Ula hacking : Double buffering of video Ram

compiled from <comp.sys.oric> by André C.

From Mickaël P.:

Here is a crazy project that came out while talking during the January CEO meeting in Paris. One of the reasons that makes it hard to do clean animations and games on the Oric, is partly due to the fact that we didn't get vertical refresh synchronisation, and also to the fact that we didn't get any way to perform «double buffering»: drawing somewhere in memory while the video chip display something else.

The vertical refresh problem has been solved, using the «RGB synch into tape in» trick. But well, it's not of real use because we can't display anything interesting. Now, if we were able to dynamically change the screen address, that synchro will come in handy to help us doing the switch at the right timing :) So, well, the question is: how can we force the Ula to read from somewhere else than $A000 or $BB80 ?

If you have a manual that shows the Oric memory map of both 48k and 16k machines, you would probably have noticed that the address of screen memory is different on these two types of machines. Considering that the Ula is probably the same in both machines, we can safely say (I hope, please tell me if I'm wrong), that the different addressing must be done using some memory addressing trick.

The important addresses to consider are: (48k/16k)
- TEXT STD CHARSET: $B400 / $3400
- TEXT ALT CHARSET: $B800 / $3800
- TEXT VIDEO MEMORY: $BB80 / $ 3B80
- HIRES STD CHARSET: $9800 / $1800
- HIRES ALT CHARSET: $9C00 / $1C00
- HIRES VIDEO MEMORY: $A000 / $2000

In binary, we have the following values:

1010000000000000 = $A000
0010000000000000 = $2000

In short, the only difference between accessing the 48k address and the 16k address is the status of the last significant bit of the address. Starting from this observation, we got the idea that perhaps it was possible to use something like PB5 to control a small switch that would dynamically perform this 0/1 switch, and thus force the Ula to read data from the top part of bottom part of memory, effectively acting as an hardware double buffering :)

Wanting to start experimenting, we simply removed an Ula, put the pin number 30 (address bus 15th bit) in top position, and switched on the machine. It freezed, but well, the display was still there.

I suppose we would have been lucky if it has worked, because when we started again looking to the Oric schematics, the relationship between how Ram address/data is accessed by the 6502 and Ula is far from being obvious. It looks like there are not 16 lines for selecting Ram, so there must be some black magic here and there. Perhaps simply acting on the bank selection would do it?

So I ask to anyone here that has a thorough knowledge of the Oric internal if this kind of address switching is doable, and if yes, what it would require hardware wise :) Thanks for reading!

Answer from Jani T.:

[How can we force the Ula to read from somewhere else than $A000 or $BB80?] Really, there isn't way.. =)

[The different addressing must be done using some memory addressing trick] Actually... It might be so that in 16k machine all $2000 addresses are really mirrored in $A000, and this is done somewhere between Ula and RAM.

[Perhaps simply acting on the bank selection would do it?] Greatly depends...

From John W.:

[Drawing somewhere in memory while the video chip display something else] Mad idea of the month: It may be possible to synchronise using 100% software... Using assembler, If we read a timer, clock, or other regular synchronisation source, then loop and read it again, and monitor the time between the last read, and log this in a buffer, we should see some side effect of the screen update in our timings as a non-uniform interval. We then try various offsets from this glitch as potential screen update points, until we get a smooth update.

If that works, then we write code to quickly find the glitch, just before we update. In high performance code, we would need to balance all the branch paths with NOPs, and could count the cycles, predicting the next Vrefresh and avoid repeated resyncs, do a resync occasionally, etc. Don't the VIAs have timers in them? I seem to remember something like this when I last programmed an Oric nearly 10 years ago! Even if the Orics timers are too hit and miss, we could develop clock divider/counter hardware, read this in software, find a glitch somewhere, then try and correlate this with something else. Connect the mic to the tape out, and try reading noise on the mic socket, anything. A refresh is a big world changing event in our little Oric...

If this worked, then semi-automatic tools could be used to time instruction streams, and pad them out with NOPs, to minimise resyncs.

Answer from André M.:

[In high performance code, we would need to balance all the branch paths with NOPs] Not actual NOPs

though, as NOP takes two cycles and the difference between a branch taken and a branch not taken is one cycle. There's also the complication that, if I recall correctly, branches take one more cycle if the target location is on a different page.

[Don't the VIAs have timers in them?] This might be the best approach. Since everything runs off the same clock, programming timer 1 to trigger an IRQ every 19200 cycles should lock you to the half-frame frequency. I haven't actually tried it but I think Mickaël has. The remaining problem is that you have to adjust the phase manually.

[Semi-automatic tools could be used to time instruction streams, and pad them out with NOPs, to minimise resyncs] The idea has crossed my mind too, but it smells like a difficult task, especially if it's to be done in a reasonable amount of time (as opposed to blindly tracing all possible code paths).

Answer from John:

[The remaining problem is that you have to adjust the phase manually] Fantastic!

[The idea has crossed my mind too, but it smells like a difficult task, especially if it's to be done in a reasonable amount of time (as opposed to blindly tracing all possible code paths)] I was thinking along the lines (for new code) of structuring the code into streams, and using an ICE/Logic analyser to check dynamic branch cost, an average cost would do... Even inserting timer reads to work out average branch cost would be OK for new code. For existing apps, the ICE would be invaluable, and just the main paths are worth syncing... The plan could be to use an ICE made from a 6502 core running in a gate array 8-)All we need to get is a dynamic address trace with timings. Or perhaps we could generate interrupts and sample the PC in s/w to discover the hot loops... We then patch these to call a stub, and balance the segment with the stub. The better and cheaper our re-sync code is, the less work we have to do...

Answer from Fabrice F.:

[We didn't get any way to perform «double buffering»: drawing somewhere in memory while the video chip display something else] Ok, here is a new idea for a double screen that could be much more easy to implement... :-) On those Oric boards with eight 4164 chips, it should be easy to replace these chips with 41256 ones (yeap, 256K x1 bits). The pinout is identical, except the additional A8 row/column line (pin #1). So, the mod just consists in cutting the tracks coming to pins #1, and tying all these pins to PB5 or PB6 for example. This way, you have access to two banks of 64 KB.

This is not very interesting for all double buffer techniques because you only have access to one bank at a time, but hey, this still means 128KB Ram on your Oric... :-) Yeap, 128KB is wasted: trying to access the full 256KB raises the same timing problems than the ones we discussed earlier... Beware that switching between two 64KB banks is not easy with the 6502 : you must do the swapping in the Rom, and you lose the current stack, page zero and so on... Is this still interesting?

I'm going to try it... it should work if the Dram refresh only needs to be done on A0-A7, this is surely true if the grid is rectangular. Ok, no doubt it works :-) I just need to have 41256 chips... Damn'd! Mike was selling those chips, and I haven't had this idea earlier! :-(

Phew, I can see that nobody bidded... Are you going to retry eBay, Mike, or may I buy your collection of Dram chips for £9?

Answer from Mickaël:

[Cutting the tracks coming to pins #1, and tying all these pins to PB5 or PB6 for example] Easier? Well, it means unsoldering all the original Rams, since they are not socketed.

[You must do the swapping in the Rom, and you lose the current stack, page zero and so on...] Not very easy to deal with that. Without modifying the Rom, I don't really see how one can perform the transfer / duplication of code that allows you to switch bank while still executing code. I suppose that in this scheme, even the overlay Ram is duplicated? I try to think of a way, but hum, does not seem straightforward at all :)

Answer from Fabrice:

[Unsoldering all the original Rams, since they are not socketed] Yeap, easier for those of us who already have all their Ram chips socketed, and it's also a simpler design... ;-)

[I suppose that in this scheme, even the overlay Ram is duplicated?] Absolutely right, the first time you switch to the second 64K bank, there's nothing in Ram you can rely on... So you must at least have a routine in Rom that moves data from one bank to the other... It would be practical to also have a routine in Rom that executes a routine in the other bank...

[I try to think of a way, but hum, does not seem straightforward at all] Yeap, the only thing that is retained when you switch from one bank to the other is the Cpu registers, and the I/O chips registers...

From Mike B.:

[Are you going to retry eBay, Mike, or may I buy your collection of Dram chips for £9?] If you have access to PayPal, drop me an Email and we can sort that out.

From Jani T.:

[It should work if the DRAM refresh only needs to be done on A0-A7, this is surely true if the grid is rectangular] Actually most Dram chips refresh them selves row-by-row basis, so this is usually not a problem. Of course you would need to deal with duplicating memory but hey, how about leaving bottom (#0000-#9FFF) 32k untouched and just bank switch upper 32k? That could even work. Of course this still leaves that access problem, now what if CPU could access different bank than Ula.

[Damn'd! Mike was selling those chips, and I haven't had this idea earlier!] I might have some 41256 chips around here... Yes, I have darn old 286 mobo that is

loaded with 41256A-12 chips =) How many you need?

Answer from Fabrice:

[How about leaving bottom (#0000-#9FFF) 32k untouched and just bank switch upper 32k?] That's a good idea: 96 KB Ram only, so switching from one bank to the other won't be a problem. Pins #1 of the 41256s would then be driven by something like PB5 AND (A15 OR NOT PHI2). Thus the Ula cycles would always take place in the bank specified by PB5, whilst the Cpu cycles would take place in the bank specified by PB5 AND A15...

[This still leaves that access problem, now what if Cpu could access different bank than Ula] I had this idea too, as an improvement over the first scheme... Using two different bits to control which bank the Cpu has access to, and which bank the Ula has access to. Pins #1 of the 41256s would then be driven by something like (PB5 AND A15 AND PHI2) OR (PB6 AND NOT PHI2).

[How many you need?] Ha, I sometimes act quickly :-) I've already sent a mail to Mike to buy his chips. Thanks for the kind offer, though. You will surely need these chips too!

Answer from Jani:

[That's a good idea .../... something like (PB5 AND A15 AND PHI2) OR (PB6 AND NOT PHI2)] Actually this is really getting close to solution used in C64 bank switching. Now it could be used so that highest address (because it's under ROM anyway) could be used to control full 8 bit banking, with 3 pieces of 3-to-8 decoders you could point 20 banks, meaning total of 640 Kbytes of Ram.. That should be enough for anybody as Mr. Gates has stated =) Actually you would end up with 640 + 32k lower RAM+ 16kb ROM... Hope that my calculations were right, but this is feasible pretty easily - necessary? I doubt... For a really tight operation, you could tweak bank switching starting from #400 (from up that there isn't any really important stuff...), so you could get almost 128 Kb of free Ram to mess with...

[You will surely need these chips too!] Actually I have at least 40 of them, and what I really want to try is to use old 256kB SIMMs that I have loads, of course SIMM needs a bit more soldering but would be easier to obtain...

Answer from Fabrice:

[Really, there isn't way] Maybe we can lure the Ula in thinking it is still reading at $A000 or $BB80, but have an external memory mapping...

[It might be so that in 16k machine all $2000 addresses are really mirrored in $A000, and this is done somewhere between Ula and Ram] Right, you can still consider the screen to be in $A000 or $BB80 on a 16K machine. If you have an issue 3 board or earlier, with 8 Ram chips (Steve has a nice collection of pictures on <http://www.48katmos.freeuk.com/oric1.htm>, you might have noticed LK1 has two possible settings (b-a for 16K Ram, a-c for 64K), and maybe this is this LK1 that gave you the idea of switching between a 16K Oric and a 64K Oric: forget about it... I guess that eight 4116 chips (or more exactly eight 5V-only flavours of these chips) were intended to be used on such boards in order to have a 16KB Oric : those chips only have 7 address lines instead of 8 address lines for the 4164 chips that are used on 64KB Orics. Remember that we are talking of Drams: the 16-bit address is split in two before being sent to the chips. When the memory cycle is initiated by the Ula (in order to read the video memory or the character sets), the Ula successively sends the two parts of the address (these are called the 'Row' part and the 'Column' part: think of the Dram as a 2D grid).

The Ula doesn't know if you have 16K or 64K, so it always uses 8 address lines for the row and the column specification. Since the 4116 chip does not have the 8th address lines, the value of this 8th address line is lost, both for the row and the column number... You would effectively have 16 Ko with those chips installed, but with a very strange memory map (A7 having no effect, each block of 128 addresses would be duplicated). LK1 is a very strange attempt to have this memory reorganized, so strange in fact that it doesn't work...

[I ask to anyone here that has a thorough knowledge of the Oric internal if this kind of address switching is doable, and if yes, what it would require hardware wise] So, you are wondering what would happen if you changed the LK1 setting during execution ? (ouch :-) First, I think it's not that easy to do (you would need to control a very fast switch with 2 positions).

Supposing you manage to do it, you will come up with your data completely reorganized in memory You might love it for some screen effects, but all your code and your data will also be reorganized...

The effect is not easy to depict... The Ula controls the two LS257 multiplexers that allow to either send A0-A7 from the Cpu address bus or an internal Ula-generated 8 bit address (including the A8-A15 Cpu address lines that the Ula has access to). At least this what happens in the normal a-c setting, when you switch to the a-b setting, you have pin #11 of IC20 (LS257) fed by A7 (from the Cpu address bus), instead of A6 from the Ula.

I find this **extremely** strange... In fact, I think that it is an error on the circuit board: pin b of LK1 is connected to the A7 line coming from the Cpu. IMHO, this is no-sense. I mean, this LK1 must be a reminder of older prototypes and can only be used with the a-c setting.

Mhhh... let's take an example of what happens with LK1's b-a setting. Let's say that you want to access address BB80 with the Cpu... Row part of the address is sent first, so the Ula controls the multiplexers in order to let A0-A7 flow in. Then, the column part of the address is sent: the Ula already has the high part of the address (A8-A15) in it, and it forwards it to the multiplexers, and it controls the multiplexers so that the address coming from the Ula is the one that reaches the Ram. But, we said that A7 from the Cpu address bus replaces A6 from the Ula when LK1 in is the a-b setting, so instead of receiving A14, the Ram receives A7. That is to say: address FB80, in overlay Ram (ok, you have found a new way to access overlay Ram without a Microdisc or such external extension ;-)

Now, what happens when the Ula wants to display the screen and accesses video memory location BB80? The multiplexers are always controlled such that the Ram address lines receive what the Ula sends to them, except for bit 6 of each part (row, column) of the address (instead, A7 from the Cpu is inserted). So, the Ula reads an undeterministic address instead! Suppose now that you agree with me and consider that the b pin of LK1 is wrongly connected to A7 from the Cpu, and thus you cut this connexion and instead you connect b to A7 from the Ula (pin #39) (this doesn't make a lot of sense either, but maybe this is what the designers had in mind?) Now you have a defined address: FBC0. Quite weird, though...

So, I don't see how you could use this for some sort of memory-map reorganisation...

Mhhh... I have been a bit long, and I don't really bring hope... Let's think again at what you want to achieve: having two memory screens, one in $3B80 and the second in $BB80. And a hardware mod that forces the Ula to read one of these two screens...

Mhhhhh.... (oops, I sound like a cow... ;-) To achieve this, I think we shall have a sort of switch inserted between Ula's pin 39 (A7) and IC20's pin #14. The aim is to replace the Ula's A7 output with a 0 **only** when the Ula sends the column address part (i.e. A8-A15), i.e. when CAS is activated. Of course, you must be able to control this switch in software, let's say with PB5 for example. Let's imagine that you insert a new LS257 (only one of the 4 multiplexer will be used): the multiplexer has two inputs (Input A: A7 from the Ula, Input B: ground), its output is sent to pin #1 of IC20, and it is controlled by «PB5 AND CAS» (CAS here is the positive signal issued by the Ula). This way, if PB5 is 0, you always select Input A, and thus you have the usual behaviour. When PB5 is 1, CAS controls the multiplexer, so the Input A is still selected when a row number is sent to the Ram chips, but Input B (0) is selected when the column number is sent. Hey... it might work :-)

Two potential problems:

1- When PB5 is 1, the multiplexer is also controlled by CAS for memory cycles of the Cpu. This means that all accesses to a Ram location in the $8000-$FFFF range is reduced to $0000-$7FFF, and so you can only use the memory screen that is currently displayed (not good for double-buffer techniques). Mhhh... if we add Phi2 to the control logic (i.e. «PB5 AND CAS AND NOT PHI2»), we should have only the Ula memory cycles affected. Full 64K range accessible by the Cpu, and PB5 selects the screen displayed by the Ula, miam :-)

2- The timing waveforms of the 4164 datasheet (thanks again, André) show that the column number should be ready before the CAS pulse arrives. This is not the case with the above design. I don't know what happens if you change the address during a CAS pulse. If the Ram chip latches the address when the CAS pulse arrives, it might not work... :-(Can anyone validate or invalidate this trick?

Again from Mickaël:

[Maybe we can lure the Ula in thinking it is still reading at $A000 or $BB80, but have an external memory mapping...] It's actually what I was hopping to get :)

[If you have an issue 3 board or earlier, with 8 Ram chips, you might have noticed LK1 has two possible settings (b-a for 16K Ram, a-c for 64K), and maybe this is this LK1 that gave you the idea of switching between a 16K Oric and a 64K Oric : forget about it...] I didn't noticed anything like this :) The idea was just that the Ula was probably the same in the 16k and 64k machines, and so it was probably just some smart addressing modification that makes it access in $2000 instead of $a000.

[So, you are wondering what would happen if you changed the LK1 setting during execution?] Not actually, since I didn't even know what was this LK1 thing before today!

[You might love it for some screen effects, but all your code and your data will also be reorganized...] I just want to get the Ula tricked, not the rest of the machine :)

[A hardware mod that forces the Ula to read one of these two screens] Keeping my breath waiting for the revelation.

[Hey... it might work .../... Can anyone validate or invalidate this trick?] Hum, can't help on that. I have «theorical» ideas of «concepts», but I'm totally unable to do any hardware thing myself :(

Answer from Steve M.:

[The Ula doesn't know if you have 16K or 64K, so it always uses 8 address lines for the row and the column specification] They did say more than once in OUM that the 16K had a different Ula. I can confirm that I've tried swapping with Ulas from Atmos and 48K Oric-1 with no difference noticed. On the subject of Ula design I think Dr Paul Johnson has the design etc. Dave Dick was trying to get detail from him around the time Mike released the Unofficial Guide, but Dr Paul was reluctant to release information, as he didn't know who owned Oric. I think his place of work and email were released in a CEO a few months back. Perhaps someone can contact him and persuade him to reveal some information. Even if he only reveals part of the information we could ask for the part, which we most need to know.

From Mike B.:

[Perhaps someone can contact him and persuade him to reveal some information. Even if he only reveals part of the information we could ask for the part which we most need to know] Because there's a big difference between asking for the full design spec and asking a «Do you remember if it works this way or that way?» kind of question. You might get away with the 2nd :)