BELUGA MODE 0

In mode 0 the Beluga cartridge will not be mapped into memory except at IO1 and IO2. It is equivalent to a situation where EXROM and GAME are set high on a conventional cartridge. The only difference is that a Commodore 128 will still start in C64 mode if bit 4 has been set to 0.

This mode is much more useful than on conventional cartridges, since you have full access to the serial flash ROM through IO1, and SRAM or SAM at IO2.

BELUGA MODE 1

In Beluga mode 1, 8KB of read-only SRAM will be mapped at $8000..$9FFF. This mode is equivalent to the 8KB cartridge mode of conventional cartridges and allows the Beluga to emulate classic 8KB cartridges. Writes to SRAM will go to the C64 under the SRAM, just like a classic cartridge. This mode is also useful if you would like to call the BASIC ROM from code stored into the SRAM.

BELUGA MODE 2

In Beluga mode 2, 16KB of read-only SRAM will be mapped at $8000..$BFFF. This mode is equivalent to the 8KB cartridge mode of conventional cartridges and allows the Beluga to emulate classic 8KB cartridges. Writes to SRAM will go to the C64 under the SRAM, just like a classic cartridge. Since you have all of the 16KB of SRAM visible this mode is quite useful for executing code that you have stored into the SRAM.

BELUGA MODE 3

In Beluga mode 3, the SRAM becomes writeable. You will have 8KB SRAM mapped at $8000..$9FFF and another 8KB mapped at $E000..$FFFF. Because Ultimax mode is used to make this mode happen, you cannot hide the SRAM using the processor port at $01, the SRAM is visible regardless of the state of $01.

This mode allows the Beluga to do a few things conventional cartridges cannot do. First of all, because the SRAM is writeable, it works like a 16KB memory expansion of the Commodore 64. You can use it to run code outside C64 memory and because the memory is writeable, it is possible to run self-modifying code outside C64 memory.

Because 8KB of the SRAM is mapped at the KERNAL region at $E000..$FFFF, the Beluga can replace the C64 KERNAL in memory. You can just load a KERNAL from the flash ROM or even diskette and install it into the SRAM, and there you go. This is very useful for testing KERNALs.

Mode 3 also allows you to switch between visible and invisible SRAM without messing up your interrupt handlers, because you have control over the 6502 interrupt vector table at $FFFA..$FFFF. On a conventional cartridge, making the cartridge visible in memory also makes the KERNAL visible, but on the Beluga you can switch to mode 3 and still execute your own interrupt service routines.

Beluga mode 4

Mode 4 of the Beluga maps 8KB of SRAM at $A000..$BFFF. This mode is very useful for replacing the C64 BASIC interpreter, something conventional cartridges cannot do. It allows cartridges that provide a more capable BASIC or even a completely different programming language, while still being able to use the bottom 40KB of memory for user programs, just like is the case with CBM BASIC.

A lot of Commodore 64 cartridges provide extensions to CBM BASIC. These cartridges work by hooking the vectors at $0300. Because conventional C64 cartridges occupy the ROM region at $8000..$9FFF when visible, these cartridges need to hide/unhide cartridge ROM continuously, often for every single byte they read from BASIC memory. Therefore these cartridges can slow down BASIC by multiple tens of percents.

With Beluga mode this is no longer necessary, because the BASIC interpreter at $A000..$BFFF itself can modified, which can see all of the BASIC memory. This interpreter then only needs to make additional cartridge memory visible if it wants to execute extensions. Therefore the Beluga makes cartridges possible that extend CBM BASIC, without any measurable slowdown.

Even if you do not want to replace the BASIC interpreter, the ability to have 8KB SRAM visible at $A000 makes you more flexible to use all of the C64's memory effectively.

Beluga mode 5

Beluga mode 5 maps 8KB of sequential access memory to the ROML region at $8000..$9FFF. Mode 5 is very important for booting a Beluga cartridge, since the Beluga does not have any parallel ROM on board.

Mode 5 is also useful for running SAM speedcode. The ROML region is 2MHz capable on the Commodore 128, so you can execute speedcode faster in the ROML region than in the IO2 region at $DF00.

Beluga mode 6

Beluga mode 6 gives you 16KB of sequential access memory into the ROML/ROMH regios. Since 16KB of sequential access memory is likely more than needed, mode 6 is currently not the most useful mode. It is possible that I will change the functionality of mode 6 in the future into something more useful.

Beluga mode 7

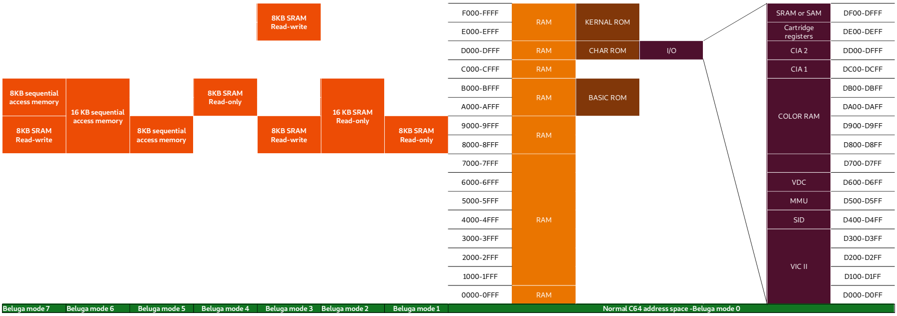
Beluga mode 7 allows you to work with a mix of SRAM and sequential access memory. You will have 8KB of SRAM at $8000 and 8KB of sequential access memory at $A000. Like mode 5, you can use mode 7 to run SAM speedcode at 2MHz.

Readable configuration register

Just like the Super Guppy, you can't just write to the configuration register, but also read from it.

Beluga memory maps

A schematic drawing of the possible memory maps of the Commodore 64 with an inserted Beluga is as follows:



Sequential access memory

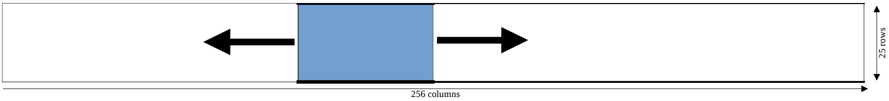
The Super Guppy and Beluga cartridges have the capability to map serial memory into the C64's address space as sequential access memory. A sequential access memory is basically a mirror of register $DE00 to an entire memory block. The address bus is ignored.

So, if you have sequential access memory mapped at $8000, instead of reading from $DE00, you can also read from $8000. But you can also read from $8001, or $8002 and so fort. Every read operation returns the next byte from serial flash memory, regardless of which address you read.

This feature allows the Super Guppy and Beluga to boot without any parallel ROM installed, but is also useful for implementing speedcode.

Example: A scrolling level

In order to better understand what you can do with SAM speedcode, let's describe an example. Suppose you have a game that runs in text mode with a level that is 256 columns wide and 25 rows high. The C64 will display a 40x25 window out of this level:



Let's assume our level contents are quite trivial, column 0 contains character $00, column 1 contains character $01 and so on. We can take advantage of SAM speedcode to draw our level very fast, but in order to do so, we need to convert our level data into speed code. This is a wasteful use of flash memory, but we have quite a bit of space in flash memory.

To convert our first line into speed code, we do:

```
LDA #$00
STA $0400,y
INY
LDA #$01
STA $0400,y
INY
LDA #$02
STA $0400,y
INY
LDA #$03
STA $0400,y
INY
;...
LDA #$FF
STA $0400,y
```

Let's assume this code is stored in flash memory at address $123456.

Let's use speedcode at $DF00 (only possible on the Beluga). Let's use Beluga mode 5. This means the KERNAL region from $E000..$FFFF is SRAM, let's place an RTS at $E000:

```
.org $E000

    LDA $DE01
    RTS
```

The horizontal scrolling position is stored in the memory location indicated by “horizontal”. Since each write from speedcode will require 6 bytes we first need to compute $123456 + 6 * horizontal. We will start with 6 * horizontal, which is a 16-bit value and store this into “flash_offset”.

```
draw_line_0:
    ; Initialize high bytes
    LDA #0
    STA flash_offset+1
    STA tmp+1

    ; Multiply by 2 and create a copy in tmp
    ASL
    ROL flash_offset+1
    LDX flash_offset+1
    STX tmp+1
    STA tmp

    ; Multiply another time by 2
    LDA horizontal
    ASL
    ROL flash_offset+1

    ; Add tmp to the result to get a multiply by 6
    CLC
    ADC tmp
    STA flash_offset
    LDA tmp+1
    ADC flash_offset
    STA flash_offset
```

Send the command for a flash read:

```
    LDA #$EB
    STA $DE00
```

Now add $123456 and use the stack:

```
    CLC
    LDA #$56
    ADC flash_offset
    PHA
    LDA #$34
    ADC flash_offset+1
    PHA
    LDA #$12
    ADC #$00
```

Send the address to flash memory and dummy cycles:

```
    STA $DE00
    PLA
    STA $DE00
    PLA
    STA $DE00
    BIT $DE02
```

And now, jump to the SAM speedcode. We want to draw 40 bytes. 256 - (6 * 40) = 16, so we need to jump to $DF10:

```
    LDY #$00
    JMP $DF10
```

Now, 40 bytes of level data will be written to the screen buffer at $0400 at high speed. Note that the multiply by 6 will only have to be recomputed when the screen scrolls, so it is not really part of the timing-critical drawing code. The overhead can be slightly further reduced by using the continuous read mode of the flash memory.

With this method we can write any number of bytes within the $0400 page, depending on the jump address and the initial value of the Y register. Therefore, this method is also suitable for partial screen updates.

6502 PHANTOM CYCLES

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 speedcode, 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, it is probably a good practise to fill dummy cycles with a NOP byte.

For example if you would like to do a PHA in your SAM speedcode, code instead:

```
PHA
NOP
```

Not all phantom cycles are a problem for SAM speedcode. For example, instructions that should be avoided on the serial flash registers at $DE00 can actually be safely used in SAM speedcode, because these instructions put their operand on the address bus, and the operand is not located in sequential access memory, so there is no dummy read from flash memory. Only instructions that generate phantom cycles that put the instruction address on the address bus need careful attention.

These instructions are:

```
PHA/PLA/PHP/PLP
CLC/SEC/CLI/SEI/CLV/CLD/SED
RTS
TXS/TSX
ROL A/ROR A/ASL A/LSR A
TAX/TXA/TAY/TYA
BRK
All branch instructions if branch taken
```

BOOTING

One of the many unique features of the Super Guppy and Beluga is their capability to boot from serial flash. Thanks to this capability they can work without any parallel ROM... which is a necessity for the Beluga, but also allows the Super Guppy to be used without a parallel ROM, which can make sense for some software to get the lowest possible cartridge cost.

ROM INITIALIZATION DURING RESET

The preparation for a boot happens during reset (or power-on). When the Beluga-controller starts its reset procedure, it will perform the equivalent of writing $FF to $DE01 two times.

This will end a possible continuous read operation and switch the serial ROM into SPI mode. Note that this is not the same as a full chip reset: For example, volatile bits in the status registers will keep their values, but this is exactly a reason why application software should not write to the status registers.

(The serial ROM does actually have a real reset command, but unfortunately the pattern generators in the SLG46620 GreenPAK that is the foundation of the Beluga controller do not have enough capacity to issue that command.)

After returning the serial ROM into SPI mode, the Beluga controller will start a read at address $000000 in the serial flash ROM. It uses command $EB, which is a quad SPI command, which means it will use all 4 bits of the ROM, but since the ROM is in SPI mode, the command itself has to be sent 1 bit at a time. $EB = %11 10 10 11, therefore what the Beluga controller does is the equivalent of writing the following bytes to $DE00: $FF $F0 $F0 $FF

It then sends the address and dummy cycles, which is the equivalent of writing $00 $00 $00 $00 to $DE00. The ROM is now ready to serve its contents and just like we sent the address using all 4 bits of the flash ROM, it will also output in 4-bit nibbles (which the logic on the cartridge converts to 8-bit bytes).

Next, the controller performs 8 times a read from flash memory, and writes the result to the configuration register. Yes, the register is written 8 times, while one time would have been enough.

(Writing the register 8 times 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.)

At this point, the Beluga controller has done its work. The flash ROM remains selected, so the C64 can read more bytes from flash during boot.

INITIAL VALUE OF THE CONFIGURATION REGISTER

Traditional cartridges normally set their configuration register to $00 during a reset. In contrast, on the Super Guppy and Beluga cartridges the configuration register is initialized from serial flash memory and this means that you as a software designer has full control over the initial value of the configuration register. This unlocks new functionality.

For example, on the Super Guppy cartridge, you can choose whether the cartridge boots in 8K, 16K or Ultimax mode. Or even normal C64 mode with KERNAL replacement enabled. And it also means that the Super Guppy cannot boot from the serial flash ROM, but it can also boot from the parallel flash ROM.

Parallel Booting

Configuring a Super Guppy for parallel boot is as simple as setting the right value of the configuration register into the serial flash ROM. For example, if the first bytes of the serial flash ROM are $88 $88 $88 $88 $88 $88 $88 $88, then the Guppy with enable its led during reset, and start the computer in 16K cartridge mode with the parallel ROM visible.

The Supper Guppy will then start just like any other 16K cartridge. Place the CBM80 signature at the right place, set the cold start vector and off you go. Keep in mind that the Beluga controller did not know you were going to do a parallel boot, therefore the serial flash ROM will be selected on reset and if you read from it, you will read bytes from flash location $000008 onwards.

In order to be able to use the serial flash memory after you have booted, you will want to switch the serial flash memory into QPI mode, and set the correct number of dummy cycles. This will be explained in the next section about serial booting.

Because the Beluga doesn't have any parallel ROM, you will have to boot it from serial flash, but still have freedom in which Beluga mode the cartridge boots. In addition, you can reboot from the parallel SRAM, see the chapter Rebooting.

Serial booting

Let's assume we would like to boot a Beluga.

The first 8 bytes of flash then 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 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.

In the speedcode that we execute we need to pay attention to 6502 phantom reads, as explained in the chapter Sequential Access Memory. We will use the TXS and PHA mnemonics and to handle the phantom cycle that follows these instructions, we add an extra NOP after them.

The next code in the boot sector then becomes:

```
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.

We want to use the serial flash memory in QPI mode, therefore switching to QPI mode is an important task that the stage 3 boot needs to perform. It does so exactly like it was documented in the chapter about the Serial Flash Memory. The bootloader however, will have to do one more thing: Dummy cycles. The chapter about Serial Flash Memory already did briefly explain that the number of dummy cycles on a serial flash memory is configurable, and that the Super Guppy and Beluga cartridges use 4 dummy cycles. The chapter did not explain how to configure 4 dummy cycles.

And unfortunately, the flash memory industry is a bit chaotic. The number of dummy cycles can be configured with the command $C0, which accepts one extra byte in which bit 4 and 5 configure the number of dummy cycles. However, which value correspondents to how many dummy cycles, is different per manufacturer.

A table which manufacturer uses which amount of dummy cycles would be one approach to do this, but it is recommended to detect the number of dummy cycles with an algorithm. The proposed stage 3 boot implements an algorithm that detects the correct number of dummy cycles by trying to read flash location $000007, which should return the initial value of the configuration register.

The stage3 boot then reads:

```
startup_config = $85

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 $11 $10 $00
    LDA #$00
    STA $DE00
    LDX #$11
    STX $DE00
    LDX #$10
    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 want to configure 4 dummy cycles with the $C0 (set read
    ; parameters) opcode. Unfortunately the evil manufacturers have
    ; managed to be incompatible which value corresponds to 4 dummy
    ; cycles. For example Winbond requires $10 while Gigadevice requires $00.
    ; We could detect the manufacturer and do an "if manufacturer"
    ; but it is better to autodetect which value works.
    ;
    ; In order to do so, we will try to read at flash location $000007.
    ; If the amount of dummy cycles is correct, we will read $85. If it
    ; is too many, we will read $ff, if it is too low we read a byte
    ; from the CBM80 signature, which is <> $85.
    ;
    LDA #$00
@1: LDX #$C0        ; Set read parameters
    STX $DE00
    STA $DE01
    LDX #$EB        ; Fast read quad IO
    STX $DE00
    LDX #$00
    STX $DE00
    STX $DE00
    LDX #$07
    STX $DE00
    BIT $DE00       ; Do not use $de02 while dummy cycles not correct
    BIT $DE00       ; It could cause bus conflict with VIC-II
    LDX $DE01
    CPX #startup_config  ; Read was correct?
    BEQ @2
    CLC
    ADC #$10        ; Try next possible value
    CMP #$40
    BNE @1
    JMP fail

@2: 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

   ; Blink the led to indicate a failure
fail:
    LDA #15
    STA $D020
    ; Blink the led
    LDA $DE03
    EOR #$80
    STA $DE03
    LDY #40
    LDX #0
:   DEC
    BNE :-
    DEY
    BNE :-
    JMP fail
```

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.

Rebooting

The Super Guppy and Beluga cartridges have a reboot mechanism that allows you to reset the computer without resetting the cartridge. This feature is very useful on the Commodore 128, because it allows you to exit Commodore 64 mode and return to the Commodore 128 native mode.

Rebooting the computer is as simple as writing to the reboot register. This register has address $DE08 on the Super Guppy and address $DE07 on the Beluga. In other words:

```
STA $DE07
```

... will reboot a computer that has a Beluga inserted. Depending on the contents of the configuration register of the cartridge, the computer will boot into C64 or C128 mode. The configuration register will keep its contents during the reboot and if the serial flash memory was selected, it will remain selected.

Suppose you want to reboot to Commodore 128 mode and you have a Commodore 128 boot sector in your flash memory at $400000. Then you can do this:

```
    LDA #$90
    STA $DE03   ; C128 will start in C128 mode, enable the led
    LDA #$EB    ; Start a read from cartridge
    STA $DE00
    LDA #$40    ; High byte
    STA $DE00
    LDA #$00    ; Mid byte
    STA $DE00
    STA $DE00   ; Low byte
    BIT $DE02   ; Dummy cycles
    STA $DE07   ; Reboot!
@1: INC $D020
    JMP @1      ; Endless loop if reboot fails for some reason
```

During boot the C128 will start reading the CBM signature from cartridge memory at $400000 and this way you can take control of the computer in C128 mode.

Of course you can also reset from Commodore 64 to Commodore 64 mode, if you see a need to do so. One possible application is to set up the SRAM of the Beluga with the contents of a conventional 8K or 16K cartridge and then reboot to start the cartridge.

The reboot feature allows you to distribute the Commodore 64 and Commodore 128 version of your game on a single cartridge. By having both versions on a single cartridge, a software publisher does not need to manufacture a separate Commodore 128 cartridge for your game and this removes a commercial obstacle to release a Commodore 128 version of your game.

Such a cartridge will have to always start in Commodore 64 mode, because a Commodore 64 won't start a cartridge without the EXROM or GAME lines being pulled, but pulling these lines with make a Commodore 128 start in C64 mode. Therefore you will start your game in Commodore 64 mode, detect whether you are running on a Commodore 128 and then reboot to Commodore 128 mode.

Commodore 128 specific topics

If you did read the manual up to here, you will have already understood that the Commodore 128 wasn't an afterthought during the design of the Super Guppy and Beluga cartridges. Instead, the cartridges are intended to remove some barriers to design games for the Commodore 128.

One of the key capabilities the cartridges have is that they enable the developer to release a Commodore 64 and 128 version of a game on the same cartridge. This is made possible by the reboot functionality and thanks to the huge capacity, there should be no reasons not to do this for space reasons. Only the game code needs to be installed twice, graphics and music can be shared between both versions.

Distributing a Commodore 64 and 128 version of a game on the same cartridge avoids the need to produce a separate C128 cartridge of a game and therefore eliminates the commercial complexity of producing and selling two different cartridges. It also removes the “buy twice” problem; it ensures that every owner of the game will own both versions. Should somebody initially use the game on a C64 and gets a C128 at a later date, he doesn't need to do anything to be able to play the C128 version.

Let's discuss how the Super Guppy and Beluga cartridges behave in C128 native mode.

Parallel memory in C128 mode

The parallel memory in C128 mode works the same as the parallel memory in C64 mode, however, the C128 has two 16KB ROML/ROMH windows, while on the C64, these are 8KB in size.

On the Super Guppy and Beluga cartridges, address line A13 is always driven by line A13 on the cartridge port, not by the state of ROML/ROMH. This means that in C128 mode, each 16KB window will show the full 16KB of parallel memory that can be visible to the C64. If you enable cartridge ROM both in the mid ROM and high ROM memory regions you will see the parallel memory two times in the memory map.

As this is not that useful, you are recommended to enable cartridge ROM in the mid ROM region only ($8000..$BFFF). The high ROM region can be used to make the KERNAL.

2MHz mode

The Super Guppy and Beluga cartridges have been designed as 2MHz compatible cartridges. Both the parallel flash or SRAM and the serial flash can be accessed in 2MHz mode.

During PHI1 high, the controller will generate cycles for read operations only. During PHI2, both read and write cycles will be generated. This means that in 2MHz mode, you can only read from memory mapped at $8000.

If you access I/O registers at $Dxxx, the Commodore 128 will do cycle stretching in order to make all access happen during PHI2 high. This is of interest if you access the serial flash, because this means that if you enable sequential access memory and access at $8000, you will be able to read from flash at slightly higher speeds than if you use $DE00 (or $DF00 for the Beluga).

Z80 mode

The Beluga and Super Guppy cartridges have not yet been tested in Z80 mode.

Writing to SRAM (Beluga)

The Commodore 128 does not have an Ultimax mode that allows the cartridge port to receive write cycles. Therefore it is not possible to write to SRAM in C128 native mode. Although the Super Guppy/Beluga can start directly in C128 mode, I expect that most software will start the computer in C64 mode and then use the reboot feature to get into C128 mode. Therefore you can fill the SRAM with data in C64 mode before you switch to C128 mode.

Because the C128 has twice as much memory as the C64, the SRAM is not as much of a game changer in C128 mode anyway.

Effects of Beluga modes in C128 mode

The Beluga decoder logic works technically the same in C64 mode as in C128 mode, however due to the difference how memory works in C128 mode, the effects are different.

In mode 0, the SRAM is never selected, therefore you will read garbage if you try to map the cartridge. In mode 1, you will be able to map the SRAM into the mid ROM region ($8000..$BFFF). In mode 2 and 3, you will be able to map the SRAM both into the mid ROM and high ROM region ($C000..$FFFF).

In mode 4, you will only be able to map the SRAM into high ROM region. In mode 5, you will be able to map sequential access memory into the mid ROM region. In mode 6, you will be able to map sequential access memory into the mid ROM region. In mode 7, you can map SRAM to the mid ROM and sequential access memory to the high ROM.

Note that you actually have to map the cartridge using the MMU, because in C128 mode, a cartridge cannot map itself.

Programming strategies

Whenever you design a program, you need to decision about the programming model for your game. Are you going to run your code from C64 RAM, or directly from cartridge? Are you going to use 8K or 16K mode? How are you going to divide the C64's memory between code, graphics and dynamic game data? Are you going to run your interrupts from the $0314 vector or from the CPU interrupt table?

These decisions are of strategic impact: Once your development is well in progress, it becomes nearly impossible to make different decisions without doing a full rewrite of your code. If you make a bad decision, your development might get stuck and you may lose your motivation to finish your project.

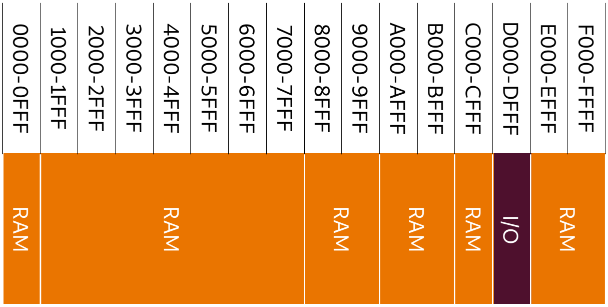
The Super Guppy and Beluga Cartridges give you more freedom in how you can use the C64's memory. Therefore you are less likely to hit into a wall with these cartridges, however, the programming model that you choose still has a huge impact on what you can do and you can't do with the C64 and therefore making the right decisions at the start of your project can still make the difference between success or failure.

In this manual we mention the Super Guppy and Beluga cartridges most of the time in a single sentence, but they are far from the same cartridge. Some programming models suit themselves to both cartridges, others are more suitable to the Super Guppy cartridge and others work best on the Beluga cartridge.

This chapter tries to assist you in choosing the right programming model for your project, and as a result, helps you to choose between Super Guppy and Beluga.

The everything RAM programming model

In an “everything RAM” model, you try to map the entire C64 address space to RAM, except normally for the I/O space. Your game may temporarily map out the I/O region to store some data under it, but in general operation without I/O visible is inconvenient so you map the I/O region most of the time.



This programming model is used by many games designed for tape and diskette for the simple reason that there is no cartridge to map, the only exceptions are games that use so little C64 memory that there is no point to unmap BASIC and KERNAL.

This programming model is quite suitable for the Super Guppy and Beluga cartridges. Because I/O is almost continuously visible, the serial ROM is also almost continuously visible. On a traditional cartridge, any cartridge access makes 16KB or 24KB of ROM visible (cartridge ROM + KERNAL), this seriously restricts the programmer from which part of the C64 address space he can access the cartridge. On the other hand, with the Super Guppy and Beluga cartridges, the programmer is always able to access the serial flash ROM anywhere from the 60KB RAM that is visible.

Because the visibility of the cartridge does not disrupt anything, the "everything RAM" programming model is a good way to extend existing games designed for tape/diskette. By moving game data from RAM to cartridge ROM, you can free up RAM, and thus add more features and content to existing games.

For newly developed games, the "everything RAM" programming model imposes less restrictions on the coder which C64 memory he uses for what, which will result in more advanced games.

The "everything RAM" programming model also is economically attractive: The Super Guppy cartridge can then be used without the parallel ROM, allowing for dirt cheap cartridge hardware.

The hidden ROM programming model

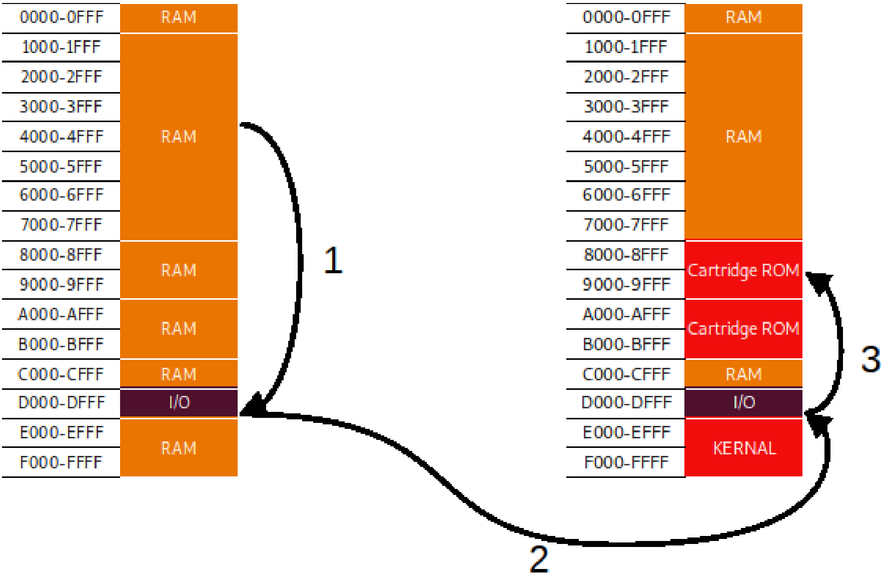
The everything RAM model is convenient, efficient and economic. You can try to free up C64 memory by moving game data to the serial flash ROM, but there is a limit how much game code you will want to store in C64 memory. In such case a hidden ROM programming model is a good way to run code outside C64 memory.

With a hidden ROM, you divide your program code between persistent code that is permanently visible in C64 memory, and some code that is used less often that you place in a normally hidden cartridge ROM. As soon as you need to call a routine in the hidden ROM, you unhide the ROM, and call the routine.

You will have the restriction that you can only call your ROM routines from below $8000 or from the $C000 high RAM region. This restriction is easy to mitigate by installing a trampoline in the IO2 region at $DF00. The trampoline makes the hidden ROM visible, calls the actual routine and hides the ROM again. This way you can still call your hidden ROM from anywhere.

Another classic restriction with this model is that making the hidden ROM visible makes the KERNAL visible. As we did discuss in this manual, the Super Guppy and Beluga allow you to overcome this restriction by replacing the KERNAL.

Both the Super Guppy and Beluga cartridges are a great choice for a hidden ROM programming model, but the Beluga has the advantage that it provides the programmer with a hidden RAM rather than a hidden ROM, and RAM is much more flexible than ROM. This means for example that you can use self-modifying code and place variables in the hidden RAM as well, which can result in more performance, less reliance on zero page variables and more C64 RAM available.



0000-0FFF	RAM
1000-1FFF	RAM
2000-2FFF	
3000-3FFF	
4000-4FFF	
5000-5FFF	
6000-6FFF	
7000-7FFF	
8000-8FFF	RAM
9000-9FFF	
A000-AFFF	RAM
B000-BFFF	
C000-CFFF	RAM
D000-DFFF	I/O
E000-EFFF	RAM
F000-FFFF	

0000-0FFF	RAM
1000-1FFF	RAM
2000-2FFF	
3000-3FFF	
4000-4FFF	
5000-5FFF	
6000-6FFF	
7000-7FFF	
8000-8FFF	Cartridge ROM
9000-9FFF	
A000-AFFF	Cartridge ROM
B000-BFFF	
C000-CFFF	RAM
D000-DFFF	I/O
E000-EFFF	KERNAL
F000-FFFF	

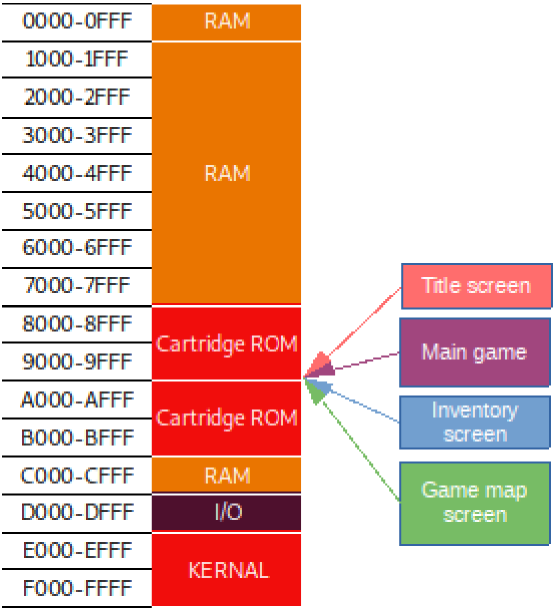
The overlay memory model

If you need even mode code, a single 16KB hidden ROM may not be enough anymore. In such cases an overlay memory model may make sense. In an overlay memory window, game code is split between persistent game code, which is always in memory, and several overlays, which are loaded on the fly.

Many games designed for diskettes use an overlay memory model: Persistent code is loaded once, and as the player switches to different areas of the game, overlays are being loaded into the overlay windows.

The overlay model is also used by cartridge games, as the 8KB or 16KB cartridge window can work as an overlay window and a bank switch is then the cartridge equivalent of loading an overlay from diskette.

Cartridge games that use the overlay memory model, generally accept that having cartridge memory visible, means that the KERNAL is also visible. The cartridge memory is normally visible and only temporary hidden if code needs to work on memory below ROMs.



As parallel ROM on the Super Guppy cartridge works like a traditional bank switched cartridge, the Super Guppy is suitable for the overlay memory model and its KERNAL replacement feature comes in handy.

However, the Beluga cartridge is a lot more flexible for an overlay memory model, because it doesn't have cartridge ROM, but cartridge RAM. With the Beluga, you do not need to puzzle which overlays can fit together in a single bank. Your overlays can be placed into serial flash ROM, where there exist no bank boundaries. Loading a different overlay requires copying some data from serial ROM to SRAM, which is of course slower than a bank switch, but should be split-second operations, especially as the Beluga's auto-increment makes such operations faster.

SRAM allows self-modifying code and also allows local variables in overlay memory, giving the Beluga cartridge quite a number of advantages over the Super Guppy for the overlay memory model.

The all-code-outside memory model

The Commodore 64 can run code directly from cartridge ROM. In some cases, the ultimate situation would be, if no C64 memory would be used for game code. After all, C64 memory is contested for graphics, sound and dynamic game data, so if you do not need any C64 memory for the code of your game, you can make more advanced C64 games.

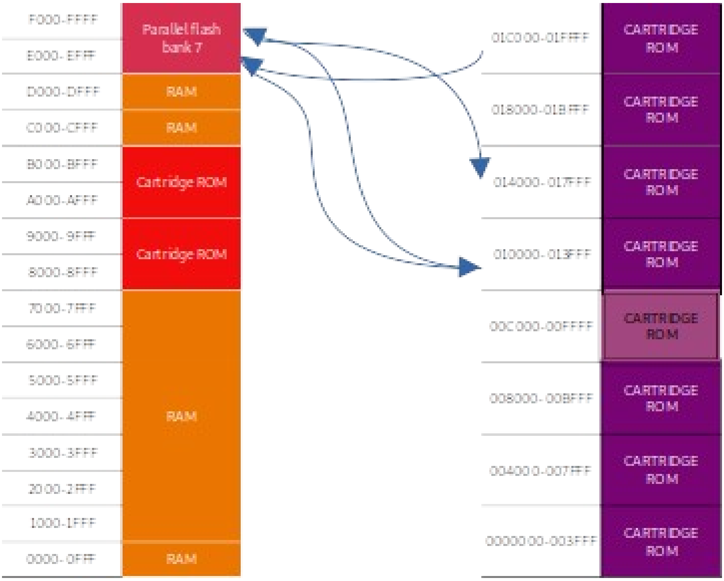
So, while this memory model would allow more advanced C64 games, very few C64 games use this memory model. The reason is: Lack of suitable cartridge hardware. Games which place all of their code in cartridge memory, need to be able to call routines in other banks. While this is not impossible with traditional cartridges, it requires inefficient helper routines.

Perhaps more important: Code that runs directly from cartridge has trouble accessing game assets elsewhere in cartridge memory, since accessing that requires a bank switch, which cannot be performed as long as the CPU is executing code from cartridge ROM.

Lastly, interrupts are handled by the KERNAL. You can hook the vectors at $0300, but interrupt latency will be higher, so you will be limited into timing critical raster effects.

The Super Guppy is possibly the first Commodore 64 cartridge that solves these problems and is therefore very suitable for games that want to run all of their code outside C64 memory:

- While code is running directly from parallel ROM, this code has full access to the serial ROM, allowing the code to access game assets.
- The readable configuration register makes far calls quite a bit easier and more efficient
- The KERNAL replacement feature gives you access to the KERNAL interrupt table, in addition, the KERNAL memory is a great place to place far call entry points.



In an all-code-outside memory model, code can freely jump from anywhere to anywhere in parallel flash memory. If code needs to call a routine in another bank, it can call a far-call wrapper in the bank 7 ROM that is mapped into the KERNAL memory space.

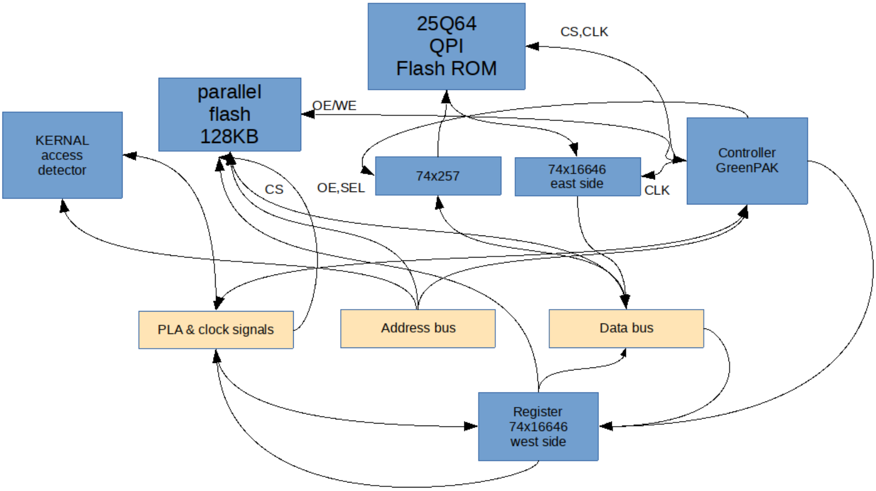
This memory model is almost only suitable to the Super Guppy cartridge, since on the

Beluga there isn't a bank-switch mechanism, code would need to be copied into SRAM, which is quite slow compared to doing a bank switch.

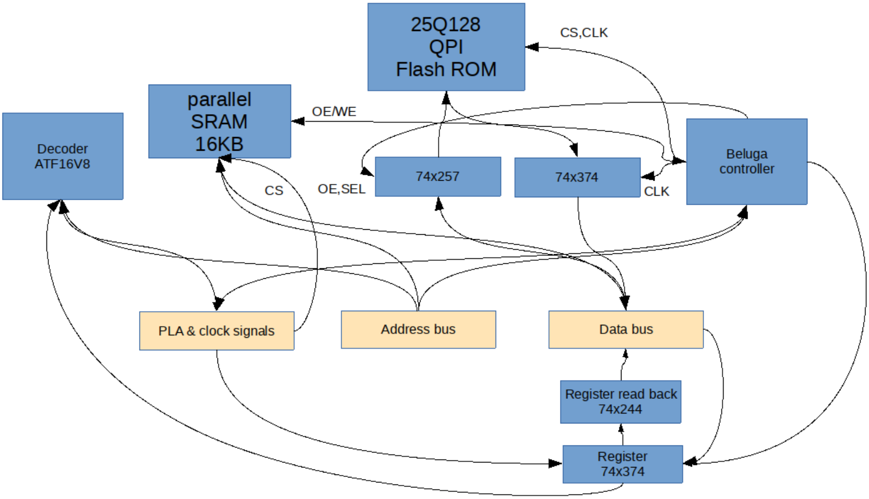
The Super Guppy's 128KB of parallel ROM can almost be fully used for the code of a game, and this is quite a lot code for C64 standards. Combined with the fact that the memory model can also free up a significant amount C64 memory, the Super Guppy should enable games of a level of sophistication that we haven't seen yet.

Block Diagrams

A schematic diagram of the Super Guppy cartridge:



A schematic diagram of the Super Guppy Cartridge



REGISTER SUMMARY

Address Beluga	Address Super Guppy	Read/ Write	Description
$DE00 56832	$DE00 56832	R/W	Read/write flash. Read or write a byte from/to the serial flash memory. If the flash memory is not selected, it will be automatically selected if you write to this register.
$DE01 56833	$DE01 56833	R/W	Read/write flash with deselect. Read or write a byte from/to the serial flash memory. If the flash memory is not selected, it will be automatically selected if you write to this register. After your read or write has been performed, the flash memory is deselected.
$DE02 56834	$DE02 56834	R/W	Read/write flash with dummy cycles. Read or write a byte from/to the serial flash memory. If the flash memory is not selected, it will be automatically selected if you write to this register. After your read or write has been performed, two extra dummy cycles are performed to the flash memory.
$DE03 56835	$DE04 56836	R/W	Configuration register The configuration register of the cartridge.
$DE07 56839	$DE08 56840	W	Reboot. Resets the computer without resetting the cartridge. If flash memory is selected, it will remain selected.