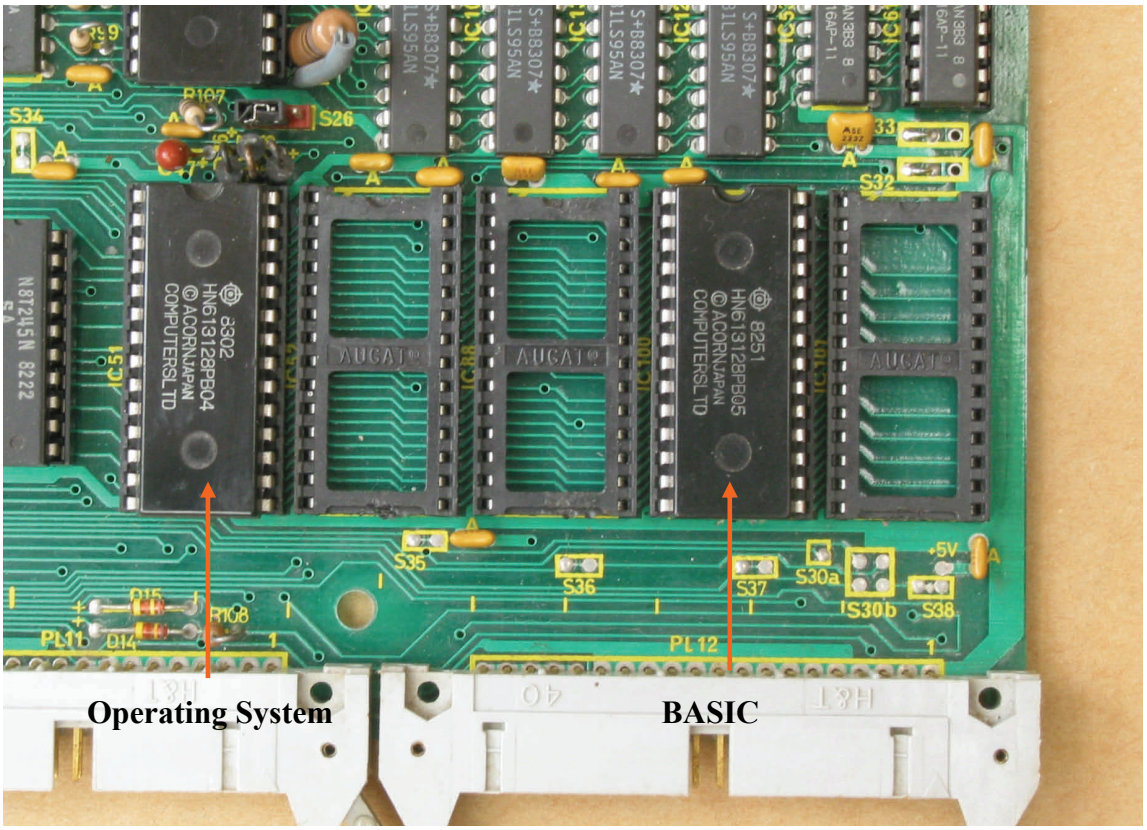
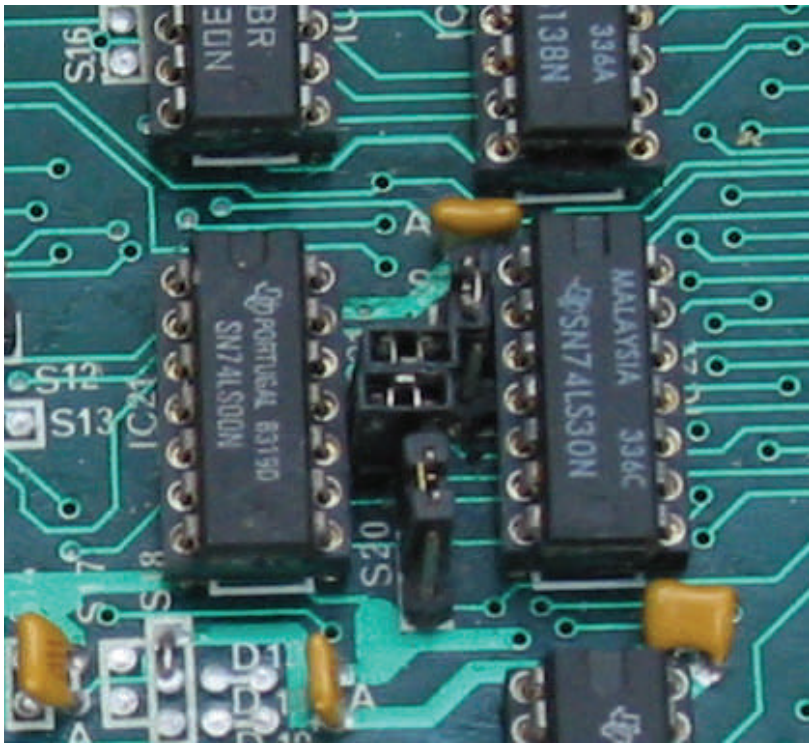
Quick installation guide

As an initial step we suggest stripping the BBC Micro down to a fairly basic configuration. Any existing ROM expansion board will need to be removed. The picture below shows the four 'sideways ROM' sockets with BASIC in one of them. The 28-pin chip on the left is the operating system ROM and is not one of the sideways ROMs.

Ensure the machine powers up properly with BASIC's '>' prompt. It may be unwise to proceed further with the installation until you have achieved this.



If the computer fails to work then check the links carefully to the left of the User VIA (40-pin 6522 chip). These are often changed when expansion boards are installed, hence when any such board is removed the links must be put back in their default position. See next page.

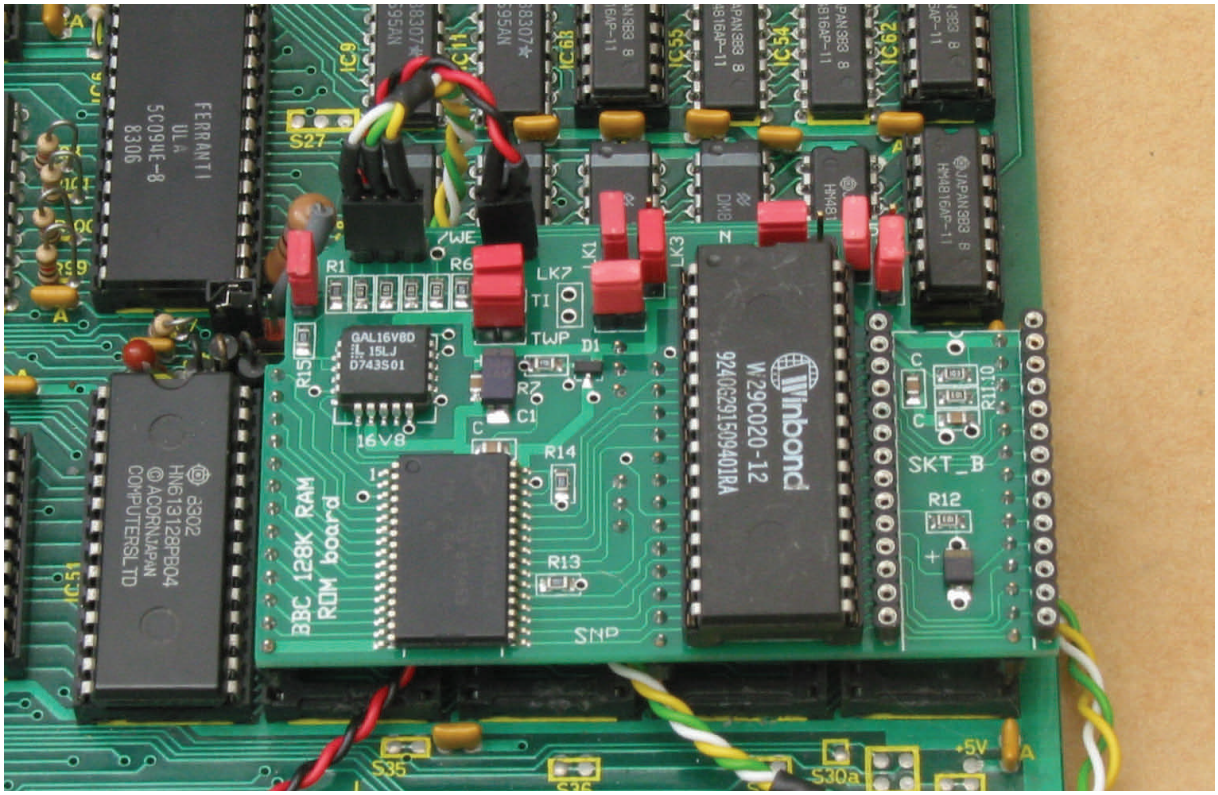


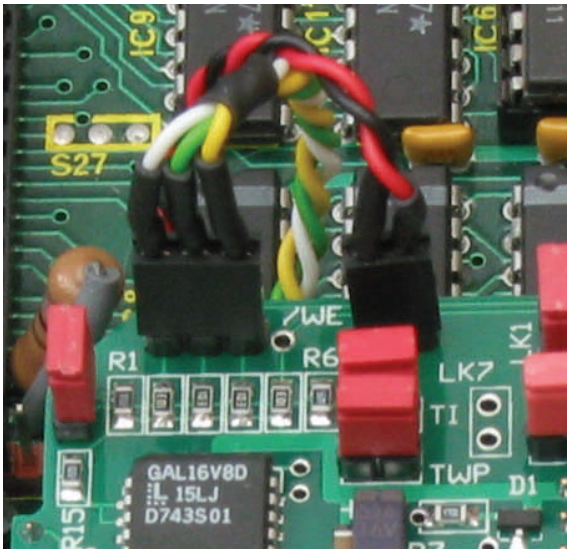
Default link settings: Links S22 and S20 are both in the North position and the two links that make up S21 both run East-West

There is a strong argument for removing the main board from the case when installing the upgrade board. Three wires are used to pick up signals for the RAM/ROM board and the neatest approach is to solder them onto the bottom of the main PCB. Red spring loaded probes can be used for those who would prefer not to solder directly to the main board.

The upgrade needs to pick up a minimum of three signals from the motherboard. Lengths of green, white and yellow wire are used for this. At the ROM/RAM board end you can either solder the wires to the pins directly, or solder them to the 3-way female housing as shown below. Some heat shrink will tidy and reinforce the joints.

Remove the BASIC ROM. The ROM/RAM board has a number of pins on the underside which go into the sockets on the motherboard. The picture shows roughly how the RAM/ ROM board fits, but note also the picture on the next page. Notice that pin 1 on the leftmost socket does not get used whereas the rightmost row of pins on the rightmost socket has all 14 pins occupied. It is difficult to see what is happening to the pins in the middle but in practice if the leftmost row of 13 pins is correctly aligned, likewise the row of 14 pins on the extreme right, all will be well and the upgrade will push home very easily.



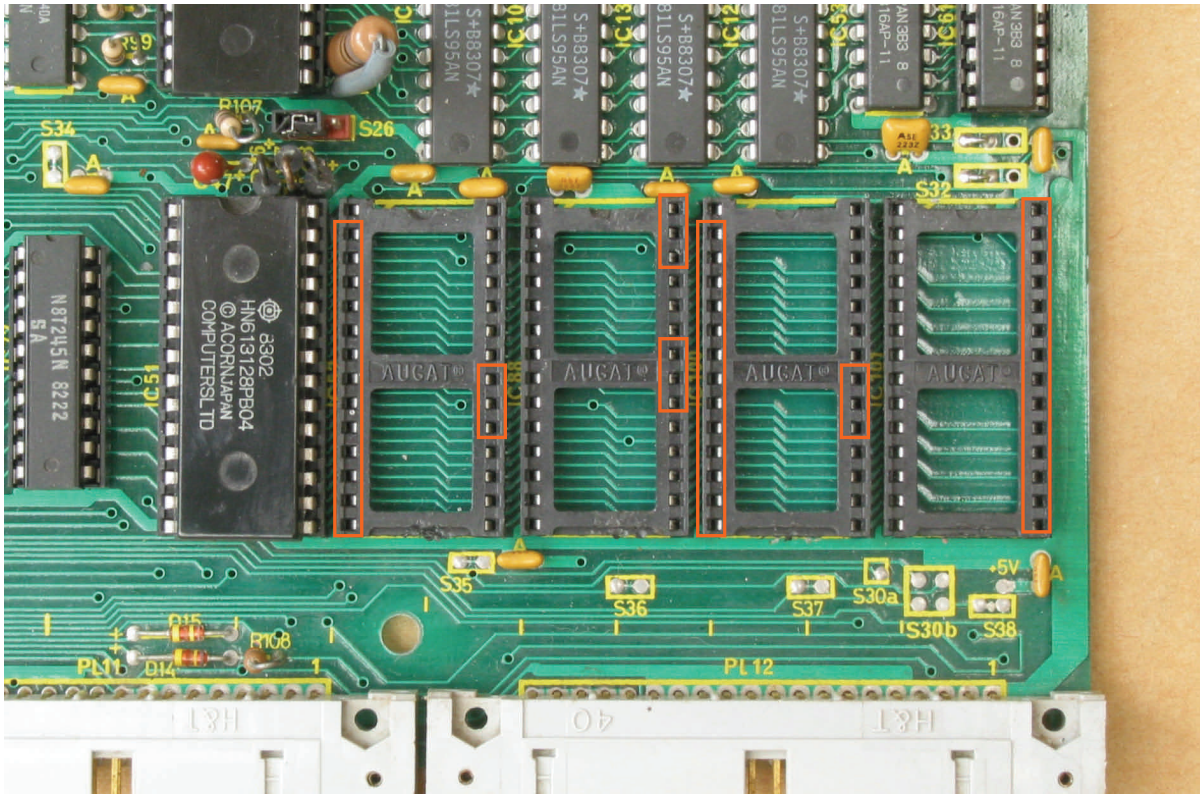


Left: Close up of the white, green and yellow wires. Also shown is the red and black pair used for the battery backup system. Note that the connector for the battery is not polarized so take care get everything the right way around. There are + and - signs on the RAM/ROM board. A diode and resistor on the upgrade board means that no harm will occur if the polarity is wrong - it just won't work properly.

The picture below shows exactly which parts of the four sockets are used by the ROM/RAM board. The parts of the sockets enclosed by an orange rectangle should have pins on the ROM/RAM board inserted into them. Study the bottom of the upgrade closely and you will see how the actual upgrade itself relates to the picture below. Specifically, there are two rows of 13 pins, one row of 14 pins and four groups of 3 pins. You will notice that pin 1 on the leftmost socket is not used.

The rationale behind this was simple maths. Pin 1 on the motherboard sockets are not used and the rows of pins come in lengths of 40. By using two rows of 13 and one of 14, wastage was kept to a minimum (13x2+14=40)

The pins are more than strong enough in a vertical sense. They are, however, very unforgiving about being bent sideways. As with ordinary DIL chips, typically this happens when a board is levered out of position, with one end suddenly springing free and the other still firmly in place.

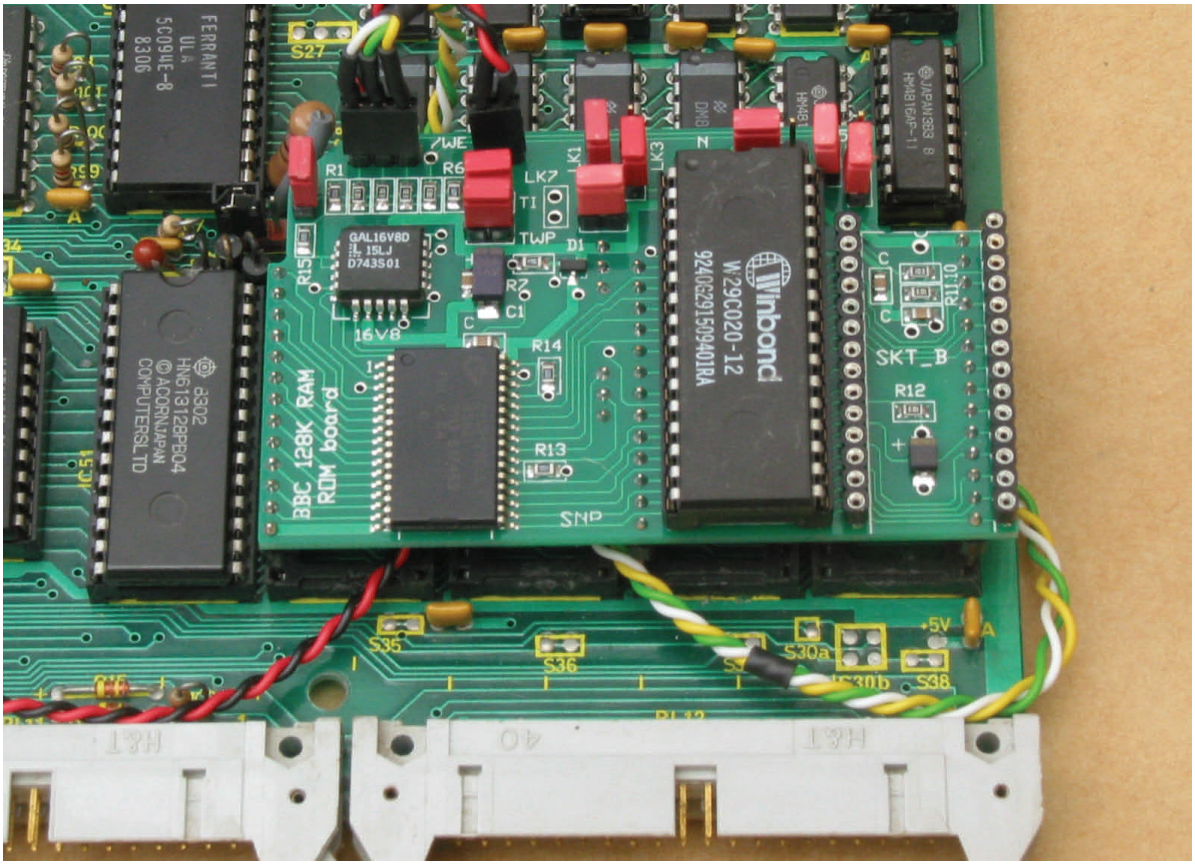


The green, white and yellow wires need to be connected as follows;

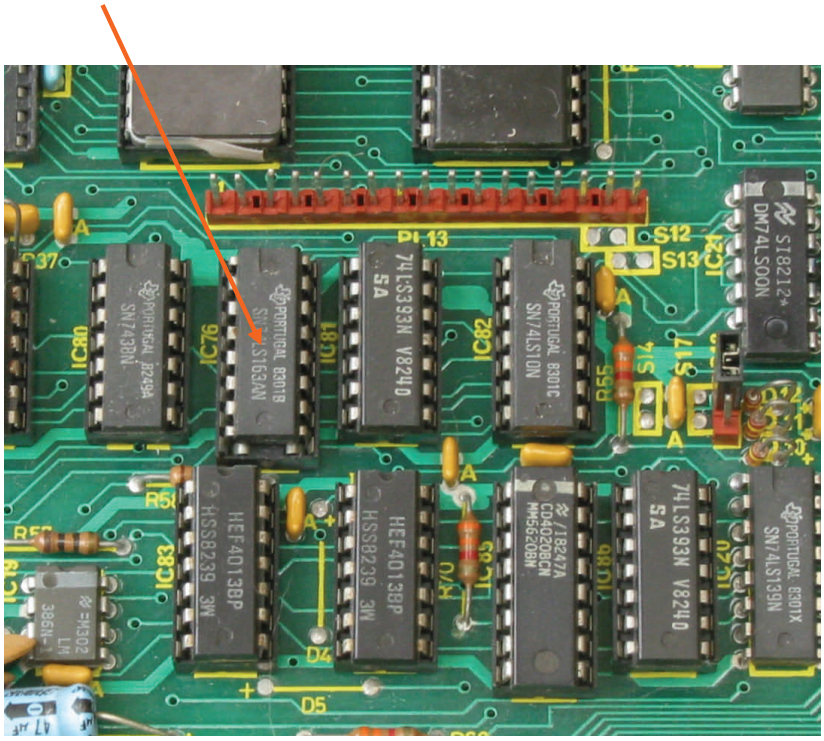
White to pin 4 of IC77 (or pin 10 of IC33, use either)
Green to pin 12 of IC76
Yellow to pin 11 of IC76

In the picture below, the three wires are twisted together for neatness with some short pieces of black heat shrink added every few inches to help prevent them untwisting. Route them round to the underside of the motherboard and solder them directly to the bottom of the relevant ICs. The red/black wires go to the optional battery pack for the eight banks of sideways RAM.

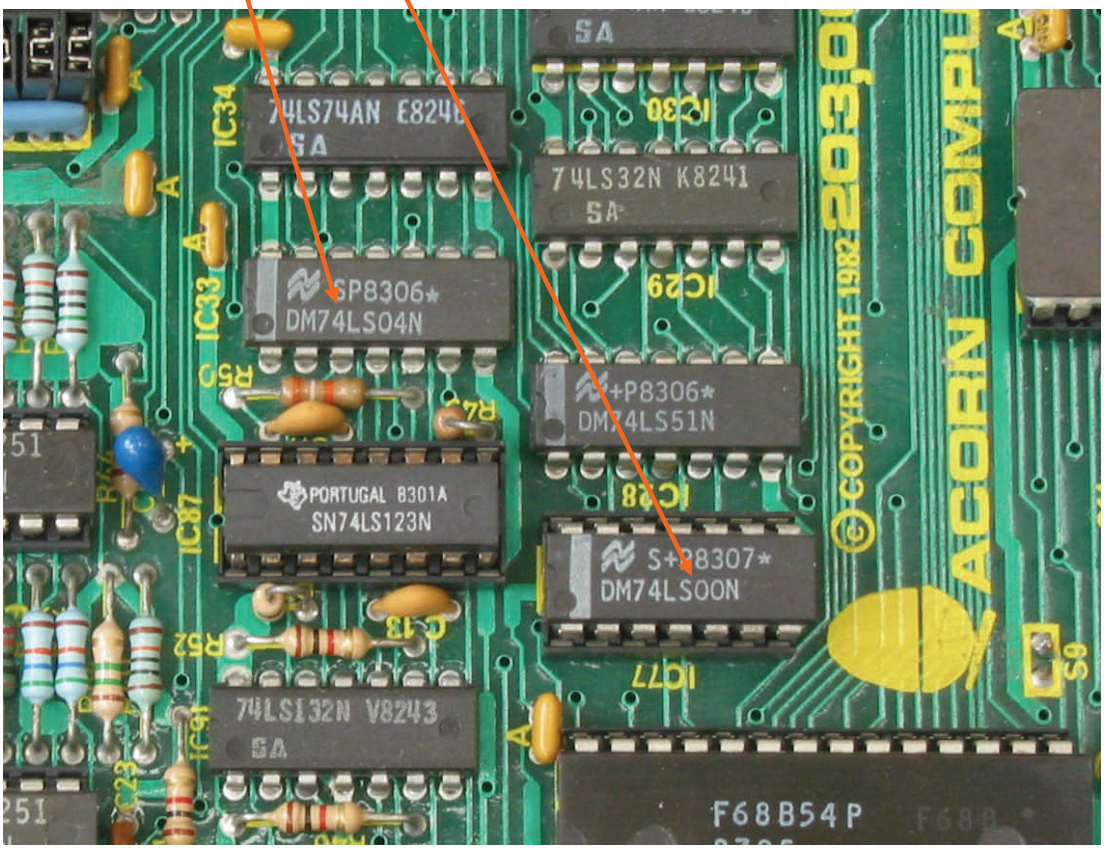
The wires can be soldered on the top of the motherboard although it is arguably less neat when done this way.



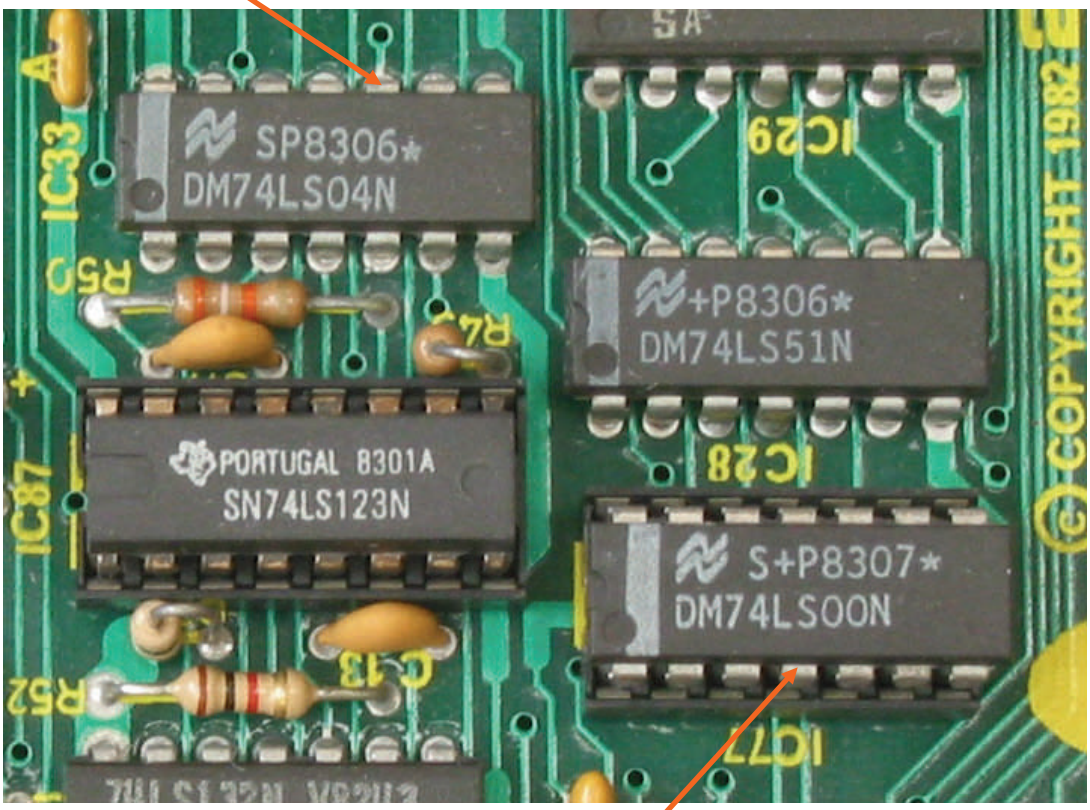
IC76 is by the keyboard ribbon cable connector.



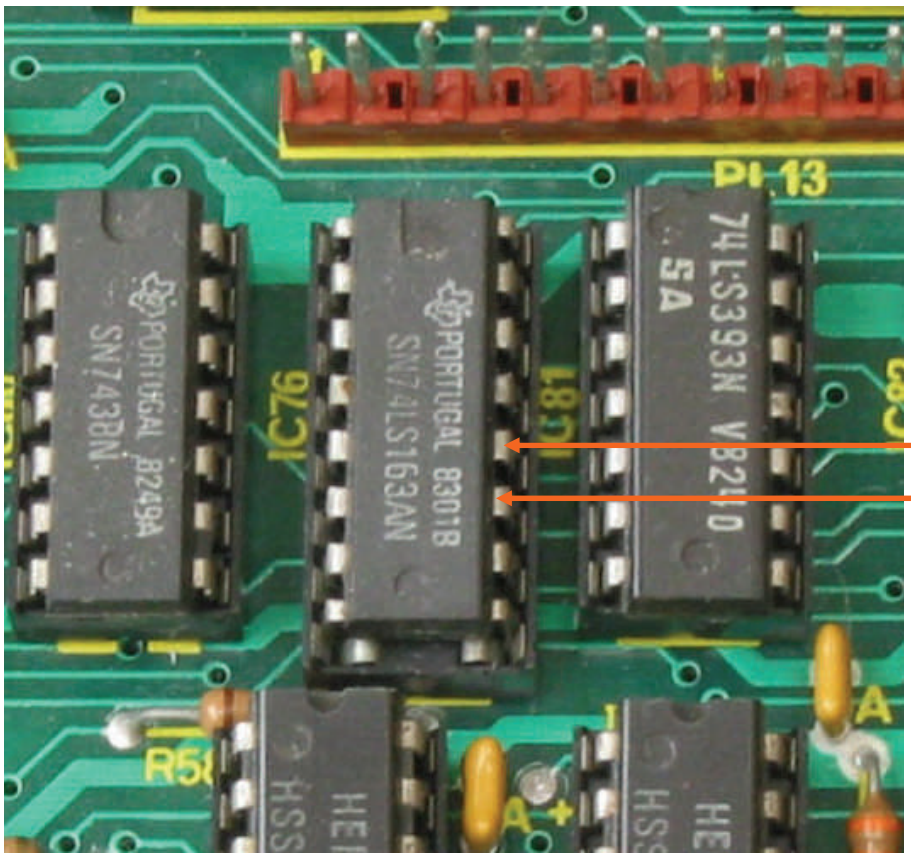
Below: location of IC33 and IC77



Pin 10 IC33



Pin 4 IC77

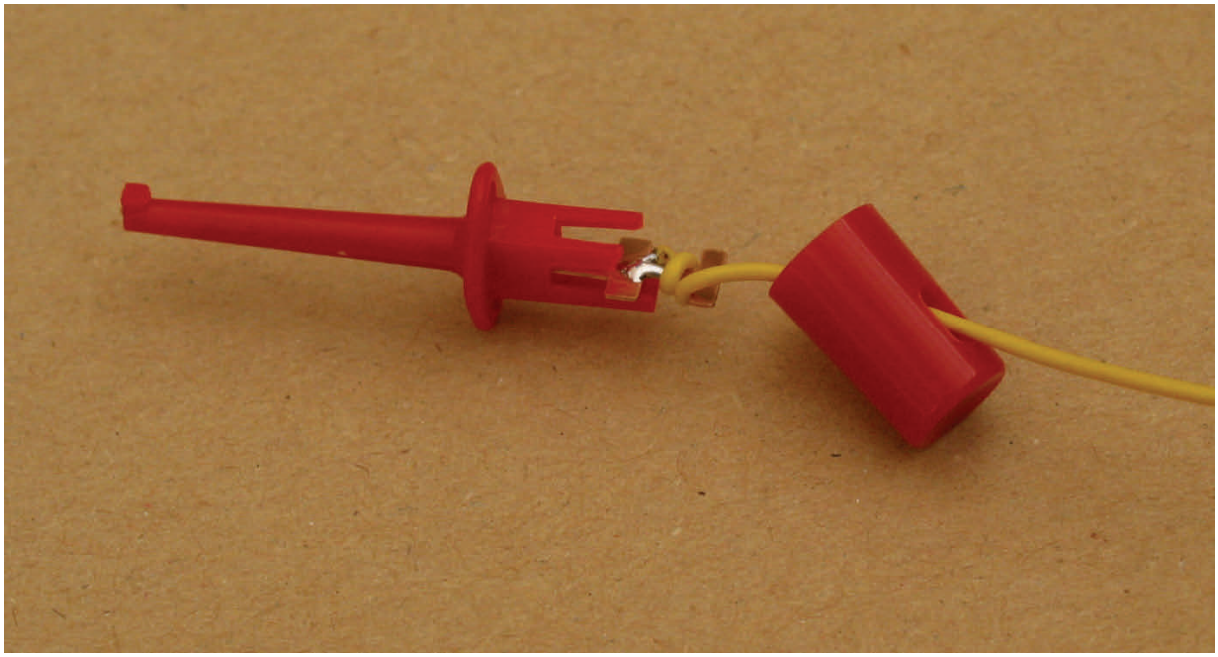


Pin 12

Pin 11

The green, white and yellow wires need to be attached to the pins of some ICs on the BBC Micro's main board. If you prefer to avoid soldering to the ICs then you can attach a probe to each of the three wires and make the connection that way. The probe can be pulled apart and the wire soldered in place. Wrap the wire around a couple of turns as shown below to act as a strain relief on the soldered joint, and remember to thread the cap onto the wire first. Finally, push the two halves together again.

Personal preference is to solder the three wires rather than use the clips. Perhaps the best approach is to use the clip initially just to make sure that everything works. Finally, take the plunge and solder the wires directly to the ICs on the motherboard.

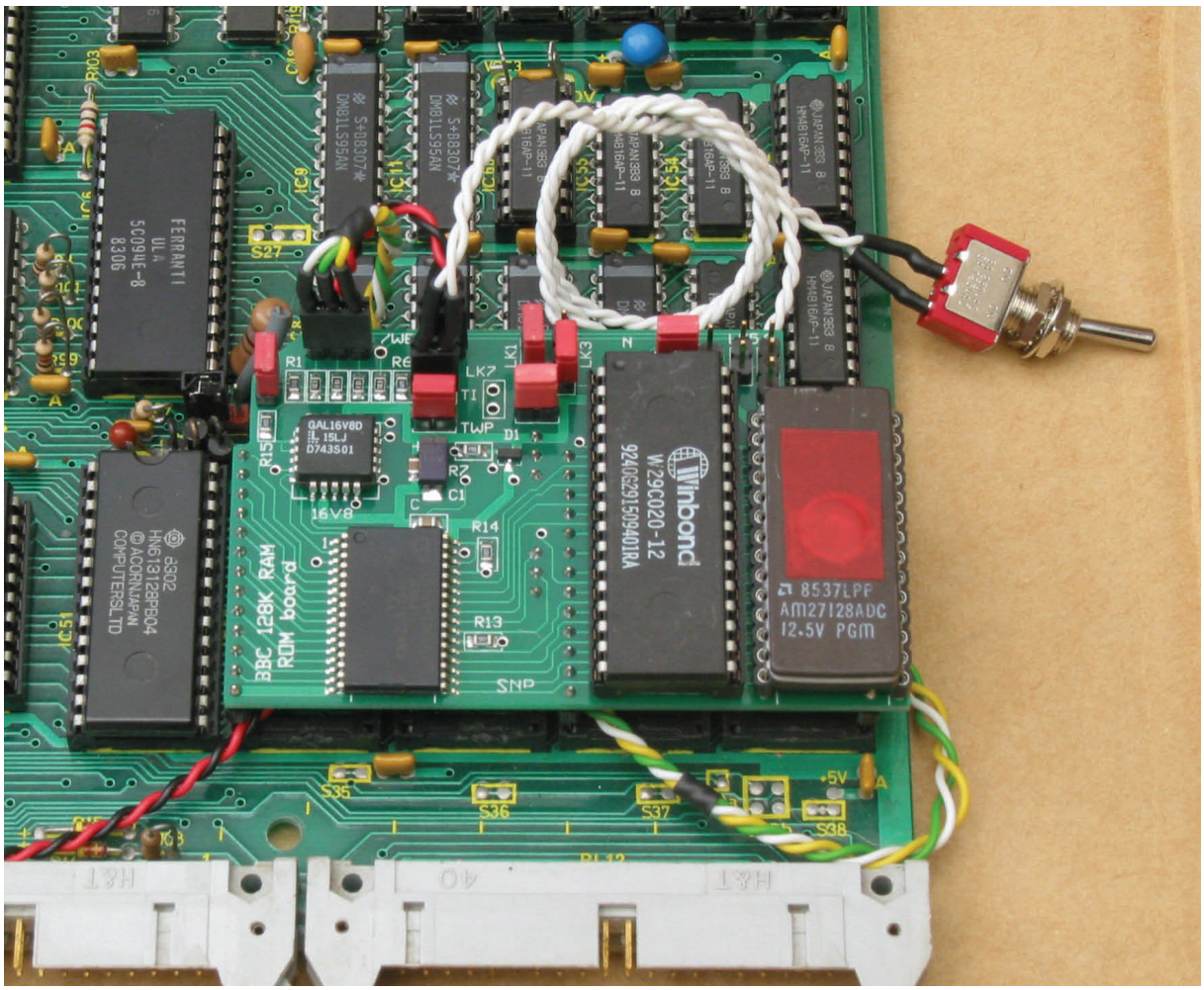


It is recommended that a toggle switch is fitted over both the links marked TI and TWP. Use the same style 2-way female housing as for the battery connection. The length of wire needed will obviously depend on where you intend to mount the switch.

TI means Total Inhibit and when open will prevent the RAM chip being read from or written to. TWP is Total Write Protect and prevents the RAM chip being written to but permits reading (providing TI is closed).

The 128kB RAM chip appears to the BBC Micro as eight separate banks of 16kB sideways RAM in socket numbers 0, 1, 4, 5, 8, 9, 12 and 13. The RAM chip is the 32-pin surface mount device immediately 'South' of the GAL16V8.

In the picture below an ordinary 16kB EPROM has been installed in SKT_B with filing system code on it. This should be your normal 'mass storage' ROM and could be for ordinary DFS, MMC, Compact Flash, etc. Notice that the two links LK5 and LK6 have been removed. The toggle switch is fitted to the link marked TI and is shown in the closed position.



With the main board back in the case it should be possible to power up the computer. Expect to see the normal BASIC '>' prompt because BASIC has been preprogrammed into one of the four banks available in the Winbond W29C020 chip (specifically, number 14).

The program needed for programming data onto the W29C020 is also stored on the W29C020 itself in ROM filing system format. To access it, type

```
*ROM
LOAD "FLASH"
RUN
```

Just as an experiment, try the 'E' (Erase) option. The program should disallow it because you would be trying to erase the BASIC interpreter from the flash ROM (W29C020) which is, of course, currently in use. Erasing it would crash the machine. Press Escape to exit.

It is suggested that you make a copy of all the programs on the ROM filing system. To do this, use

```
*ROM
*CAT
```

to perform a catalogue and see the names of the files. Then use

```
*ROM
LOAD "filename"
*DISC        (or whatever filing system you are using)
SAVE "filename"
```

Repeat as required. This simply copies the files to your normal storage medium. Although the FLASH program is stored on the W29C020 you may not wish to keep it there because it is taking up a ROM socket. Hence the need to copy it to another location.

If you get a ‘Bad program’ message after trying to LOAD a program, it is does not indicate a fault. It is more likely that the file being loaded is not a valid BASIC program.

By typing *OPT 1,2 before cataloguing the ROM filing system, more information is printed out in response to *CAT. The last three columns of numbers are important because they are the length and also the load and execution addresses of the file. Load and execution addresses may be the same but this is not always the case. To copy a non-BASIC program to your normal filing system (we will assume DFS in this example), follow these steps. The file RFS will be used as an example.

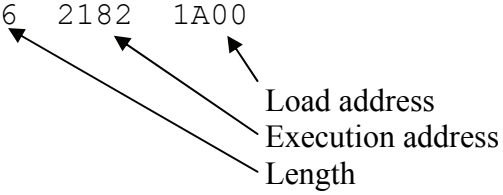
RFS shows up in the catalogue as;

```
RFS        0BE6   FFFF1A00   FFFF2182
```

Here, &BE6 is the length of the file and &FFFF1A00 is the load address. &FFFF2182 is the execution address. The &FFFF prefix relate to the way addresses in the second processor were distinguished from those in the I/O processor and can be ignored for our purposes.

To copy the RFS file, we will use *LOAD to forcibly load the file to a particular address in memory, then *SAVE to save it as a block of memory to the required filing system.

```
*ROM
*LOAD  RFS  2000
*DISC
*SAVE RFS  2000 + BE6  2182  1A00
```



Remember, you only need to go through this slightly more complex procedure if the file is not a BASIC program. In fact, though, the above approach involving *SAVE and *LOAD will work for BASIC programs too.

If you don't supply the load and execution addresses after the *SAVE command then they take on default values. If these are incorrect then it will not be possible to *RUN a machine code program. The Advanced Disk Toolkit (ADT) provides commands that allow you to alter file parameters. ADT200 is the name of the ROM image and needs to be run as a sideways ROM. This is not quite the same as the ROM filing system.

The W29C020 can hold up to four ROM images and they appear to the BBC Micro as ROM numbers 2, 6, 10 and 14. You do not have to keep BASIC in the flash ROM although you may wish to do so. However, in order to overwrite it you will need to make a copy of BASIC somewhere else, one of the eight banks of sideways RAM being a good choice. You can use B-Utility's RCOPY (ROM Copy) command to do this

```
*RCOPY  E 0
```

copies the contents of ROM number 14 (part of the W29C020) to number 0 (in sideways RAM). The RCOPY command always expects hex digits (&E is 14 decimal) and B-Utility is also in the W29C020 as ROM number 10. A Ctrl-Break followed by *FX142,0 will then ensure you are running BASIC in socket 0. You could now, if you wished, select the Erase option.

The situation to avoid, if possible, is one where the W29C020 is completely erased and the sideways RAM contents are also lost or corrupted. It's not impossible to recover from this but it means fiddling about inside the machine inserting and removing ROMs because as a minimum you will need to reinstall the original BASIC ROM.

Using the FLASH program is very easy. Use the 'S' option to decide which ROM socket you wish to program. The possible choices are 2, 6, 10 and 14. However, the program will not allow you to overwrite BASIC if BASIC is stored in the W29C020 and is currently in use. The W29C020 as supplied does indeed contain BASIC in socket 14, so 14 will not be a legitimate choice until you have moved BASIC elsewhere.

Also, B-Utility is quite a handy toolkit ROM (in socket 10) so before you overwrite it, use the SAVEROM program to store it on disk, perhaps to use later in sideways RAM.

The FLASH program is stored on the ROM filing system format in socket 6. Again, make sure you have saved it safely before deleting it from the W29C020. FLASH is on the CD in SSD format and also as a text listing. But it is over 300 lines long so you will not wish to type it in just for the fun of it.

Use option L to load your chosen ROM image into memory. Some simple checking is carried out to make sure it is a valid ROM image (and a warning given if it appears not to be valid) but it is largely up to you to supply sensible data.

The 'P' option actually programs the W29C020. You will be warned if you are about to overwrite your current filing system. Think carefully before agreeing to this. It may be wise to ensure that you have a copy of the filing system in one of the banks of sideways RAM. It needs a full YES response in uppercase letters to proceed. Programming, including a verify, takes around a second.

'E' will erase the entire W29C020 and again requires a YES to proceed. You are also warned if the current filing system is about to be deleted. Erasure will not be permitted if BASIC is currently stored in the W29C020 and in use. Note that you do not need to erase the chip before reprogramming it.

'X' exits the program. Answering Y to the prompt about performing a reset will perform a fairly effective simulated power-on reset without actually turning the machine off and on. This may be useful to allow the machine to recognize the new ROM images and permit them to claim workspace if required.

The FLASHER program is the one that is used to program all four banks of the W29C020 in quick succession. It’s based on the FLASH program described above, but the menu and most of the prompts have been taken out in order to speed things up. The ROM images to be loaded, together with the socket number that they should be programmed into, are simply included in a DATA statement right at the end of the program. Modifying it to suit your own machine should be simple enough.

Battery backup for sideways RAM

Keeping the sideways RAM's contents when the machine is turned off requires the connection of a battery pack. Various possibilities exist, for example two alkaline AAA cells connected in series in a suitable dual holder. Three AAAs also work well. It is even possible to use a 3.6V rechargeable cell.

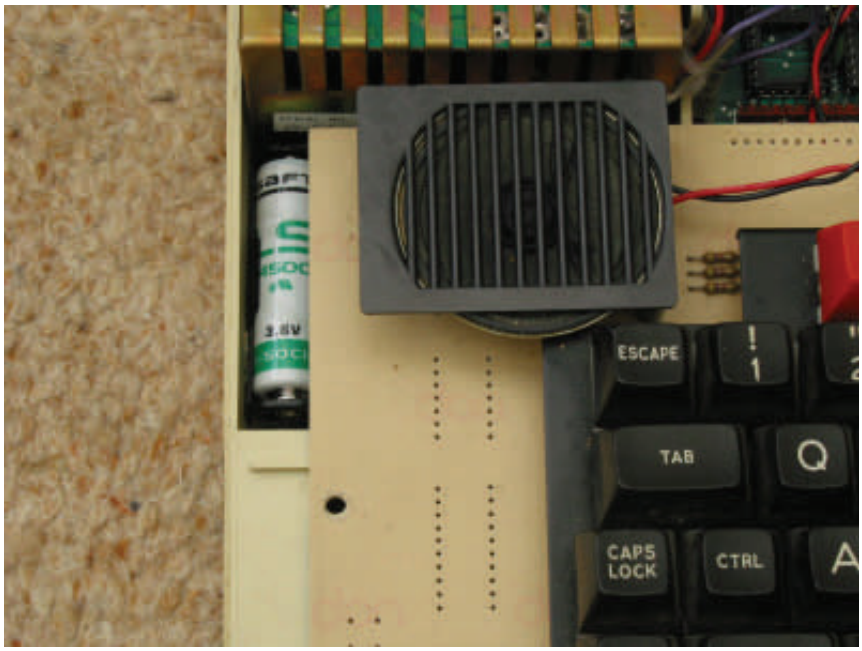
Perhaps the best option is a lithium thionyl chloride (LTC) cell, something that is specifically designed for long term low current drain applications. The Cypress 128kB RAM chip draws about 2 microamps in standby mode and the expected life of an LTC cell should be at least five years and probably much longer. Despite its initial higher cost, it should require absolutely no attention whatsoever for many years.

Do not use an ordinary AA alkaline cell in the single AA holder supplied. The nominal voltage of 1.5V is not sufficient to maintain the contents of the 128kB static RAM chip. The LTC battery is usually about 3.6V.

[If you wish to use a rechargeable 3.6V NiMH cell then it will be necessary to short out LK7 on the upgrade board. This will allow the cell to receive a small trickle charge of around 2.5mA when the computer is turned on. **On no account fit a battery which is not designed to be recharged when LK7 is shorted.** The LTC cell, with LK7 in its default open state, is the recommended route at a cost which should work out at less than 1p per week]

You will need to solder a length of black and red wire to the AA holder, the black going to the end with the spring fitted (-ve). The polarity is in fact marked on the holder itself, and use heat shrink over both connections. The end which attaches to the RAM/ROM board can have a 2-way female connector attached. See previous photos. The red wire must go to the terminal marked '+' on the board.

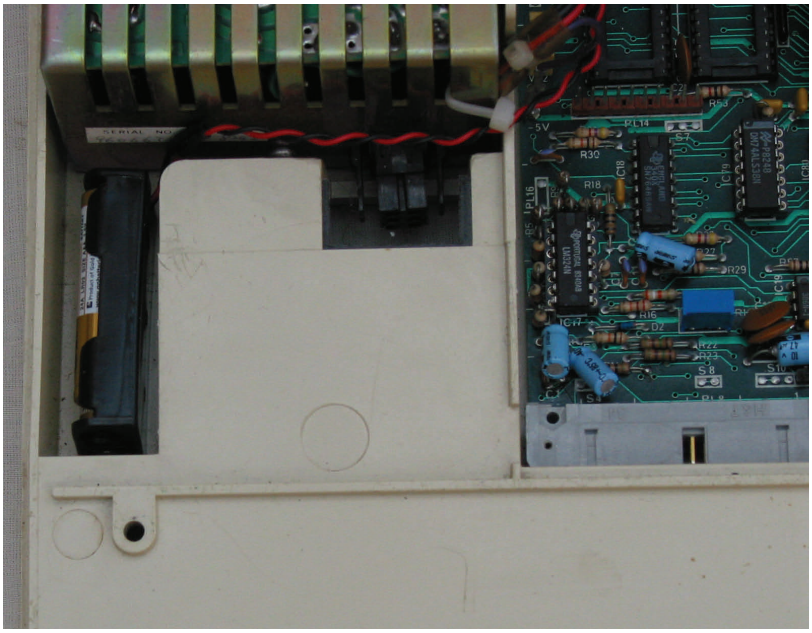
There are various places in which the LTC cell can be located and three possibilities are illustrated. **When using the method below, it is imperative that you take steps to ensure that the end of the battery cannot short out against the metal case of the power supply.**



These pictures show other possible locations for the AA cell holder. It can be held in place by double sided adhesive tape or perhaps a spot of silicone rubber sealant. The only disadvantage with attaching it to the keyboard is that if you want to remove the keyboard for any reason, you will most likely need to disconnect the backup battery and the RAM's contents will then be lost. That said, reloading the data, particularly from a modern solid state drive, is matter of a few seconds and really should not be a problem.

A perfectly satisfactory battery backup system can be made using the twin AAA holder. A couple of AAA alkaline cells are included in the kit. A twin AAA battery system such as this has been in use for over two years and in that time the voltage has only dropped from an initial 3.2V to 3.1V. Clearly, it is a very cost effective solution.

As with the lithium thionyl chloride battery, simply solder black and red wires to the battery holder with a 2-way female connector at the other end (to push onto the RAM /ROM board). The AAA holder can be located as shown with a piece of foam to prevent it moving about.



Writing to the sideways RAM

The RAM/ROM board provides eight banks of 16kB sideways RAM in sockets 0, 1, 4, 5, 8, 9, 12 and 13. The LOADROM program provided will load any named file from the current filing system and write it into any chosen bank of sideways RAM. Some ROMs may provide commands such as *SRLOAD to load data into specific banks of sideways RAM. It is possible to load ROM images from cassette tapes but it is very slow and tedious.

Another program is called LOADER. This is a machine code program and the source assembly language used to create it is called LOADSRC. The general idea behind LOADER is that you set BASIC's resident integer variable S% to the number of the socket into which the data is to be written, load the LOADER code at some suitable address, then CALL it. &4000 bytes of memory, starting at address &3000, will then be written to the socket indicated by S%. Hence addresses &3000 to &6FFF inclusive are written to the sideways ROM space. The BBC Micro will obviously need to be in MODE 7 for this to work.

LOADER is quite short, only around 50 bytes. An example of how to use it might be;

```
10 MODE 7
20 *LOADER LOADER 2F00       (the machine code program)
30 *LOAD ROMDATA 3000        (the ROM data to be loaded)
40 S%=4 : REM The ROM socket to be written to
50 CALL &2F00 : REM Write the data to socket 4
```

The CALL address in line 50 must be the address of the LOADER code. Here it was loaded at &2F00 but the code is in fact relocatable at any address within reason. The following would have worked just as well;

```
*LOAD LOADER &2E80
CALL &2E80
```

It would be trivial to implement some kind of FOR-NEXT loop to read filenames and socket numbers from DATA statements, load the images in turn and write them into sideways RAM as part of a !BOOT sequence. In fact, loading all eight banks of sideways RAM in this way can be done in around 5 seconds with a suitable filing system (around 0.6s per socket).

If you have the MMC storage system then another technique might be to use something like this;

```
*DBOOT SETUP
```

This will select the named ‘disk’ and act upon the !BOOT file.

Whatever the method used, it will of course necessary to ensure that the TI and TWP links on the upgrade are both closed, otherwise writing to the RAM is impossible. Another link next to R1 should also be closed.

After loading an image into sideways RAM you will normally need to perform a Ctrl-Break to allow the machine to recognize it. *HELP will show a list of the ROMS in the machine. Certain types of 'toolkit' ROMs will also show exactly which socket numbers are occupied.

There is an SSD image on the CD called RSU1.ssd (ROM Set Up 1). If you have the MMC system it is possible to copy this image from the CD to the multimedia card itself. Entering

```
*DBOOT RSU1
```

will then select the named disk, *EXEC the !BOOT file and load some example ROM images into the sideways RAM. It should be clear how the program can be modified to load any images that are required. You can also have several different disks, each one setting up the machine in different way. Eg

```
*DBOOT RSU2
*DBOOT RSU3
*DBOOT 420
```

Another program is RLOAD. This is a machine code program and requires the format;

```
*/RLOAD  <filename>  <socket number in hex>  (Q)
```

This loads the named file into the specified socket number. The use of the 'Q' option will speed up the process at the expense of corrupting main memory (by using it as a buffer). An example of the syntax would be;

```
*/RLOAD  EDITOR  C  Q
```

to load the ROM image EDITOR into sideways RAM in socket 12 (&C),

Recovering from a ROM image in sideways RAM that causes the computer to hang

There are two main approaches. One is to turn off the computer and briefly remove the battery backup system. You can then turn the computer back on and all should be well.

Another method is remove the link marked TI (Total Inhibit) then perform a Ctrl-Break. The RAM cannot then be read and any data that it contains is ignored. Then replace the TI link (or close the switch, if fitted) and simply load a new ROM image over the offending one.

The use of a toggle switch over the TI pins is the recommended approach because it is best to ensure that TI is open whenever the computer is turned on or off. Clearly, this is much more convenient to do by means of a switch rather than an internal link.

ROM images that write to themselves

This technique was used extensively back in the BBC Micro's heyday in order to discourage software, normally supplied on EPROM, from being distributed on floppy disk and loaded into sideways RAM. Essentially the code would be writing to a suitable part of the sideways ROM address space (&8000 to &BFFF) to see if the data at that address could be altered. If so, the assumption was made that the program was running in sideways RAM instead of an EPROM. It then refused to work properly and perhaps attempted to disable itself.

Of course, this protection mechanism is easily defeated by write protecting the sideways RAM. Whatever the copyright issues might be, it is also a fact that many of the ROM images now available on the internet have been modified in order to remove the relevant piece of code. Such images can generally be used in sideways RAM without difficulty.

The W29C020 flash ROM is not changed by casual writes to particular addresses. Every write operation, if it is to be successful, must be preceded by the special command sequence specified in the datasheet. In this respect it is immune to the 'self write' system used by some ROMs. However, a write to any address of the flash ROM can disrupt several of the subsequent read cycles if those read cycles are from the W29C020 itself.

You can see this for yourself by typing the following. (This assumes that BASIC is running from the W29C020. To find out, type `PRINT ?&F4` If the result is 2, 6, 10 or 14 then BASIC is indeed in the flash ROM)

```
?&9000=0
```

You will probably get a spurious error here. The reason is that the CPU has attempted a write to address &9000 and that will be a write to the W29C020. The next opcode fetch will be from the W29C020 itself (because that's where the BASIC interpreter is located) but the previous write to &9000, although it doesn't change the data, confuses the flash memory's internal state machine. The effect is that the next read produces invalid data (ie instruction code). This is not a fault of the W29C020 or any other flash ROM. The inbuilt protection against undesired Write operations is mainly to prevent corruption during transient power up and power down situations whilst allowing the code to be legitimately changed if needed (eg in BIOS chips). It is not designed to thwart self-modifying code.

In practice, what this means is that self-writing ROM images can cause an issue in the flash ROM but for slightly different reasons compared to sideways RAM. Fortunately it is not likely to be a problem because either the ROM image never used any kind of self-write protection in the first place (BASIC, filing system ROMs etc) or, as previously mentioned, the code has been hacked to remove the protection. The point is made again that the flash ROM data will not get altered by self-writing code but it can affect subsequent read operations from that ROM for a few tens of microseconds afterwards.

The Partial Write Protection (PWP) link by R1 will, when open, write protect banks 8, 9, 12 and 13 of the eight RAM banks. However, it will also write protect the flash ROM by holding the Write Enable pin high. This overcomes the problem described above because no Write to the flash ROM can take place. Additionally it write protects a 32kB static RAM (if fitted) in SKT_B. It would be possible to fit a toggle switch over the PWP link but in practice this should not be required.

The best arrangement of ROMs

There isn't really a 'one size fits all' answer to this. Although the BBC Micro can support up to 16 ROMs in the machine, it is perhaps unlikely that in any one computing session every single one of them would be needed. Additionally, dozens of different ROMs are available and everyone's requirements will differ. The ideal arrangement of ROMs may well depend on whether you have access to an EPROM programmer. Remember that the W29C020 is best for storing ROM images that are important and don't need to be changed very often. In contrast sideways RAM is ideal for an E00 DFS, printer buffer and so on.

If you are able to program a 27512 (64kB) EPROM, then perhaps a good choice would be BASIC, your main filing system, some kind of BASIC toolkit and maybe the BASIC Editor. These four ROMs, now in a single chip, could go in the rightmost socket (SKT_B) as ROM numbers 3, 7, 11 and 15. This would leave the both the flash W29C020 (sockets 2, 6, 10 and 14) and the 128kB RAM chip (sockets 0,1, 4, 5, 8, 9, 12, 13) available for any other purpose. That is effectively 12 free sockets.

You could probably get away without battery backup on the RAM, as the W29C020 can be programmed in situ and of course doesn't need a battery to retain data.

The best point about the above setup is that you can never find yourself with no language ROM and no filing system, because those are both safely stored on the EPROM.

If you cannot program your own EPROMs then one option might be to put the original BASIC in the rightmost socket (SKT_B) and your main filing system in the W29C020. This would leave three free spaces in the W29C020 and all eight banks of sideways RAM are available. Even if the sideways RAM gets lost or corrupted, BASIC is still present and the filing system data should be safely stored in the flash ROM. The W29C020 is virtually completely immune from accidental writes or corruption. The FLASH program supplied will warn you if you are about to delete your current filing system, so inadvertently erasing it should not be an issue. If you delete the filing system ROM then of course you could just load it back into sideways RAM from disk, MMC or Compact Flash. But you need a filing system ROM in the machine in order to do this...

Losing your filing system ROM is a nuisance but not disastrous. As outlined elsewhere the steps to recover from this are fairly easy. One option might be;

1) Put BASIC in SKT_B, setting LK5 and LK6 as required. Use a program like COPYBAS to copy BASIC into sideways RAM which must have battery backup fitted.
2) Turn off the machine and change BASIC for your normal filing system ROM (DFS, MMC etc). You are now in a position where you have both BASIC and your filing system available, the former in sideways RAM and the latter as an ordinary ROM. It is now possible to use the FLASH program to program the W29C020 as needed.

ROM clashes

With such a large number of ROMs written for the BBC Micro it was almost inevitable that some ROMs might interfere with the operation of others. For example, two different ROMs might both try to respond to a command like *SLOAD, with usually the highest numbered ROM getting priority.

Sometimes the clash might be that two or more ROMs were unofficially trying to use a few bytes of memory for their own special purposes (some kind of flag perhaps), but each using it in a different way.

A more modern example is that of the MMC system which needs to access the User Port through the User 6522 VIA. There is another project developed largely by Martin B called UPURS which allows transfer of SSD images from a PC to BBC Micro, but also through the User Port. UPURS has its own code in ROM and it will come as no surprise that the Turbo MMC ROM and UPURS cannot be used simultaneously.

Fortunately there is usually no need to physically remove ROMs. Various 'Manager' ROMs can accomplish the same thing through software tricks. B-Utility itself has two commands, *ON and *OFF which can be followed by a list of ROM numbers (in decimal) which you wish to temporarily disable. For example;

```
*OFF 3  5  10
```

disables the given socket numbers, even after Ctrl-Break. Use the *ON command to reverse the process.

Other 'ROM Managers' include the Advanced ROM Manager.

A simple way to disable ROM number <n> is to enter from BASIC;

```
?(&2A1+n)=0
```

However, Ctrl-Break will usually reset the memory location back to its correct value so this technique is not effective in all situations. Also, writing zero to the &2A1 table does not prevent vectored entry into sideways ROMs, a topic which is covered in the Advanced User Guide.

Programming a Winbond W29C020 Flash ROM

There are basically two ways to program this device, either externally with a stand alone programmer or installed in the ROM/RAM board. Either way, the chip can contain up to four ROM images as far as the BBC Micro is concerned.

With the chip in a separate programmer (usually attached to a PC), you will need to have the ROM image data to program into the device. The W29C020 is a 256 kilobyte chip but when used in the BBC Micro only the first 64kB are relevant. In other words, only addresses &0000 to &FFFF on the device will be recognised. All remaining addresses (&10000 to &3FFFF) are not used.

A ROM image must always start on address zero or an address that is a multiple of &4000. Hence images must begin at &0000, &4000, &8000 or &C000. Remember that we are talking about the address within the device as if it had no connection with the BBC Micro. All four ROM images, when installed in the Beeb, will appear within the address range &8000 to &BFFF but will have a different 'sideways ROM' number. The HxD hex editor may be useful in joining together multiple ROM images into a single block ready for programming. The connection between the device addresses and the ROM number as seen by the Beeb is as follows;

Addresses range	ROM number
&0000-&3FFF	2
&4000-&7FFF	6
&8000-&BFFF	10
&C000-&FFFF	14

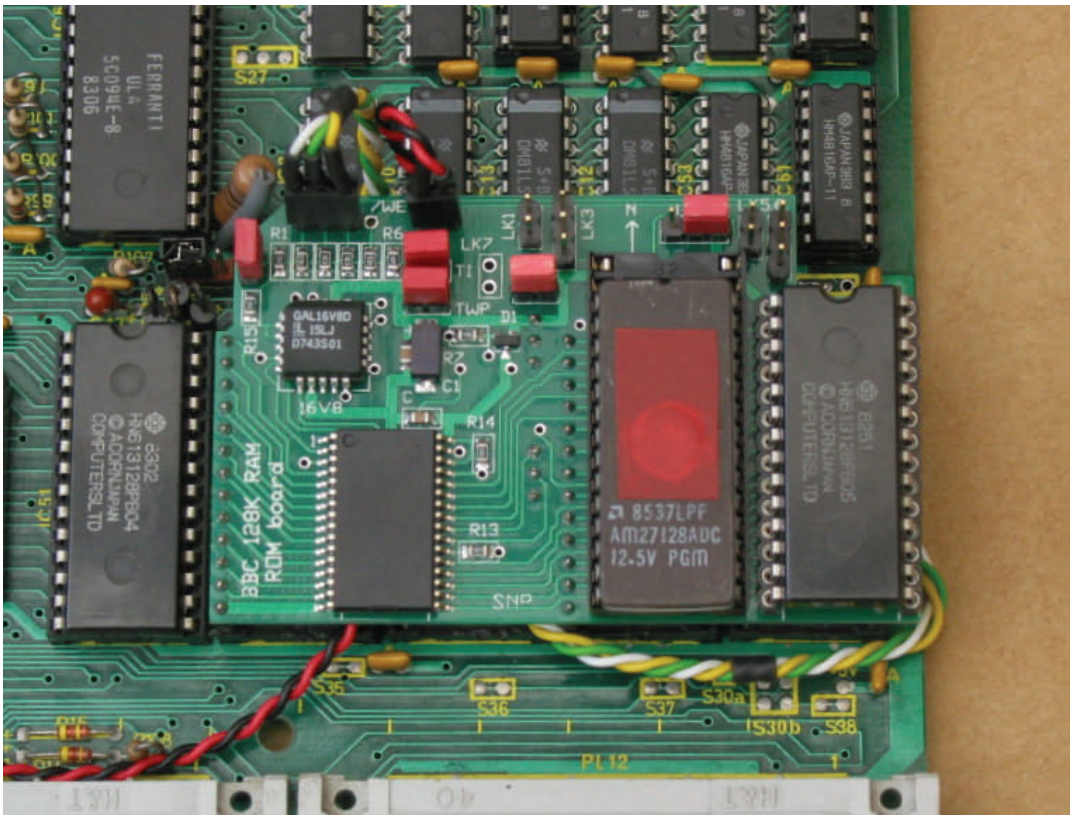
Programming a W29C020 in the ROM/RAM board

Part of the reason for using the W29C020 is that it can be programmed in circuit using only the existing 5V supply. A simple BASIC program is available to perform this programming, but you need to bear the following in mind.

It will obviously be necessary to have a copy of the BASIC interpreter present in the machine. Additionally, you must consider how you will load ROM images from the storage system into the computer. It is clear that you normally need a minimum of two ROM images available at the same time, BASIC and your usual filing system ROM (eg DFS, ADFS, MMC etc). If we assume, worst case, that the eight banks of sideways RAM are all empty and that the W29C020 is also empty, you only have the socket on the extreme right of the expansion board (SKT_B) to accommodate two ROM images (BASIC and filing system). Of course, if you have BASIC and the filing system programmed into a single 32kB EPROM then you're ready to go (although LK5 and LK6 will need to be set for a 27256 chip). But here we will assume that BASIC and your filing system ROM are on two separate 16kB chips, the eight banks of sideways RAM are all empty and the W29C020 is completely blank. With care this situation need never arise, but if it does then the following is one way to get back to normal.

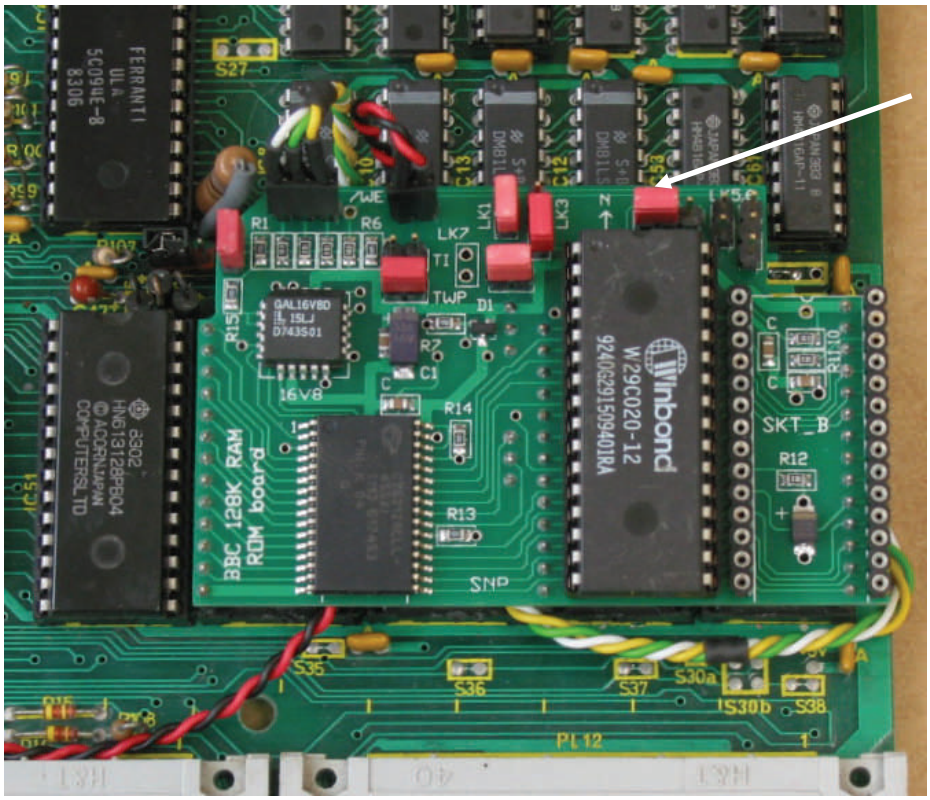
Step one is to set up the expansion board as shown below. BASIC is in the rightmost socket and the DFS (or appropriate filing system) in the other. Links are arranged to suit 16kB ROMs. For reasons that will become evident, you will need to have a battery backup system fitted to the sideways RAM. The eight banks of sideways RAM are in sockets 0, 1, 4, 5, 8, 9, 12 and 13. **Note that when 28 pin chips are used in SKT_A the 4 unused pins are towards the rear of the machine. Also, LK4 is set East.**

Switch the machine on and it should power up in BASIC. You now need to use the SAVEROM program to save copies of the BASIC ROM and also the filing system ROM to your normal storage system. A modern solid state storage system will be quicker and more convenient than a floppy drive but both methods are entirely acceptable. SAVEROM will ask you for the ROM number that you wish to save, and with the arrangement as shown below BASIC is in socket 15 and the DFS in socket 14.



You now need to run LOADROM in order to load copies of BASIC and the DFS back into sideways RAM. LOADROM also asks for a ROM number into which data will be loaded, and the valid options are 0, 1, 4, 5, 8, 9, 12 or 13. You must choose a different number when loading BASIC and the filing system code. Now remove the link marked TI (or open the switch if fitted) and turn off the machine. Remove both the BASIC ROM and the DFS and then install the W29C020. Set the links as shown on the next page. NB changing LK4.

[Technical notes. It would have been possible to have left BASIC in the rightmost socket and only bothered to copy the DFS code into sideways RAM. The important thing is that when the machine is turned back on with the W29C020, both BASIC and the filing system code are present somewhere in the sideways ROM system. The reason for removing the TI link when switching the machine on or off is to reduce the chances of corruption in the 128k sideways RAM chip.]



Notice that LK4 is to the left when using the W29C020

The system should now look as above. Switch on and you will be confronted with a 'Language?' error. This is normal. With the TI link removed the machine cannot read the contents of sideways RAM and this is where BASIC and the filing system code are now located. Simply place the link back on the TI pins and do a Ctrl-Break. You should now be back with the familiar BASIC prompt.

It is at this stage that the supplied BASIC program will be needed. It is a simple menu driven arrangement that will allow data (ie a ROM image) to be loaded from the current filing system and programmed into the flash ROM. The ROM must be a W29C020. It can contain up to four separate ROM images as ROM numbers 2, 6, 10 and 14.

The required BASIC program is supplied as a text file on the CD so as a last resort it can be typed in by hand. It is also on the CD in the form of an SSD (Single Sided Disk) image, so if you have one of the modern solid state storage systems then getting it onto the Beeb is easy enough.

A third method, providing the Winbond chip was obtained from IFEL, is that the required program is stored on the W29C020 itself in the ROM filing system. If so, then it is suggested that the program is copied to another storage medium. To do this type;

```
*ROM              (Selects the ROM Filing System)
LOAD "FLASH"      (Load the BASIC program)
*DISC             (or use *ADFS, *CARD etc)
SAVE "FLASH"      (Save a copy of the BASIC program)
```

When the 'FLASH' program is run, you get a straightforward menu of options. The first of these is to set the ROM number that you wish to program and a default value is shown. This is always either 2, 6, 10 or 14. This is because the W29C020 can store up to four ROM images and they are seen by the BBC Micro as being ROM numbers 2, 6, 10 or 14.

[Technical note. There is nothing to stop you from programming a copy of BASIC into the W29C020, likewise your usual filing system ROM. If you have already done this then the program will either warn you about certain options, or indeed prevent them from being selected. Suppose for example that you have already programmed ROM number 10 (within the W29C020) with BASIC and that ROM number 10 (ie BASIC) is the current language. If you tried to reprogram ROM number 10 with new data then the machine would crash because the BASIC interpreter, currently in use, would suddenly disappear. For this reason the program will disallow certain choices.

Note also that the program is geared towards programming a Winbond W29C020. It is in fact possible to use software to identify the exact type of chip that is installed (ie both manufacturer and device type). If the program fails to detect the presence of the W29C020 then you will receive an error to this effect. If you want to try programming another type of device (eg AM29F010) then you will need to study the chip datasheet and modify the program to suit.]

Option (S) allows you to set the ROM number to be programmed. As discussed above this must be one of 2, 6, 10 or 14. This option doesn't attempt to program the chip in any way, it just selects the ROM number for the (P) option.

Option (L) allows you to load data into the computer's memory ready for programming onto the chip. The data in question would normally be a valid ROM image and some rudimentary checks are carried out on the data that is loaded. The program also checks that the file to be loaded does not exceed &4000 (16kB) in length.

Option (P) programs the data in the computer's memory into the chosen bank (ROM number) of the W29C020. If, by doing so, you will overwrite the current filing system then you will receive a warning to this effect. The program will then select the tape filing system before programming the new data onto the chip. The action requires confirmation and a full YES must be entered in uppercase letters.

The programming only takes a few moments and is followed by a verify to ensure that programming has been successful. A 'Pass' or 'Fail' message is self-explanatory.

Option (E). This attempts to erase the entire chip. As with the programming option this will not be permitted if the BASIC interpreter is currently in the W29C020 and in use. It also requires a full YES to confirm the action. Note that there is no need to erase the chip before programming it so the Erase option is not really essential.

Option (*). This simply allows any 'star' command to be entered. It might be useful to change filing systems (*DISC, *ADFS), to catalogue the current disk, change drives etc.

Option (X). Exit the program. You will be offered the choice of performing a fairly effective system reset (almost equivalent to a power on reset), and this requires a 'Y' response to proceed and any other key to merely quit the program normally.

Deleting BASIC from the W29C020

There is nothing to stop you from programming BASIC into the W29C020. If you have already done this, however, when you run the BASIC program just described, you may find that it is not possible to completely erase the chip and you cannot overwrite BASIC with something else. Both actions could crash the machine.

The way to work around this, if needed, is to temporarily load two copies of BASIC at the same time. To do this, save a copy of BASIC to your main filing system using the SAVEROM program. Then load it into one of the banks of sideways RAM (numbered 0, 1, 4, 5, 8, 9, 12 or 13) using LOADROM. Type Ctrl-Break then enter;

```
*FX142,n
```

where n is the sideways RAM bank number that BASIC has just been loaded into. You can check if the action has been successful by typing

```
PRINT ?&F4
```

The result should tie up with the value of <n>,

It will now be possible to load and run the FLASH program and alter the contents of any of the ROM numbers 2, 6, 10 or 14. You can also select the option to erase the entire chip.

Erasing a single bank of the W29C020

There is no specific option to do this but remember that it is not necessary to erase a 16kB bank before reprogramming it with new data. If you really wish to delete a ROM image from the W29C020 then you can achieve the required result just by using the (L) option to load almost anything (other than a valid ROM image) into the machine prior to programming it onto the chip. This will effectively delete the existing ROM image. The reason this works is that sideways ROMs on the BBC Micro have to follow a certain format near the start of the code (ie from &8000 onwards). If this format is not observed exactly, then the machine will not recognize it and will flag the socket as being empty.

The are various software tricks for disabling ROMs that seem to be causing problems. One is to enter

```
?(&2A1+n)=0
```

where n is the number of the ROM that you wish to disable. Certain toolkit or utility ROMs also provide commands for achieving the same thing, for example the Advanced ROM Manager and also B-Utility. Pressing Ctrl-Break will normally attempt to restore the above memory location to its correct value although ‘toolkit’ ROMs are usually a bit more robust in keeping ROMs disabled after Ctrl-Break.

A few notes on the programming algorithm

Although the FLASH program is a combination of BASIC and assembly language, there is always some satisfaction to be gained from getting things to run as quickly as possible. The W29C020 datasheet is clear about the requirements for programming the chip. Essentially, programming is done a block at a time, each block ('page') being 128 bytes long. Generally, you will want to write all 128 bytes to each page because those that have not been defined (in that same page) will be turned into &FF.

When the last byte of a page has been written and a time interval of approximately 200us has elapsed with no further write to the chip, the W29C020 will then commence its internal writing to the memory array. This writing process does not happen instantaneously and the datasheet specifies a maximum time for this of 10ms. The W29C020 makes provision for a software technique for determining when the internal writing is complete. This can be done by reading the last byte that was written to the chip. While the chip is still busy with its internal write cycle, reading the last byte will obtain the complement (inverse) of the correct value on data bit D7.

Testing for this in assembly language is extremely easy. If the data byte that was last written to the chip is in the X register and the address that it was written to is defined by (dest%),Y then a suitable polling loop might be;

```
.busy
  TXA
  EOR (dest%),Y
  BMI busy
```

The question then arises whether we can get on with anything useful while we are waiting for the loop to end. Although not strictly needed, the program shows the progress of the programming process by printing out the current page that is being programmed into the W29C020. There are 128 pages, because 128 lots of 128 bytes (the page size) is 16384 (&4000, the size of sideways ROM space). Immediately after the last byte of each page has been written to the chip, the current page number (1 to 128) is printed out on the screen in a particular place. It is the assembly language equivalent of;

```
PRINT TAB(20,19)RIGHT$("00"+STR$X%,3)
```

Only when this has been done does the program fall into the 'busy' loop shown above to ensure that the internal write is complete. In this way, any overhead converting X to ASCII digits and printing them out at the right place on the screen does not add significantly to the total programming time.

The effect is that programming the complete bank of 16kB takes only a second or so. The maximum time taken for the chip to program its internal array for each 128 byte page is given as 10ms in the datasheet and this is unlikely to be too far out. So if we assume 10ms per page and there are 128 pages to program, simple maths tells us that the total programming time will be about 1.28 seconds at most. The programming time achieved by the FLASH program is slightly less than this figure (even including a final verify) so there seems little point in trying to achieve any further speed increase.

Programming another type of flash ROM

The Winbond W29C020 is entirely satisfactory for this ROM/RAM board and there should not be any advantage in using another type of flash memory. That said, the Beeb is a good platform for experimenting with both hardware and software and there is no reason why you should not try programming another type of chip, for example an AM29F010, just for the satisfaction of doing it. The procedure involves writing to the ROM select latch at &FE30 and this can only really be done from a machine code program. Therefore, a knowledge of how to use BASIC's inbuilt assembler will be needed.

The key point to grasp is that the datasheet for the chip will refer to writing a certain byte of data to a specific address in order to achieve a particular result. Sometimes several bytes must be written to different addresses in a certain order. The datasheet authors cannot possibly know what the target system for the chip will be, so the addresses that they refer to will obviously have a tenuous connection with the BBC Micro. Using the W29C020 as an example, to completely erase the device initially involves writing data &AA to address &5555 on the chip. However, just writing;

```
LDA #&AA
STA &5555
```

in assembly language is not going to work. Address &5555 is right in the middle of user memory and all the sideways ROMs (of which the flash memory is one) will be disabled. Some ‘address translation’ will be needed although this turns out to be very straightforward. Here are the key points to keep in mind (a separate section covers much of this more detail);

1) A16 and A17 on the flash chip (pins 2 and 30) are permanently grounded
2) A15 and A14 on the chip (pins 3 and 29) respectively come from the QD and QC outputs of the ROMSEL 74LS163 latch on the motherboard (at address &FE30) and NOT directly from the processor’s address bus.
3) A13 through to A0 come directly from the CPU address bus.
4) When writing to the chip, CPU address A15 must be high and A14 must be low.

In practice therefore, when the datasheet refers to writing to address &5555 you need to achieve this in two stages. The first is to write a number to number to ROMSEL such that;

D0 on the data bus is 0
D1 on the data bus is 1
D2 on the data bus is whatever you want A14 on the W29C020 to be
D3 on the data bus is whatever you want A15 on the W29C020 to be
D4 to D7 should be zero

The reason for setting D1 and D0 to 10 (binary) is to ensure that the correct socket on the motherboard (and by implication on the ROM/RAM upgrade) is activated.

In the FLASH program you will see how this has been done using a couple of BASIC functions called FNlatch and FNaddress.

FNlatch takes the full address in the argument and shifts the binary equivalent right by 12 places (divide by 4096). This moves bit 15 down to bit 3 and bit 14 down to bit 2. The result

is then ANDed with &C (binary 1100) in order to force all the remaining bits to 0. Finally, the result is ORed with 2 (binary 10). The result of this calculation is the eight bit immediate constant for an LDA instruction, and this is then written to ROMSEL (at &FE30) in the normal manner. That is to say, the same number is written to &F4 and then &FE30. Writing to ROMSEL in this way not only makes sure that the correct socket on the main board is active, but also sets A15 and A14 on the W29C020 to the correct logic state.

Another simple function is FNaddress. This takes the number given in the argument and retains only bits 0 to 13 by ANDing it with &3FFF. Sideways ROMs can only be accessed when A15 on the CPU is high and A14 is low, so the number is then ORed with &8000.

The assembler built into BBC BASIC is extremely flexible. The full power of the expression evaluator is available and you can see how this has been used to advantage in the FLASH program. As an example, when the datasheet specifies data byte &55 must be written to address &2AAA, this has been implemented by;

```
LDA #FNlatch(&2AAA)        \Extract bits for A15 and A14
                           \and ensure D1 and D0 are 10
JSR romsel                 \Write to &F4 and &FE30. This sets
                           \A15 and A14 on the chip correctly
LDA #&55                   \Data byte needed
STA FNaddress(&2AAA)       \Compute correct address and store
:
:
:
.romsel
STA &F4
STA &FE30
RTS



DEFFNlatch(a%)
=((a% DIV 4096) AND &C) OR 2

DEFFNaddr(add%)
=(add% AND &3FFF) OR &8000
```

Link settings summary

PWP. The Partial Write Protect link is by resistor R1 near the back of the upgrade board, on the left. When open, four of the eight banks (8, 9, 12, 13) become write protected and the remaining four can still be written to as normal. The 128K static RAM chip produces eight banks of RAM in sockets 0, 1, 4, 5, 8, 9, 12 and 13. PWP also write protects the W29C020.

TWP. When open, the TWP link (Total Write Protect) causes all eight banks of sideways RAM to be write protected. It overrides the setting of the PWP link (above). In other words, when TWP is open the state of the PWP link makes no difference. The TWP link effectively just acts as an open/close switch to either allow or prevent the Write signal getting through from the programmable chip to the 128K static RAM.

TI (Total Inhibit) can be used to disable the 128K RAM chip completely. When open, it is not possible to read from the 128K RAM chip and it cannot be written to either. One use is to easily recover from a corrupted ROM image in the sideways RAM which is causing the machine to crash. Simply open TI and do Ctrl-Break.

It is strongly recommended that an easily accessible switch is fitted over the TI pins so that it can be in the open state whenever the machine is turned on or off. Past experience indicates that this is a foolproof method of ensuring that no spurious Writes get through to the RAM chip during power on or off.

LK1. When closed, QD from the motherboard 74LS163 latch is connected through to pin 1 on a 28-pin chip in SKT_A, or pin 3 on a 32-pin chip (eg W29C020). When open, the onboard resistor R8 pulls this pin high.

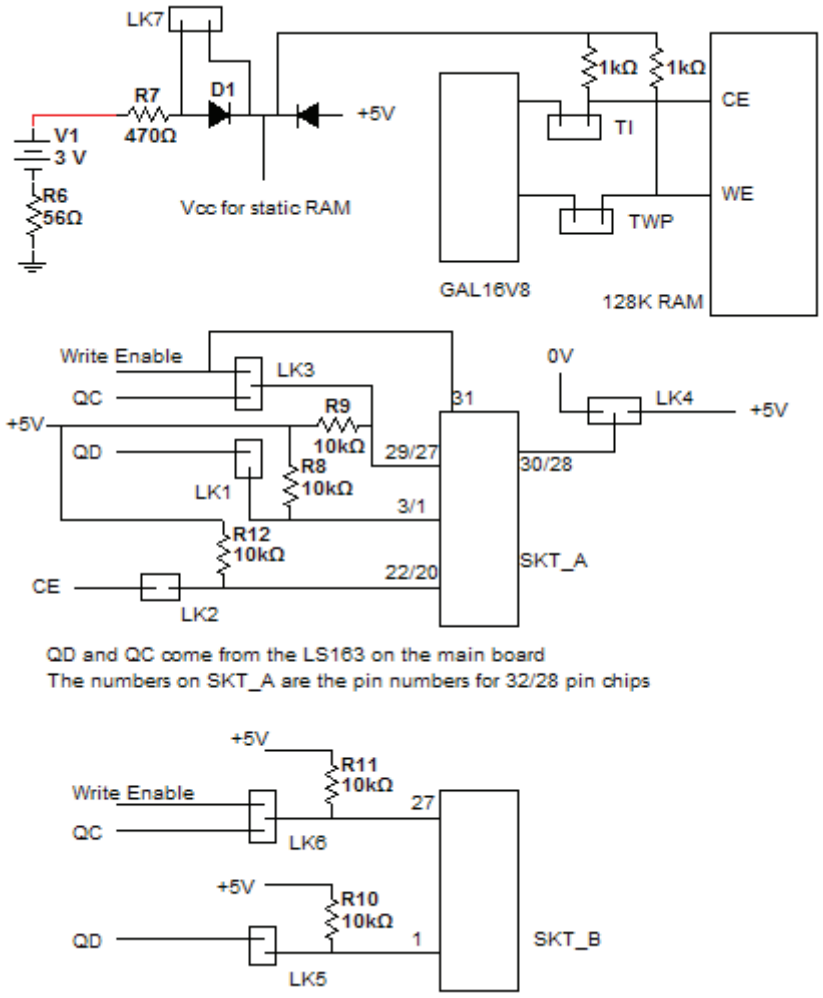
LK3. When North, a Write Enable signal is routed through to pin 27 on a 28-pin chip (or pin 29 on a 32-pin chip). When South, QC from the motherboard 74LS163 latch is routed through to these pins instead. Left open, the pin will be pulled high by R9.

Here is a quick summary of LK1, LK3 and LK4 settings which relate only to SKT_A

Device	LK1	LK3	LK4	Number of ROMs	ROM numbers
16kB (27128)	Open	Open	East	1	14
32kB (27256)	Open	South	East	2	10,14
64kB (27512)	Closed	South	East	4	2,6,10,14
32kB SRAM	Closed	North	East	2	6,14
W29C020 Flash	Closed	South	West	4	2,6,10,14

LK2 must be closed for SKT_A to work at all. If LK2 is open, the Chip Enable pin on 20 (28 pin device) or pin 22 (32-pin flash ROM) will be pulled high by R12. Removing this link temporarily may be useful if code in the flash ROM crashes the machine.

When using the 32 pin W29C020 LK4 must be in the West (left) position to put a logic 0 on pin 30. For any kind of 28-pin chip LK4 must be East to route the +5V supply to what is then pin 28.



LK5. When closed, QD from the motherboard 74LS163 latch is connected through to pin 1 on SKT_B. When open, the onboard resistor R11 pulls this pin high.

LK6. When North, a Write Enable signal is routed through to pin 27 on a 28-pin chip in SKT_B. When South, QC from the motherboard 74LS163 latch is routed through to this pin instead. Left open, the pin will be pulled high by R10.

LK7 shorts out the diode D1 which normally prevents the backup battery, if fitted, from being charged. **Only short out this link if you are using a rechargeable battery such as a 3.6V NiMH cell. Otherwise it must be left open.** LK7 is in fact not fitted to make it less likely that it will be made accidentally when the backup battery is not of the rechargeable type.

This is a summary of the links for SKT_B

Device	LK5	LK6	Number of ROMs	ROM numbers
16kB (27128)	Open	Open	1	15
32kB (27256)	Open	South	2	11,15
64kB (27512)	Closed	South	4	3,7,11,15
32kB SRAM	Closed	North	2	7,15

Notes:

When using a 28 pin chip in SKT_A, make sure the unused pins are at the back of the machine. See this photo here.

When using a 32kB static RAM chip in either SKT_A or SKT_B it will not be battery backed. Write protection for SKT_A and SKT_B can be achieved by respectively removing LK3 and LK6 as shown on the previous page. With the 128kB surface mount RAM present at all times, battery backed and easily write protectable with a switch, it is not envisaged that using 32kB RAM chips in SKT_A or SKT_B will be especially useful.

Suitable 32kB RAM chips, should you require them, typically require a search of the numbers 43256 or 62256 on eBay. Be sure to get the DIL or DIP package rather than any kind of surface mount device such as SOIC or SOJ.

The above arrangement of switches has proved very convenient for the 128K RAM board. The rearmost switch marked RD is connected across the TI (Total Inhibit) pins and the switch is closed when moved forwards in the direction of the arrow. The RAM chip can then be read from in the normal way. The other switch marked WR is the write protect switch, and the RAM is write protected when the switch is to the rear (open) as in the picture above. The RD switch is pushed rearwards whenever the computer is turned on or off and only moved to the forward position if and when needed. A Ctrl-Break is usually required in this type of situation, ie after moving the RD switch to the forward (closed) position.

Write protection is less useful than it was back in the 1980s because many of the ROM images now available on the internet have been modified to remove the anti-sideways RAM protection.

The BBC Micro's sideways ROM system

Here we will take a look at how the BBC Micro's paged ROM system works. Those with no interest in electronics and/or assembly language will hopefully get an understanding of what's going on. Readers who are au fait with such material should be able to fathom the system without difficulty. The Beeb's paged ROM system is very flexible but not particularly complex in hardware terms.

Examine the circuit diagram of any typical microcomputer system and various key features will become apparent. The first is that the microprocessor is very much in command and can talk to various different pieces of hardware through the system data bus. On the 6502, as used in the BBC Micro, this data bus is 8 bits wide. The CPU (Central Processing Unit, the 6502 in this case), always reads and writes eight bits at a time. The individual data lines are normally referred to as D0 through to D7, D0 being the least significant bit.

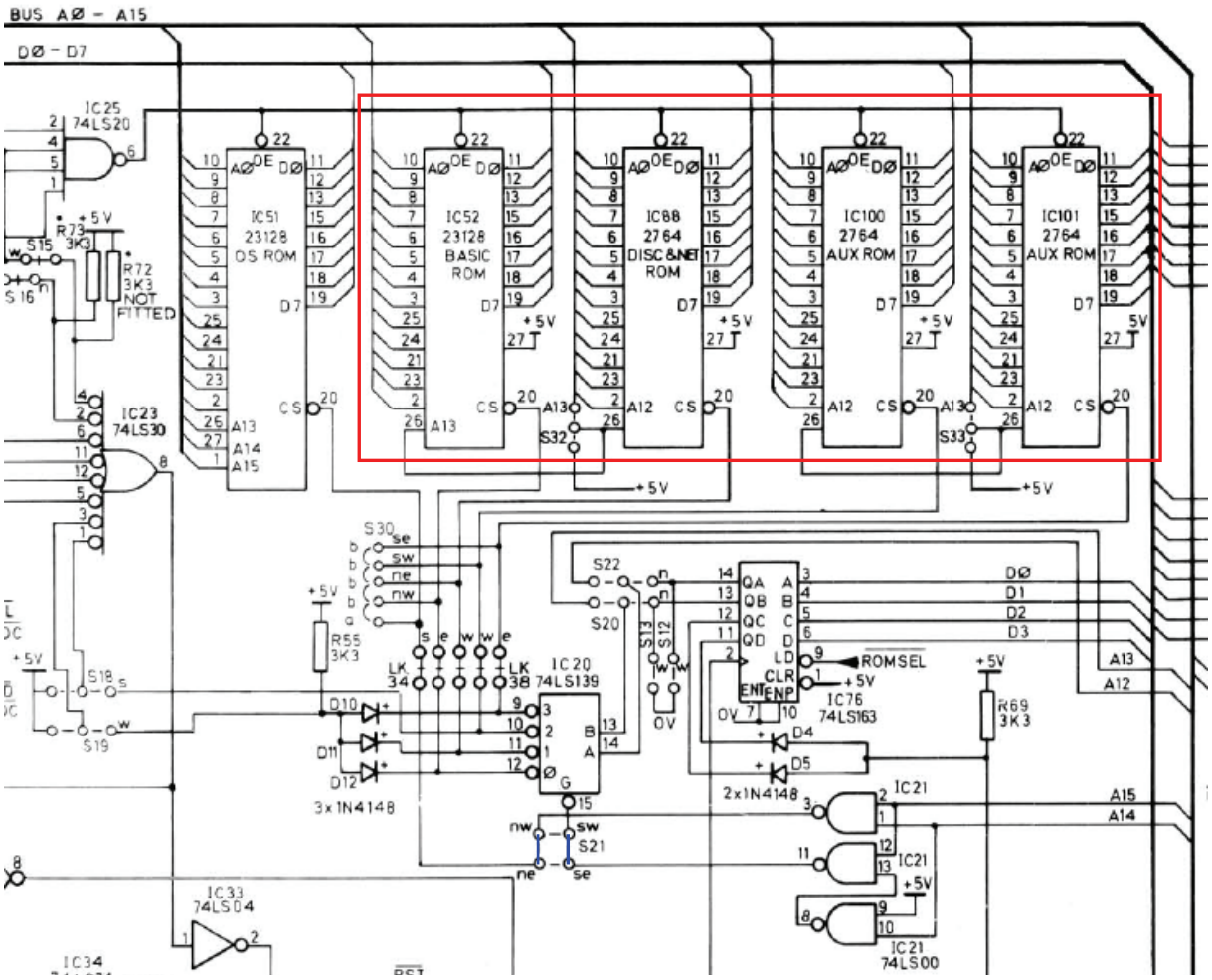
Examples of other pieces of hardware in the machine include the Operating System ROM, the System 6522 VIA (Versatile Interface Adapter), User VIA (optional), floppy disk controller (typically an Intel 8271 or perhaps a 1770 device), main memory and so on. All of these will be connected to the system data bus, and the obvious question arises as to how the processor can communicate with a specific device - the floppy disk controller chip for example - and temporarily ignore all the others.

Most chips have some kind of 'Enable' signal associated with them. This Enable signal will be generated by decoding circuitry on the main board, and the system is normally designed in such a way that any particular device (for instance the User VIA), can be accessed if and only if the CPU reads from or writes to a very specific address. Sometimes a chip can contain several different special purposes registers, each being located at a different address. Staying with the example of the User VIA in the BBC Micro, it contains 16 separate registers and these are located in the memory map between addresses &FE60 and &FE6F inclusive. When the CPU reads from or writes to those addresses, the Enable signal on the User VIA is activated. The User VIA alone will respond, other devices will remain in a disabled state because their individual Enable signals instruct them to remain inactive. The BBC Micro uses the '&' prefix to indicate that a number is hexadecimal. The CPU uses its R/W signal to indicate whether it is trying to read from or write to other chips in the circuit.

The fact that the User VIA is located at addresses &FE60 to &FE6F is not an essential requirement the 6502. The designers of the BBC Micro decided that the User VIA would be located at those addresses and the address decoding circuitry was implemented accordingly. A completely different machine might also use a 6522 VIA but locate it at addresses &EE80 to &EE8F. The 6502 does have a few addresses which are reserved for special purposes, but other than that the system designers have a great deal of leeway in terms of what hardware goes into the finished system and where.

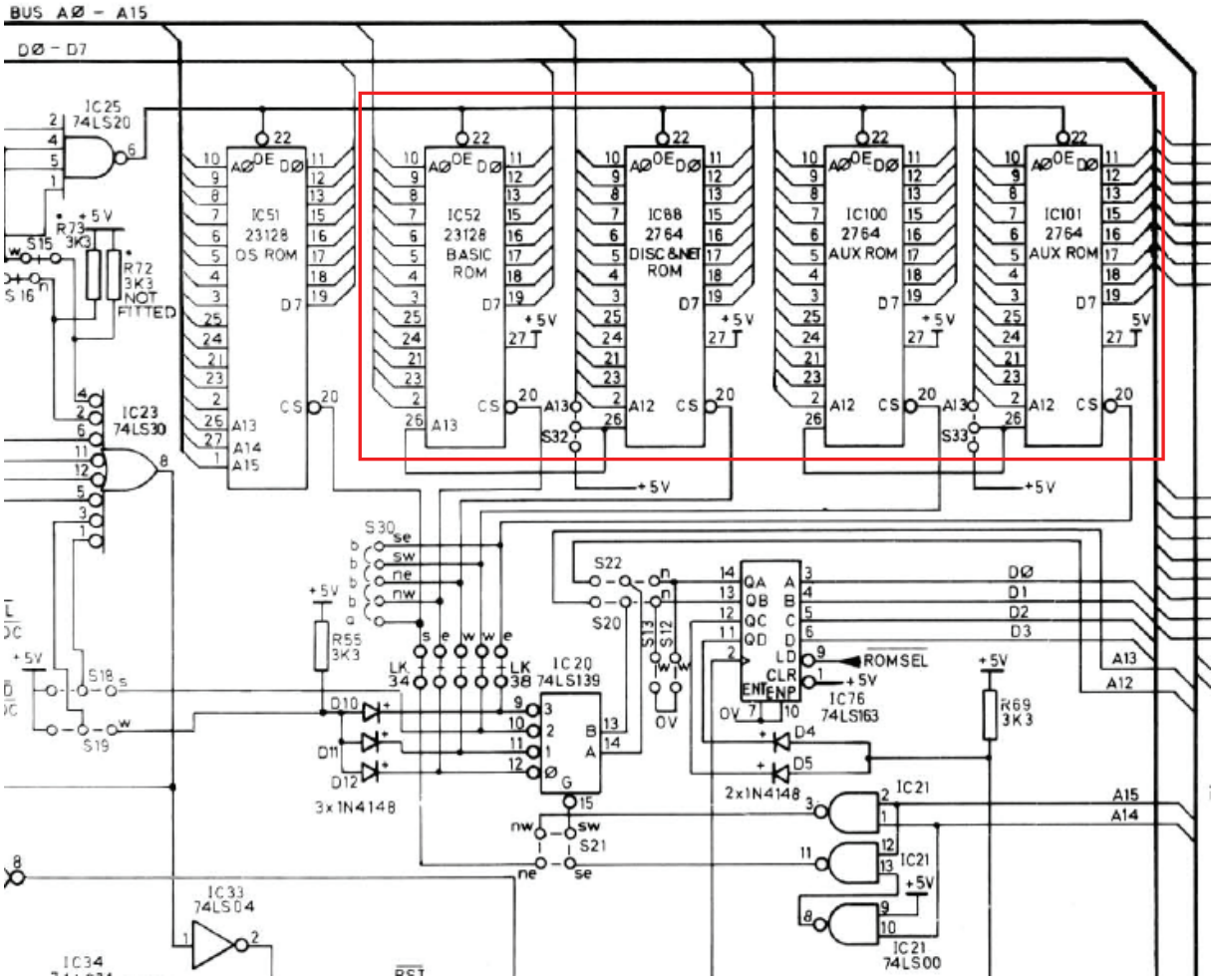
It is essential that only one device should try to control the data bus at any one time. Chaos would ensue if the floppy disk controller and one of the VIAs attempted to place data on the data bus at the same time. That is why a major fault on the motherboard, such as solder splash which permanently grounds one of the data lines, will stop the CPU in its tracks and prevent the system from showing any signs of life.

With a little bit of scene-setting now over we are in a position to look at the Beeb's sideways ROMs in a bit more detail. The diagram below shows the relevant circuitry. The four sideways ROMs are enclosed by a red rectangle, the ROM just outside it on the left is the Operating System ROM. The Operating System ROM occupies addresses &C000 through to &FFFF and is therefore 16kB in length (&4000 bytes). However, some addresses within that range are devoted to specialist chips such as the VIA mentioned previously. Furthermore, addresses &FC00 through to &FDFF are dedicated to the 1MHz bus, one of the areas of memory mapped I/O (Input/Output). We need not concern ourselves with this here.



The first point to observe from the above is how the eight data lines on each ROM are seen to be joined together. Hence D0 on one ROM is connected to D0 on all the others. Similarly for D1 to D7 inclusive. Fouteen address lines from the CPU, A0 to A13, are likewise connected to each of the address pins on the ROMs.

So far this is everything that we would expect. Four separate ROMs are connected to the system data bus to feed information to the CPU when required. The key to this is the CS (Chip Select) signal on pin 20. These are NOT all joined together. Each chip has its own unique CS signal generated by 2-to-4 line decoder, IC20 (74LS139).



Working backwards from the ROMs, the active low CS pin (number 20 on the ROM) is connected to one of the four decoder outputs (74LS139, IC20). This decoder has two inputs designated A and B on pins 14 and 13 respectively. These inputs can of course be in one of four possible states, 00, 01, 10 and 11. For each state of the A/B inputs just one of the outputs can go low and the other three will remain at a logic high level. The outputs are called 0, 1, 2 and 3 to tie in with the binary number on the inputs A and B. The actual pin numbers are seen to be 12, 11, 10 and 9. There is a further input called G on pin 15 which we will return to in a moment.

It is now necessary to see how the inputs to the decoder are controlled. Due to the links marked S20 and S22 it is not immediately clear where the signals go, but in a standard model B the A input on the LS139 comes from the QA output on IC76, the 74LS163. The B input on the LS139 comes from the QB output on the LS163.

IC76, the 74LS163, is a four bit synchronous counter. It is not used in a counting mode here and instead relies on the LD (Load) input to preset the counter with a known value. The LD input has been labeled ROMSEL by Acorn and is a signal that is activated by the CPU writing to address &FE30. The effect is that whenever a Write occurs to &FE30, the LS163 captures the least significant four bits of the data bus (D0 to D3), and stores them in the latch. The stored values are then available on QA, QB, QC and QD where they remain until a further write to &FE30.

So, as an example, the instructions;

```
LDA #2                  \Binary 0000 0010
STA &FE30
```

will result in the number 2 being stored in the LS163. Specifically, QB will be logic high and QA, QC and QD will all be logic low.

Recall that the CPU always deals with complete bytes on the data bus but it is clear that the values of D4 to D7 when writing to ROMSEL are irrelevant. Only the least significant four bits (nibble) are stored in the latch and the upper nibble is disregarded. The lower nibble will have a value in the range 0 to 15. It is best to ensure that D4 to D7 are all zeroes when writing to ROMSEL to ensure compatibility with later machines such as the Master.

When the required number (0 to 15) has been written to ROMSEL we now know that this same number will appear on the outputs QA, QB, QC and QD of the LS163 latch. The QA and QB outputs are fed into the And B inputs on the 2-to-4 line decoder, IC20. This in turn drives one of the decoder outputs low as previously described. Or rather, it does when the G input referred to earlier is also at the right logic level.

The sideways ROMS are all designed to appear in the computer's memory map between the addresses &8000 to &BFFF inclusive. If the CPU is accessing any other area of memory, such as the operating system ROM or perhaps the machine stack in page 1 (&100 to &1FF), it would be undesirable to activate any of the sideways ROMs. For this reason the G (Global) input on the LS139 decoder is used.

The G input on pin 15 of IC20 is used as a master control input. When it is logic high all four decoder outputs will be off (high) also. The state of inputs A and B makes no difference. It is only when the G input goes low that the output defined by A and B goes low too.

G is required to go low when the CPU is accessing the sideways ROM space, &8000 to &BFFF. In binary these addresses are;

1000 0000 0000 0000 (A15 on the left, A0 at the extreme right)
1011 1111 1111 1111

What we can see is that common to all the addresses in the given range is that A15 is high and A14 is low. The decoding required to do this is very simple and can be seen in the bottom right of the diagram on the previous page. Three dual input NAND gates are used. The NAND output on pin 11 feeds the G input on the LS139 decoder, and will only go low when A15 is high and A14 is low. The bottom NAND gate in the group of three acts as an inverter because the second of its two inputs is tied high.

So we are nearly there. Whenever the CPU reads from or writes to an address in the range &8000 to &BFFF, it will be accessing one of the sideways ROMs because the G input on IC20 is low. The programmer must decide in advance which ROM is needed and this in turn is achieved by writing the required number to ROMSEL at &FE30

There are three more points to make to complete the picture. One is that it is not possible (in the Model B) to read from ROMSEL (eg LDA &FE30) and obtain meaningful information. In other words, you will not read back the number that was written there. For this reason there is always a copy of the value in ROMSEL in zero page location &F4. This copy does not appear by magic and the onus is on the programmer to make sure that a copy is stored there whenever you write to ROMSEL. For example;

```
LDA #3
STA &F4
STA &FE30
;
;
```

Notice the order of the STA instructions. Always write to &F4 first and then &FE30. The reasons for this are to do with interrupts. There is no need to disable interrupts when performing the above sequence. The code;

```
SEI
LDA #3
STA &F4
STA &FE30
CLI
```

will work but the SEI/CLI is not needed.

Very often as a programmer you will need to ascertain the number of the currently active ROM so that it can be activated again before your code exits That is one of the reasons why the copy in &F4 is important. Eg

```
LDA &F4             \Get the currently active ROM
PHA                 \Save it
LDA #4              \Activate ROM 4 (in this case)
STA &F4
STA &FE30
                    \Your code here
PLA                 \Retrieve original ROM number
STA &F4             \Activate it in the normal way
STA &FE30
RTS                 \Exit
```

The second point is that the Beeb's design was geared up to handle up to 16 sideways ROMs numbered 0 to 15. In the basic Model B, the QC and QD outputs of the LS163 latch are not connected to anything useful. Like the whole of the upper nibble, D2 and D3 become "don't care" states when writing to ROMSEL. The effect is that (again in an unexpanded model B), exactly the same data will appear in four different sockets, the only difference being the value on QC and QD.

Consider the following binary numbers

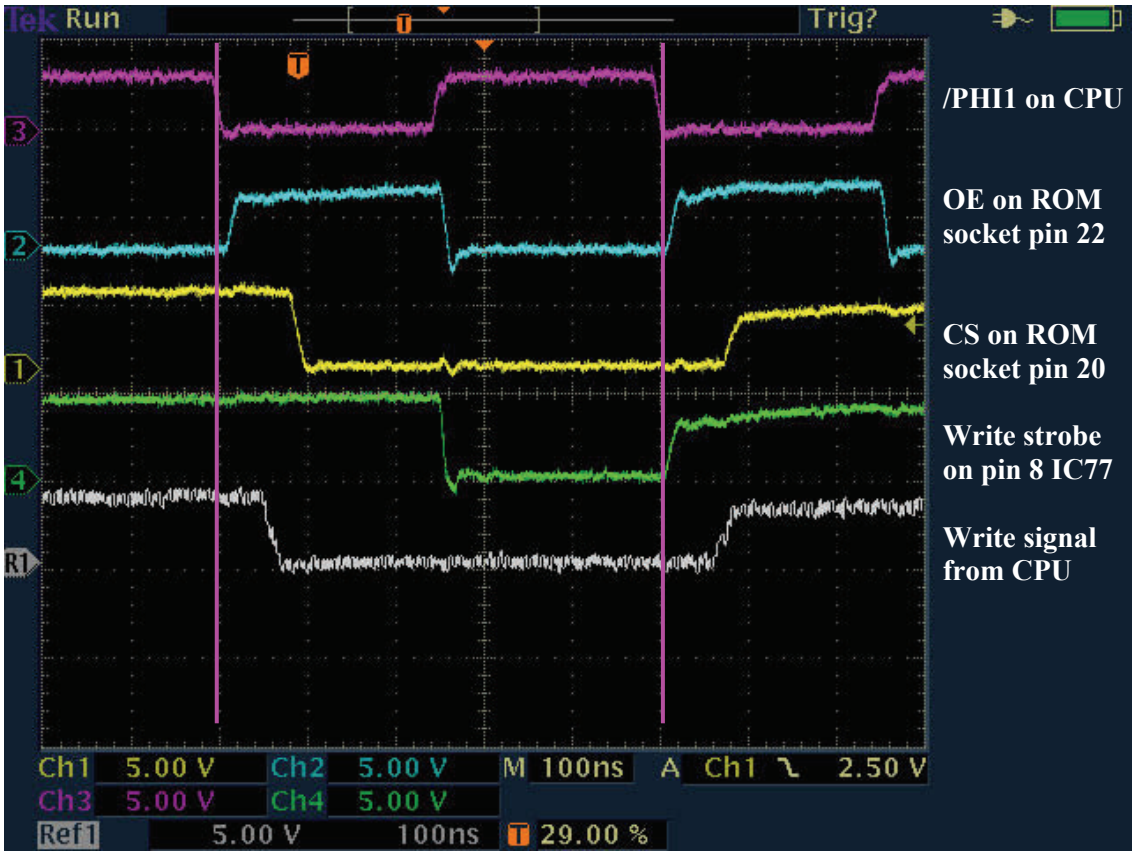
```
0010        (decimal 2)
0110        (decimal 6)
1010        (decimal 10)
1110        (decimal 14)
```

When written to ROMSEL in an unexpanded model B, they will all result in the same physical sideways ROM being selected. This is because the lowest two bits, latched into QA and QB on the LS163, are the same in all cases. When the computer is powered up or Ctrl-Break performed, it scans all the sockets 15 to 0 (in that order) to work out which ones appear to contain valid ROM images. As a rule the computer uses only the highest numbered socket and disregards what it thinks are identical copies in lower numbered sockets. That is why the socket numbers on a normal Beeb's motherboard are numbered 12 to 15 (left to right). The default language is the language with the highest ROM socket number. So for example, with BASIC in socket 14 and Wordwise in socket 12, the machine will normally power up with BASIC active. To use Wordwise it is necessary to type in the required command, *WORDWISE in this case. Swap the two chips around, however, and the machine will power up in Wordwise. It will now be necessary to enter *BASIC if that's what you wish to use. Needless to say, most users arrange their ROM numbers so that it powers up in their language of choice.

The final hardware consideration is the OE (Output Enable) signal on pin 22 of the ROMs. As far as the ROMs are concerned this is the last piece of the jigsaw. The implication up until now has been that the ROMs place data on the data bus when the CS (or CE, Chip Enable) signal goes low, but this is not entirely accurate. A low on CS changes the chip from a low power standby mode to one where it consumes a bit more power but is ready to respond very quickly. A low signal on OE as well will finally cause the chip to place data on the system bus. A high level on OE places the data lines on that ROM in a high impedance state - their loading on the bus is negligible. There are various reasons for this apparently elaborate arrangement. Power consumption is one and another is to do with the access time - the time taken for the chip to produce valid data after its address and control lines are stable.

```
10 DIM A 100
20 P%=A
30 [
40 LDA &F4
50 PHA
60 LDA #12
70 STA &F4
80 STA &FE30
90 STA &9000
100 PLA
110 STA &F4
120 STA &FE30
130 RTS:]
140 CALL A
```

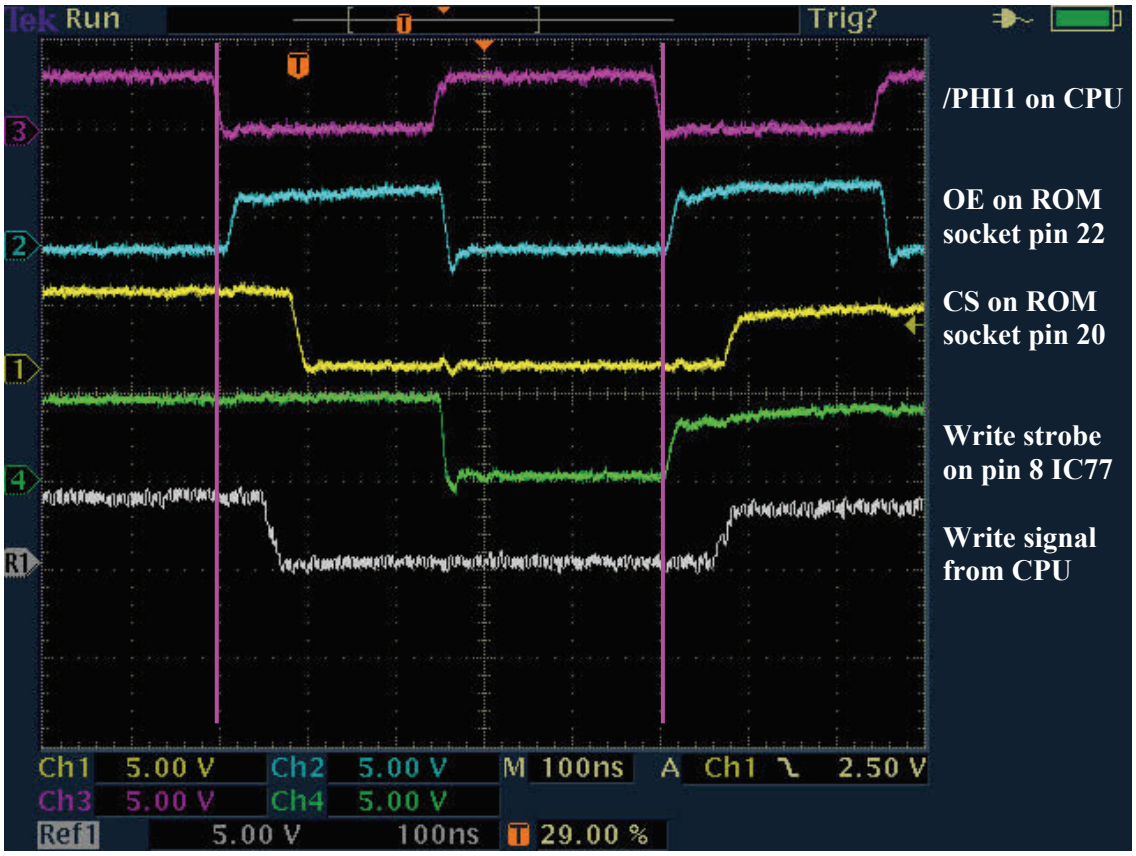
This program was used to produce the oscilloscope screen on the next page. It generates a low pulse on the CS signal, pin 20 of ROM socket 12. It is the STA instruction in line 90 which causes the low pulse on CS because it culminates in the CPU writing to address &9000 in the sideways ROM address range (&8000 to &BFFF)



This shows most of the relevant signals when a write operation occurs to one of the ROM sockets. Starting from the top, the purple trace is the main PHI2 clock (actually obtained in the Beeb by inverting PHI1 on pin 3) used by the processor. The two vertical cursors show a complete clock cycle and is in fact the Write operation resulting from the STA &9000 instruction (see program on previous page). The cursors are seen to be 500ns apart and this is to be expected from a 2MHz clock. It is the falling edge of PHI2 that is usually the critical moment. During a Read, the CPU latches data from the data bus on the falling edge of PHI2. At the completion of a Write, the data is guaranteed to be correct on the falling edge of PHI2 and it is up to the device being accessed to receive the data in a timely manner. The receiving device, perhaps a VIA or disk controller chip, will normally have some kind of "Write Strobe" or "Write Enable" input pin to accomplish this.

The blue Output Enable signal is from pin 22 on the ROM socket. The OE signal is common to all the ROMs and is generated on pin 6 of IC25, a 74LS20. This is a quad input NAND gate and one of the inputs is the 2MHz clock. When any input to a NAND gate is low, the gate's output will be high and it can be seen above how OE goes high shortly after the falling edge of PHI2. The slightly noticeable delay is the propagation delay in the NAND gate.

It is odd that Acorn designed OE to be low during a Write cycle. For a few hundred nanoseconds, there is contention on the data bus with the CPU trying to place a certain data pattern on the data bus and the ROM trying to output another.



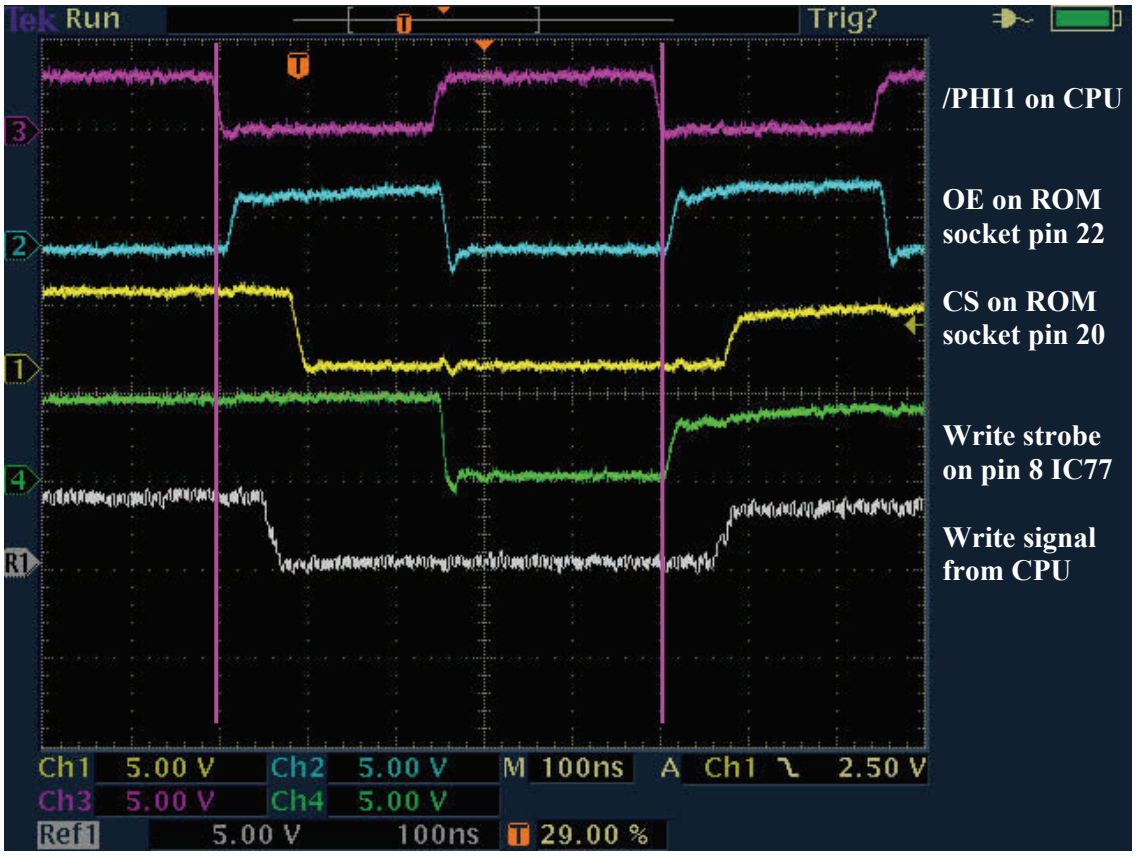
Same diagram as the previous page for ease of reference.

The yellow signal is the low Chip Select pulse on pin 20 of the ROM under test. Each ROM, remember, has its own unique Chip Select signal generated by 2-to-4 line decoder. We saw how one of the outputs of the decoder can only go low when a suitable address appears on the address bus. Specifically, A15 must be high and A14 must be low - when this happens a group of three NAND gates creates a logic 0 (low) on the G enable input on the decoder. The leftmost cursor is on the falling edge of PHI2 and marks the completion of a Read cycle by the CPU. (As a point of detail, the CPU was reading the most significant byte of the absolute address in the STA &9000 instruction. In other words, the data read was &90 but we cannot tell (using the information given) from what address that data was read. We only know that it was in user RAM, probably in page &19 somewhere).

Immediately after the falling edge of PHI2 marking the end of the Read cycle, the CPU started its Write cycle in order to place the contents of the accumulator at address &9000. It takes a while for the address lines to change and stabilise to the correct destination address, and only when this happens will CS go low. That is why there is very conspicuous delay between the falling edge of PHI2 and a change in CS.

The bottom white trace is the Read/Write signal from the CPU. A low indicates a Write operation is taking place.

After the Write cycle shown above, a read operation will occur. It will be an opcode fetch from the address immediately after the three-byte STA &9000 instruction.



Same diagram as the previous page for ease of reference.

The fact is that the Write signal will go low before the data bus and address bus have both settled to stable logic states. We know this is so because the above illustration shows that the CPU Write (bottom trace) goes low well before the falling edge of CS, which in turn only occurs when A15 and A14 are 1 and 0 respectively.

That is why you cannot use the R/W signal directly from the processor when trying to write to a static RAM chip installed in one of the sideways ROM sockets. Random addresses will be flashed with unknown data and cause havoc. The falling edge of the Write signal needs to be delayed and the way this is normally done is to use the signal available on pin 8 of IC77 (green trace). Yet again an arrangement of NAND gates ensures that this Write Strobe signal is only low when PHI2 is high and also the CPU Write signal is low (bottom white trace). The moment PHI2 falls, the green Write Strobe signal rises a few nanoseconds later.

It is fortunate, when checking the data sheets for a 32kx8 static RAM, that the OE pin on the chip is a “don’t care state” during a Write cycle. In other words it can be either high or low and the write will still be successful. If it were necessary for the OE signal to be high during a write to a static RAM then it would not be so easy to add a 32kB RAM chip to an ordinary model B.

It is worth noting that a Read cycle from the sideways ROM space looks almost exactly like the above diagram. The only difference is that, fairly obviously really, the green Write Strobe remains high as does the Read/Write signal from the CPU (bottom trace).

Using the spare space in the W29C020 - method 1

The RAM/ROM board treats the W29C020 chip as being four banks of 16kB in sockets 2, 6, 10 and 14. This is a total of only 64kB, yet the Winbond 29C020 is a 256kB chip.

To achieve a capacity of 256kB the W29C020 has 18 address lines, A0 to A17. The two most significant ones, A16 and A17, are permanently grounded on the expansion board. If, however, we wished to access the chip's full capacity then normally we would want to be able to control the logic level on these address pins. With each pin having two possible logic states, the total number of combinations of A16 and A17 is four (00, 01, 10 and 11). Four lots of 64kB is of course the full 256kB.

One possibility would be to implement a writable latch similar to the one at &FE30 (ROMSEL). Let us assume that such a latch was implemented at address &FF30, the idea would be that a write to &FF30 would capture D0 and D1 from the data bus in, for example, a device such as a 74HCT74 dual D-type latch. The outputs of the latch would then be connected to A16 and A17 on the W29C020. It would be important to include the R/W signal in the decoding so that a read operation from address &FF30 would be ignored and that only a write would trigger the latch.

Just putting a bit more detail here, the requirement is to decode some of the address lines and control signals from the CPU such that when a Write operation occurs to address &FF30 (for this example), a positive going edge is generated which is fed to the clock input of the HCT74 D-type latch. D0 and D1 from the system data bus are fed to the two D inputs of the latch and the positive going edge generated by the decoding logic clocks the values on D0 and D1 onto the Q outputs of the latch. It is these Q outputs which would be fed to the A16 and A17 address pins of the W29C020. These Q outputs should only change when a Write occurs to address &FF30. Of course, you might not wish to fully decode all the address lines and this might mean that a Write to FF3x would trigger the latch. In other words, all addresses between and including &FF30 and &FF3F have the same result. You would probably want to put some kind of RC network on the RST input of the latch so that the machine power up in a known state, that is with the two Q outputs set to zero.

Effectively, this approach provides a software means of paging in the previously unused areas of the W29C020. For instance

```
?&FF30=0              ( or LDA #0:STA &FF30 )
```

would set A16 and A17 to zero (which they normally are), and

```
?&FF30=2              ( LDA #2:STA &FF30 )
```

would set A16 to zero and A17 to one. Every time the state of A16 and A17 is altered, a new block of four ROMs appears in the Beeb's sideways ROM sockets at number 2, 6, 10 and 14. With this method you could have 28 sideways ROMs installed in the machine, but of course only 16 would be visible at any one time. It would be easy to write a kind of 'Bank Manager' in the form of a sideways ROM. This would make possible commands such as;

```
*BANK 2
```

this being easier to remember than ?&FF30=2. Yet another possibility is to capture D0, D1 and D2 in three D-type latches (part of an HCT174) and use these to page in up to eight banks of a chip such as an A29F040 (512kB). That's 32 sideways ROMs in a single chip (available as eight groups of four), plus eight banks of sideways RAM, plus (potentially) four more ROMs programmed into a 27512 in SKT_B. Forty four ROMs in all. Then there's the 1MB M27C801 which could offer sixteen banks of four ROMs…

NB. On the RAMROM_B2 board A16 and A17 are tied to ground (the latter by means of a link). The modification outlined above and below would involve bending up the pins on the W29C020 before connecting them to the outputs of any logic devices. This alteration would be at your own risk. Note also that the FLASH program as supplied assumes that the logic state of both A16 and A17 is zero. If A16 and A17 are controlled by your own logic then the program would need a few changes to take account of this.

Using the spare space in the W29C020 - method 2

This approach will allow you to have eight banks of flash memory (sockets 2, 3, 6, 7, 10, 11, 14 and 15) in addition to the 8 banks of sideways RAM (sockets 0, 1, 4, 5, 8, 9, 12 and 13). The advantage is that you can potentially have the full 16 ROM sockets without the need for an EPROM programmer and it's much easier to implement than method 1 above. The main disadvantages are that the rightmost socket (SKT_B) becomes unusable unless the modification is undone. Additionally, if you accidentally corrupt or erase the W29C020 then again the modification will probably have to be removed, at least temporarily, in order to recover from the situation. The procedure (untested, I add) is as follows.

1) Remove the W29C020 from its socket and bend outwards pins 22 (Chip Enable) and pin 30 (A17). Replace the chip in the socket, ensuring that pins 22 and 30 are not touching anything.
2) Using two gates from a 74LS00 (for example), implement a dual input AND gate. Feed one of the inputs from pin 22 of SKT_A (with the W29C020) and the other from pin 20 of SKT_B.
3) Take the output from the above gate and attach it to pin 22 of the W29C020.
4) Connect pin 22 of SKT_A (which also goes into the AND gate) to pin 30 of the W29C020.

This modification works by ensuring that CE on the W29C020 (pin 22) goes low when either CE on SKT_A or SKT_B goes low. When CE on SKT_A goes low, A17 on the W29C020 will be low too. This is in fact the default arrangement. To get A17 high, you will need to set CE on SKT_B low (because the CE signals for the ROM sockets come from a 1 of 4 decoder). This in turn is achieved by writing either 3, 7, 11 or 15 to ROMSEL located &FE30.

As with method 1, you will need to modify the FLASH program slightly. When writing the command sequence to the W29C020 (see data sheet), A17 needs to be low. Therefore the number written to ROMSEL must be 2, 6, 10 or 14. When writing the actual data to the W29C020, A17 will need to be correctly set as described in the above paragraph.

Understanding the GAL16V8

The ROM/RAM board contains a single GAL16V8 to perform all of the required decoding. It is an example of a simple PLD or Programmable Logic Device. Devices such as these can often replace a dozen or so discrete logic parts, saving both on board space and cost. The 16V8 can have up to 16 inputs and 8 outputs, although not simultaneously as there are not enough pins. The 'V' stands for versatile and is supposed to be descriptive of the OLMC (Output Logic Macro Cell).

Anyone programming in 6502 assembly language will normally write their code in assembly language and then use an assembler to produce the executable machine code. Few people would choose to write machine code by poking numbers into memory. Similarly, although the 16V8 can have several operating modes, it is normal to write the required logic equations with nothing more than a text editor and then produce the fuse pattern for the PLD with suitable software. Such software will automatically select the correct operating mode and flag up illegal syntax (such as trying to define a pin to be some kind of logic output when that same pin can be used as an input only).

Use of PLDs can help to keep commercial designs secure by making it difficult to fathom what their functions are. The presence of a security bit prevents simple copying of the fuse pattern. However, in this case the 16V8 has been used purely as a matter of design convenience rather than to make the whole project into a closely guarded secret.

The precise syntax used to write the logic equations will vary from one software package to another. Here, the "+" symbol implies the logical OR with the "*" being used for AND. A preceding "/" acts a logical NOT. Writing an equation to produce the required result can be very simple indeed, for example;

```
Z = A * B + C
```

Z = (A AND B) OR C

Some equations may be easy enough to write down by inspection. Truth tables, Karnaugh maps or other reduction techniques may be useful for more complex problems.

The Write Enable for the flash ROM (WE) is a function of the Output Enable signal on the motherboard socket (OE_MB), the Read/Write signal on one of the flying leads (RW_FL) and also the partial Write Protection link next to R1 (WP_PART). WP_PART will be high when the link is removed and this must prevent WE on the flash ROM from going low. Therefore;

```
WE = OE_MB + RW_FL + WP_PART
```

We could also have written;

```
/WE = /OE_MB * /RW_FL * /WP_PART
```

using DeMorgan's Law. The second equation tells us that WE on the flash ROM goes low when the OE signal on the mother board is low AND a processor Write is occurring AND the WP_PART link is in place (thus producing a logic 0 on the input to the GAL.). Of course, the Chip Enable pin must also be low.

The flash ROM data sheet is very clear on one point. The Output Enable on the flash ROM chip must remain high during a Write cycle, yet a glance at the waveforms on page 38 shows that OE on the motherboard (OE_MB) is low during the second half of the 2MHz clock period for both reads and writes. OE_MB cannot therefore be connected directly to the OE pin on the flash ROM (pin 24) because this would inhibit the write. OE on the flash ROM must be generated by the GAL;

```
OE_FLASH24 = OE_MB + /RW_FL
```

OE_FLASH24 is now always high whenever the CPU's Read/Write signal is low.

The 128kB static RAM chip is the 32 pin SOIC surface mount IC on the left of the board. It has two Chip Enable pins, one being active high and the other active low. Both must be in the correct state in order to activate the chip, but in fact the active high CE is permanently pulled to +5V. Thus, only the active low pin (22) is used.

The idea is that the 128kB RAM is activated whenever one of the leftmost two sockets on the motherboard is selected, that is when either of the Chip Enable signals (pin 20 on the socket) goes low. When this happens, the CE pin on the RAM chip must go low too. Therefore;

```
/CE_RAM22 = /CE1 + /CE2
```

where CE1 and CE2 are the two Chip Enables from the motherboard. In words, CE on the RAM chip goes low if CE1 goes low OR CE2 goes low. Strictly speaking, CE on the RAM will also go low if both CE1 and CE2 are low simultaneously. However, CE1 and CE2 are generated by a 2-to-4 line decoder so in practice this can never happen.

A15 and A14 on the RAM chip merely track the signals present on the green and yellow flying leads (which come from the 4-bit latch on the motherboard). That is;

```
A15_RAM31 = A15_FL
A14_RAM3  = A14_FL
```

The most significant address line (A16) is determined directly by the state of the Chip Enable on the leftmost motherboard socket. Hence;

```
A16_RAM2 = CE1
```

This means that when CE1 is low, A16 on the 128kB RAM will be low too. When CE1 goes high then A16 will do so also. If both CE1 and CE2 are high - as they will be when one of the two sockets on the right are active - then the state of A16 is irrelevant because the RAM chip will not be enabled.

The Write Enable signal on the 128kB RAM is generated from the Output Enable OE_MB on the motherboard sockets (OE is common to all four sockets), the R/W signal from the CPU, the partial write protect link (WP_PART) and also A15 (A15_FL)

```
WE_RAM29 = OE_MB + RW_FL + WP_PART * A15_FL
```

This is the SAVEROM program written in BBC BASIC. When you have finished entering the program type RUN. This trivial program helps to show just how useful the assembler built into BBC BASIC really was. The assembler turns the assembly language section (lines 140 to 320) into *machine code*, and the CALL statement causes that code to be executed.

As a rule the program should be entered exactly as it appears below. Some parts are case sensitive and be careful not to confuse similar characters (eg the letter 'O' with a zero).

```
 10REM > SAVEROM
 20PROCassemble
 30INPUT "Socket number "X%:X%=X% AND 15
 40CALL code
 50OSCLI("SAVE ROM"+STR$X%+" 3000+4000")
 60END
 70DEFPROCassemble
 80DIM code 100
 90FOR pass% = 0 TO 2 STEP 2
100P%=code
110src=&70:dest=&72
120!src=&8000:!dest=&3000
130[OPT pass%
140LDA &F4          \Save current socket
150PHA
160STX &F4          \Activate chosen socket
170STX &FE30
180LDY #0           \For indirect indexed addressing
190.loop
200LDA (src),Y      \Read from ROM space
210STA (dest),Y     \Write to RAM
220INY              \Increment offset
230BNE loop
240INC src+1
250INC dest+1
260LDA src+1
270CMP #&C0         \Check for all done
280BNE loop
290PLA              \Retrieve original ROM number
300STA &F4          \Activate it
310STA &FE30
320RTS
330]:NEXT
340ENDPROC
```

```
  5 REM > LOADROM
 10MODE 7
 20PROCassemble
 30INPUT "Filename to load "f$
 40INPUT "Socket number (0-15) "X%
 50X%=X% AND 15
 60OSCLI("LOAD "+f$+" 3000")
 70?(&2A1+X%)=0:CALL A
 80INPUT "Perform reset (Y/N) ";
 90yn%=GET AND &DF
100IF yn%=ASC"Y" THEN ?&FE4E=127:CALL !-4 ELSE PRINT"N"
110END
120:
130DEFPROCassemble
140DIM A 100
150P%=A
160src=&70:dest=&72
170!src=&3000:!dest=&8000
180[OPT 2
190LDY #0
200LDA &F4
210PHA
220STX &F4
230STX &FE30
240.loop
250LDA (src),Y
260STA (dest),Y
270INY
280BNE loop
290INC src+1
300INC dest+1
310LDA dest+1
320CMP #&C0
330BNE loop
340PLA
350STA &F4
360STA &FE30
370RTS
380]
390ENDPROC
```

The LOADROM program will load ROM images from the current filing system into a bank of sideways RAM. Observe the trick in line 100 of a simulated reset by writing 127 to address &FE4E, then calling the 6502 processor reset vector with CALL !-4

```
 10REM COPYBAS
 20REM Copies BASIC to a S/W RAM socket
 30DIM buff% 260, code 100
 40count%=&80:buffptr%=&70
 50romptr%=&72:destrom%=&81
 60!buffptr%=buff%
 70!romptr%=&8000
 80PROCassemble
 90INPUT "ROM socket (0,1,4,5,8,9,12,13) "n%
100?destrom%=n% AND 15
110CALL code
120END
130:
140DEFPROCassemble
150FOR pass%=0 TO 2 STEP 2
160P%=code
170[OPT pass%
180  LDX &F4
190  LDA #64
200  STA count%
210  LDY #0
220.mainloop
230  JSR romsel  \Select BASIC
240.loop1
250  LDA (romptr%),Y
260  STA (buffptr%),Y
270  INY
280  BNE loop1
290  LDA destrom%
300  STA &F4:STA &FE30
310.loop2
320  LDA (buffptr%),Y
330  STA (romptr%),Y
340  INY
350  BNE loop2
360  INC romptr%+1
370  DEC count%
380  BNE mainloop
390.romsel
400  STX &F4:STX &FE30
410  RTS
420]
430NEXT
440ENDPROC
```

This is the FLASH program for programming the W29C020 in the machine. This is a fully working version although the one supplied on the W29C020 itself or on the CD may have had some slight tweaks.

The FLASH program is on the W29C020 in ROM Filing System format. It is also on the CD as part of an SSD image (for use with the MMC system) and also as a text file listing. Typing this program in by hand should be something of a last resort.

```
   10REM >FLASH
   20REM Programs the Winbond W29C020
   30IF FNsp THEN PRINT'"Please turn off 2nd processor":END
   40IF !&D000<>&A8086918 PRINT "Not a Model B":END
   50PROCinit
   60MODE7
   70PROCassemble
   80PROCwinbond:IF fail% END
   90file$="no data"
  100REPEAT
  110  CLS
  120  PRINT TAB(2,5)"S) Set ROM number ("STR$rom%")"
  130  PRINT TAB(2,6)"L) Load data to buffer ("file$")"
  140  PRINT TAB(2,7)"P) Program ROM "STR$rom%
  150  PRINT TAB(2,8)"E) Erase entire ROM"
  160  PRINT TAB(2,9)"X) Exit"
  170  PRINT TAB(2,10)"*) Star command"
  180  PRINT TAB(2,12)"Choice ?";
  190  REPEAT
  200    a%=GET:IF a%<>42 THEN a$=CHR$(a% AND &DF) ELSE a$="*"
  210  UNTIL INSTR("SLPEX*",a$)>0
  220  PRINT a$'
  230  IF a$="S" THEN PROCromnum
  240  IF a$="L" THEN PROCload
  250  IF a$="P" THEN PROCprogram
  260  IF a$="E" THEN PROCerase
  270  IF a$="X" THEN PROCreset
  280  IF a$="*" THEN PROCstar
  290UNTIL a$="X"
  300END
  310:
  320DEFPROCromnum:LOCAL J%,m%
  330PRINT "Enter ROM number  ";
  340FOR J%=1 TO 4
  350 m%=rsock%(J%)
  360 IF m%<>0 THEN PRINT STR$m%" ";
  370NEXT
  380PRINT
  390INPUT k%:fail%=TRUE
  400FOR J%=1 TO 4
  410IF rsock%(J%)=k% THEN fail%=FALSE
  420NEXT
  430IF fail% THEN PRINT '"Illegal ROM socket":VDU7:PROCpak ELSE rom%=k%
  440ENDPROC
  450:
  460DEFPROCload:LOCAL f$,h%,s%
  470PRINT "Enter filename ";
  480INPUT f$:h%=OPENINf$
  490IF h%=0 OR LENf$>20 THEN PRINT "Invalid name":PROCpak:file$="no
data":ENDPROC
  500s%=EXT#h%:CLOSE#h%
```

```
  510IF s%>&4000 THEN PRINT "File too long":PROCpak:file$="no
data":ENDPROC
  520file$=f$
  530$oscli%="LOAD "+file$+" "+STR$~buff%
  540X%=oscli% AND 255:Y%=oscli% DIV 256
  550CALL &FFF7
  560s%=!((buff%?7)+buff%)
  570IF s%<>&29432800 AND ?buff%<>&C9 THEN PRINT '"Data may be invalid.
Warning only.":VDU 7:PROCpak
  580ENDPROC
  590:
  600DEFPROCprogram
  610IF file$="no data" THEN PRINT "Load data into buffer":VDU
7:PROCpak:ENDPROC
  620IF fsrom%=rom% THEN VDU7:PRINT"Warning"'"About to overwrite filing
system"
  630PROCconfirm:IF fail% THEN PRINT '"Aborted.":PROCpak:ENDPROC
  640IF fsrom%=rom% THEN *TAPE
  650?(&2A1+rom%)=0
  660!src%=buff%:!dest%=&8000
  670?r%=rom%
  680PRINT TAB(3,19)"Programming page     of 128"
  690CALL program%
  700PRINT '"   Verifying...";
  710!src%=buff%:!dest%=&8000
  720CALL verify%
  730VDU7:IF ?flag%=0 THEN PRINT '"Fail":VDU 7 ELSE PRINT '"Pass"
  740PROCpak
  750ENDPROC
  760:
  770DEFPROCerase
  780IF n%<4 THEN PRINT "Chip in use - cannot erase":PROCpak:ENDPROC
  790IF fsinflash% THEN VDU7:PRINT"Warning"'"About to overwrite filing
system"
  800PROCconfirm
  810IF fail% THEN PRINT '"Aborted.":PROCpak:ENDPROC
  820IF fsinflash% THEN *TAPE
  830?&2A3=0:?&2A7=0:?&2AB=0:?&2AF=0
  840CALL erase%
  850PRINT"Done.":PROCpak:ENDPROC
  860:
  870DEFPROCstar
  880INPUT '"*"com$
  890IF LENcom$>30 THEN PRINT"Too long":PROCpak:ENDPROC
  900$oscli%=com$
  910Y%=oscli% DIV 256:X%=oscli% AND 255
  920CALL &FFF7:PROCpak:ENDPROC
  930:
  940DEFPROCconfirm:LOCAL a$
  950PRINT'"Confirm programming/erasure ";
  960INPUT a$
  970fail%=a$<>"YES"
  980ENDPROC
  990:
 1000DEFPROCpak:LOCAL G%
 1010PRINT 'pak$;:G%=GET
 1020ENDPROC
 1030:
 1040DEFPROCreset
 1050PRINT"Perform reset? (Y/N)";:X%=GET
 1060IF X%=ASC"y" OR X%=ASC"Y" THEN ?&FE4E=127:CALL !-4 ELSE PRINT
```

```
1070ENDPROC
1080:
1090DEFPROCvalidroms:LOCAL m%,j%
1100n%=0
1110FOR m%=1 TO 4
1120j%=4*(m%-1)+2
1130IF j%<>?&F4 THEN n%=n%+1:rsock%(m%)=j%
1140NEXT
1150ENDPROC
1160:
1170DEFPROCwinbond:CALL id%
1180fail%=(?flag%=0)
1190IF fail% THEN PRINT"Check chip and/or links"
1200ENDPROC
1210:
1220DEFFNsp
1230A%=234:X%=0:Y%=255
1240K%=(USR&FFF4 AND &FFFF) DIV 256
1250=K%>0
1260DEFFNmodelB
1270IF !&D000=&A8086918 THEN =TRUE ELSE =FALSE
1280:
1290DEFPROCassemble
1300FOR J%=0 TO 2 STEP 2
1310P%=code%
1320[OPT J%
1330.id%
1340  LDA &F4
1350  PHA
1360  SEI
1370  LDX #&80
1380  JSR sequence1
1390  LDX #&60
1400  JSR sequence1
1410  LDA #FNlatch(&0000)
1420  JSR romsel
1430  LDX #10     \Short delay
1440.delay
1450  DEX
1460  BNE delay
1470  LDA FNaddr(&0000)
1480  CMP #&DA    \Winbond?
1490  BNE storeX  \X=0
1500  LDA FNaddr(&0001)
1510  CMP #&45    \Device type
1520  BNE storeX
1530  DEX         \X=255
1540.storeX
1550  STX flag%
1560  LDX #&F0
1570  JSR sequence1
1580  CLI
1590  PLA         \Original ROM
1600  JMP romsel
1610:
1620.erase%
1630  LDA &F4
1640  PHA:SEI
1650  LDX #&80
1660  JSR sequence1
1670  LDX #&10
```

```
1680  JSR sequence1
1690  CLI
1700  PLA
1710  JMP romsel
1720:
1730.program%
1740  LDA &F4
1750  PHA
1760  LDA #1
1770  STA pagenum%
1780  LDA r%
1790  JSR romsel
1800.progloop
1810  SEI
1820  LDX #&A0
1830  JSR sequence1
1840  LDA r%
1850  JSR romsel
1860  LDY #0       \Initial offset
1870.write
1880  LDA (src%),Y
1890  STA (dest%),Y
1900  INY
1910  BPL write    \End when Y=128
1920  CLI
1930  TAY          \Last byte stored
1940  LDA #31      \PRINT TAB(20,19)
1950  JSR osascii
1960  LDA #20
1970  JSR osascii
1980  LDA #19
1990  JSR osascii
2000  LDA pagenum%
2010  LDX #ASC"0"
2020  CMP #100
2030  BCC cc1
2040  INX
2050  SBC #100
2060.cc1
2070  PHA
2080  TXA
2090  JSR osascii  \100s
2100  PLA
2110  SEC
2120  LDX #ASC"0"
2130.lp10
2140  SBC #10
2150  BCC cc2
2160  INX
2170  BCS lp10
2180.cc2
2190  ADC #10+ASC"0"
2200  PHA
2210  TXA
2220  JSR osascii  \10s
2230  PLA
2240  JSR osascii  \1s
2250:
2260  TYA:TAX      \Save last byte
2270  LDY #127     \End of page
2280.busy
```

```
2290  TXA          \Toggle check
2300  EOR (dest%),Y
2310  BMI busy
2320  LDA src%     \Update pointers
2330  EOR #128
2340  STA src%
2350  BMI cc3
2360  INC src%+1
2370.cc3
2380  LDA dest%
2390  EOR #128
2400  STA dest%
2410  BMI cc4
2420  INC dest%+1
2430.cc4
2440  INC pagenum%
2450  LDA pagenum%
2460  CMP #129
2470  BNE progloop
2480  PLA
2490  BPL romsel   \BRA
2500:
2510.verify%
2520  LDA &F4
2530  PHA
2540  LDA r%
2550  JSR romsel
2560  LDX #0       \Assume fail
2570  LDY #0
2580.verloop
2590  LDA (src%),Y
2600  CMP (dest%),Y
2610  BNE store
2620  INY
2630  BNE verloop
2640  INC dest%+1
2650  INC src%+1
2660  LDA dest%+1
2670  CMP #&C0
2680  BNE verloop
2690  DEX          \=&FF, pass
2700.store
2710  STX flag%
2720  PLA
2730  BPL romsel   \BRA
2740:
2750.sequence1
2760  LDA #FNlatch(&5555)
2770  JSR romsel
2780  LDA #&AA
2790  STA FNaddr(&5555)
2800  LDA #FNlatch(&2AAA)
2810  JSR romsel
2820  LDA #&55
2830  STA FNaddr(&2AAA)
2840  LDA #FNlatch(&5555)
2850  JSR romsel
2860  STX FNaddr(&5555)
2870  RTS
2880.romsel
2890  STA &F4
```

```
2900  STA &FE30
2910  RTS
2920]:NEXT
2930ENDPROC
2940DEFFNlatch(a%):LOCAL k%
2950=((a% DIV 4096) AND &C) OR 2
2960DEFFNaddr(add%)
2970=(add% AND &3FFF) OR &8000
2980:
2990DEFPROCinit
3000DIM rsock%(4),buff% 16390,oscli% 35,code% 300
3010flag%=&80:src%=&70:dest%=&72:pagenum%=&75
3020r%=&81:osascii=&FFE3
3030pak$="Press a key"
3040IF ?&213=&FF THEN fsrom%=?&DBC ELSE fsrom%=-1
3050fsinflash%=((fsrom% AND 3)=2)
3060PROCvalidroms:rom%=rsock%(1)
3070ENDPROC
```