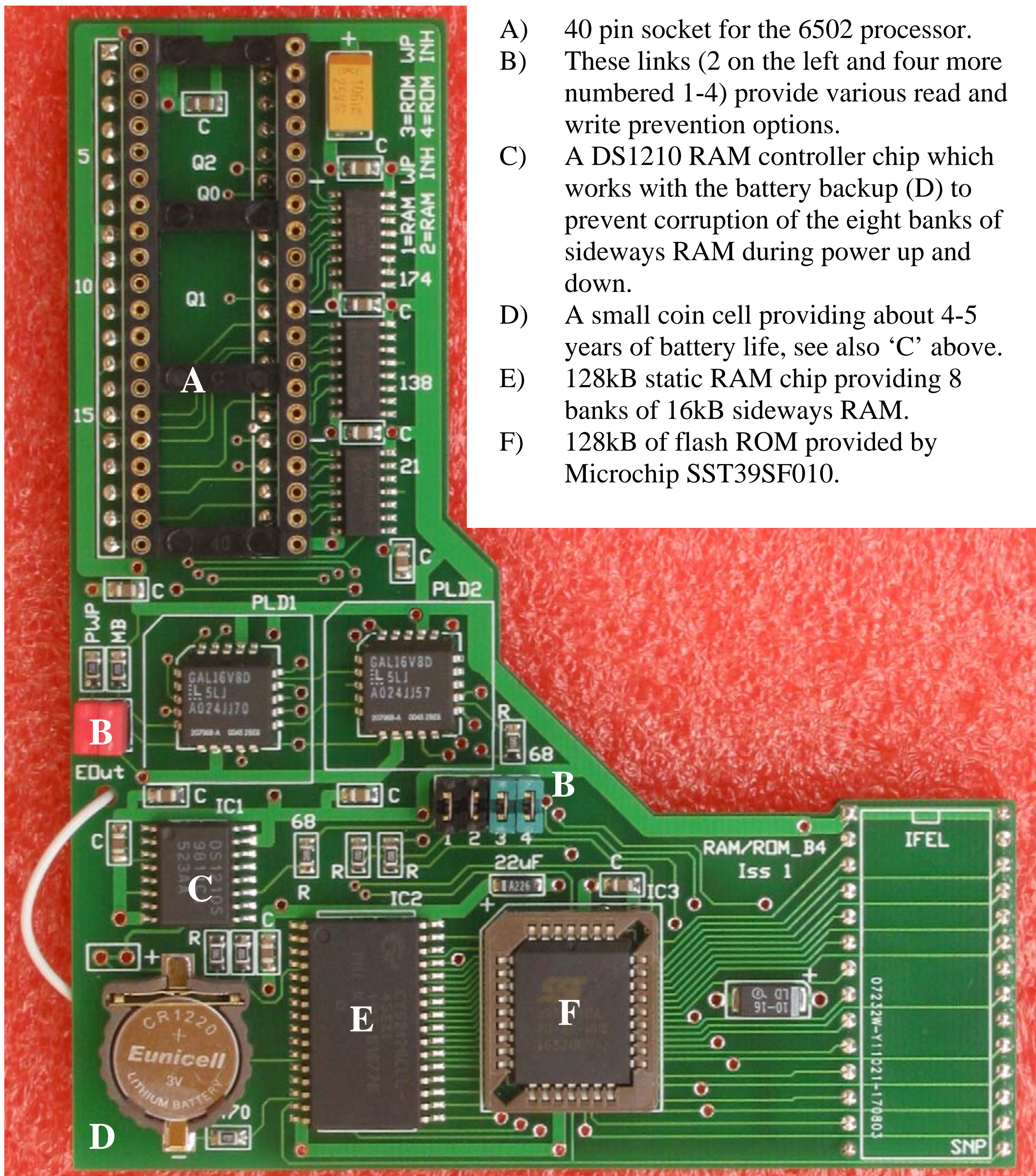
A quick board summary



A) 40 pin socket for the 6502 processor.

B) These links (2 on the left and four more numbered 1-4) provide various read and write prevention options.

C) A DS1210 RAM controller chip which works with the battery backup (D) to prevent corruption of the eight banks of sideways RAM during power up and down.

D) A small coin cell providing about 4-5 years of battery life, see also 'C' above.

E) 128kB static RAM chip providing 8 banks of 16kB sideways RAM.

F) 128kB of flash ROM provided by Microchip SST39SF010.

The eight banks of 16kB sideways RAM are in sockets 0, 1, 4, 5, 8, 9, 12 and 13. The sockets containing flash ROM are 2, 3, 6, 7, 10, 11, 14 and 15.

When this RAM/ROM board is installed, the machine will usually have RAM and flash ROM in the following arrangement;

Socket number	RAM or flash ROM	Default contents (easily altered)
15 (F)	Flash (Note 1)	Meteors game
14 (E)	Flash (Note 1)	Chucky Egg game
13 (D)	RAM	Invaders game
12 (C)	RAM	
11 (B)	Flash	ROM utilities
10 (A)	Flash	BASIC 2
9	RAM	
8	RAM	Exmon 2.02 (*E to start)
7	Flash	B-Utility toolkit
6	Flash	Snapper game (*SNAPPER)
5	RAM	Disk Toolkit (*HELP ADT for commands)
4	RAM	BASIC Editor (*BE to start)
3	Flash	Turbo MMC code, 0.A3
2	Flash	Rocket Raid game (*RAID)
1	RAM	
0	RAM	

One way to remember whether a socket contains RAM or ROM is to visualise the socket number in binary. If bit 1 is a one then it's flash ROM, otherwise it's the static RAM chip. For example, socket number 10 (decimal) is 1010 in binary. Bit 1 is set so it is flash ROM (the rightmost bit is bit 0).

Although you can have dozens of ROM images stored on some kind of suitable filing system (ordinary floppies, MMC, Datacentre, GoSDC etc), the Beeb's operating system is only geared up to recognise a maximum of 16 sideways ROMs at any one time. They are numbered from 0 to 15. Sideways ROMs, as they are often called, always occupy addresses &8000 to &BFFF in the computer's memory map. That is exactly &4000, 16384 or 16kB of memory. Only one of the maximum of 16 can be active at any one time and the computer rapidly switches them on and off as required. & is the symbol used by BBC BASIC to indicate that a hexadecimal number follows.

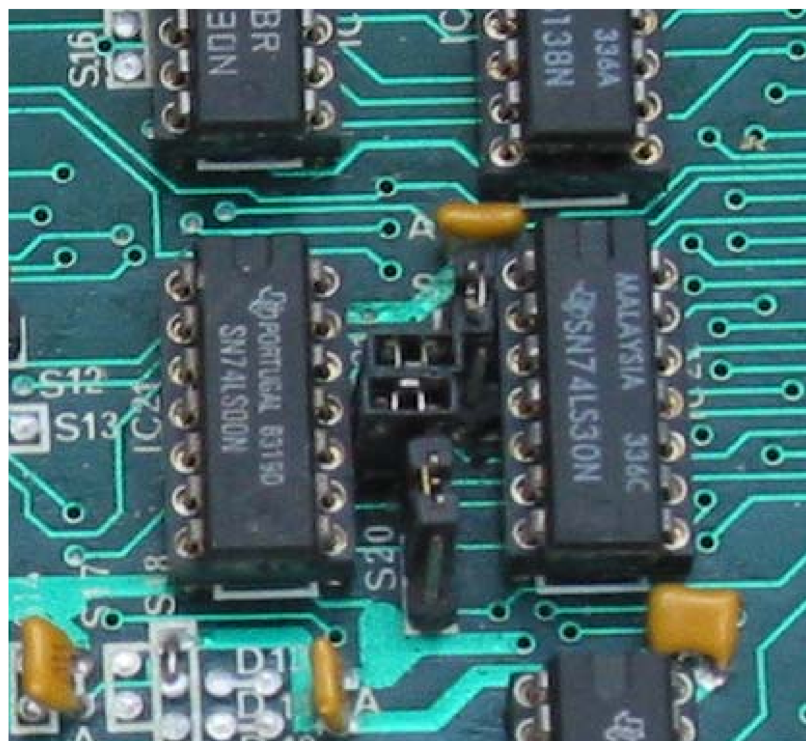
As the ROM socket numbers go from 0 through to 15 you will notice how the RAM and flash ROM alternate, two of one followed by two of the other. It has been designed this way because the higher numbered sockets can sometimes be thought of as having a higher priority than others. Most of the time this 'priority' issue will not matter but sometimes it might be convenient to not only have a particular ROM in a higher numbered socket than another, but also to make sure it is in RAM rather than ROM. This type of situation will be covered later.

Note 1: A link on the RAM/ROM board marked 'MB' can be used to disable the flash ROM in these two sockets and enable the rightmost two sockets on the motherboard instead.

The first step is to remove the computer's lid in the usual way (two screws marked 'fix' underneath and two more at the back). It will also be necessary to remove the two bolts holding the keyboard in place so that, as a minimum, the keyboard can be slid forward by an inch or so to give better access to the main board.

Any existing RAM/ROM expansion board must be removed before fitting this upgrade. It is suggested that the machine is reduced to a configuration where just the rightmost two ROM sockets of the machine are occupied by BASIC and your normal filing system ROM (DFS, ADFS, MMC etc. The two sockets to the left should be empty and the operating system (OS) ROM must be in IC51. On some ROM boards, the OS chip was taken out of IC51 and put back on the ROM board somewhere. You need to make sure that the OS is removed from any existing ROM board and placed back in IC51.

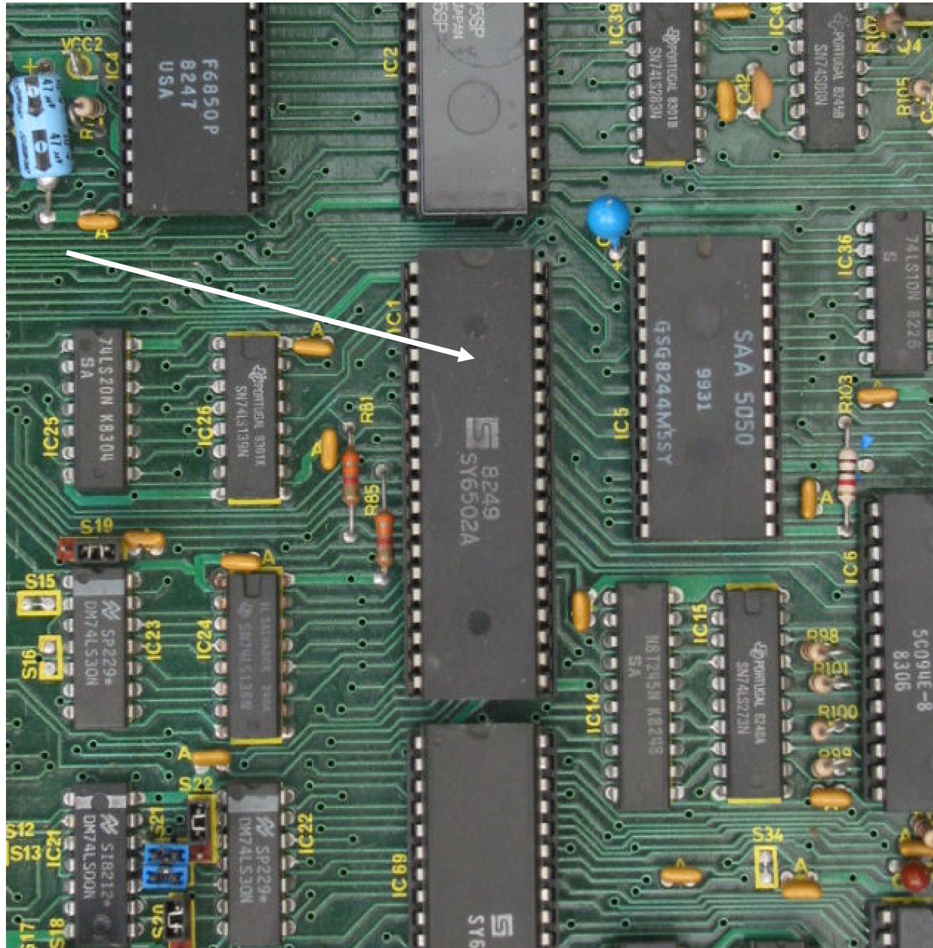
At this stage it is worth briefly switching on the machine to make sure that all is well. You will only have BASIC available and your filing system ROM. If the machine doesn't power up properly, check the link settings in the vicinity of S20 to the left of the 40-pin 6522 VIA. These were often altered when a ROM board was installed, so they need to be put back to their default settings when taking any such board out again. Remember too that the computer will only power up normally if the keyboard is connected. See below for link details.



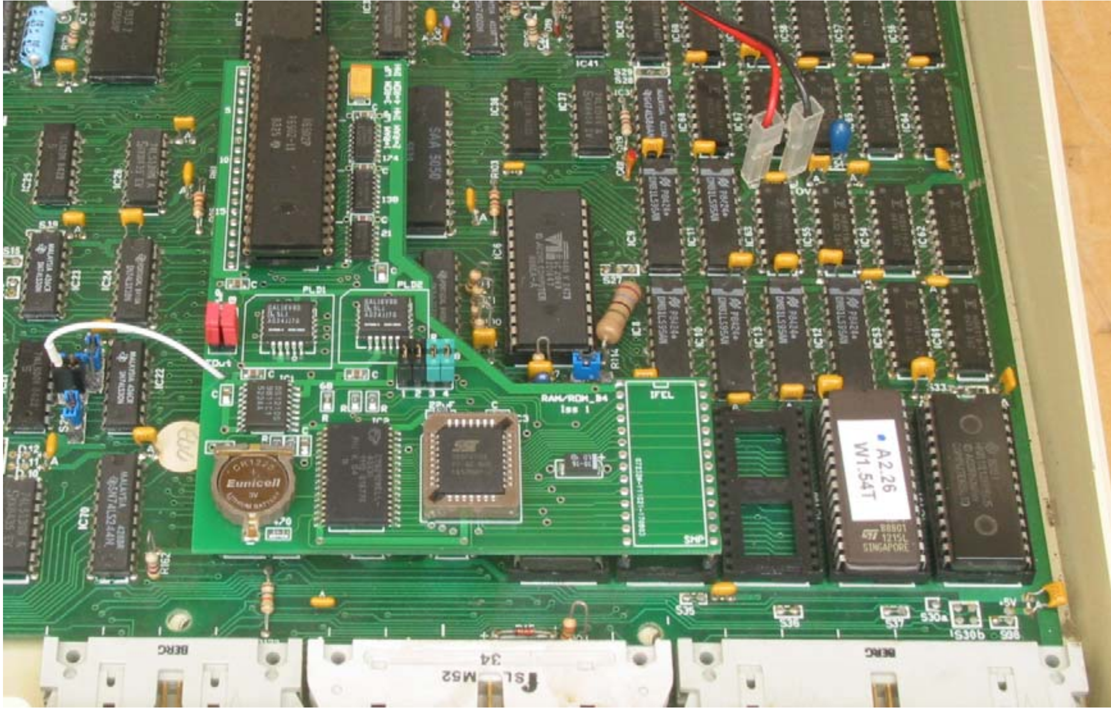
Default link settings: Links S22 and S20 are both in the North position and the two links that make up S21 both run East-West.

Switch off the machine again. The keyboard can readily be removed completely if preferred simply by unplugging the ribbon cable.

Carefully remove the main 6502 processor, IC1, from its socket.



The 6502 socket and the 28-pin socket immediately to the right of the OS ROM should now be empty. The RAM/ROM board occupies these two sockets and should be carefully pushed into them as shown below.

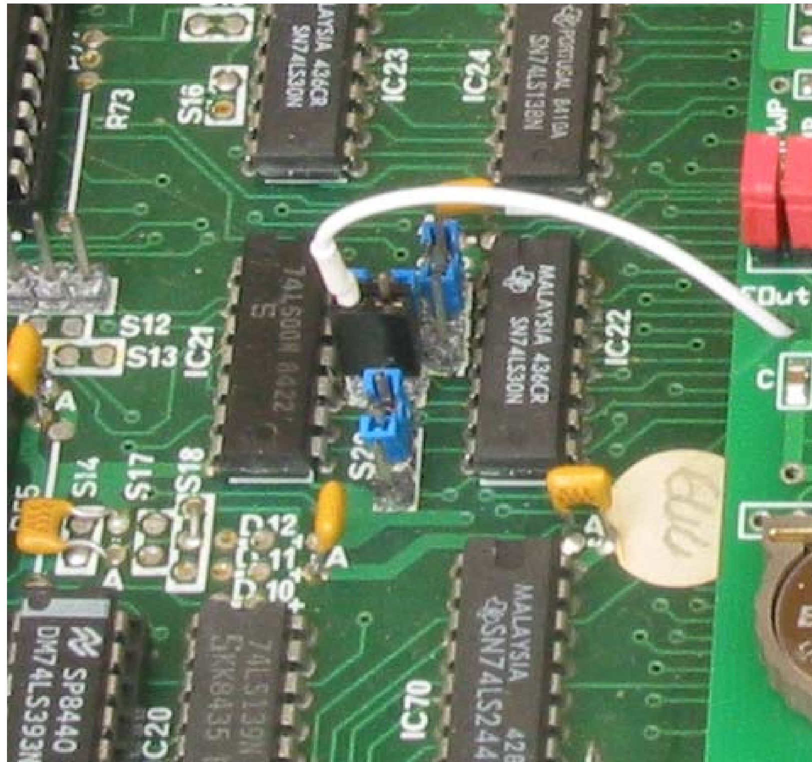


The 6502 itself goes into the socket on the RAM/ROM board. As you can see here, the rightmost motherboard socket contains BASIC and the one to its left a DFS filing system. The point is made again that most reasonably fast storage systems will be suitable, such as DFS, ADFS, MMC, Datacentre and so on.

The white flying lead goes to part of S21. Study these links again and you will see that S21 consists of two ‘jumpers’ both running left-right. The jumper nearest to the keyboard must be removed. The 2-way connector on the end of the white lead pushes onto the two pins of S21 so that the white wire is nearest to the power supply. See picture on next page.

Note: The wire may in fact be a colour other than white, and a single rather than 2-way female connector attached to the end. Obviously the colour is unimportant, but ensure that the wire is connected to the left hand pin of S21 as viewed from the front.

All the links on the RAM/ROM board should be in place. These are the two on the left marked as PWP and MB, and also the group of four a couple of inches to the right (numbered 1-4).



Flying lead connected to part of S21

At this stage and with the keyboard connected, the computer can be switched on. You can work with the computer's lid removed as there are no hazardous voltages on the board. Precautions need to be little more than working tidily so that, for example, the end of a reel of solder cannot unexpectedly spring into gaps in the power supply.

The machine should have powered up successfully in BASIC. In fact, both of the ROMs on the motherboard are disabled and cannot be accessed at present. BASIC has been stored in one of the eight banks of flash ROM, ROM socket number 10 to be precise. You can verify this for yourself by entering

```
PRINT ?&F4
```

Location &F4 contains the currently active ROM number which in this case is BASIC. If the BASIC ROM on the main board as shown on the previous page was being used you would get a result of 15.

As a brief diversion, try entering one of the following;

```
*RAID
*SNAPPER
```

You will need to press Break to exit from these games.

Some of the flash memory contains files in ROM Filing System format. Type

```
*ROM
```

followed by one of these;

```
CHAIN "CHUCKY"          (CH."CHUCKY" also works)
CHAIN "METEORS"
CHAIN "INVADER"
```

These games are slower to load than Rocket Raid and Snapper above. This is because of the slightly different way in which the data is loaded from the RAM/ROM board. It is not a fault of the game or the hardware. Perform a Ctrl-Break and then type `*ROM` again. Enter

```
CHAIN "CHECK"
```

What this does is perform a simple checksum on some of the RAM and ROM sockets, identified by a number in the range 0 to 15, to ensure that it is not corrupted. In fact, a simple checksum is not a hugely reliable test of data integrity, because clearly if two bytes are swapped around then the checksum remains the same yet the data has changed. But it flags up the majority of problems if they occur.

It would normally be the case that all banks would pass, especially the flash ROM banks which are virtually impossible to corrupt by accident. The obvious exception would be if the RAM/ROM board was supplied with data specific to the user's requirements or with no backup battery fitted and the 'CHECK' program had not been updated to reflect these changes.

Another command to try out would be

```
*ROMS
```

This is not the same as *ROM, this being used to select the ROM filing system. In fact, *ROMS is not normally recognised by the standard Model B at all and this command is actually being handled by the Advanced Disk Toolkit (ADT) in socket number 5. This toolkit ROM understands the *ROMS command and lists out the names of the various ROMs in each of the sockets. Notice that everything is listed out as being a "Rom" because ADT does not bother to check whether a socket contains RAM or ROM.

ADT is an example of several types of utility ROMs that were available many years ago. If you erase socket 5 or replace the data in it with something else, then the *ROMS command will no longer work. But of course with ROM images stored on MMC memory, Compact Flash etc, it is very quick and easy to load them when needed.

The MB link

If the installation has gone according to plan and you have followed the steps outlined on previous pages, you should have a fully working machine. The RAM/ROM board has completely replaced the all of the original ROM sockets and provides eight banks of sideways RAM in sockets numbers 0, 1, 4, 5, 8, 9, 12, 13 and eight banks of flash ROM in sockets 2, 3, 6, 7, 10, 11, 14 15. All sixteen banks are 16kB long and occupy the address space &8000 to &BFFF.

The two rightmost sockets on the motherboard are not used when the link marked MB is in place. They can be empty or have chips in them, it doesn’t matter either way because the machine will not attempt to access them.

The situation changes slightly when the MB link is removed. Socket numbers 14 and 15 are no longer mapped to the flash ROM chip and instead the two sockets on the motherboard become active (also as 14/15, 15 being the one on the right). The primary reason for including this feature is to enable the user to recover from the situation, albeit an avoidable and unlikely one, where all of the banks of sideways RAM and sideways ROM are empty. This would lead to the machine showing ‘Language?’ and you cannot proceed any further. Another scenario would be where one of the banks of flash memory has been loaded with a faulty ROM image that causes the machine to crash whenever that ROM is called and results in the machine being unusable. The way to deal with this is covered later. Here, we have merely removed the MB link (it stands for MotherBoard) in order to activate the Beeb’s two rightmost ROM sockets. You should have the original BASIC ROM in the one on the extreme right and a suitable filing system in the one immediately to its left.

When you power up the machine it should be in BASIC once again, but it is now using the original BASIC ROM on the main board and not the one in flash memory. You should also have the filing system in ROM socket 14 active. All your normal filing system operations, *CAT and so on, should be working as usual.

Note: The RAM/ROM board normally has the Turbo MMC code stored by default in socket number 3, one of the flash memory banks. This is suitable for the MMC interface board with a small 74HCT125 chip on it. If you have this type of system then you don’t actually need to remove the MB link. This because you have BASIC in flash socket 10 and also the MMC filing system code in socket 3. Thus, you are already in a position to copy the files described in the next section from the ROM filing system to the MMC. It may also be that the board was supplied in a bespoke form with a particular filing system, a certain version of a 1770 DFS for example, in socket 3.

Copying important programs to floppy, MMC etc

Enter `*ROM` as before and then `*CAT`, just or `*`.

Now enter `LOAD "COPY"` followed by `LIST`

This short program is a very simple way of copying some useful programs to your usual mass storage media. That can be an MMC system, ordinary DFS, ADFS and so on. As the program does not know what storage system you are using, you will have to add a line or

two to change the filing system so that the files are copied correctly. For example, you might add the line

```
*DISC
```

somewhere, and perhaps follow it with `*DRIVE 1` if that's where you want the files saved. An MMC user might wish to select the required 'disk' before saving the files, for instance

```
*CARD
*DIN 234
```

Obviously 234 would be replaced with a number of a suitable disk that had already been formatted. Once you have added the necessary lines, simply enter RUN and the required files will be copied over. It is very easy to see how the program works, with a list of files to be copied held in DATA statements at the end of the program.

SAVEROM and LOADROM etc

The BASIC program SAVEROM can be used to copy the contents of a specified ROM socket to the current filing system. Using RUN will prompt for the socket number and the data in that socket, exactly 16,384 bytes (hex 4000), will be saved to the filing system. The filename used is just "ROM" followed by the given number, so you may want to use `*RENAME` to change it to something more meaningful. Eg

```
*RENAME  ROM5 ADT200
```

LOADROM can be used to load data into any bank of sideways RAM but not the banks of flash ROM. It was written many years ago, long before the idea of using flash memory in a Beeb.

Remember also that various sideways ROMs may offer their own RAM loading commands (eg *SRLOAD in the Acorn 2.26 DFS). These should work perfectly providing the target socket number contains RAM and not the flash ROM.

*RUN SST, or */SST or *SST

The program SST is one of the files that should have been copied, by the above procedure, to your normal filing system such as DFS, Datacentre, MMC and so on. You can usually use any of the above versions of the command to run it, *SST being the shortest and most obvious choice. */ is short for *RUN.

*SST is used to load ROM images into one of the sixteen possible 'ROM sockets'. Unlike the simple LOADROM just mentioned, it can be used to program not only the sideways RAM but also the eight banks of flash ROM. Clearly, this modern RAM/ROM board has no spare physical sockets on it and you just have to specify a number in the range 0 to 15 to decide where a ROM image should be loaded. *SST on its own just prints out a quick summary of the syntax. At its simplest this would be

*SST <name of file> <socket number as a single hexadecimal digit>

For example;

```
*SST  DISCDOC  9
*SST  ARM113  C          (or *SST  ARM113  &C)
```

The second example loads the file ARM113 into socket number 12 decimal, 12 being C in hex. You can include the normal Acorn '&' prefix to indicate a hex number if you wish, as shown in the second example. You will get an error message if the file named in the *SST command cannot be found.

The named file cannot be longer than &4000 bytes, as this is the size of the sideways ROM address space (&8000 to &BFFF inclusive, exactly &4000 bytes). The next byte at &C000 is part of the operating system ROM in IC51. Also, the file cannot be shorter than 16 bytes, as this is unlikely to be a valid ROM image.

When you use the *SST command, the named file is in fact loaded into memory at address &3500 before being written into the sideways RAM or ROM. A special character can be included which causes the data already in the buffer to be used instead of loading data from the filing system. This is the circumflex ^ above the '6' key

```
*SST  ^  7
*SST  ^  D
```

Data invalid warning

Before the data beginning at address &3500 is written to the RAM/ROM board, a quick check is made to ensure that it appears to be a valid ROM image. An appropriate warning is given if this appears not to be the case. However, answering "Y" to the prompt will allow the programming to proceed.

Filing system or Language deletion warning

If you are about to load data into a particular socket and that socket contains the current language, you might want to think twice about overwriting it. If the Beeb cannot find any language when it is reset, it just displays a 'Language?' message from which you cannot really do very much.

Likewise, deleting your current filing system could also be inconvenient. As soon as you have deleted DFS, ADFS, MMC (or whatever) from the system, do you have something else in place to reload ROM images and run the SST program itself? You can see that deleting the only language in the system and/or the main filing system is usually unwise. It is not impossible to recover from this situation but it is generally best avoided.

For this reason, you will be warned if the current language or filing system is about to be overwritten. A 'Y' will allow the operation to proceed and any other character will abort the action.

This is a quick reminder of the socket numbers and whether they contain flash ROM or RAM.

Socket number	RAM or flash ROM	Default contents (easily altered)
15 (F)	Flash	Meteors game
14 (E)	Flash	Chucky Egg game
13 (D)	RAM	Invaders game
12 (C)	RAM	
11 (B)	Flash	ROM utilities
10 (A)	Flash	BASIC 2
9	RAM	
8	RAM	Exmon 2.02 (*E to start)
7	Flash	B-Utility toolkit
6	Flash	Snapper game (*SNAPPER)
5	RAM	Disk Toolkit (*HELP ADT for commands)
4	RAM	BASIC Editor (*BE to start)
3	Flash	Turbo MMC code, v 0.A3
2	Flash	Rocket Raid game (*RAID)
1	RAM	
0	RAM	

It is slightly quicker to program sideways RAM than flash ROM.

Checksum

After programming a RAM or ROM socket with the *SST command, a checksum result of the 16kB of data is printed out. This provides a reasonably reliable and rapid check of data integrity. You can run a checksum using the very simple CSUM program which was supplied on the ROM filing system. Providing it has not been deleted from socket 11 (&B), you can run it at any time with

```
*ROM
CHAIN “CSUM”
```

Then enter the required socket number. Various utility ROMs such as the Advanced ROM Manager also provide a checksum option.

Erasing the data in a RAM/ROM socket

Instead of loading data into a ROM socket you might wish to delete the contents of that socket. Example situations would be a ROM which appears to be clashing with something else or removing a temporary filing system ROM. You can erase the contents of a socket by using the # character in place of the filename in the *SST command. For example;

```
*SST  #  6
*SST  #  0
```

The warnings described earlier will also be given as needed (erasure of current language or filing system).

There is a subtle difference, largely unimportant to the user, in the erasure of RAM and ROM. When one of the banks of flash ROM is to be erased, special commands are sent to the chip to instruct it to erase four consecutive sectors, each 4kB in length. A check is then made to make sure that every byte of the entire 16kB (four 4kB blocks) has been correctly set to &FF (decimal 255). An erase pass/fail message is issued as appropriate.

When erasing one of the eight 16kB banks of RAM, every one of the 16,384 bytes is individually set to &FF. When this is complete, the code falls into the 'checksum' section used by the normal programming algorithm and should therefore show a result of &3FC000. This is simply 16,384 times 255 displayed in hexadecimal.

I and R options in the *SST command

When you use *SST to load a ROM image into one of the sixteen possible sockets, it is normally necessary to perform a Ctrl-Break so that the machine will become aware of its presence. If this is not done then you will probably find that the ROM does not identify itself in response to *HELP (most ROMs respond to *HELP, but not all), and any commands that the ROM is expected to understand will not work.

By using the 'I' (Initialise) option in the *SST command it has the effect of making the computer aware of the new ROM image. For instance

```
*SST  DISCDOC  7  I
```

loads the named file into socket 7 and then informs the computer that the new ROM has been installed.

There are some cases where the 'I' option will be ignored. If, for example, the warning was given that the data that you were about to program into a RAM/ROM socket was invalid, then the 'I' option will not have any effect. Similarly, the command

```
*SST  #  6  I
```

is contradictory. If you erase the contents of a socket, trying to initialise it is pointless. The socket's contents will be erased and the 'I' disregarded. Indeed, erasing valid data from a socket will always result in that socket being flagged as empty by *SST. Failing to do so would cause the machine to crash at the slightest provocation. (This is not a fault of the Beeb or the RAM/ROM board, but merely a consequence of how and when the Beeb activates the various ROMs and executes code within them).

There are some instances when the 'I' option cannot be entirely effective. One is when a particular ROM can only properly initialise itself when a hard reset occurs. An example would be a ROM that needs to claim workspace from main memory. Filing system ROMs often fall into this category. So it may be necessary to perform a Ctrl-Break after loading such ROMs. A very effective simulation of a power-on reset can be achieved with

`?&FE4E=127` (or `*FX 151,78,127` with a second processor in use)

Follow this by pressing Break.

There is another option for the *SST command which will perform a fairly comprehensive reset automatically after programming the given socket. This is achieved by using ‘R’ in the *SST command.

```
*SST  MYROM  &D  R             (The ‘&’ is optional)
```

The ‘R’ causes the machine to perform a reset after programming the file MYROM into socket 13 decimal (&D). Be aware that the contents of main memory will be lost by doing this (including any BASIC program that might be in memory at the time), but the contents of the sideways RAM or the flash ROM are not affected in any way by the reset action.

Sometimes the machine will automatically reset itself even if the ‘R’ switch has not been included. An example would be when you have approved the deletion or replacement of the current language. Usually, when the SST program has finished executing, control will be returned to the current language (normally BASIC). However, this isn’t going to work smoothly if the language has suddenly vanished, hence the need for a system reset.

On a technical note, the machine uses a 16 byte table from &2A1 to &2B0, to indicate whether a socket contains a valid ROM or not. It often happens that the operating system offers all the ROMs present in the machine the opportunity to take a particular course of action. It does this by activating them in turn and then calling code within them as a subroutine. The OS must obviously ignore sockets that do not appear to contain valid 6502 machine code, and it does this by maintaining the table referred to above, one byte per ROM. Strictly speaking, the table contains the ROM ‘type’. If the most significant bit of this byte is zero then the corresponding socket is largely ignored and the socket considered to be empty. For instance

```
?(&2A1+5) = 0
```

will prevent socket 5 from being called by the OS through its service entry point at address &8003. A Ctrl-Break will cause the OS to set the entry to the correct value again, depending on what it actually finds in socket 5.

Inconsistent checksum results

You will have gathered that the use of the *SST command to load data into the flash ROM or RAM will produce a checksum after programming. This is likely to show the same result each time that particular block of data is used but occasionally different results may be produced, with no obvious cause.

In fact, what is almost certainly happening is as follows. ROM ‘images’ (essentially an exact copy of the bytes of data in a ROM but stored as a file) are usually created by copying every single byte of those ROMs and writing them to a file. And so, because the ROM is almost certainly a 16kB type (eg 27C128), the file is precisely 16kB (hex 4000) bytes long too.

When you issue the *SST command (for instance `*SST MYDATA 7`), the file MYDATA is loaded into memory beginning at address &3500. Because it is probably &4000 bytes long, every address between &3500 and &74FF inclusive will be overwritten by this data.

Hence whatever was originally in this address space doesn't matter, as it will be overwritten. The checksum after programming should be the same each and every time.

However, suppose that the original ROM (and associated image file on the disk) is only 8kB long (eg a 2764 type device). What happens now is that the file loaded by the *SST command will only use addresses &3500 to &54FF inclusive, exactly &2000 bytes. Addresses &5500 to &74FF will not be affected in any way and their contents may well be uncertain. But *SST always programs the entire buffer (&3500 to &74FF) into the bank of sideways RAM/ROM and performs a checksum on this basis, and this is why the checksum result can vary.

The solution, if it is an issue, is to make sure that all ROM images are exactly &4000 bytes long. You can do this with commands such as;

```
*LOAD  FILE8  2000
*SAVE  FILE16  2000 + 4000
```

The ROM image called FILE8 is now saved as FILE16, should be functionally identical and will be exactly &4000 bytes long. An alternative format to the `*SAVE` command above would be

```
*SAVE  FILE16  2000  6000
```

The first *SAVE command saves a block of memory with the specified name, starting at address &2000 and extending for &4000 bytes. The last address saved would therefore be &5FFF and not &6000 as it might first appear. The second command does exactly the same, and saves the block of memory starting at address &2000 and continuing as far as address &6000-1. This is, when scrutinised, the required &4000 bytes.

Write protection options

The RAM/ROM board provides three main write protection mechanisms, all of which are designed to prevent unwanted changes to the RAM or ROM chip from being made. Exactly why such changes might occur is outlined below in the technical note. If you just want the details about which link does what, the next paragraph can be skipped.

Technical note: When the operating system has passed control over to one of the sixteen possible ‘sideways ROMs’, the code in that ROM (address range &8000 to &BFFF) has almost total control over what it does. This can include writing to the sideways ROM address space itself. Sometimes this is desirable, as the code can use some of the available space for temporary storage rather than claiming workspace from the computer’s main memory. Equally, however, these writes are sometimes used by software to determine if the code is being run in sideways RAM rather than the EPROM on which the software was perhaps originally supplied. If the writes succeed (in other words the data is alterable), the software will usually disable itself and refuse to run properly. This method of copy protection is easily overcome just by write protecting the sideways RAM.

There is a group of four links between the chips PLD2 and IC3. Links 1 and 2 relate to the RAM chip (Cypress CY62128 or similar). Links 3 and 4 control the options for the flash ROM (probably an SST39SF010). Links 1 and 2 are normally a different colour to link 3 and 4, as a reminder that links of one colour relate to the RAM and the other ones to the flash ROM. Electronically, the links do nothing more than short together the two pins on which they are placed.

Link 1, default position closed: Removing this physically breaks the electrical connection between the Write signal produced by the decoding logic and the Write Enable pin on the RAM chip. This is the ‘brute force’ approach to write protecting the static RAM chip. Clearly, it write protects the entire chip, and thus the RAM in sockets 0, 1, 4, 5, 8, 9, 12 and 13 all become write protected. Replacing the link permits changes to be made again. The link can be removed and replaced without turning the machine off.

Link 3, default position closed: Removing this physically breaks the electrical connection between the Write signal produced by the decoding logic and the Write Enable pin on the flash ROM chip. This is the ‘brute force’ approach to write protecting the flash ROM. Clearly, it write protects the entire chip, and thus the ROM in sockets 2, 3, 6, 7, 10, 11, 14 and 15 all become write protected. Replacing the link permits changes to be made again. The link can be removed and replaced without turning the machine off.

It will be apparent that the actions of links 1 and 3 are basically identical, other than one relates to the RAM chip and the other to the flash ROM.

Partial Write Protection

In the technical note above about why write protection can be required, the situation was outlined where allowing writes to occur to sideways RAM could in fact be useful. This would allow software executing within the sideways address space (&8000 to &BFFF) to use some of that RAM for data storage. The Copier utility provided with the Turbo MMC system is an example of this, as it uses some of the sideways RAM to hold a list of files to be copied between the different filing systems.

Clearly, the situation could arise where you want some of the sideways RAM to be writable whilst other banks of sideways RAM are write protected. This can be achieved by removing the PWP link at the left hand side. It stands for Partial Write Protection. What it does is write protect **ALL** the banks of flash ROM and four of the banks of sideways RAM. Four banks of sideways RAM are still writable. Specifically, regarding the RAM;

Socket number	Write protected by PWP?
0	Yes
1	No
4	Yes
5	No
8	Yes
9	No
12	Yes
13	No

With the PWP link removed you will not be able to load new data into RAM sockets 0 4, 8 and 12 (multiples of 4, as an aide memoire). It will also be impossible to make any alterations to the flash ROM sockets.

Remember too that removing link 1 overrides the setting of the PWP link. The Write signal from the logic is completely disconnected from the RAM chip when LK1 is removed, hence all eight banks of sideways RAM are write protected and the setting of the PWP link becomes irrelevant.

Self-write protection

Write protection links are all very well and entirely effective, but the self-write protection system provides a form of write protection that operates automatically on certain sockets. It applies to all the flash ROM sockets (2, 3, 6, 7, 10, 11, 14 and 15) and also to RAM sockets 0, 4, 8 and 12. These are the same sockets that are affected by the PWP link described above.

Previously, it was stated that certain software running as a sideways ROM would attempt to alter the data somewhere in the sideways ROM address space, &8000 to &BFFF. If the alteration was successful, the software would conclude that it was being run in sideways RAM instead of a ROM and therefore refuse to work. What the self-write protection mechanism does is prevent any kind of store instruction from writing to the current bank of sideways RAM, providing the store instruction occurred within the sideways ROM address space. For example

```
STA &9000
```

when executed from address &8159 (which is within the address space &8000 to &BFFF) will be blocked and the contents of memory location &9000 (also within &8000 to &BFFF) will not be altered. But the instructions `STA &70` or `STY &4000` would both work, because these destination addresses are not in the sideways ROM address space.

Now consider `STA &9000` but executed from address &2450 in main RAM (NOT the sideways ROM address space). This will not be blocked because the instruction doing the writing is not located in the sideways ROM address space. Remember, that the self-write feature only operates on all flash ROM sockets and RAM sockets 0, 4, 8 and 12.

Obviously, it is necessary to be able to load data into a bank of sideways RAM, even if that bank has the self-write protection (SWP) feature. That is where the automatic side of SWP comes in. The code to load data into sideways RAM is invariably running somewhere in main memory, and that will be well below &8000. The SWP feature does not prevent the data from being loaded because the machine code instruction doing the writing is not located in the sideways ROM address space (&8000 to &BFFF). As soon as the code in the new ROM is being executed, any attempt by that code to alter itself will fail due to the SWP operating. No fiddling about with links or switches and no special software commands either before or after loading the ROM image. It works very well.

If you're confused by the operation of SWP, it is probably best to just remember some golden rules. Don't attempt to load any kind of E00 filing system (covered in more detail elsewhere) in RAM sockets 0, 4, 8 or 12. Also the Copier program supplied with the Turbo MMC system requires sideways RAM that is not write protected, so this too must not be used in sockets 0, 4, 8 or 12. E00 filing systems and the Copier utility are examples of programs that should be used in sockets 1, 5, 9, or 13. This RAM is not normally write protected and indeed this can only happen if LK1 is removed. Removing the PWP (Partial Write Protection) link does not protect RAM banks 1, 5, 9 and 13.

Remember also that any sideways ROM code that has no specific need to be used in sideways RAM may be better off if written to one of the banks of flash ROM instead.

8kB ROM images

The standard Beeb's ROM sockets in the front right hand corner were normally intended to accept either a 16kB or 8kB EPROM. If an 8kB device was used then it appeared in the address range &8000 to &9FFF, precisely &2000 or 8192 bytes long. For fairly simple technical reasons, the data would be exactly duplicated in addresses &A000 to &BFFF. Thus address &8000 contained the same data as &A000, &8001 the same as &A001 etc.

Certain software originally supplied on an 8kB EPROM made use of this fact in order to prevent an 8kB ROM image from being run in a 16kB bank of sideways RAM. Various tricks were used, for example some subroutines could have &2000 added to them. In this way `JSR &9024` became `JSR &B024`. The point to note is that if the code was being run in the original 8kB ROM (often an EPROM) it would work correctly. But it would fail if loaded into sideways RAM because the data at address &B024 (in this example) would almost certainly be wrong.

The solution is similar to solving the 'problem' of inconsistent checksums. Specifically, you need to make sure the data at addresses &8000-&9FFF is exactly duplicated at address &A000 to &BFFF. You can very easily do this as follows;

```
*LOAD  FILE8  2000
*LOAD  FILE8  4000
*SAVE  FILE16  2000 + 4000
```

You should now find that FILE16, effectively FILE8 concatenated with itself, can be loaded into any bank of sideways ROM or flash ROM and have it working successfully.

Loading E00 Filing Systems

In brief, an E00 filing system must be used in sideways RAM and not one of the flash ROM sockets. Immediately, that initially narrows the options down to socket numbers 0, 1, 4, 5, 8, 9, 12 or 13. In fact, because sockets 0, 4, 8 and 12 have self-write protection, the choice is limited to sockets 1, 5, 9, or 13. Trying to use an E00 filing system in any other socket will almost certainly not work.

What is an E00 filing system?

When the original BBC Micro was first introduced, the machine would power up in BBC BASIC. Programs entered from the keyboard would be stored in memory beginning at address &E00 ('PAGE' as BASIC called it), all addresses lower than this being potentially used by the machine for some other purpose. The end address of your BASIC program ('TOP') would move up or down as lines of code were added or deleted, but clearly TOP would never be less than PAGE. Programs could be loaded and saved to cassette tape if you had a suitable cassette player.

When floppy disk upgrades first became available, they typically consisted of three main parts. The floppy drive itself, a handful of chips to be plugged into sockets on the Beeb's main board (usually based on the Intel 8271 disk controller) and an EPROM containing the filing system code. The problem was that the EPROM would claim some workspace for its own use (such as buffers for open data files and the directory catalogue of filenames) and this memory had to come from the computer's RAM. The effect was that the figure of &E00 increased to typically &1900. This 'loss' of about 2800 bytes became more significant in screen modes other than mode 7.

One method to overcome this was the concept of the E00 filing system. Such systems would use filing system code which was located in sideways RAM instead of an EPROM or ROM. The idea was simple—part of the sideways RAM address space beginning at &8000 would be used for the filing system code and the remainder (between the end of the code and extending to &BFFF) would then be free to use as general workspace. Thus, no RAM needed to be claimed from main memory and the value of PAGE could remain on &E00. Hence the general name of an 'E00 filing system'. It will also be apparent why you should not attempt to use an E00 filing system in one of the four banks of sideways RAM with the self-write protection feature (0, 4, 8, and 12). The code running in these sockets cannot write to the sideways ROM area so it is unable to use the RAM for workspace. That is why you need to use one of the four RAM sockets numbered 1, 5, 9 or 13. Equally obvious is that Lk 1 must be in place, otherwise the entire RAM chip (ie all eight banks of sideways RAM) is write protected.

An E00 filing system was usually something of a trade-off. There is less space for the filing system code because a certain amount must remain free for use as workspace. Typically, to create room, commands to format and verify disks were taken out and had to be run from a floppy. Similarly, the number of files permitted to be open at any one time might be reduced slightly. But generally, any disadvantages were outweighed by having PAGE remain on &E00.

The other obvious problem was that the E00 filing system ideally needed to be stored in battery backed sideways RAM. If this was not done then the E00 filing system was lost every time the computer was turned off. Therefore, a non-volatile filing system (FS) had to be present in the machine in order to load the E00 filing system, then the non-volatile FS disabled in some way. This is all unnecessarily inconvenient and the presence of several banks of battery backed RAM on this upgrade makes it easy to use an E00 FS and have it available for immediate use when the computer is turned on. E00 filing systems are available for ordinary DFS, ADFS (Advanced Disk Filing System) and also MMC systems.

Something that you will definitely need to avoid doing is using *LOAD to load to an address in the sideways ROM address space (&8000 to &BFFF). For example

```
*LOAD  DATA  8000
```

This will have the effect of loading the file over the filing system code which will, being an E00 filing system based in RAM, become corrupted and certainly need to be reloaded. The result of this type of *LOAD may even crash the machine.

In general, *LOAD loads a file into memory. If no load address is given in the command then the load address associated with the file is used by default, but it is possible to override this by including an address after the filename.

```
*LOAD  DATA
*LOAD  DATA  3200
```

The second of these commands will force the file to be loaded at address &3200 (hex 3200).

*SST load address, memory usage and *HISST

SST is a machine code program that will be loaded and run when you type *SST. The load address is &1200 and the SST program is just short enough not to encroach above &1900. &1900 is often the value of PAGE when a DFS is installed.

This means that any data in the memory range &1200 to just below &1900 will be overwritten by running SST. Normally this will not matter because not all of the workspace claimed by a filing system is in constant use. It may, for example, only be used if data files are open. Nevertheless, you should be aware of the memory occupied by SST and only run SST when you are certain that no data files are open and the space is not being used for some other purpose.

The load address of SST is particularly relevant when you are using an E00 DFS, something that was described on previous pages. If your BASIC program starts at &E00, it cannot be very large before the end of the program in memory (‘TOP’) extends up to &1200 and beyond. In this case, using *SST will wipe out part of your program and you will get the dreaded ‘Bad program’ message.

SST also uses certain zero page memory locations and these are not restored after use. The memory locations used will not cause any problems when SST is run from BASIC, but using it from any other language will almost certainly corrupt that language’s data in memory.

A good rule is to always save your work (BASIC program etc) before running SST unless you know with absolute certainty that no data loss will occur. Really, with modern data storage devices for the Beeb being quick and convenient, accidental data loss ought to be a thing of the past.

HISST

A slightly different version of SST is also supplied called HISST (High SST). It is nothing more than SST but assembled to load and run at address &2E00. This is obviously higher in memory than &1200, hence the name HISST. It may be the better option if the workspace between &E00 and PAGE (PAGE is sometimes called OSHWM or Operating System High Water Mark) is in use. It is run in exactly the same way as SST and uses the same syntax and the only difference is that the load and execution addresses are both &2E00 rather than &1200. For example;

```
*HISST  DATAROM  7
```

Renaming SST or HISST

SST and HISST are nothing more than files on the storage medium (floppy, MMC etc). SST was only chosen as being the first three characters of the flash ROM part number, SST39SF010, but clearly there is nothing to prevent you from altering the filename with the *RENAME command and using that new name from then on.

```
*RENAME  SST  FLASH
*FLASH  ANYDATA  7
```

‘39SF010 not found’ and ‘MB open’

These two errors are self-explanatory. When attempting to use the SST utility to program the flash ROM (normally socket numbers 2, 3, 6, 7, 10, 11, 14 and 15), the ROM is not responding to commands. Usually, this will be caused by incorrect link settings. The links that must be in place are PWP, LK3 and LK4.

Technical detail: LK3 breaks the Write Strobe signal from the logic to the flash ROM chip so writing to it is impossible. Valid data can still be read from the flash ROM. However, LK4 intercepts the Chip Enable signal and removing it puts the flash ROM into standby mode. Reading from the chip will not produce valid data and writing to it is not possible.

Socket numbers 14 and 15 are also mapped to flash ROM when the link marked MB is in position. When this link is removed, socket numbers 14/15 are no longer mapped to the flash ROM and instead the two rightmost sockets on the motherboard are activated. The contents of the flash ROM are not altered by removing MB but you cannot then program socket 14 or 15 with the SST utility. Trying to do so produces the ‘MB open’ message.

In summary, you should be able to program any of the eight flash ROM sockets providing links PWP, MB, LK3 and LK4 are all in place.

Recovering from a corrupted ROM that causes the machine to ‘hang’

You’ve loaded a new ROM image into sideways RAM or ROM, pressed Ctrl-Break and the machine hangs up with a flashing cursor. Turning the machine off and back on again doesn’t fix the problem. This section explains how to get back to a working computer again. It can be helpful to understand why and how a ROM can crash the computer in this way and that is what the following few paragraphs attempt to illustrate. If you are already familiar with this concept then you can skip forward to the underline.

When the computer is turned on, one of its tasks is to figure out which of the possible sixteen sideways ROM sockets contain valid code. ‘Valid code’ here means machine code instructions and data which conform to a certain protocol that the operating system (OS) recognises. The check is not particularly robust but is sufficient to weed out sideways RAM containing random power-on data or indeed empty sockets. Specifically, it looks for the copyright string (C) together with a zero byte. The start of this string is not in a fixed position, although it’s typically somewhere in the first thirty or so bytes, and the OS uses another byte of the sideways ROM address space to know where to look for it. If the string is not found then the OS assumes that the current ROM number does not contain valid code and flags it as empty. It does this by writing a zero to a table in memory beginning at address &2A1. Address &2A1+n will contain zero, where n is the ROM number, if a socket is deemed by the OS to be empty.

On the other hand, successfully identifying the (C) string will cause the OS to write the ‘ROM type’ to the table. The ROM type byte is always in a fixed position in the sideways ROM space (&8006). If bit 7 of this byte is set then it indicates that the ROM has a ‘service

entry' point, this always being located at address &8003. The BASIC interpreter does not have a service entry point but all user ROMs should have one.

Occasionally, the OS will offer the sideways ROMs the chance to take a particular course of action. Just two examples are identifying themselves when *HELP is entered or acting upon an unknown OSByte call. It will do this by selecting the chosen socket and then performing a subroutine call to address &8003 with JSR &8003. Remember that &8003 is the service entry point for the ROM and the instruction there is invariably an unconditional JMP (op code &4C). For example, the data at the three addresses &8003, &8004 and &8005 might be &4C, &28 and &81. This corresponds to the instruction JMP &8128, so execution continues from that address. The contents of the processor accumulator indicate why the ROM was called and at this point the author of the sideways ROM/RAM has almost complete control over what the machine does. Using the above examples, the accumulator contains 9 when *HELP has been issued and contains 7 if an OSByte call has been made which the OS doesn't recognise.

In this way the programmer writes suitable code to handle the expected service calls. In the example of reason code 9, it means that *HELP has been entered and the ROM's title is often printed out. In this case the ROM code exits with an RTS instruction with not only the X and Y registers unchanged, but also the accumulator. (Clearly, the registers can be used during the service call but they may need to be preserved on exit.) If a service call exits the ROM with A set to zero, the OS will not offer the same service call to other ROMs lower in the list. The ROM is said to have 'claimed the call'. So in the case of *HELP, a ROM should not claim service call 9 because other ROMs will not get the chance to reveal their presence.

The example of service call 7 is slightly different. It means that an OSByte command has occurred in the system and it is not one that the OS recognises. It therefore allows the various sideways ROMs the chance to deal with it instead. So, the service entry of a particular ROM would initially check for the accumulator containing 7 ('unknown OSByte') and then establish exactly what the OSByte number was and any associated parameters. If the ROM can deal with the OSByte command, it takes the required course of action, and then hands control back to the OS ***with A set to zero***. This informs the OS that the service call has been claimed and it should not be offered to the other ROMs.

On the other hand, if the ROM did not recognise the OSByte call, it would conclude that another ROM must be given the chance to claim it instead. It will therefore exit with A still containing 7 and the OS offers the call to the next ROM. That, in general, is how service calls work on the Beeb's sideways ROM system.

We have digressed slightly because the problem to be addressed here is where a ROM causes the machine to hang. In the above example, the service entry point was assumed to contain JMP &8128. Technically it could JMP to any 16-bit address but it would normally be within the sideways ROM address space &8000 to &BFFF. Suppose, however, the code at address &8128 is JMP &8128. In other words;

```
8128 4C  JMP 8128
8129 28
812A 81
```

The processor now falls into an endless loop. Obviously this is rather unlikely to happen by accident, but it is apparent that any serious bug in the service entry code is going to cause the processor to crash. Recall that the sixteen byte table beginning at &2A1 is used by the OS to indicate whether a ROM should be called via its service entry at &8003. In fact, if bit 7 of the table entry is set then the ROM will potentially be offered service calls and otherwise the socket will be skipped. (That is how the OS can avoid activating an empty socket and then trying to call code within it. Empty sockets have zero in the table.) The ROM may well have passed the basic test of having a valid copyright string but the OS cannot examine the service entry code to check for major programming errors.

The OS doesn't just check for the presence of ROMs when the computer is turned on, it also does it when Ctrl-Break is pressed. ROMs are offered service calls surprisingly often and the previous examples of *HELP and an unknown OSByte command are just two of many possibilities. You should now be able to see why a corrupt or flawed ROM image loaded into sideways RAM or flash ROM will cause the computer to hang—the processor may have just got stuck in a loop somewhere or fallen into random data, executing undefined instructions and so on. Pressing Break will not recover from this because the processor just goes through its reset ritual again, eventually crashing for the same reason and probably in the same place. Nor will turning the computer off/on if the problem ROM image is in flash memory or battery backed RAM.

ROM Priority

The above illustration of why corrupt or faulty ROM data can prevent the computer from properly powering up also introduced the idea of ROM priority. When service calls are being offered to the various ROMs in the machine by the OS, it does so by starting with ROM number fifteen and working down to zero. Empty sockets are of course skipped as previously discussed.

Generally, the ROM priority is of no consequence but occasionally it can cause problems. An example is the multitude of 'Utility' ROMs which became available, and these offered programmers some handy extra commands such as *SHIFT to move blocks of memory about and 6502 machine code disassemblers. These relied on the fact that unknown 'star commands' were offered to the sideways ROMs as service call number 4 (ie A set to 4), complete with a pointer to the command in memory. The ROMs would then try to match this string with one of several commands that they recognised, taking action if they managed to do so. The problem arose when two or more ROMs were fitted to the machine which all acted upon exactly the same command,. The ROM in the higher numbered socket would always grab it first, claim the call and any other ROMs in the system which also recognised it would be denied the chance to take action.

Sometimes ROM authors got around this by allowing the various commands that the ROM understood to be prefixed with a certain character, 'Z' for example. Hence *EDITMEM might also be entered as *ZEDITMEM, the rationale being that no other ROM would then claim it. There were also certain 'Manager' ROMs that were intended to selectively disable other ROMs in the system. These usually worked by writing a zero into their table entry (beginning at &2A1) in order to fool the OS into thinking that the socket was empty.

It is customary for ROMs to allow the use of upper or lower case letters and also for commands to be abbreviated by the use of a period. For example;

```
*MEMORY  3456         (Example command in full)
*mem.  3456
```

However, this does rely on the programmer putting the necessary code in his ROM to permit the use of letters of either case and also a period. It doesn't just happen on its own.

With the RAM/ROM board, the question of ROM priority should never really be a problem as such. This is because ROM images can so quickly be deleted and reloaded into another socket, or deleted from the system completely until required at a later date.

Default language

Related to the concept of ROM priority is the language that the machine will normally use after switching on or after a hard reset (ie Ctrl-Break). As a general rule, if two or more language ROMs are present in the computer, it will try to use the language in the highest numbered socket. Other languages can be selected if required by their appropriate star command, such as *BASIC, *WORDWISE or *SHEET.

So, if Wordwise was loaded into socket 14 and BASIC into socket 10, Wordwise will be the normal start up language. *BASIC can be used if required to invoke the BASIC language. However, swap Wordwise and BASIC around and BASIC becomes the default language with Wordwise available if you need it with *WORDWISE (or some suitable abbreviation).

A point of detail in all this is that the 'ROM type' byte, the seventh byte of the ROM located at address &8006 in the memory map, can be used to decide if a language is to be considered for selection when a hard reset is performed. Specifically, this will occur if bit 6 of the ROM type is set. If bit 6 is clear then you will need to enter a suitable star command to start that language even if it is in the highest numbered socket. Using the above example of Wordwise in socket 14 and BASIC in socket 10, providing you alter the ROM type byte of Wordwise to be &82 instead of &C2, BASIC will still become the default language with Wordwise being invoked with *WORDWISE as normal. Here's how you might do this using an example file called WWISE;

```
*LOAD  WWISE  2000
?&2006 = &BF AND ?&2006     (Could also just use ?&2006=&82 )
*SAVE  WWISE  2000 + 4000
```

Another way of starting up a language is to use

```
*FX 142 , <n>
```

where <n> is the socket number in decimal of the required socket. Hex numbers are not allowed in FX commands on the Model B.

Recovering from corrupt ROM images cont'd…

At this point you should know why corrupt or faulty ROM images can crash the computer. There now follows a sequence of events which should always get the machine back up and running again. Once you understand what's being done and why, you will almost certainly be able to see various shortcuts to perhaps reduce the number of steps required. For instance, if everything was fine until a ROM image was loaded into socket 5, you can be fairly sure that the contents of the RAM chip are causing the problem. In this case, erasing any of the banks of flash ROM should not be necessary.

The two key points to bear in mind are that removing Link 2 (second from the left in the group of four) will completely disable the RAM chip. Therefore the banks of sideways RAM numbered 0, 1, 4, 5, 8, 9, 12 and 13 will not be recognised by the computer and any data in them is ignored. However, removing LK2 does not actually cause the data in the RAM to be lost, merely unreadable until LK2 is replaced again.

Similar comments apply to LK4, the only difference being that LK4 relates to the flash ROM chip (socket numbers 2, 3, 6, 7, 10, 11, 14 and 15). Bear in mind also the MB link. When this is removed, the two rightmost sockets on the main board are usable as socket numbers 14 and 15.

The steps to take that should always work are;

Remove LK2, LK4 and the MB link. Make sure LK1 and LK3 are in place and also PWP. Ensure that BASIC and a filing system (FS) ROM are present in the rightmost two sockets on the main board. Switch on and you should have BASIC and your FS working properly, be that MMC, DFS, Datacentre and so on. The RAM chip and flash ROM are both completely disabled (removal of LK2 and LK4). Insert the media (electronically with some kind of star command, or maybe a physical floppy) so that the SST file is available. This is the one used for writing to sideways RAM and the 39SF010 flash ROM chip.

Replace LK2 and LK4 so as to enable the RAM chip and flash ROM again. You will need to do this with the computer turned on and, having done so, do not press Ctrl-Break. This will almost certainly cause a crash once more.

Now use the *SST command on every socket in turn from 0 to 13. This is done using commands such as

```
*SST  #  0
*SST  #  1
:
:
*SST  #  D   R
```

Notice that for the last of these commands the 'R' option was added. This will cause the machine to perform a reset after erasing the contents of sideways RAM socket 13 (decimal) and the machine should now be up and running normally. You still have only BASIC and the FS in sockets 14/15 because all the eight banks of sideways RAM have been cleared along with six flash ROM sockets 2, 3, 6, 7, 10 and 11. Remember, flash ROM sockets 14 and 15 are not available with link MB removed.

There is one more possibility that needs to be considered. Perhaps the rogue ROM image was in socket numbers 14 or 15 in the flash ROM. By removing the MB link, these two banks of flash ROM are disabled and the two sockets on the main board enabled instead. If you replace the MB link and the computer crashes again, it implies that the problem was with socket 14 or 15 all along. If so, how do you erase flash ROM banks 14 and 15?

One method would be to use the SAVEROM program to save the contents of sockets 14 and 15 (ie BASIC and the FS) as ROM images to disk (MMC or whatever). This must be done with the MB link removed. Then load those two ROM images back into sideways RAM. Any two banks will do, chosen from numbers 0, 1, 4, 5, 8, 9, 12 or 13. For example;

```
*SST  ROM14  5
*SST  ROM15  8
```

Replace link MB but remove LK4 to disable the flash ROM, then do a Ctrl-Break. The machine should power up in BASIC and with your FS available, both now running in sideways RAM. Replace LK4 (to enable the flash ROM chip) and enter

```
*SST  #  E
*SST  #  F
```

These commands erase sockets 14 and 15 and at this point you can almost certainly perform a Ctrl-Break or ‘genuine’ power on reset. You can now reprogram the various banks as needed.

It will be apparent why the MB link was included in the RAM/ROM board design. If you somehow end up with all the banks of sideways RAM and flash ROM corrupted or deleted, you would be completely stuck unless there were a means of activating two motherboard ROM sockets so that BASIC and an FS ROM could be used to boot the machine and reload data.

Whilst the above process may seem very long-winded, it is a situation that is easy to avoid if you follow some basic rules. Keep the flash chip for ROM images that are known to work properly, especially BASIC and the main FS. The flash ROM is virtually impossible to corrupt by accident. Put experimental ROM images or untested ones into sideways RAM. If there is a problem, you really only have to remove LK2, do a Ctrl-Break, put LK2 back in place and erase the socket causing problems. You probably won’t have to erase all eight banks of RAM.

Remember too that the *SST program will warn you if you are about to erase or overwrite the current language (normally BASIC), or the current filing system. It is difficult to do these things by accident.

The best arrangement of ROMs

As a convenient reminder, this is the configuration of RAM and ROM.

Socket number	RAM or flash ROM	Default contents (easily altered)
15 (F)	Flash (Note 1)	Meteors game
14 (E)	Flash (Note 1)	Chucky Egg game
13 (D)	RAM	Invaders game
12 (C)	RAM	
11 (B)	Flash	ROM utilities
10 (A)	Flash	BASIC 2
9	RAM	
8	RAM	Exmon 2.02 (*E to start)
7	Flash	B-Utility toolkit
6	Flash	Snapper game (*SNAPPER)
5	RAM	Disk Toolkit (*HELP ADT for commands)
4	RAM	BASIC Editor (*BE to start)
3	Flash	Turbo MMC code, 0.A3
2	Flash	Rocket Raid game (*RAID)
1	RAM	
0	RAM	

Note 1: With link MB removed, flash ROM is not available in these sockets and the two rightmost sockets on the main board are enabled instead.

Previous material has covered the idea of ROM priority and default language selection. The reason for designing the board with alternating RAM and ROM (two of one, two of the other and so on) was so that it is fairly easy to choose not only whether a ROM image is loaded in RAM or ROM, but also the priority. Higher numbered sockets are offered service calls before lower numbered ones.

The SST program will load data into RAM slightly quicker than the flash ROM and the RAM chip theoretically has an unlimited number of write cycles. The flash ROM may have a quoted 100,000 write cycle lifetime, but putting that in perspective, you could write to it dozens of times every day for many years. The decision to load data into RAM or ROM should not usually be based solely on the loading time or degradation of the ICs.

E00 filing systems MUST go into sideways RAM 1, 5, 9, or 13. Flash ROM definitely will not work and RAM banks 0, 4, 8 and 12 have the self-write protection. The Copier program, supplied with the MMC storage system can also only be used in 1, 5, 9, or 13.

It is best to put only tried and tested ROM images into flash ROM. Accidental corruption is almost impossible. The procedure for recovering from a corrupt ROM image causing the machine to permanently hang up has been described, and life is somewhat simpler if you know with 100% certainty that it must be in the RAM chip. Another setup which may be worth using is to keep the original BASIC ROM and a filing system ROM on the main board and have the MB link open. These two ROMs, effectively numbers 14 and 15, cannot be accidentally deleted and you still have six banks of flash ROM available and all eight banks of sideways RAM.

Disabling ROMs

1) Manager ROMs.

Even though a ROM socket in the Model B may be physically occupied (or contain an electronic ROM image), there are some tricks that can be used to fool the machine into thinking that it is empty. It is clear how this could be useful if two different DFS-type filing system EPROMs were installed but we wish to instruct the machine to disregard one of them. Essentially, the computer would be switched on at which point it thinks, quite correctly, that two filing systems are present. A command would then be issued to tell the machine to ignore a particular socket number and the machine reset with a Ctrl-Break. At this stage the computer should ignore the specified socket, leaving only the single filing system that you want to use. Multiple sockets could be disabled in this way.

For technical reasons, 'Manager ROMs' like this are often at their most effective when installed in the highest priority socket, number 15. There are a few memory locations that are not used by the Beeb's OS 1.20 and software writers sometimes made unofficial use of them. This can lead to clashes with other programs which are also trying to use the same memory locations in a similar - or indeed a completely different - way. It is generally unwise to have two or more ROMs installed which offer any kind of ROM management facility. The Advanced ROM Manager, Advanced Disk Toolkit and various other utility ROMs provided the option to selectively disable ROMs.

It would be possible to write a very effective ROM Manager for use in a bank of sideways RAM. The ROM would need to respond to commands like *ON and *OFF followed by the ROM numbers to be affected, and perhaps use a couple of bytes in the sideways RAM itself to acts as flags to decide which ROMs were to be disabled. Obvious programming safeguards would be to make sure that the Manager never disabled itself and to make sure that there was always at least one language ROM available.

2) Another software trick

This method of disabling certain ROMs can work well. It relies on the way in which the computer can be made to execute some machine code every time Break or Ctrl-Break is used.

When reset (which is essentially what the Break key does), the machine will test the contents of memory location &287. If it is 76 (hex 4C), then the computer will treat this as a JMP instruction with absolute addressing. The address from which program execution will continue is given by the contents of the next two memory locations, &288 (low byte of address) and &289 (high byte of address).

Technically, what happens is the computer examines the contents of &287. If it is anything other than 76 then nothing happens. But if it is indeed 76, then the code will be executed. It will either be a JSR &287 or JMP &287 but from the user's point of view the effect is the same. A piece of machine code will be run and you only have to put an RTS at the end of it to return control to the operating system. What we can do is ensure that the code turns off one or more ROMs of our choosing. This can be done by writing a zero into the appropriate place of the 16 byte table beginning at &2A1 and ending at &2B0.

Here is the assembly language program needed to do this.

```
10 code = &A80 : REM Code goes at this address
20 P%=code
30 [
40 LDA #0
50 STA &2A1+7 \Change to the ROM number you want to disable
70 RTS
80 ]
90 !&287=(!&287 AND &FF000000) + code*256 + &4C
```

Line 90 sets location &287 to a value of &4C, &288 and &289 to the correct jump address and &28A is unchanged. If you wanted to turn off more than one ROM then you just add another line similar to 50. For example, to turn off ROM 3 as well, simply add;

```
60 STA &2A1+3
```

Using this sort of technique you could turn off one particular filing system leaving only another active, and thus prevent PAGE being raised more than necessary.

Some technical notes;
Page &A (addresses &A00 to &AFF) is normally reserved for features such as the RS423 serial interface and cassette usage. Providing these are not used then generally you can put small code snippets in there. However, it is always possible that some games might make temporary use of this area of memory as well and you will run into problems if memory location &287 contains &4C and the code jumps into garbage. Of course, it's nothing that turning off the machine won't fix.

If you want to stop your code from being run, then just set location &287 to anything other than 76. Zero is the obvious choice (?&287=0)

This simple example code is not compatible with the 2nd processor. It is possible make it Tube compatible by using the correct OSBYTE calls instead of writing to &287-&289 directly. Refer to the Advanced User Guide and OSBYTE 249.

Also, many games ensure that when you press the Break key, virtually all of the machine's memory is wiped. This will almost certainly include the small block of memory where you placed your 'Reset code', so you will have to enter it again. You can't really do much about this. Likewise, the use of the 'R' option in the *SST command will also clear memory after programming the chip.

You can cause the machine to clear memory (but not sideways RAM) when Break is pressed by entering *FX 200,3 Be warned, it really does work so make sure you don't have any valuable data in memory that hasn't been saved first. *FX 200 turns off this memory clearing effect.

Games using sideways RAM

Quick summary: If a game tries to use sideways RAM and yet it crashes when doing so, try removing the PWP link.

More detail: Sometimes games make use of sideways RAM as temporary storage. This might be to store data for screen layouts or indeed any other purpose that the game author wanted, and this in turn can reduce or eliminate accesses to any kind of mass storage system (floppy, MMC etc).

What normally happens is that early on some code will be loaded into to main memory which will then scan the various ROM sockets looking for writable sideways RAM. If found, it may just go ahead and use it anyway. On the other hand, the game may ask the user's permission before doing so. Whether or not any problems will occur as a result depends on how the sideways RAM is used.

If blocks of data are simply written into sideways RAM and retrieved later, there should not be any issues. If executable code is stored in sideways RAM, which is then copied down into main memory before being executed, this should work properly too. A problem may well occur if executable code is stored in sideways RAM and executed from the sideways ROM space. In other words, it is NOT copied down to main RAM first.

The potential problem arises because of the self-write protection on RAM banks 0, 4, 8 and 12. Remember, self-write protection prevents code running in these banks of RAM from altering itself in any way and it does this by blocking writes to any address within the sideways ROM space, &8000 to &BFFF. Crucially, the write is blocked if, and only if, the instruction doing the writing was read from the sideways ROM address space.

It should now be a bit clearer what may happen. The code loaded into main memory will scan for sideways RAM and perhaps find it in socket 0. The software may reasonably conclude that this bank of sideways RAM can be freely written at any time. Providing it is just used for temporary storage of data for the game, all should be well. But if the following are all true;

1) Executable code is stored there.
2) The code is executed from the sideways ROM address space (&8000-&BFFF)
3) That code tries to write to any address in the range &8000-&BFFF

This will not work properly because the software would be unaware of the self-write protection on socket 0.

That is also why removing the PWP (Partial Write Protection) link may cure the problem. With PWP removed, sockets 0, 4, 8, and 12 are all write protected. Therefore if and when a scan is performed to look for sideways RAM, these same sockets will appear to contain unalterable ROM and be ignored. However, sockets 1, 5, 9, and 13 do still contain writable sideways RAM and these sockets do not have the self-write protection.

Battery discharge rate

Near to the negative terminal of the batter holder is a surface mount resistor and is effectively in series with the battery. Typically, this resistor has a value of 470 ohms although its precise value is not important. The DS1210 battery back up chip contains all the circuitry necessary to safeguard against the backup battery being charged when the computer is turned on. Maxim IC confirmed that no external components are required in this respect.

There are two reasons for including it. One is that it provides an easy method of judging the current drawn by the static RAM when the computer is off. A high impedance voltmeter on a millivolt range can be used to measure the voltage across this resistor. Typically it is around 0.35mV. From Ohm's law and using the example of a 470 Ohm resistor, the current drawn is under 1 microamp.

The resistor may be marked 4700 and could easily be mistaken for 4.7kOhms. In fact, the last digit would be a multiplier and the value is therefore translated as being 470 x 10^0, ie 470.

The second reason is as follows. When the computer is plugged into the mains and even when switched off, the computer's digital ground will be connected to the mains earth. If a mains operated soldering iron (earthed tip) were touched on the positive battery terminal, you would short out the battery. A small coin cell is unlikely to provide enough current to do any damage but certain types of cell might result in a damaged track on the PCB. The use of a series resistor connected to the ***negative*** terminal will limit the current to a safe value.

Battery life

Batteries can deliver a certain amount of *charge* before becoming exhausted, charge being measured in Coulombs and is the product of a steady current in Amps flowing for a certain number of seconds. It may help, even if it is a slight oversimplification, to think of any given battery as being able to deliver a current of 'I' Amps for a time 't' seconds, where the product of 'I' and 't' remains constant. For example, an alkaline AA cell rated at 2000mAh might theoretically deliver;

1A for 7200 seconds (2hrs)
2A for 3600 seconds (1hr)
100mA for 72000 seconds (20hrs)
2 microamps for $3.6x10^9$ seconds (41667 hours or 114 years)

However, the above figures become inaccurate at extremes of current and batteries do not deliver a constant current before instantaneously failing. The terminal voltage falls during discharge and would eventually become too low to work the equipment properly. An AA alkaline cell wouldn't be able to supply 20A even if short circuited, so the corresponding time would be meaningless. The apparent capacity of a battery tends to be higher at the lower currents but at very low currents (and therefore long discharge times), the cell will degrade over time and the skew the result
.
The RAM/ROM board draws less than a microamp from the backup battery when the machine is off. For a CR1220 battery the capacity is around 40mAh and the life would be in

the region of four years. A CR1225 will also fit in the on-board holder and might be expected to last about 25% longer.

It is possible to make up an alternative battery pack from suitable cells together with appropriate holder. Such battery packs can often be fitted in a hollow just to the left of the keyboard and in front of the power supply (PSU)**.** ***The two points to note are that the coin cell in the on-board battery holder must be removed when attaching an extra battery source in this way, and also no part of the alternative battery or the holder's terminals must be allowed to touch the metal case of the Beeb's PSU.***

Three obvious types of battery pack for this RAM/ROM board might be;

1) Two AAA alkaline cells. With a capacity of typically 1000mAh, the theoretical life would be 114 years, and degradation over time is likely to be the limiting factor. Check such cells periodically for any signs of leakage and replace them before it can become a corrosion problem.
2) A single AA lithium thionyl chloride cell, typical voltage 3.6V. Theoretical life over 200 years and cell degradation over time is likely to be the limiting factor. Lithium cells such as these are well suited to long term, low drain applications. Nevertheless, check such cells periodically for any signs of leakage and replace them before it can become a corrosion problem.
3) A CR2032 coin cell, typical voltage when new 3.2V. Capacity around 200mAh giving a theoretical life of around 25 years and cell degradation over time is likely to be the limiting factor. Lithium cells such as these are well suited to long term, low drain applications. Nevertheless, check such cells periodically for any signs of leakage and replace before it can become a corrosion problem. This is an inexpensive and compact solution and is perhaps the solution of choice if you want something that lasts longer then the default battery on the upgrade.

In all cases, simply take the +ve and -ve leads from the battery holder and solder to the tabs on the surface mount battery holder on the RAM/ROM board. The +ve terminal is clearly marked and is nearest to the back of the computer. ***Remember to remove the coin cell from the surface mount holder.***

A small sticker somewhere to indicate when the battery was last installed is no bad thing.

Example installation of a 3.6V lithium thionyl chloride battery in a suitable AA size holder.

Link settings summary

The group of four links numbered 1 to 4 will be covered first. The general point to remember about all these four links is that they physically break an electrical signal from the logic to the RAM and flash ROM chips. A pull-up resistor on the board draws the respective pin on the chip to a logic high and prevents it from ‘floating’ (an input that is not connected to anything). A link that is in position simply electrically joins the two pins on which it is placed.

1) Removing this breaks the active low Write signal to the RAM chip and therefore write protects all eight banks of sideways RAM (0, 1, 4, 5, 8, 9, 12 and 13). Normal position would be closed.
2) Removing this breaks the active low Chip Enable signal to the RAM chip and therefore disables all eight banks of sideways RAM (0, 1, 4, 5, 8, 9, 12 and 13). It is not possible to write to the RAM chip or read valid data from it with LK2 removed. Data is not lost when the link is open. Removing this link might be done temporarily when recovering from a corrupt or faulty ROM image but the normal position would be closed.
3) Removing this breaks the active low Write signal to the flash ROM chip and therefore write protects all eight banks of ROM (2, 3, 6, 7, 10, 11, 14 and 15). Normal position would be closed. It is almost impossible to alter the contents of the flash ROM by accident and it should rarely be necessary to remove this link.
4) Removing this breaks the active low Chip Enable signal to the ROM chip and therefore disables all eight banks of ROM (2, 3, 6, 7, 10, 11, 14 and 15). It is not possible to write to the ROM chip or read valid data from it with LK4 removed. Data is not lost when the link is open. Removing this link might be done temporarily when recovering from a corrupt or faulty ROM image but the normal position would be closed.

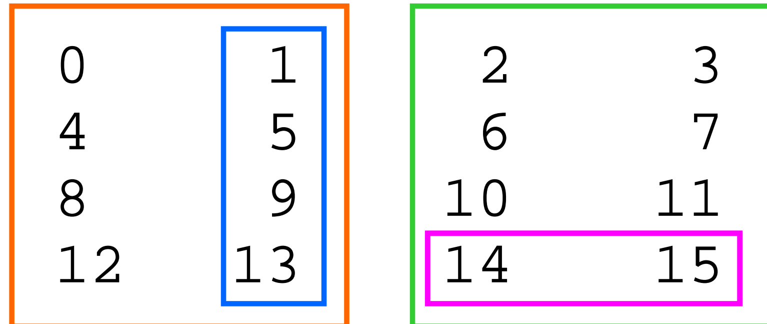
The two links marked PWP and MB are what might be called ‘decoded links’. Whereas links 1-4 physically break a signal path to the RAM or ROM, PWP and MB are inputs to the logic and decoded to achieve a particular effect.

PWP. The Partial Write Protect link, when removed, write protects the entire flash ROM chip and also banks 0, 4, 8 and 12 of the sideways RAM. RAM banks 1, 5, 9, and 13 can still be written to in the usual way, hence the ‘partial’ write protect aspect of it. PWP must be closed in order to program the flash ROM chip and closed is the default state.

MB. The MotherBoard link, when removed, activates the two rightmost sockets on the main board as numbers 14 and 15. When closed, sockets 14 and 15 are mapped to the flash ROM and the RAM/ROM board effectively completely replaces the Beeb’s ROM sockets.

Remembering the socket numbers

The choice of socket numbers for RAM and EEPROM on the expansion board isn't as random as it might first appear. It can be helpful to visualise the socket numbers 0 to 15 as four rows of four, thus;



At its simplest, the orange rectangle contains the numbers of the sockets with sideways RAM. The numbers in the green rectangle are mapped to the flash ROM.

We can also add a blue rectangle enclosing numbers 1, 5, 9, and 13. These are the sockets with no self-write protection and are not affected by the removal of the PWP (Partial Write Protection) link. By implication, therefore, all the remaining sockets do have self-write protection and are completely write protected when link PWP is open.

Finally, the purple rectangle encloses sockets 14 and 15. These are the socket numbers that are mapped to EEPROM when the MB link is made and the two rightmost sockets on the motherboards are active when the link is open.

As has been explained previously, the Beeb's sideways ROM system implements a certain 'priority' in that higher numbered sockets are offered service calls first (starting with 15 and working down to 0). Therefore, when loading ROM images, the numbering system used by the RAM/ROM board should allow considerable flexibility not only in terms of whether to use RAM or EEPROM, but also which ROM images have priority over others.

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.

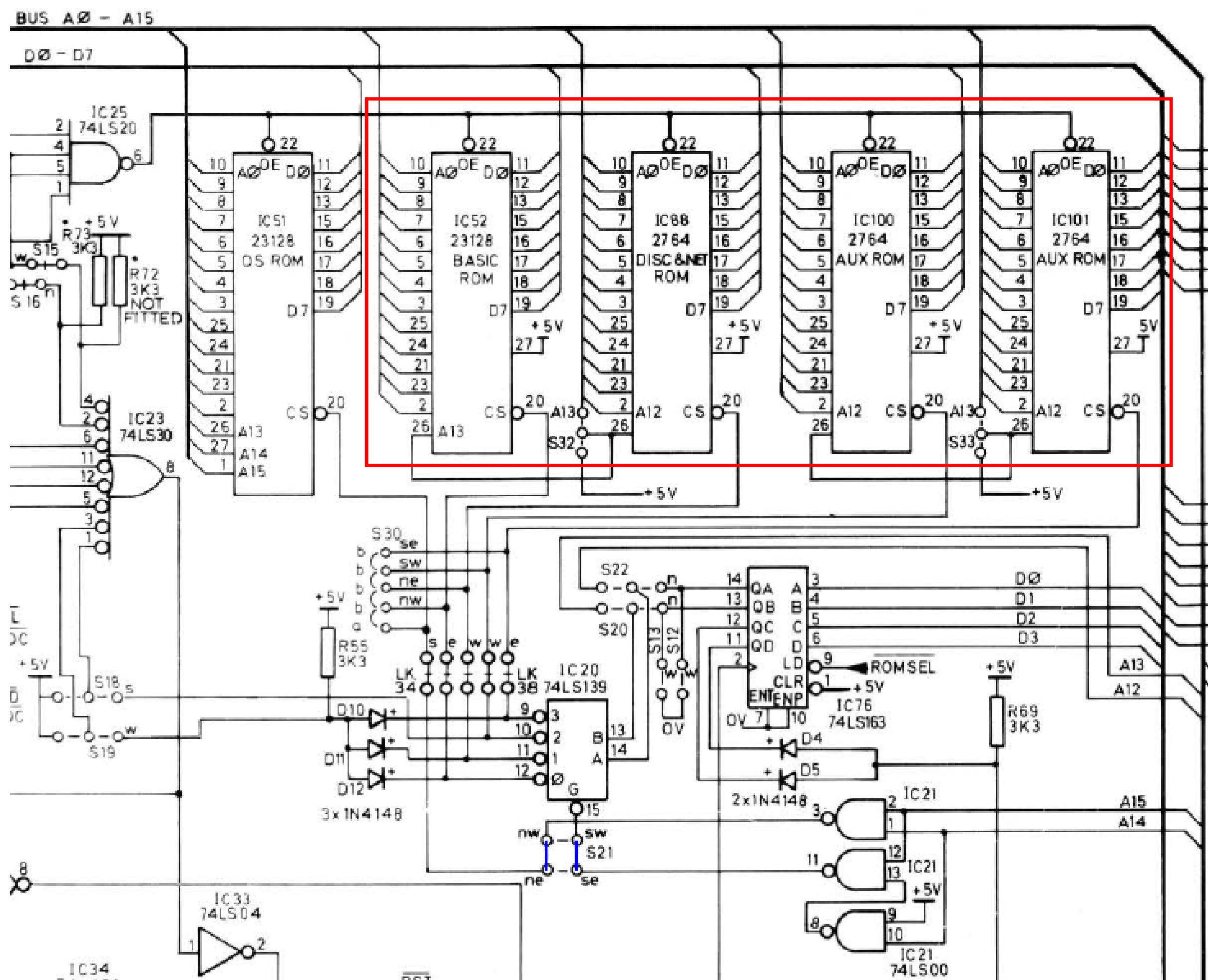
Examples of other pieces of hardware in the machine include the Operating System ROM, the System 6522 VIA (Versatile Interface Adapter), User VIA (optional), floppy disk controller (typically an Intel 8271 or perhaps a 1770 device), main memory and so on. All of these will be connected to the system 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 appears in the memory map.

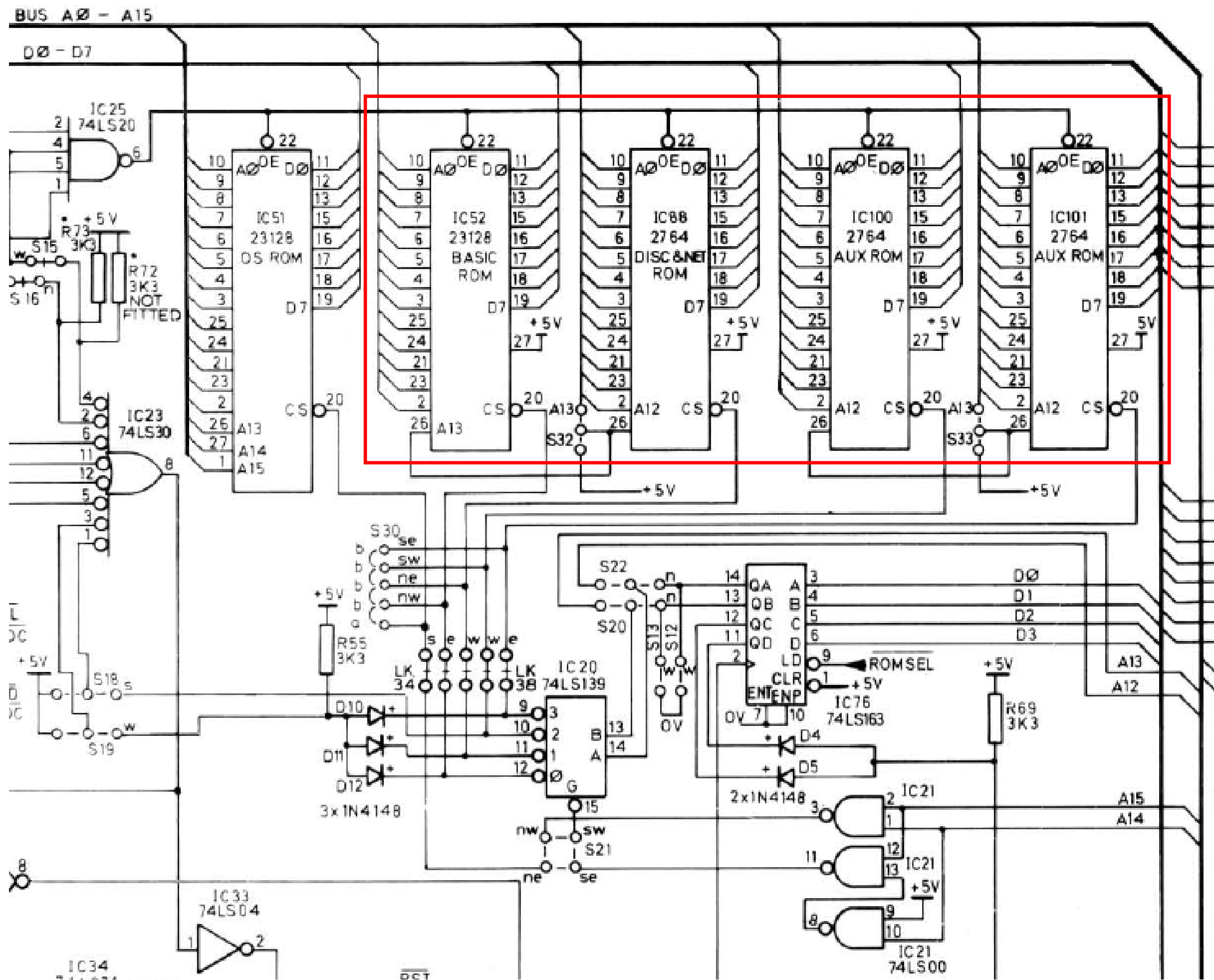
It is essential that only one device should try to control the data bus at any one time. Chaos would ensue if the floppy disk controller and one of the VIAs attempted to place data on the data bus at the same time. That is why a major fault on the motherboard, such as solder splash which permanently 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 and in fact these are the four sockets in the front right corner of the motherboard. 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. The system Reset signal does not in itself alter the contents of the latch, but the OS will write to the latch as part of its reset sequence.

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 A 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 D4-D7, 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

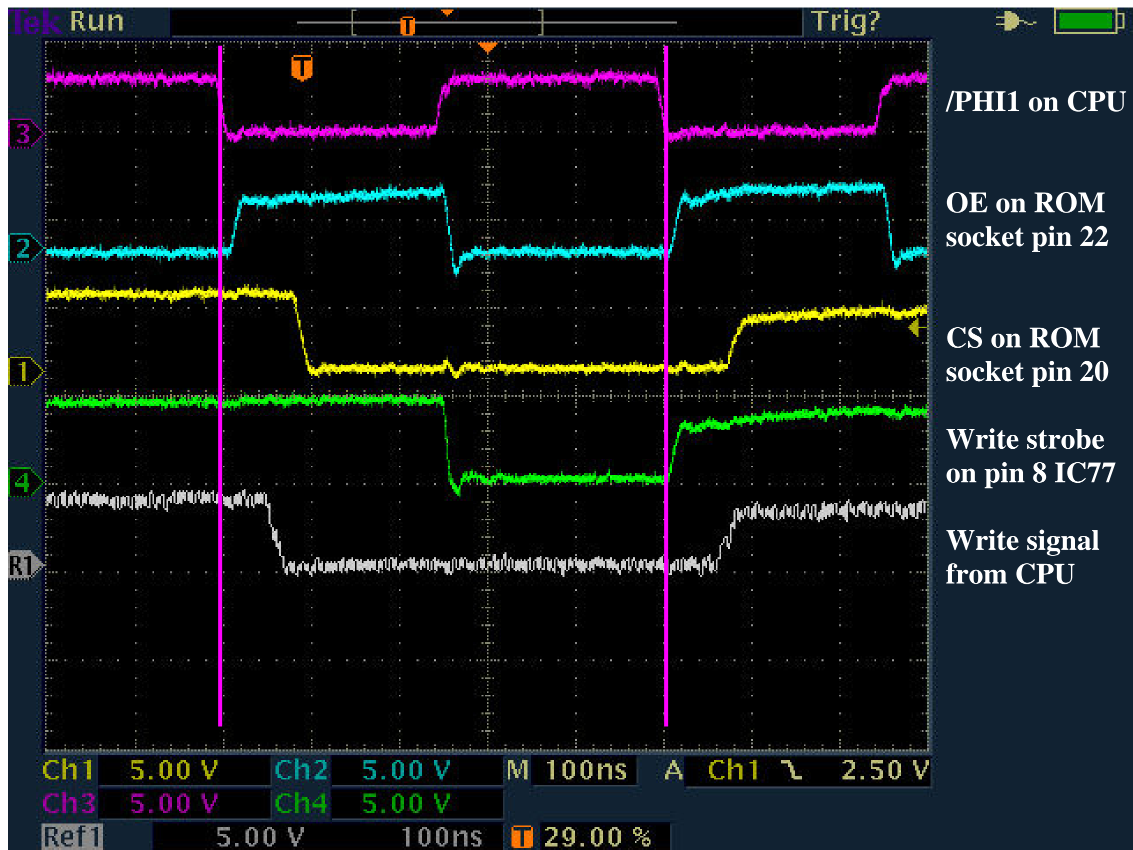
```
0010        (decimal 2)
0110        (decimal 6)
1010        (decimal 10)
1110        (decimal 14)
```

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
```

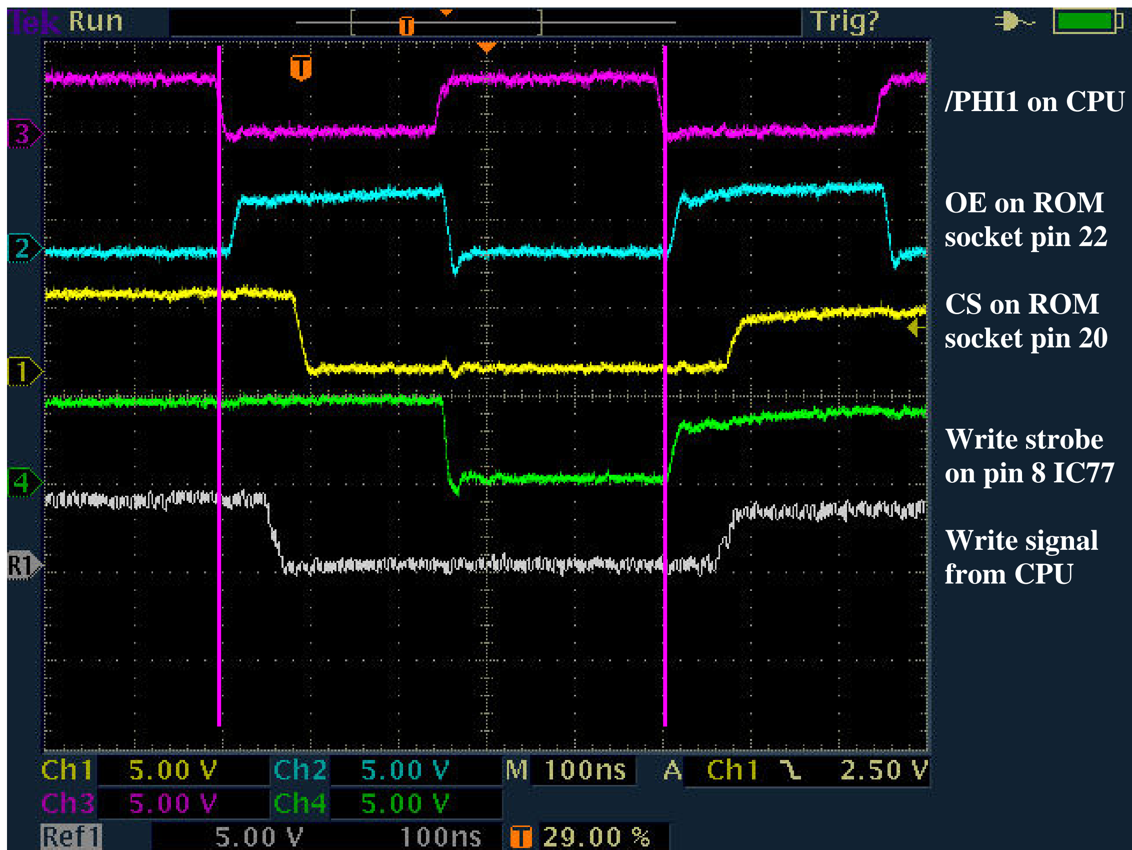
This program was used to produce the oscilloscope screen on the next page. It generates a low pulse on the CS signal, pin 20 of ROM socket 12. It is the STA instruction in line 90 which causes the low pulse on CS because it culminates in the CPU writing to address &9000 in the sideways ROM address range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



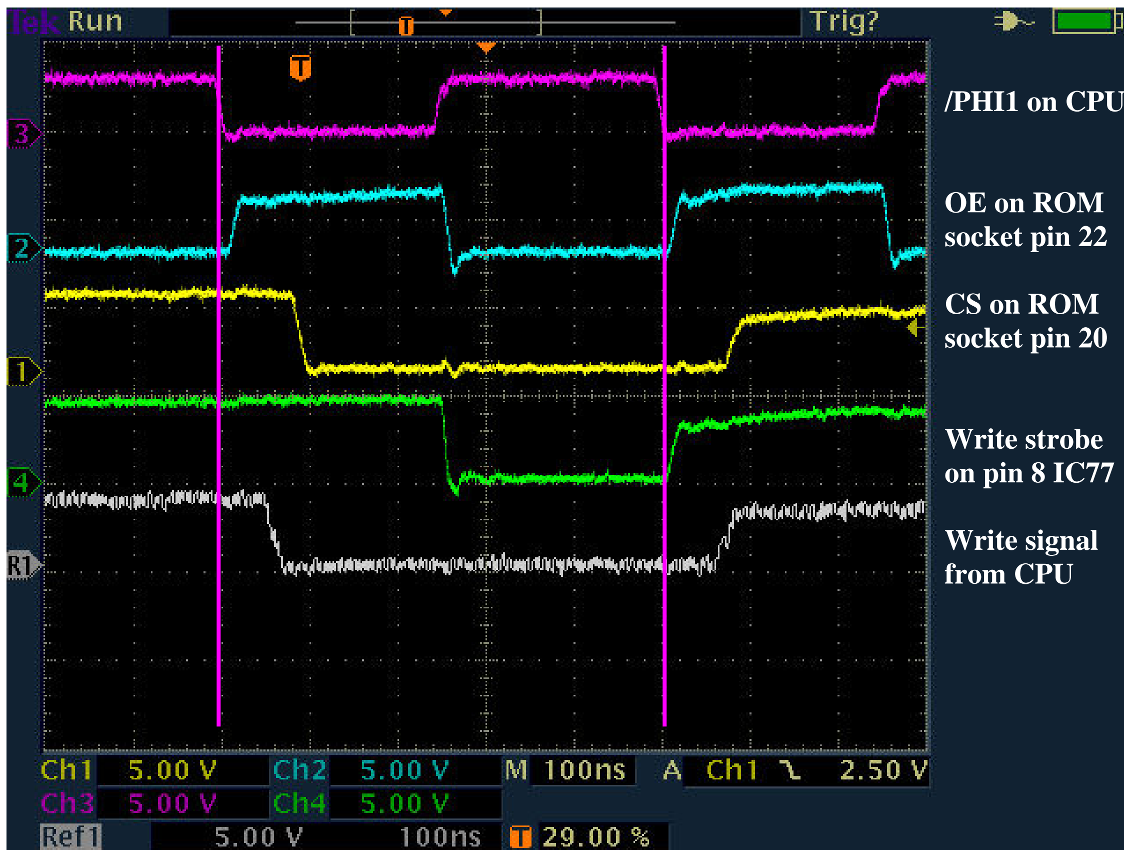
Same diagram as the previous page for ease of reference.

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 op code fetch from the address immediately after the three-byte STA &9000 instruction.



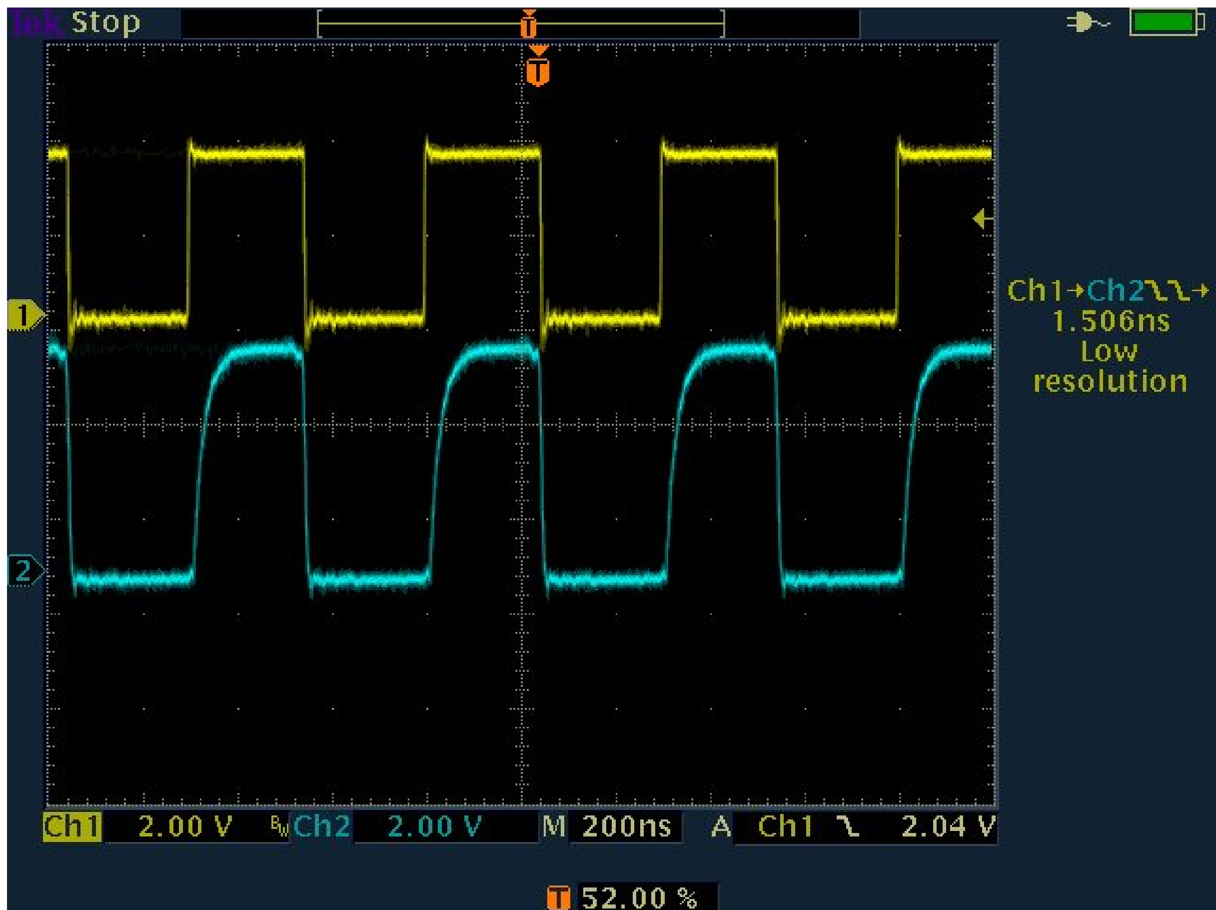
Same diagram as the previous page for ease of reference.

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.

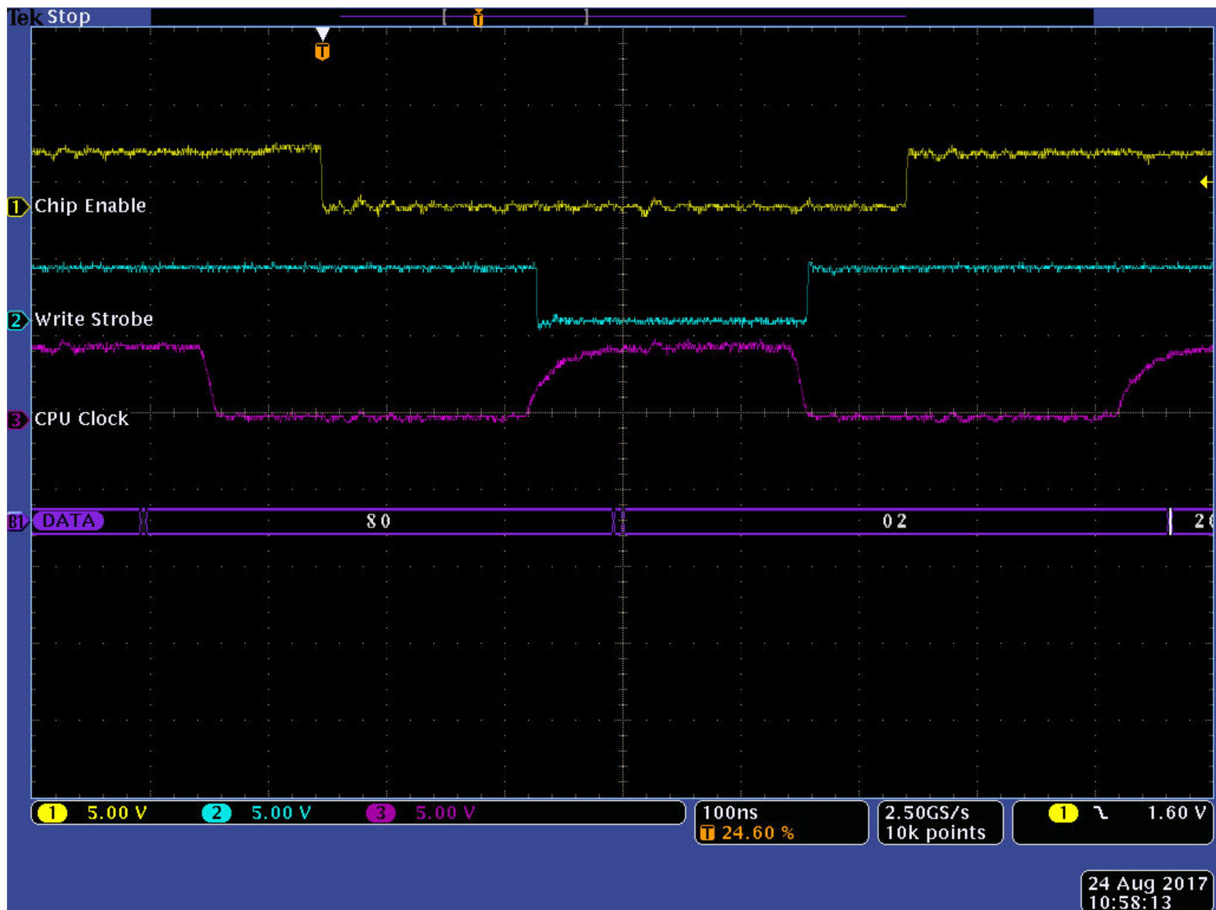
It is worth noting that a Read cycle from the sideways ROM space looks almost exactly like the above diagram. The only difference is that, fairly obviously really, the green Write Strobe remains high as does the Read/Write signal from the CPU (bottom trace).



Ch2 (blue): 2MHz PHI2 from the CPU
Ch1 (yellow): Pseudo PHI2 generated on the RAM/ROM board.

The bottom trace simply shows the ‘real’ 2MHz PHI2 clock from the CPU. Many of the timings of the signals described in the 6502 data sheet are referenced to the falling edge of PHI2.

The top trace is a “pseudo PHI2” generated on the RAM/ROM board. It has been designed to mirror the real PHI2 as closely as possible, particularly the falling edge. Here, the two falling edges differ in time by a mere 1.5ns. The top pseudo PHI2 clock is used extensively on the expansion board.



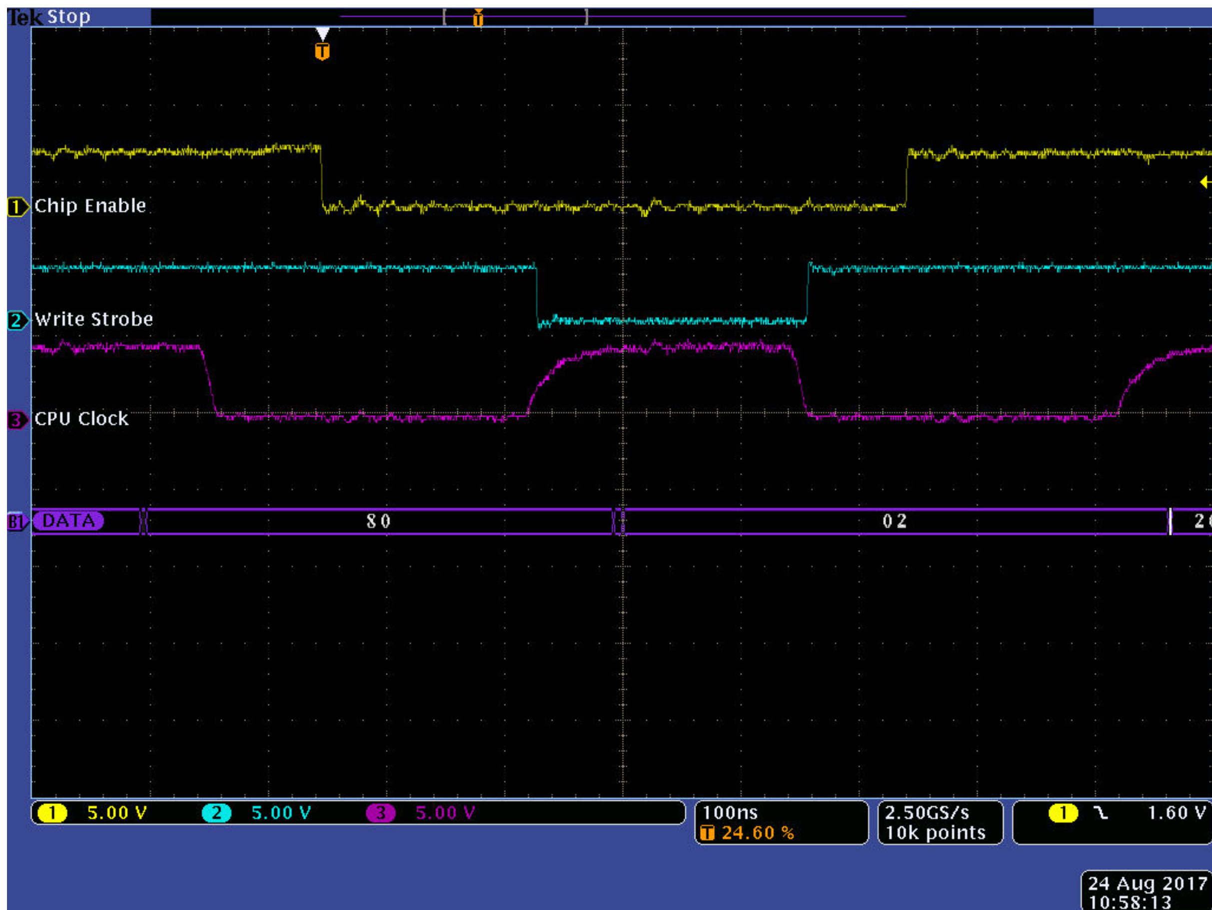
Write cycle to the SST39SF010 flash ROM on the RAM/ROM board (STA &8000)

The above signals are fairly self-explanatory and are those seen on the SST39SF010 when a write operation is performed.

Technically, it is a WE controlled Write cycle as opposed to CE controlled. The data sheet explains the difference but in the former, Chip Enable must initially be low and Output Enable needs to be logic high. With CE and OE maintaining their respective logic levels, a low pulse is applied to the WE pin, the rising edge of this pulse latching the data into the device. A restriction is that the width of the negative going WE pulse must be greater than 70ns, something that is clearly not a problem here. It is about 250ns wide.

The instruction that produced the above signals was STA &8000 with the accumulator set to 2. The contents of the data bus start off as &80 because this is the last byte of the three-byte STA &8000 instruction. The data bus changes to 02 as the processor outputs the accumulator shortly after the rising edge of the clock. The 6502 datasheet uses the symbol t_{WDS} for the time between the rising edge of the clock and valid data appearing on the data bus. For the 2MHz part it is given as a maximum of 110ns although on this particular cycle it's only about 60ns.

The write timing implemented for the RAM chip is almost identical to that for the flash ROM although the CE falling and rising edges are delayed slightly (10ns or so) by the DS1210 battery backup controller.



Write cycle to the SST39SF010 flash ROM on the RAM/ROM board (STA &8000. Same diagram as previous page)

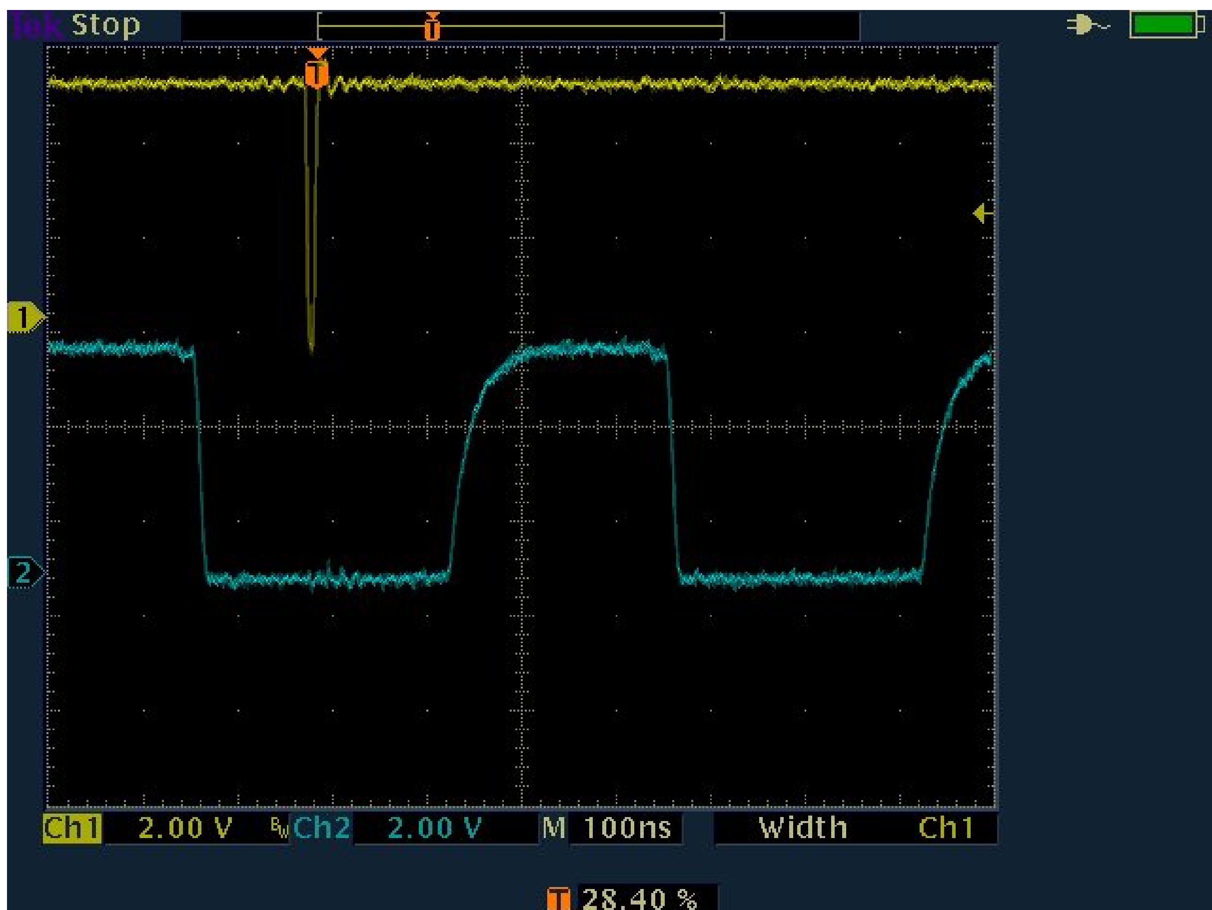
As just mentioned, the Write cycle to the static RAM chip is very similar to that for the flash ROM shown above, and the only difference is that the yellow Chip Enable signal is shifted right by about 10ns. But it shows an important aspect of the static RAM's operation.

The Write cycle to the static RAM starts when Write Strobe goes low with Chip Enable already in its active low state. But the data bus initially has the wrong value of &80. It changes to the correct value about 60ns later and continues to hold this correct value until Write Strobe goes high. In fact, the data bus can change as often as it likes because the data inputs to the static RAM are 'transparent'. The byte written to the RAM chip will be whatever was on the data bus at the moment Write Strobe or Chip Enable goes high, whichever occurs first. A point of detail is that the 128kB static RAM chip data sheet specifies a certain minimum time for which the data must be valid ***prior*** to the end of the Write cycle, typically 25ns. From the above diagram, it is clear that valid data (&02 in this case) is present on the bus for about 150ns before Write Strobe goes high.

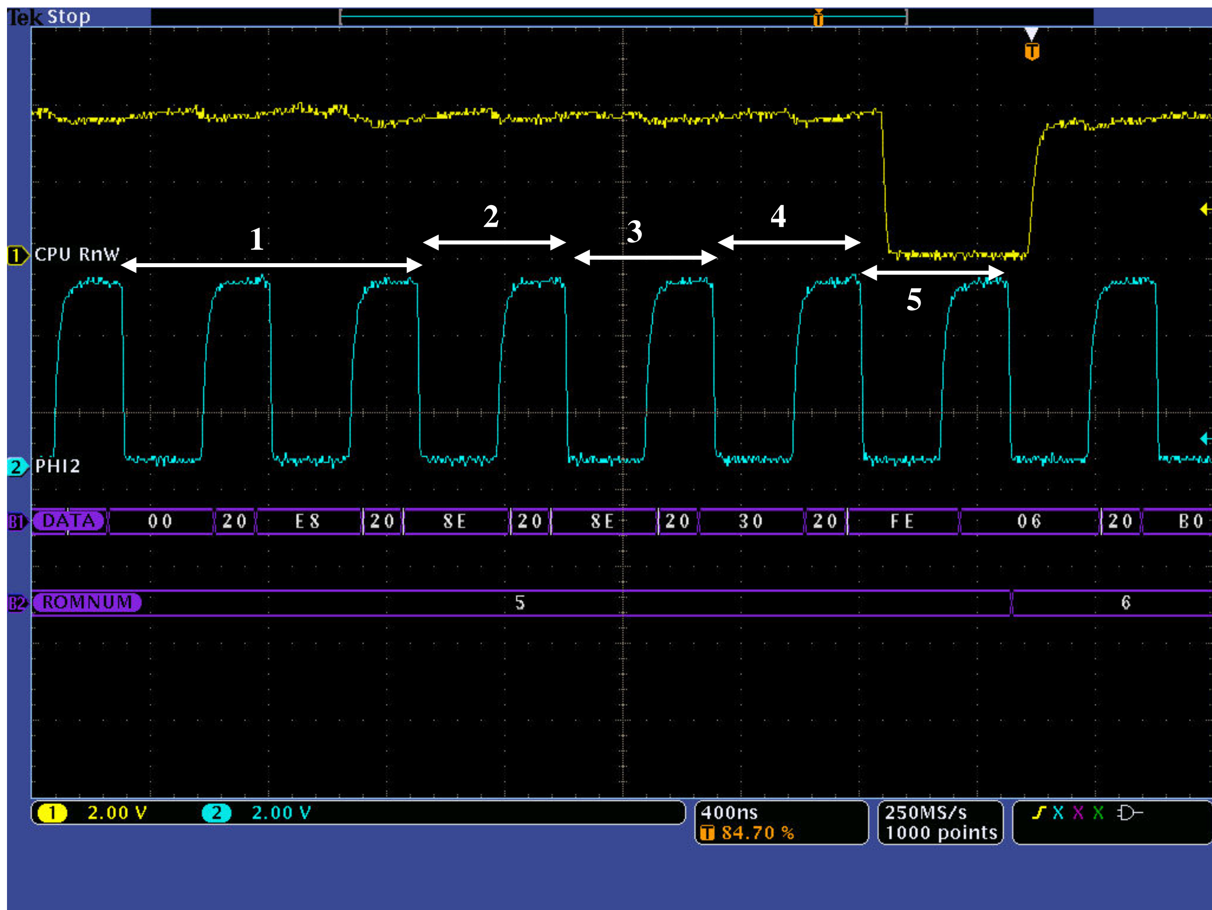
The same 'transparency' issue also affects the address lines. The various CPU control signals, particularly the address and Read/Write signals, typically change to their new values about 100ns after the falling edge of PHII2 (purple trace). But they don't all change at exactly the same time and the effect is that if the CPU RnW signal on pin 34 is connected directly to the RAM chip, multiple addresses will be written to. The problem is made worse by the fact that the data bus also initially contains the incorrect value. RAM corruption is the

inevitable result and that is why the Write Strobe signal must be delayed until much later in the machine cycle, usually until PHI2 goes high.

A typical decoding glitch, about 10ns wide in this case, that can result from changing address lines is shown below. These glitches aren't a problem as such providing the system is designed to ignore them. For example, feeding channel 1 (yellow) directly into a clock input would cause all sorts of strange results, with the narrow glitch probably being interpreted by the rest of the circuit as a valid clock pulse.



Decoding glitch on channel 1 that can be produced by changing address lines. Bottom trace is the CPU PHI2 signal on pin 39.



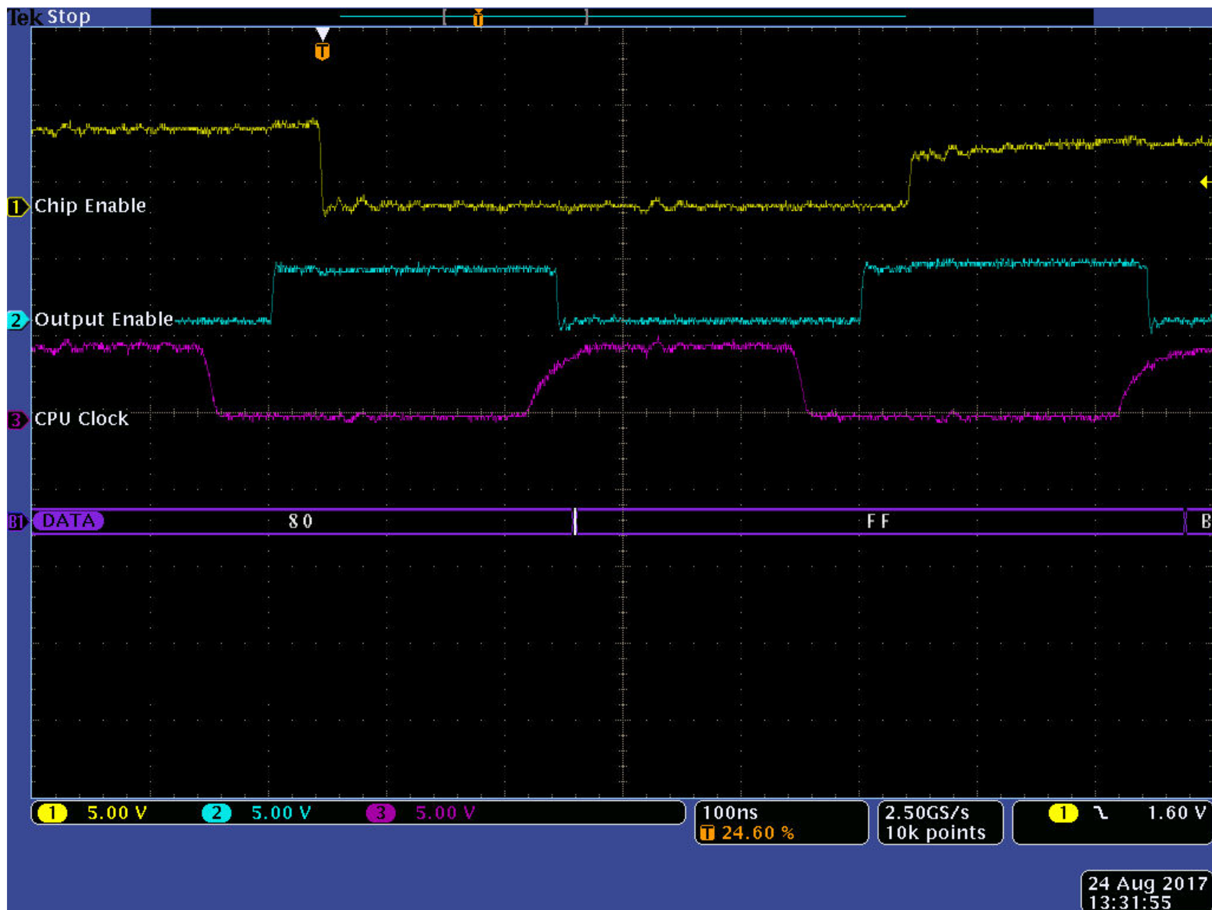
Writing to the ROM select latch at address &FE30

Fundamental to the operation of the Beeb's sideways ROM system is the writable 4-bit latch at address &FE30. Writing to this address causes the least significant four bits of the data bus to be written into the latch, and the latch's outputs will retain this data until such time as it is written to again. The system reset signal does not in itself clear the latch although the operating system's reset code will write to this latch. The above signals were produced by a very simple loop with the carry flag known to be set and interrupts disabled;

```
.loop : INX : STX &FE30 : BCS loop
```

In the above picture, DATA is the system eight bit data bus D0 to D7. ROMNUM is the output of the four bit latch accessed by writing to &FE30. PHI2 is the 2MHz clock on pin 39 of the CPU and CPU RnW is the Read/Write signal on pin 34. The clock cycles have been numbered to clarify exactly what is happening.

1) Op code &E8 (INX) is read from memory and executed. This take two cycles in all.
2) Op code &8E (STX absolute) is read from memory.
3) The low byte of the target address, &30, is read from memory.
4) The high byte of the target address, &FE, is read from memory.
5) The contents of the X register, previously 5 but now incremented to 6 by INX, is placed on the data bus and written to address &FE30. The falling edge of PHI2 transfers the lower four bits of data into the ROMNUM latch which now takes on the new value of 6.



Read cycle from the SST39SF010 flash ROM
(LDA &8000)

The read cycle doesn't require anything particularly complex. Both the Chip Enable and Output Enable (nOE) pins must be low for valid data to appear on the chip's output pins. It can be seen that the access time from nOE going low (T_{oe} on the SST39SF010 data sheet) is about 15ns and the data read from the chip in this example is &FF. Capacitance on the data bus also ensures that this value stays on the bus for about 200ns after nOE has gone high. DATA just starts to change at the extreme right of the screenshot as the next op code is fetched from memory.

The 6502 data sheet also specifies a 'Read Data Setup Time', t_{DSU}. This is the time for which valid data must be on the 6502 data bus before the falling edge of PHI2 during a read cycle and is given as 60ns for the 2MHz part. In the above diagram, this would be the time between the white vertical line where the DATA becomes valid (roughly in the centre of the screen on the purple DATA bus) and the next falling edge of CPU clock, PHI2. From the time base of 100ns per division we can see that this time difference is a little under 200ns, giving a comfortable safety margin over the 60ns minimum.

IFEL, 21 Glenfield Road, Glenholt PLYMOUTH PL6 7LL (01752) 777106
Email: ifelsales@tiscali.co.uk