Memory expansions for the Commodore 64

Marko Mäkelä <Marko.Makela@FTP.FUNET.FI>
Pekka Pessi <Pekka.Pessi@HUT.FI>

January–February, 1987
April 17th, 1994
[last modified December 22nd, 1999][*†‡]

Commodore 64 becomes remarkably more efficient by adding memory to it. Then the worst slow-down, incredibly slow disk drive, can be worked around by using a part of the memory as a RAM disk.

In 1986, when the original article was written, there were no commercial memory extensions for sale in Finland.[1] *Of course there were some in the USA, but they were quite useless, as they had only 64 kB of memory, and the price was high as well. When built by oneself, the following memory expansion should have costed 300–400 Finnish marks.*[2]

Many goals were set to the expansion. As many programs as possible should work also with it installed. This means that there could not be any radical changes to the memory map.

The address space of an expanded machine really remains same for a usual programmer. After reset, the computer doesn't differ from an unexpanded one practically at all.

[*] This document is based on Pekka Pessi's two articles describing an 256 kB internal memory expansion for the Commodore 64. The articles were originally published in the largest Nordic and Finnish home computer users' magazine, MikroBITTI, in its first two issues in the year 1987. Six years later, they were translated to English and edited by Marko Mäkelä, with help from Pekka Pessi. This is a third, corrected and improved version of the document.

[†] August 1996: Thanks to Wolfgang Scherr from Austria, who noticed my mistake in the schematic diagram. The inputs of the 74LS153 chip were mixed, which caused the address block decoding to fail.

[‡] By now, I know of two Commodore 64 (mine and Pekka Pessi's) where this expansion has been built, and of four Commodore 128s (of which two expanded by me, and the other two by Wolfgang Scherr and Matthias Kirchner from Germany). The expansion never became a success, although the banked concept is technically better than e.g. the Commodore REU. Meanwhile, there is one more program that supports the expansion: my transfer system prlink.

[1] Commodore's RAM Expansion Units came to our market in the year 1987.

[2] One American dollar (USD) is equivalent to five or six Finnish marks (FIM). I constructed my expansion in 1993 March, and it costed 111 FIM. It could easily have been about 20 FIM cheaper.

Even taking a look at the machine from outside doesn't reveal the expansion, as Pekka needed the expansion port for his IEEE-488 interface cartridge, also designed by himself.

The memory expansion is built in the machine on an add-on card. As the mother board remains the same — no traces are cut — the expansion can be removed without a soldering iron.

The design aimed to a hardware that supports programming. While making the RAM disk program, Pekka changed the hardware several times, until he was satisfied with it.

Contents

1 Some basics

1.1 Expansion memory in 16 kB blocks

The processor of Commodore 64, MOS 6510, has an 8-bit data bus, and its address bus is 16 bits wide. Like other 8-bit processors, it can address only 64 kB of memory at a time. In most 8-bit computers, the memory is limited to these 64 kilobytes. How could one add memory above this limit?

The solution is simple: the memory is divided into banks of no more than 64 kB, which are switched on and off. Some processors have been added a special circuit for this purpose, in which case the executing program can be in its own 64 kB bank and the processed data in another bank. For example, MOS 6509, a fellow processor of MOS 6510, works in this way, enabling access to one megabyte. The Commodore 128 uses a sophisticated chip, MOS 8722 MMU (Memory Management Unit), which lets you to activate one 64 kB memory bank of a total of two memory banks at a time.

Our expansion divides the 256 kB of memory to sixteen blocks of sixteen kilobytes each. The processor can address up to four of them at a time. Every four 16 kB segment of the address space can be mapped to any 16 kB block. Figure 1 shows the mapping right after startup.

However, the video chip VIC-II — MOS 6569[3] — retrieves its data from the memory outside the normal bus. The internal address registers of VIC-II are 14 bits wide, so it can address only 16 kB without external logic. The required two extra bits for accessing the whole 64 kB video bank are provided from the second CIA chip. Our extra logic provides additional two address bits for accessing the whole 256 kB of video memory.

1.2 Memory chips

Commodore 64 uses 64 kb dynamic RAM chips of JEDEC standard. In 1982, when the computer was introduced, they were most modern technology, they needed only one operating voltage supply instead of traditional three.

The semiconductor memories have developed fast, however, and now a chip in a DIP of equal size can hold 256 kilobits. The pinout of these 256 kb chips differs minimally from the 64 kb ones. The smaller 64 kb chips, at least the ones used in C64 and C128, have one unused contact. The address line to handle three times bigger memory is tied to this pin. In the DRAMs the address lines are multiplexed: two address bits use the same pin successively.

In the MikroBITTI article Pekka wrote that 256 kb chips are rather cheap, and the price would lower as the production rate increases. Nowadays the production must have almost stopped. When Pekka bought his chips between March and April of 1986, they costed about 50 FIM each. When the original article was published, they costed less than 20 FIM. After that the prices rose

[3]6567 for NTSC

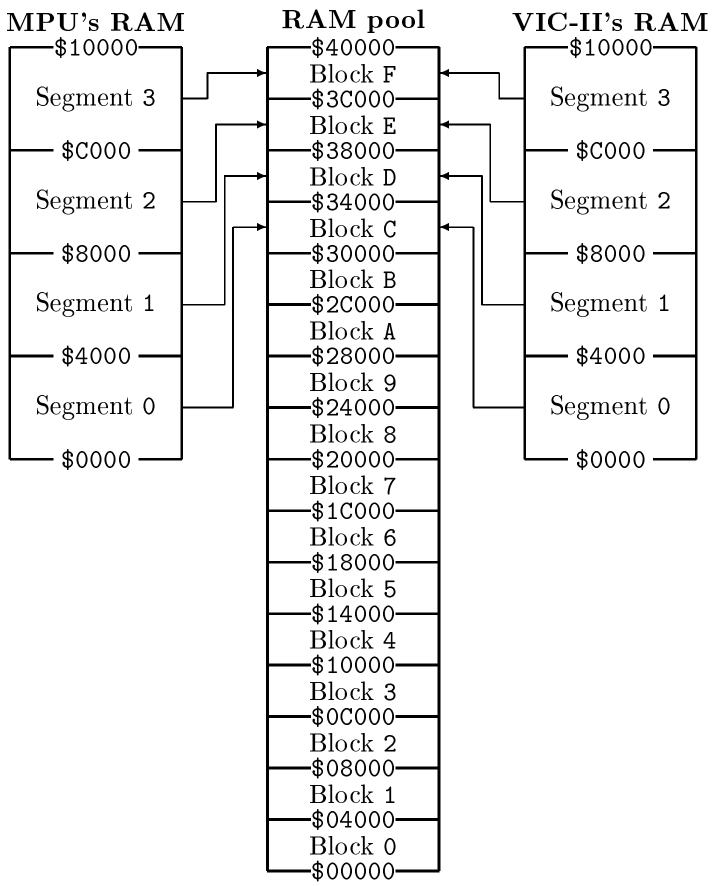

Figure 1: Memory mapping right after power-up

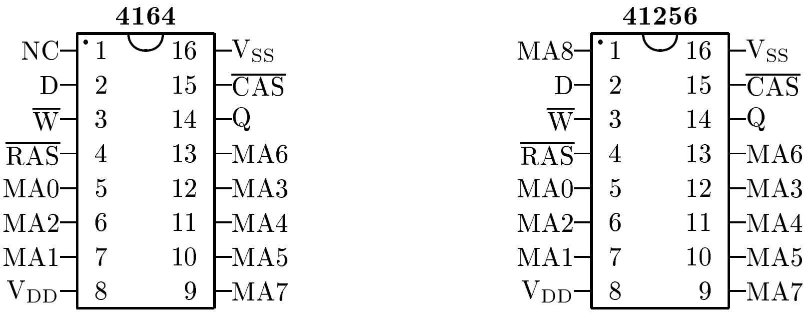

Figure 2: The Dynamic Random Access Memory Chips 4164 and 41256

due to a memory shortage. But nowadays the chips don't cost practically anything, if you're lucky. Many users of IBM PC compatibles want to upgrade their system memory with 1 Mb chips or alike and would like to get rid of their old 256 kb chips. I bought eight second-hand chips with total 35 FIM, and later 36 chips with 117 FIM, including shipping. The lowest price of unused chip I encountered was 13 FIM a piece and the highest was 30 FIM, almost 10 times the price I paid!

The 256 kb chips don't consume significantly more power, so there is no need for a bigger power supply. However, devices that take their power directly from the computer can cause problems. You can find this out by experimenting.

The speed of the chips doesn't prevent the replacement either. According to its schematics diagram, Commodore 64 can use chips with access time of 200 nanoseconds.[4] Even the slowest 256 kb dynamic RAMs are not that slow.

It might be wise to replace the bypass capacitors near the memory chips with bigger ones. On the other hand, my machine works well with the default 10 nF capacitors. If your computer starts to work unreliably, too small bypass capacitors could be the culprit.

1.3 Dynamic headaches

The dynamic RAM chips are organized in rows and columns. In 64 kb chips, a row is 256 bits wide, and in 256 kb ones it is 512 bits wide. Also the memory address is divided into row and column addresses. When a bit is being accessed in the dynamic RAM, the row address is asserted before the column address.

First the interfacing circuitry puts the row address on the Multiplexed Address bus, while the video chip asserts the $\overline{\text{RAS}}$ (Row Address Select) signal for a short period. After that, the video chip pulls the $\overline{\text{CAS}}$ (Column Address Select) line low, and the interfacing circuitry places the column address on the

[4]Besides, the oldest Commodore 64 I have uses 300 ns chips housed in ceramic packages.

bus. After all this, the bits in that position can be read or written.

The computer has two circuitries that take care of this multiplexing. The multiplexers U13 and U25 form the address when the processor has the bus, whereas the video chip produces the row and column addresses itself when it needs its screen data.

You could access a set of nearby locations faster, if you specified the row address only once, and then produced only the column addresses for each location. This technique is supported in the Acorn Archimedes computer on the processor level, and some other computers utilize it with external circuitry as memory interleaving. The Commodore, however, does not have to hassle with this, as its system clock rate is so slow. As a matter of fact, it actually uses the least significant processor address bits (A0–A7) as a row address and the most significant bits (A8–A15) as the column address.

1.4 Memory refresh

A dynamic memory chip stores the data bits as charged tiny capacitors, which discharge among the time. The data must be refreshed periodically, every 2–4 milliseconds, by recharging the capacitors.

If the whole contents of the memory was refreshed simultaneously, the power peak would cause enormous problems. Only a block of one or two rows can be refreshed at a time. The 64 kb chips have 128 blocks to be refreshed, which implies a 7-bit refresh counter (2^7 equals 128).

In order to avoid disturbance, 256 kb chips must have more blocks. Thus they require a longer refresh counter (8 bits). As the amount of refresh cycles has increased, the capacitors' ability of keeping charge has been improved. The 64 kilobit DRAMs required 128 refresh cycles every 2 milliseconds, now the 256 kb chips need 256 cycles but every 4 ms.

Whenever you select a row address,[5] the block to which the row address belongs gets refreshed. As the 64 kilobit chips have a 7-bit refresh counter, the lowest seven row address bits specify the row address, and the highest bit is ignored. The 256 kilobit memory chips have an 8-bit counter, so they ignore the 9th row address bit and determine the block to be refreshed by the eight lowest bits.

The VIC-II chip refreshes the memory systematically, 5 rows in the end of each screen scan line. It does this by selecting a row address determined by its internal counter, and then increases this counter by one. The video chip could have only 7-bit refresh counter, and it would still operate with 64 kb chips, but fortunately it has an 8-bit counter, so all of the 256 kb chips get refreshed.

Newer memory chips can be refreshed using a CAS-before-RAS technique. In this technique, you pull first the $\overline{\text{CAS}}$ signal low, and then the $\overline{\text{RAS}}$ signal. The memory chips recognize this as a memory refresh condition, and they refresh a

[5]See Section 1.3.

block and increase their internal refresh counter. However, this technique was not available when the Commodore 64 and its video chip were designed.

2 Building the expansion

2.1 Disclaimer

Although this procedure worked perfectly for me, I cannot guarantee that anyone else can perform this upgrade without damaging their computer. I therefore disclaim any responsibilty for any damage that may occur as a result of attempting this upgrade. It will also void any warranty on your computer.

On a more positive note, there is no reason why someone who is experienced in wielding a soldering iron, and has done some electronic construction or troubleshooting, should not be able to perform this upgrade successfully.

2.2 Getting started

A termostate soldering iron, desoldering pump or other desoldering tool, a screwdriver, a spoon and a continuity tester are the only tools needed. The spoon is for removing the chips. A bottle top remover is not suitable for that.[6] The continuity tester is vital for checking suspicious connections. If your tester does not automatically select proper measuring range, use the coarsest (MΩ) range, as it uses smallest current, which shouldn't damage any chips on the board.

The installation begins of course by opening the machine and removing the keyboard and LED cables. It is useful to memorize, photograph or draw how the parts were initially connected.

After removing the cables, remove the aluminum cardboard shield and open the screws that hold the motherboard with the case, and remove the board.

The expansion consists of one daughter board, which contains most of the new logic, one piggy-backed chip, and a spaghetti of wires.

In Figure 3, there is a schematics diagram of the daughter board for the expansion. There are some signals that you must wire to the mother board. You can take the $\overline{\text{I/O2}}$ and A7 signals from the cartridge port, or from some through-put location near the daughter board. The $\overline{\text{I/O2}}$ signal should be on the pin 9 of the chip U15 (74LS139). The A7 can also be taken from the 6510's pin 14, or from the multiplexor U13 (74LS257), pin 10.

The MA8 signal is the new Multiplexed Address line for the memory chips and should be soldered to the pin 1 of each chip. All the remaining five signals on the right edge of the diagram interface to the multiplexor chip U13. The $\overline{\text{CAS}}$ signal goes to pin 1. To interface the address lines A14, A15, B14 and

[6] A tiny screwdriver is equally good. Just insert the screwdriver tip under one end of the chip and wound it a bit in upward angle so that the chip moves slightly. Then insert it to the other end of the chip and try to lift it a bit. You may have to repeat this procedure. Be careful not to wound the pins too much.

Electronic Components	
Symbol	Description
IC1	MC 6821
IC2	74LS153 (or 74LS253)
IC3	74LS151 (or 74LS251)
IC4	74LS05
U9–U12, U21–U24	80256 or compatible
C1	100 nF polyester capacitor
R1	33 Ω resistor
R2	4.7 kΩ resistor

Other Parts	
Quantity	Quality
2 pcs	20-pin through-put connectors (piggy-back socket halves)
1 pc	40-pin socket (if U2 is not socketed)
1 pc	16-pin socket
plenty of	connection wire

Table 1: Parts list for the expansion

B15, you have to desolder two pins of the multiplexor, 11 and 2. The signal A15 should then be wired to the mother board contact under the multiplexor pin 11, or to the 6510's pin 23, and the signal B15, the relocated address line should be soldered to the multiplexor's pin 11. Similarly, the contacts A14 and B14 should be connected to the system bus line A14 and the U13's pin 2, respectively. Figure 4 shows the pinout of the multiplexor chip U13.

2.3 Removing the old memory chips

First you have to remove the memory chips U9–U12 and U21–U24. If you look at the mother board from the front of the computer as if you were typing, the chips are on the front left in two rows of eight chips. They are of type 4164 (or 3164 or 6665 or 6664 or 8064 or...). You could install the new chips into sockets, but I thought that it is a waste of money.

If these memory chips are already on sockets, the most of the work is done for you. It helps a lot, if you remove the bypass capacitors before removing the chips. Removing the components is easiest with a desoldering pump. It becomes even easier, when you first solder the pins with fresh solder, so that the hartz from it makes the removal of old solder easier.

Using much power is questionable, as the copper folio comes off the board in

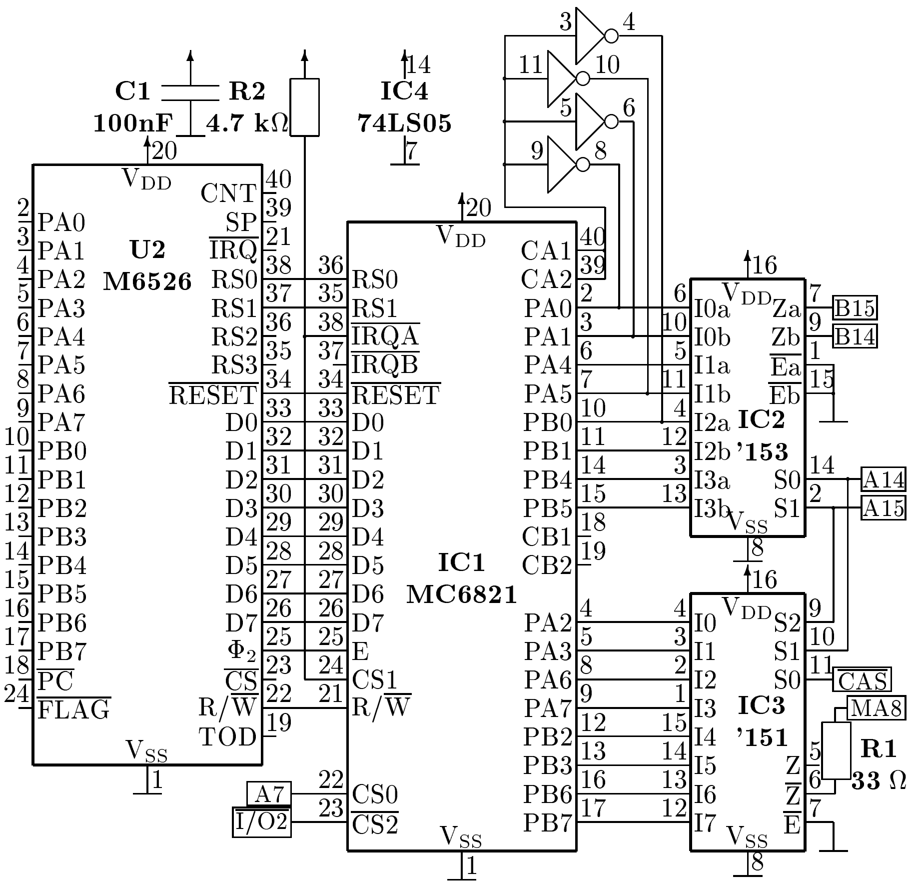

Figure 3: The schematics diagram of the expansion. See text.

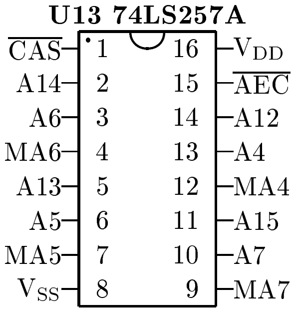

Figure 4: Pin-out for the multiplexer chip U13

a surprisingly easy way. As usual, I used a screwdriver like a crowbar, and the through-coppering got lost from several places. This was not crucial, as those pins were connected only to the down side of the board. However, three or four routes broke on the top side also. This made it far more difficult (and slower) to solder the new chips in, but I succeeded on the first try.

After you have removed the 4164s, you can solder the 16-pin sockets (or the 41256 memory chips) into their places. You can solder the capacitors back as well, if you removed them.

2.4 Adding the new address line

You must connect the pin 1 of each memory chip (or socket). It is the extra address line (MA8) to the switcher. The best way is to solder a Wire-Wrap wire to each contact under the mother board, but any thin and pliable uni-strand wire should do. The wire does not affect in any way the computer's operation with 64 kb chips.

After the pins have been connected together, they must be temporarily connected to +5 V, which is in the pin 8 of the memory chips. Comparing to TTL chips, the operating voltages are 'reversed' in dynamic memories.

Now the new 256 kb memory chips can be installed to the sockets (preferably right-side forward), and you can try switching the power on. You do not have to connect anything except the power cable and the cable to the TV set or monitor. It is a good idea to turn on the monitor first and let it warm up, so that it will show the picture from the very beginning.

If the screen shows up normally, you may not (yet) have made any mistakes. If it does not show up at all, you have to find possible cut-outs and shorts. Multi-colored '`@`'s show up usually because of too small bypass capacitors. Another cause is that the pin 1 is not connected to +5 V. In this case the screen may come up normally, but a little disturbance in the operating voltage locks the computer up. Now the computer should operate exactly like an unexpanded

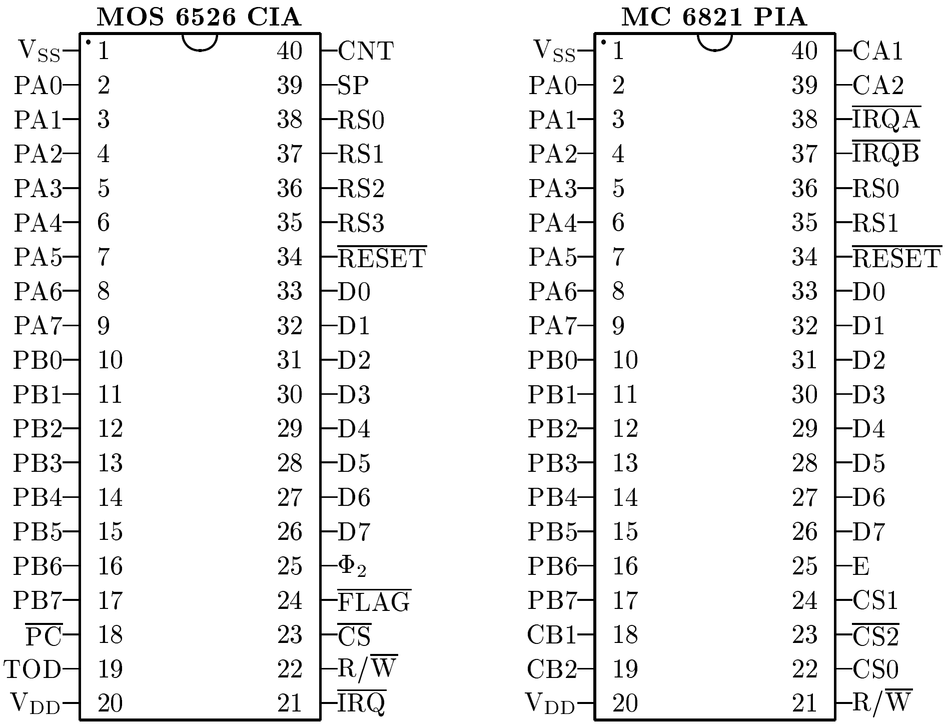

Figure 5: The CIA and the PIA

C64, so any previously working program should work with it.

2.5 Prepare for the final step

Next you remove U13 (74LS257, to the right of the memory chips) and U2 (MOS 6526, near the keyboard connector). Either or both of these chips may already be on sockets, and you must remove the rest. Reinsert the chips and check if the machine boots up.

If the computer does not work on first try, remember to disconnect any cables from it before trying to fix the problem. The soldering iron may occasionally give little electric pulses to the computer, and this might burn some expensive chips, especially if the computer is hooked to a wall outlet or a television set.

When you have completed the preparations, you can start building the control logic. You could build the whole expansion by piggy-backing chips, that is, by soldering new chips on the top of old ones, bending some feet to the side, and connecting messy wires all over your computer. However, the best way is to put most of the chips on a daughter board. I used only a small daughter board, and piggy-backed two or three chips, but you can be wiser and put all new chips on the daughter board.

My daughter board interfaces the heart of the expansion, MC 6821 PIA, to the bus of the computer through the pins of U2, the MOS 6526 CIA near the keyboard connector. The CIA is raised on the board on a normal socket, whose pins are lenghtened with two through-put socket halves, so that they can reach the socket on the mother board. I built the daughter board on an uncoppered prototype board, a plastic board with holes punched in it at a 1/10 inch grid.

After raising the CIA on the daughter board, it is a very good idea to insert the board to the socket and check if the machine boots up. The next step is to add the PIA on the board. The contacts from the CIA except the operating voltages may be difficult to route. I solved the problem by putting the wires through the small holes that were left between the biggy-back socket halves and the down surface of the daughter board. It was very painful, but the design is very compact. After soldering all CIA contacts to the PIA, I wired the inverter and the rest of the chips. To increase reliability, I used thin multi-strand wire, as uni-strand wire gets easily loose when you push it.

Since I had finished the daughter board, I bent up the pins 2 and 11 of the U13 multiplexer chip, and connected its pins 1, 2, 10 and 11 to the daughter board with wires. First I inserted the wires for A14 and A15 directly to the chip socket, but as it turned out to be unreliable, I located a through-put place for each line, and soldered the wires there instead.

When you have wired the multiplexer U13, remove the jumper wire between MA8 and +5 V and connect that address line to the daughter board. Then connect the PIA's $\overline{\text{CS}}$ line to $\overline{\text{I/O2}}$, which is in U15's pin 9 (or one of the through-put places along the trace's path to the cartridge port), and insert the daughter board to the socket. Switch the power on and pray that your dear computer works.

If you get only crap consisting of `@`'s or some randomly changing graphics, check that all CIA pins have a good contact to the piggy-back socket, and that the wires from U13 and its socket are firmly connected. If it doesn't help, you have to check all daughter board connections with the continuity tester. Don't panic, you can ensure that the computer works by connecting the MA8 line back to +5 V, by bending the U13 lines back down, and by inserting the CIA directly on the motherboard.

2.6 Testing

After you have installed the boards to your machine, it is time to test the connections. You can connect LED, keyboard and probably disk drive in addition to the power cable and the TV cable, but do not fasten the mounting screws yet. If the screen shows up and if the machine seems to operate, input the following test program:

```
10 PB=57282
20 POKE PB,255:POKE PB+1,4:POKE PB,255
30 PRINT"PRESS A KEY AFTER THIS HAS DISAPPEARED":
   FOR I=0 TO 3000:NEXT
40 POKE PB,14:WAIT 198,15:GET A$:POKE PB,255
```

On the line `10` a variable `PB` is set up. It is the address of the peripheral and data direction registers for the 6821 port B, and the block selection register of the segments 2 and 3 and the VIC-II.

The line `20` contains initialization of PIA: the lines PB0–PB7 are set outputs, the data direction register is switched to data register with '`POKE PB+1,4`', and the PB lines are set high.

On the line `40` VIC-II is given block `0` (`$00000`–`$0FFFF`), and then the program waits for a keypress and restores the block `F` (`$30000`–`$3FFFF`).

If this test program works as expected, the screen will be filled with '`@`'s and other random characters.

At this point, you may want to run the `TEST` program, which is among the distribution files.[7]

3 Using the expansion

3.1 The operation of the block switcher

There are four new micro chips in the expansion. The most important of them is the PIA chip MC 6821, which holds the values of the block selections. The PIA has two 8-bit ports set up in the addresses `57280` and `57282`. The upper and lower four bits (nybbles) of each port determine which 16 kB block is mapped to each 16 kB segment of the processor's address space. IC2 and IC3 participate in forming the memory block control signals.

There is a chip equivalent to the PIA even in Commodore's own 6500 series, but it is not suitable for this connection, as it is not TTL compatible. The 6821 from Motorola 6800 series, which contains also processors reminding those in the CSG[8] 6500 and 8500 series, is bus compatible and suitable for this purpose.

Commodore 64 asserts the 16 bit addresses to the original 64 kb chips in two parts. First it asserts the lower eight bits, then the higher eight. The 256 kb chips require two additional address bits, so the chips are given nine bits at a time. Due to this address multiplexing, the block selection bits cannot be directly input to the memory chips, but they must be lead through the multiplexer circuitry of IC2, IC3 and U13.

IC4 contributes to the operation during power-up. It ensures that the C64 gets reasonable memory blocks to its different segments. In the beginning the segments are filled with four upmost memory blocks.

[7]See Section 5.1.

[8]Commodore Semiconductor Group; former Mostek or MOS Technologies

3.1.1 PIA's location in address space

The PIA's data bus and E, $\overline{\text{RESET}}$ and R/$\overline{\text{W}}$ signals have been connected directly to the 6526 chip. Similarly are the RS0 and the RS1, which select a PIA register, connected to A0 and A1.

The I/O block decoder (U15) tells us when the second I/O block is selected. This block resides in the area `$DF00`–`$DFFF`. The signal $\overline{\text{I/O2}}$ is connected to the PIA's chip selection pin $\overline{\text{CS}}$, and it forms most of its addressing. The address line A7 limit PIA's area in I/O2 to `$DF80`–`$DFFF`, because it will be tied to the CS pin.

3.1.2 Block selection

As the address space has been divided to four segments of 16 kB, the A14 and A15 cannot be lead directly to the memory chips, but they participate in the block selection. These two address bits determine which of the four blocks is in use. For each segment, the PIA ports tell which memory block to map. Original A14 and A15 are connected to IC2 and IC3, which select the right output lines of PIA. For each 16 kB segment there are 4 output lines which form the block address for the segment.

IC2 selects two lowest bits of the block address and feeds them to the address multiplexer chip U13 as B14 and B15. They are practically equivalent to the A14 and A15 signals. After the address bits A0–A7 have been asserted during the first addressing cycle, IC2 asserts B14 and B15 during the second (CAS) cycle.

The 256 kb memory chips still need two extra address bits. The expansion must multiplex them with IC3, which is a 'one-of-eight' multiplexer. Its eight inputs are tied to the two upmost bits of the four block addresses. A14 and A15 are connected to the IC3, but it needs yet another control signal to handle all eight input bits. This signal is $\overline{\text{CAS}}$, which controls multiplexing other address bits (MA0–MA7) as well.

While the $\overline{\text{CAS}}$ signal is low and the memory chips are fed the lowest bits (A0–A7) of the address, the IC3 selects the third bit of the block address determined by A14 and A15. This bit is called address bit A16, and it is asserted to the 'extra' address line MA8 simultaneously with the lowmost bits. When $\overline{\text{CAS}}$ is high, the upper address bits are fed, and IC3 selects the fourth bit of the block address determined by A14 and A15. It corresponds to the address bit A17 and is fed through the same MA8 with all the other upmost bits.

When the video chip accesses the bus, the address and data lines from the processors are in high-impedance state, driven to logical '`1`' level with very weak current, so that the video chip can change their state easily. As the address lines A14 and A15 are not connected to the VIC-II, they remain as logical '`1`' whenever the video chip has the bus. Thus, the switcher logic 'thinks' that the address range `$C000`–`$FFFF` is being addressed, and it selects the block for that segment also for the VIC-II.

The resistor on the MA8 line protects the IC3, because the inputs of dynamic memories are not fully TTL compatible.

3.1.3 Startup settings

In order to enable the operation of the machine, each of the four segments must be mapped to a unique memory block. The Commodore 64 Kernal tests the lowmost continuous area of writeable memory and would hang up, if the same block was mapped to both `$0000` and to `$4000`, for instance. Modifying the startup routines would cure this problem, but in that case the Kernal ROM chip should be changed.

The bootup state can be achieved otherwise. The $\overline{\mathrm{RESET}}$ signal sets all the PIA port lines to inputs. As input a line has an impedance of several megaohms. A TTL chip reads such a signal as a logical '`1`'. IC4 can force four block selection pins (PA0, PA1, PA5 and PB0) low, so that the memory segments of C64 point to the four upmost memory blocks in ascending order. The port A contains the bits `1101 1100` and the port B `1111 1110`. As the IC4 has open collector outputs, it doesn't disturb the port's operation when outputting high state. That is why the initialization routines[9] write the value `52` to the address `57281`, which forces the IC4 outputs high by lowering the CA2 line.

3.2 Segmented memory

The address space of Commodore 64 consists of four 16 kB segments which are at the address ranges `$0000`–`$3FFF`, `$4000`–`$7FFF`, `$8000`–`$BFFF` and `$C000`–`$FFFF`. An expanded C64 uses the topmost four 16 kB blocks of the memory after startup. It considers them as its whole world and does not know anything of the other memory blocks. Figure 1 describes the situation. A total of twelve memory blocks are left out of the C64's world.

With expanded memory we can cheat the C64 by writing a suitable number to a known address, in order to make it consider the lowmost block as the second segment, for example. Then all operation that the C64 does at the second segment's area alter in fact the lowmost block, although the computer has no idea of it. This is the idea behind the whole expansion circuit.

What is the benefit of it? The second segment (segment `1`) is actually a good example of the function, because it resides in the middle of the RAM reserved for C64 BASIC programs. If we make a little C64 BASIC program that holds an array exactly in the third segment, we can switch another memory block to that area while the program is running, and we have another 16 kilobytes to expand our table. In this manner all unused memory blocks (12×16 kilobytes) can be taken in use, and the memory is able to hold enormous arrays, which

[9] See Section 3.4.

can be accessed simply by switching the memory block. See Example 4.1 for an example of this technique.

Another and more useful way to exploit the extra blocks is to use them as a RAM disk. A RAM disk means that you can copy even a whole disk to these blocks and consider it as a new disk drive, from which you can load program and data at a very fast speed. For a RAM disk you need a smart program that redirects disk commands and executes them on the expanded memory.[10]

3.3 Critical addresses

The critical addresses of the device are `57216`–`57343` (`$DF80`–`$DFFF`). There is the PIA chip to which you `POKE` the values to switch memory blocks. The PIA does not have 128 registers, as one might think. There are sixteen copies of its 4 addresses in that memory area. For instance, the addresses `57216`, `57284`, `57288` and `57340` are equivalent to each other.

`57280` is a memory place whose lowmost four bits (bits 0–3, low nybble) determine, which of the sixteen memory blocks is accessed through the lowmost segment (segment `0`) of Commodore 64. The upper four bits (bits 4–7, high nybble) specify, which of the blocks show up at the second segment (segment `1`). In a similar manner the low nybble of the address `57282` determines which block resides at segment `2`, and the high nybble tells the block addressed via the upmost segment.

These addresses have even another function. They can act as data direction registers as well, i.e. tell if the port lines are inputs or outputs. However, this application uses only some of the PIA's characteristics. For normal operation, all the port lines should be set to outputs. The function of these addresses depend on the bit 2 of the next address. For instance, the function of `57280` is defined with the address `57281`. If you `POKE` there a value with its third bit set, the values written to `57280` will go to the data direction register. Inputs have the corresponding data direction register bits reset, and outputs have them set. See Tables 2–5 for a complete description of PIA registers.

[10] See Section 5.4.

3.4 Initializing the expansion

Before using the expansion memory, you have to first initialize the PIA. Every time when a $\overline{\text{RESET}}$ is issued, the PIA registers change to the default state.[11] In the beginning of your program you will initialize the PIA registers so that the default block division of memory remains:

```
pia  .equ $DFC0

     LDA pia+1    ; Select Peripheral Registers
     ORA #4
     STA pia+1
     TAX
     LDA pia+3
     ORA #4
     STA pia+3
     TAY

     LDA #$FE     ; Set the default memory block data
     STA pia
     LDA #$DC
     STA pia+2

     TXA          ; Select Data Direction Registers
     AND #$FD
     STA pia+1
     TYA
     AND #$FD
     STA pia+3

     LDA #$FF     ; Set the ports to output
     STA pia
     STA pia+2

     TXA
     AND #$C7
     ORA #$30     ; Set CA1 and
     STA pia+1    ; select Peripheral Registers
     STY pia+3
```

[11] See Section 3.1.3.

You may want to use an array instead. That will save both space and processing time but lose generality. Someone may have CB1 or CB2 in use,[12] and changing all the command register bits causes side effects on these pins. Anyway, here is a BASIC example of using an initialization table:

```
10 PIA=57280
20 FOR I=11 to 1 STEP -1:READ A:POKE PIA+I,A:NEXT
30 DATA 4,254,4,220,0,255,0,255,4,254,52
```

Many Commodore 64 games do not like any extra hardware in the area `$DE00`–`$DFFF`, as it is used by many "freezer" cartridges and alike. If you need to use such software with the memory expansion, you can completely disable the PIA from the address space until a system $\overline{\text{RESET}}$ occurs. To do this, change the last `DATA` value on the line `30` from `52` to `53`. By the way, the expansion can be easily enhanced to be the most powerful freezer cartridge. See Section 6.1 if you are interested in this.

3.5 Programming the PIA in machine language

Think it in hexadecimal numbers. There are sixteen memory blocks, numbered from `0` to `F`. The address `$DFC0` holds two hexadecimal digits. The less significant digit, the one at right, selects the memory block for the segment `0` (`$0000`–`$3FFF`), whereas the other digit is for segment `1`. The other important PIA address, `$DFC2`, selects the blocks for segments `2` and `3` with is low and high nybble, respectively.

For instance, if you want to switch block `E` (`$38000`–`$3BFFF`) to segment `1`, initialize the PIA and execute the following. Note that your program must run outside segment `1` (`$4000`–`$7FFF`), or otherwise the next instruction will be fetched from the new block, thus probably crashing the processor.

```
pia   .equ $DFC0

      LDA pia      ; Segments 0 and 1
      AND #$0F     ; Preserve segment 0
      ORA #$E0     ; Select block E for segment 1
      STA pia
```

If you used our initialization routine before this, the memory areas `$4000`–`$7FFF` and `$8000`–`$BFFF` should now mirror each other. This is an easy way to peek under ROMs and I/O with a simple machine language monitor that does not play with the 6510's I/O registers to switch ROMs and I/O temporarily out.

[12] See Section 6.

3.5.1 An exception: video memory

As the video chip's address bus is only fourteen bits wide, it can access only sixteen kilobytes directly. The two additional lines needed to address 64 kB are provided by the second CIA. Its lines PA1 and PA0 are the inverse of the VIC-II's address lines VA15 and VA14.

The VIC-II needs another two address lines to see full 256 kB of memory. The PIA lines PB7 and PB6 (the uppest two bits of `$DFC2`) serve as VA17 and VA16. So, the VIC-II memory does not necessarily have to be accessible to 6510, but there is a restriction: As the block selector for the upmost segment uses the same two lines, both the VIC block and the block for segment `3` cannot be chosen freely.

For instance, if you want the VIC to 'see' its RAM at `$04000`, the lines VA17–VA14 must be '`0001`'. You can select only blocks `0`–`3` for segment `3` to fulfill this condition. Let's assume that you want block `2` to be mapped there:

```
cia2  .equ $DD00
pia   .equ $DFC0

      LDA cia2+1  ; First set the CIA2 lines
      ORA #$03    ; PA0 and PA1 to output.
      STA cia2+1

      LDA cia2    ; Then set PA1 and reset PA0.
      AND #$FC    ; Remember, the lines VA15 and VA14
      ORA #$02    ; are the inverse of them.

      LDA pia+2   ; Segments 2 and 3
      AND #$0F    ; Preserve segment 2
      ORA #$20    ; Select block 2 for segment 3
      STA pia+2
```

If you want the video bank selection to work exactly like in a stock computer, you have four alternative memory block configurations. The addresses `$DFC2` and `$DFC0` must contain one of the following bytes: `$FE` and `$DC`, `$BA` and `$98`, `$76` and `$54`, or `$32` and `$10`. You can use the expansion to debug or examine programs that occupy full 64 kilobytes of normal memory. When you issue a $\overline{\text{RESET}}$, the program's memory will remain totally unaltered, if it is outside the topmost four blocks. There is no need for a 'freezer' cartridge.[13]

4 Programming the expansion in BASIC

With BASIC the use of the extra memory is a bit limited. In the upmost segment (segment `3`) there is operating system ROM, under which you can place different

[13]See Section 6.1.

Peripheral Register A *(Peripheral Lines PA7–PA0)*	
Bits	Description
7–4	Block Selection, Segment `1`
3–0	Block Selection, Segment `0`

Data Direction Register A	
Bits	Description
7–0	Data Direction of Peripheral Lines PA7–PA0 When a bit is set, its corresponding Port A line is an output. Otherwise it is an input.

Table 2: The PIA address `$DFC0` (`57280`)

memory blocks, but reading them with BASIC is naturally impossible. However, in some cases writing data to this segment partially under Kernal ROM and I/O area may be a working solution. The lowmost kilobytes are free RAM, and it can be utilized by switching memory blocks. But the benefit of the extra memory decreases, as you can use only the lowmost four kilobytes of each block.[14]

The highest segment but one, segment `2`, is halfly under BASIC ROM, and only its lower half can be freely used.[15] When utilizing it, you have to take in consideration that those 8 kilobytes can be under a ROM cartridge, if one is connected, or they could hold some of the variables and tables that are stored in the top of the BASIC memory. You have to construct your programs so that they do not collide with the segment's area.

The lowmost segment, segment `0`, contains Kernal's and BASIC interpreter's system variables. Normally you cannot change its contents, since the operating system would not find its status information. This can be worked around by copying those vital bytes to the new memory block and switching the block with a machine language routine.

The only segment that can be wholly used with plain BASIC is the segment `1`, the second one from the bottom (`$4000`–`$7FFF`). It resides in the middle of the space reserved for BASIC programs. If you construct your BASIC programs wisely, that is short enough, and ensure that the information used by BASIC are located exactly on this area, you can switch the blocks in this segment freely and exploit all the twelwe extra blocks as a huge data storage.

Example 4.1 shows how you can create a table on this area and hold its data simultaneously in all the extra memory blocks.

[14] You can read and write the area `$C000`–`$DFFF` of this segment using BASIC. Accessing the area `$D000`–`$DFFF` without machine language is tricky but possible.

[15] You can always write under ROM (except in the UltiMax game cartridge configuration).

Control Register A	
Bit(s)	Description
7	IRQA1 Interrupt Flag Goes high on active transition of CA1; Automatically cleared by MPU read of Peripheral Register A. May also be cleared by hardware $\overline{\text{RESET}}$.
6	IRQA2 Interrupt Flag When CA2 is an input, IRQA2 goes high on active transition of CA2; Automatically cleared by MPU read of Peripheral Register A. May also be cleared by hardware $\overline{\text{RESET}}$.
5–3	CA2 Control `00x` Input, triggered on falling edge `01x` Input, triggered on rising edge When `x` is `1`, $\overline{\text{IRQA}}$ Interrupts by CA2 active transition are enabled. `10x` Output, Read Strobe for Peripheral Register A CA2 goes low on first high-to-low E transition following a read of Peripheral Register A. When `x` is `0`, it remains low until next active CA1 transition. When `x` is `1`, CA2 remains low for one E cycle. `110` Reset CA2 `111` Set CA2
2	Register in address `$DFC0` `0` Data Direction Register `1` Peripheral Register
1	Determine Active CA1 Transition `0` IRQA1 set by high-to-low transition on CA1 `1` IRQA1 set by low-to-high transition on CA1
0	CA1 Interrupt Request Enable/Disable `0` Disable $\overline{\text{IRQA}}$ Interrupt by CA1 active transition. `1` Enable $\overline{\text{IRQA}}$ Interrupt by CA1 active transition.

Table 3: The PIA address `$DFC1` (`57281`)

Peripheral Register B *(Peripheral Lines PB7–PB0)*	
Bits	Description
7–4	Block Selection, Segment 3
3–0	Block Selection, Segment 2

Data Direction Register B	
Bits	Description
7–0	Data Direction of Peripheral Lines PB7–PB0 When a bit is set, its corresponding Port B line is an output. Otherwise it is an input.

Table 4: The PIA address `$DFC2` (57282)

4.1 Processing a huge array

```
10 I=0:J=0:K=0:A=0
20 DD=56576:PIA=57280
30 FOR I=11 TO 0 STEP -1:READ A:POKE PIA,A:NEXT
31 DATA 4,254,4,220,0,255,0,255,4,254,52,220
40 K=16384-7:POKE 47,K AND 255:POKE 48,K/256:
   POKE 49,K AND 255:POKE 50,K/256
50 DIM A%(8191)
60 FOR I=0 TO 15:POKE PIA,I*16+12
70 PRINT I":";:FOR J=0 TO 9:PRINT A%(J),:NEXT:PRINT:NEXT
80 POKE PIA,220:END
```

The program displays ten first integers of each memory block. The table it reserves fills the whole segment `1`, because each integer (notice the `%` sign) takes two bytes and 8192 of them are reserved. The contents of the table `A%` can be changed to another memory block by simply `POKE`ing PIA's corresponding register. On the line `40` the table is ensured to start at `$4000` by changing the start and end addresses of tables in the addresses `47`–`50`. Saving the name, size and dimensions of the table takes the seven bytes, which are subtracted from the start address.

Reserving the table to an arbitrary address has its drawbacks. The size of the program and its variables may not exceed 14 kilobytes so that they could fit to the memory before the beginning of the table. All variables must definitely be declared before allocating the table. If you do not declare them by giving them a value, the interpreter finds really exotic values for them or transfers the table off its position.

Control Register B	
Bit(s)	Description
7	IRQB1 Interrupt Flag Goes high on active transition of CB1; Automatically cleared by MPU read of Peripheral Register B. May also be cleared by hardware $\overline{\text{RESET}}$.
6	IRQB2 Interrupt Flag When CB2 is an input, IRQB2 goes high on active transition of CB2; Automatically cleared by MPU read of Peripheral Register B. May also be cleared by hardware $\overline{\text{RESET}}$.
5–3	CB2 Control `00x` Input, triggered on falling edge `01x` Input, triggered on rising edge When `x` is `1`, $\overline{\text{IRQB}}$ Interrupts by CB2 active transition are enabled. `10x` Output, Read Strobe for Peripheral Register B CB2 goes low on first high-to-low E transition following a read of Peripheral Register B. When `x` is `0`, it remains low until next active CB1 transition. When `x` is `1`, CB2 remains low for one E cycle. `110` Reset CB2 `111` Set CB2
2	Register in address `$DFC2` `0` Data Direction Register `1` Peripheral Register
1	Determine Active CB1 Transition `0` IRQB1 set by high-to-low transition on CB1 `1` IRQB1 set by low-to-high transition on CB1
0	CB1 Interrupt Request Enable/Disable `0` Disable $\overline{\text{IRQB}}$ Interrupt by CB1 active transition. `1` Enable $\overline{\text{IRQB}}$ Interrupt by CB1 active transition.

Table 5: The PIA address `$DFC3` (`57283`)

4.2 Storing graphics

```
10 I=0:J=0:A=0:A$=""
20 DD=56576:PIA=57280:V=53248:COLOUR=50176
30 FOR I=11 TO 0 STEP -1:READ A:POKE PIA,A:NEXT
31 DATA 4,254,4,220,0,255,0,255,4,254,52,220
40 POKE V+24,16+8:POKE V+17,59
50 FOR I=0 TO 11:POKE PIA+2,I*16+14:
   POKE DD,PEEK(DD) AND 252 OR (NOT I AND 3)
60 FOR J=0 TO 999:POKE J+COLOUR,3:NEXT
70 GET A$:IF A$="" THEN 70
80 NEXT I
90 POKE PIA+2,254:POKE DD,PEEK(DD) OR 3:
   POKE V+24,23:POKE V+17,27
```

The extra memory can be used as a store of high resolution pictures as well. This program shows all twelve memory areas that could contain reasonable pictures. The pictures can be created with a BASIC extension that resides in RAM and saves its graphics under Kernal ROM. In those memory blocks that contain no pictures, you see random memory contents.

High resolution graphics is enabled on the line `40`. The beginning of the line `50` switches block `I` to the segment `3` and switches the VIC chip to the same memory area. The second `POKE` statement selects the block for the video chip. '`(NOT I AND 3)`' filters the extra bits off and inverts the essential ones so that they can be stored to the lowmost two bits of `$DD00`. The line `60` sets the picture's colour to black-cyan. The next line waits for a keystroke before showing another picture. After all blocks have been shown, the original state of the I/O chips will be restored on the line `90`.

5 RAM disk and other programs

As memory expansions are expensive, programs making use of extra memory are rare. Besides, different expansions are not compatible with each other. However, it does not mean that you could not fully utilize the expansion. The most obvious utilization method is a RAM disk. When using VC-1541, it is not only luxury but almost vital condition.

Pekka Pessi has made a couple of programs that utilize the expansion. The software is distributed in two self-extracting archive files (`SFX`es).[16]

The file `ROS-V1.SFX` (for RAM Operating System) holds the source code of the RAM disk program, and some miscellanous files. To load the archive file, use the command "`LOAD"ROS-V1.SFX",`*device*". Then change a blank disk to

[16] If you haven't got the files with this document, you can retrieve them via anonymous FTP from `FTP.FUNET.FI`. The directory `/pub/cbm/documents/projects/memory/c64/256kB` contains files related to this expansion.

the drive before RUNning. The second archive file, UTIL256.SFX, contains the following software:

5.1 Memory test

The program TEST tests the block selection and the whole memory. If it jams before reporting "Test passed", something has gone wrong. Its source code is in the file TEST.A, which requires a library STRING.A. I translated the executable to English by patching the binary file.

5.2 Poor man's multitasking

With the MULTI51200 program you can run four different programs. The program does no multitasking, it only holds four environments in the memory, each in its own memory block. MULTI.A is the source code. After loading it with "LOAD"MULTI51200",*device*,1" and initializing the BASIC pointers with "NEW", you can switch the environments with "SYS51200,f,b". The parameter f is a flag determining if the current block should be copied to the destination block (0) or not (1). The b selects the destination block (0–3). The initial block is 3.

5.3 Machine language monitor

If you don't like to switch the memory blocks manually in your favourite machine language monitor, the MON256 utility is for you. The memory is again divided to four 64 kB blocks, numbered from 0 to 3. The commands are explained in Table 6.

The source code for the monitor is split in the files MON.A, CONSOLE.A, COM.A, ROUTINES.A and TABELS.A.

5.4 RAM disk

The most important utility is a RAM disk program, which occupies about 9 kilobytes of memory. It transfers Kernal and BASIC interpreter to RAM and patches the serial bus routines. The actual program is in the area $00800–$03FFF.

It emulates all VC-1541 functions except relative files. For example, the commands U1, U2, B-A etc. work. UI+ and UI- make no difference. The program also detects some fast loaders and works with them installed.

The loader is called RAM DISC, and the patched Kernal is in RAM.K. You need only patches to the low-level serial bus routines, so you may want to restore the original colors and keyboard definitions. To minimize incompatibility problems, you should replace the ROM chip that holds the BASIC and Kernal ROMs with an EPROM containing the patched Kernal. The RAM disk routines are in RAM.C.

`a` *nnnn cmd* or `.` *nnnn cmd*	Assembles instruction *cmd* to memory address *nnnn*.
`b` *bb ff*	Selects a block. The *bb* holds the block number, and *ff* is a flag. If it is `1`, it directs all memory accessing to RAM. If it is `0`, you can access the ROMs and I/O.
`c` *hhhh iiii jjjj*	Compares the memory area *hhhh*–*iiii* with the area beginning from *jjjj*.
`d` [*hhhh* [*iiii*]]	Disassembles memory.
`f` *hhhh iiii nn*	Fills the memory between *hhhh* and *iiii* with the byte pattern *nn*.
`g` [*hhhh*]	Executes program until a `BRK` is encountered.
`h` *hhhh iiii nn mm...* or `h` *hhhh iiii* `'`*text*	Hunts the memory area *hhhh*–*iiii* for the byte sequence *nn mm...* or for *text*.
`j` [*hhhh*]	Calls a subroutine.
`l` `"`*filename*`"`[`,`*n*]	Loads a program. Default device number is 8.
`m` [*hhhh* [*iiii*]]	Hexadecimal dump of memory.
`>`*hhhh nn mm...*	Stores bytes in memory.
`r`	Dumps the registers (for `g` and `j`).
`;`	Modifies the register values.
`s` `"`*filename*`"`,*n*,*hhhh*,*jjjj*	Saves the memory area *hhhh*–*jjjj*.
`t` *hhhh iiii jjjj*	Copies the memory area *hhhh*–*iiii* to *jjjj*.
`v` `"`*filename*`"`[`,`*n*]	Verifies a program. Default device number is 8.
`x`	Exits the monitor.
`@` [*command*]	Sends *command* to device 8. If it begins with `$`, the disk directory will be read. If no command is given, the disk drive's status will be displayed.

Table 6: Commands for the `MON256` utility

5.4.1 Disk copiers

The program `RAM DISC COPY` copies a regular 1541 disk to RAM. It utilizes the slow `U1` command, and it is included as an example only. The source code `DUP.A` exposes the RAM disk's storage format.

A faster and more useful tool is `FDUPLICATE`. Using it, you can copy a regular disk to the RAM disk or vice versa. You can make multiple copies of a disk easily. This program's fast transfer routines are designed for PAL systems, and the utility cannot be used in NTSC machines without little modification. Its source code is in the two files `S/SUCK` and `S/DSUCK`.

6 Enhancing the expansion

There are a couple of unused contacts in the PIA. In addition to that, two 7405 ports are not connected. The extra PIA lines include an input, CB1, an input/output line CB2, and the Interrupt Request line $\overline{\mathrm{IRQB}}$.

If you connect the $\overline{\mathrm{IRQB}}$ line to the $\overline{\mathrm{IRQ}}$ or $\overline{\mathrm{NMI}}$ input of your system, you can have one or two new interrupt sources, useful for interfacing your custom hardware. And if you are running out of User Port pins, the lines CB1 and CB2 can save you from designing an I/O cartridge. Besides, you can use the $\overline{\mathrm{IRQB}}$ as an output if you wire the CB1 line to something that you can control with software, CA2 for instance. Just remember to add a pull-up resistor to the $\overline{\mathrm{IRQB}}$ line if necessary.

6.1 Built-in freezer

For the Commodore 64, there are several 'freezer' cartridges that let the user to halt theoretically any program (game) to alter it (make the player immortal), or more often merely to make a 'back-up copy' of it. Alas, anything cannot be frozen with these cartridges. If the programmer of the game is clever enough to inhibit IRQ and NMI interrupts in his program, and if the code runs at the area `$0000`–`$0FFF`, no external cartridge will be able to halt it without asserting the $\overline{\mathrm{RESET}}$ signal, which would lose most status information of the computer.

This freezer expansion will let you to replace the program's memory with previously initialized RAM by pressing the Restore key. If the NMI interrupts are disabled, freezing will be done with the BRK instruction. As the circuit forces the $\overline{\mathrm{HIRAM}}$ line to logical zero, the interrupt vectors will always be fetched from RAM. All of the freezer software can be stored to RAM, so it is easy to change, and no EPROM programming devices are needed. Another advantage is that the freezer software may freely use 128 kilobytes of memory, and there are 64 kilobytes of working storage, way more than in the best freezer cartridges.

For this expansion, you need a double ON–OFF or ON–ON switch, four 1N4148 diodes and two 10 kΩ resistors. First do some preparations. Break the connection between the PIA's $\overline{\text{RESET}}$ input (pin 34) and the system $\overline{\text{RESET}}$, and replace it with a diode with the marked end towards the system bus, so that the pass direction is from the PIA to the system $\overline{\text{RESET}}$. Then you have to solder a pull-up resistor between the PIA's $\overline{\text{RESET}}$ line and the +5 V power outlet. Locate the U8 chip (7406) on the motherboard and desolder its pin 6. Solder a diode between the motherboard connection and the pin with the mark pointing to the chip. Finally, solder the other pull-up resistor between the U8's pin 6 and +5 V. After these modifications, the PIA should continue to reset normally, and the Restore key should remain functional.

Now you can mount the switch to the system. If you have an ON–ON switch, hold it in your hand in such an angle that you see two rows of three pins. Solder two diodes from the right-hand contacts of the switch to the U8's pin 6, with the mark pointing to the U8. Solder the PIA's $\overline{\text{RESET}}$ line (pin 34) to either middle contact of the switch, or to either free contact if you have an ON–OFF switch. Finally, mount the $\overline{\text{HIRAM}}$ line, which is on the 6510's pin 28, to the remaining middle contact.

The switch affects in the operation of the Restore key. When it is open, the Restore key operates normally. When the switch is closed, the Restore key also resets the PIA, switching the default memory blocks in, and forces the $\overline{\text{HIRAM}}$ line low, so that the interrupt vectors will always be fetched from RAM. The idea of the expansion is that the program runs in some other memory blocks, say `3`–`0`, with the PIA totally disabled from the address space. The default memory blocks (`$F`–`$C`) will be initialized mostly with null bytes, which is the opcode of the `BRK` instruction. You also need the freezer interrupt vectors (`$FFFA`–`$FFFF`) and a small interrupt handler that stores the processor registers and switches the main freezer program into the address space.

If you want to freeze hanged programs, too, you can do it without losing the state of the I/O chips. By using a custom Kernal ROM, you could even store all 6510 registers except the Program Counter. You just have to be able to reset the 6510 without resetting the rest of the chips. To do this, desolder the $\overline{\text{RESET}}$ signal, pin 40 on the 6510. Connect a 1N4148 diode between the system $\overline{\text{RESET}}$ signal and the 6510, with the mark pointing away from the processor. Add a pull-up resistor to the signal, and a switch between the logical ground and the signal.

Now, whenever you push the switch, only the 6510 will be reset. You should then use the Autostart code (`$C3 $C2 $CD $38 $30` at `$8004`–`$8008`), and the Kernal will jump to the vector `($8000)`. Unfortunately it will destroy all registers except the `Y` register when searching for that code. You could make a slight modification to the ROM that first saves all registers on some memory area, like somewhere in `$90`–`$FF`. There are some `$AA` bytes in the Kernal ROM that you can take into use. Just relocate the $\overline{\text{RESET}}$ vector (`$FFFC`) to your routine, and remember to jump to `$FCE2` at end.

Alas, I just have invented a software hack that cannot be frozen with this circuit. If a program runs on the I/O area (`$DE00`–`$DFFF` except the range where the PIA is mapped to while the I/O is switched on) and has disabled the NMI interrupts, the program will continue running even if you have activated the freezer and hit the Restore key. Should you ever encounter this type of a software hack, you can use a triple switch and add the $\overline{\text{LORAM}}$ signal (6510's pin 29) to one extra contact. Connect the remaining contact to the U8's pin 6 through a diode with the mark pointing to the U8. This will disable the I/O area for the time the Restore signal is active, so the processor can fetch a `BRK` from the memory underneath, and will always freeze correctly.

I haven't completed the freezer expansion yet, as the daughter-board in my faithful C64 stopped working when I opened the cover to start the surgery operation. My first attempt of rebuilding it did not succeed, and now I am in Germany, more than one megameter away from the computer. When returning to Finland in August 1994, the freezer expansion will be one of my first projects I am going to finish.[17]

Naturally you also need a program to utilize the freezer circuitry. You don't need to code so much, only the routine to jump back to the freezed program and the routine that saves all registers upon freezing need to be written from scratch. You can patch existing machine language monitors, sprite editors and utilities like that, only the data fetch and data store subroutines must be rewritten. Developing the freezer software is very simple, since you can access the frozen program through a 16 kB window, and you can use the Kernal routines all the time.

6.2 New operating system

I had purchased a PC board that allows you to choose between the Kernal ROM and a custom 8 kB EPROM (2764). I never added a fastloader or other useful routines to the Kernal, since I should have deleted some routines, i.e. the Datassette and RS-232 handling.

But if you had more ROM, you could keep your new Kernal fully compatible with the old Kernal while adding new features to it. A 32 kB EPROM would easily hold the RAM disk routines, the disk copier and routines for adequately fast loading and saving. Still you would have plenty of space for controlling your own expansions, like a IEEE-488 interface or a SCSI bus.

Figure 6 shows the pinouts for the original Kernal ROM (2364) and the replacement chip (27256). Connect the address lines A0–A12, the data bus (D0–D7) and the power supply lines (V_{CC} and GND) together, and connect V_{PP} to V_{CC}.

The signals $\overline{\text{CS}}$, $\overline{\text{CE}}$ and $\overline{\text{OE}}$ are for chip selection. Connect $\overline{\text{CE}}$ and $\overline{\text{OE}}$ together, and add 4.7 kΩ resistors between both chips' VCC and chip selection lines. Then add a ON-ON switch between 2364's $\overline{\text{CS}}$, 27256's $\overline{\text{CE}}$ and $\overline{\text{OE}}$, and

[17] So I thought at that time—as of December 1999, I have neither fixed that particular C64 nor completed the freezer expansion. Besides, the expansion would not have much advantage over freezer cartridges or the features present in some emulators.

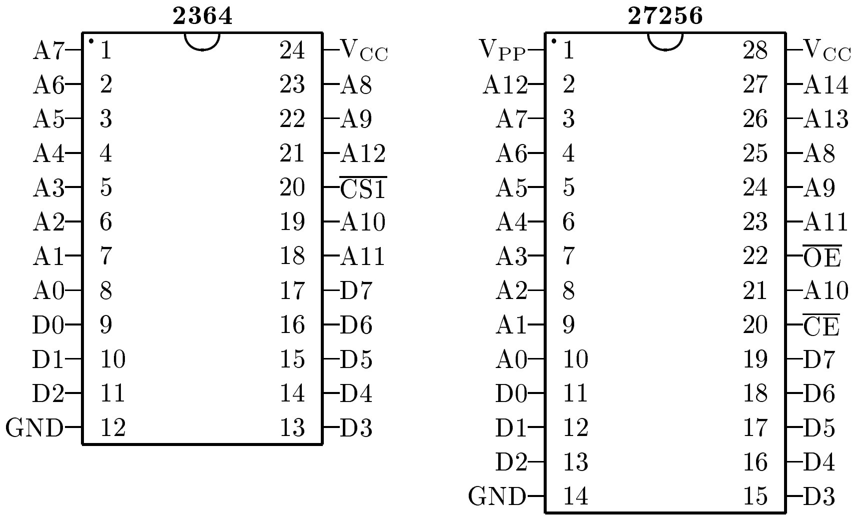

Figure 6: The Read-Only Memory Chips 2364 and 27256

the $\overline{\text{KERNAL}}$ line that comes from the motherboard originally to 2364's $\overline{\text{CS}}$ pin.

Now you have only the lines A14 and A13 left. PIA's CB2 can control one of them, but what about the other? No problem, the $\overline{\text{IRQB}}$ line can be used as an output if the CB1 is connected to CA2 or CB2. Add 4.7 kΩ pull-up resistors from CB2 and $\overline{\text{IRQB}}$ to +5 V, and lead the signals to the EPROM's free address lines.

Now those A14 and A13 lines are high upon start-up, and the computer will see the EPROM area `$6000`–`$7FFF` instead of the original Kernal ROM, if the switch is in correct position. That part of the EPROM should initialize the PIA. Note that altering the CB2 or the $\overline{\text{IRQB}}$ lines immediately causes a jump to another EPROM bank.

7 Contacting the author

If you have anything to ask or comment, feel free to contact me. If you complete the expansion, I would like to receive an E-mail message or a postcard from you, so that I know I didn't write this document in vain. Finally, if you make any software for the expansion or patch existing programs to make use of the extra memory, I would love to put the software available on the FTP site `FTP.FUNET.FI`.

My addresses are:

InterNet: `Marko.Makela@FTP.FUNET.FI`
Mail: Marko Mäkelä
Harakankuja 6 D 19
FIN-02600 Espoo
Finland