PETTESTE2K – Version 04. Copyright © 2020 David E. Roberts.

This test program was developed as a 'cheap and cheerful' test aid for the various versions of the Commodore PET computer (irrespective of whether it contains a CRT (Cathode Ray Tube) controller device or not).

On start-up, the 6502 CPU (Central Processing Unit) starts executing instructions from a RESET VECTOR found in high memory. In a standard Commodore PET machine this is where the Kernal ROM (Read Only Memory) resides at address $Fxxx.

The general problem with the PET is that most of the ROMs were soldered into the board – making it difficult to install a replacement ROM containing diagnostic code. However, in order to customise the PET for different countries, Commodore decided to place all of the customisation and internationalisation code into what they refer to as an EDIT ROM residing at $Exxx. This is a 2K ROM device and is generally installed in an IC (Integrated Circuit) socket. As a result, the EDIT ROM can be removed from the socket and a diagnostic replacement installed.

On start-up, the 6502 resets to a vector contained within the high memory of the Kernal ROM and starts to initialise the hardware. After only a few instructions, the Kernal ROM executes initialisation code stored in the EDIT ROM. This is our chance to 'get in' and slip our diagnostic code into the PET unnoticed!

The downside of this technique is that a small portion of the Kernal ROM must be working correctly in order for the 6502 instruction execution to reach our diagnostic ROM. This was considered a worthwhile trade-off – but should be considered if the diagnostic ROM fails to even start execution.

Of course, the diagnostic ROM outputs the results of the various tests to the screen; therefore a functioning video sub-system is required. This, however, is one of the first tests the diagnostic ROM performs!

On entry from the Kernel ROM, the diagnostics initialises any CRT controller (if present on the PET). The initialisation mode and parameters (e.g. 40/80 columns, 50/60 Hz etc.) can be configured by the user by editing the source code for the diagnostic ROM and inserting the appropriate table of register values at label CRTC_INIT. The binary image can also be hand-patched after loading it into an EPROM programmer if so desired rather than editing the assembler source code and re-assembling it. If a CRT controller is not present on your particular PET, don't worry, the initialisation will not take effect and no damage will occur. The electrons just spill out onto the floor ☺!

You only need to patch the CRTC initialisation values if you are using a 40 column PET with a CRTC device (e.g. a PET 4032 'Fat 40'). No CRTC – no problem!

There are details related to PETs and CRTCs you will find at:

http://cbm-hackers.2304266.n4.nabble.com/PET-50Hz-editor-ROMS-td4658493.html

http://www.6502.org/users/andre/petindex/crtc.html

Disclaimer: There also appears to be inconsistent data around as well (as is the nature with disassembling things). If you have any problems, please contact me at http://www.vcfed.org/forum/activity.php user daver2 (on the Commodore Forum) and I will be glad to help out.

VDU TEST

This test writes an incrementing character code from $00 to $FF into all 8 pages (2K for an 80-column PET) of the VDU memory. If your machine is a 1K 40-column PET this should still be OK – the last 4 pages of data writes should either be discarded or overwrite the first 4 pages again. Either way, this should be fine.

The test continues by verifying that the first 4 pages (1K) of VDU memory contain the correct values. If not, the test program loops so that the test operator can see the results.

If the test fails – the diagnostic program loops, re-writes the test pattern and performs a further check. This process continues until such time as the correct data pattern is read from the VDU memory.

Obviously, a completely black picture could indicate that a number of problems exist:

- The monitor may be faulty.
- No video or synchronisation signals may be getting from the mainboard to the monitor.
- The timing chain is faulty.
- If a CRT Controller is fitted – it may be faulty.
- The CPU is not correctly executing instructions. This may be due to a number of reasons…

If the test pattern is not correct – the test operator can observe the actual characters that are being displayed as a result of the read/write operations from/to the VDU memory. This should give you some clue as to the potential fault (or at least where to start probing). Obvious errors could be a stuck data bit (either a '1' where a '0' should be or a '0' where a '1' should be). Less obvious errors are addressing faults where the correct character is stored at an incorrect address – thus either not being stored to the correct location, or overwriting another location. Of course, this may occur on the read or write cycle – or be a function of the memory device itself.

Data could also be changing. This would be indicated on the display as flickering characters (say alternating between 'A' and 'B' in the same character cell).

When the diagnostic reads the correct test pattern from the VDU memory – it performs a delay (to let the human see the results…) and moves on automatically to the next test.

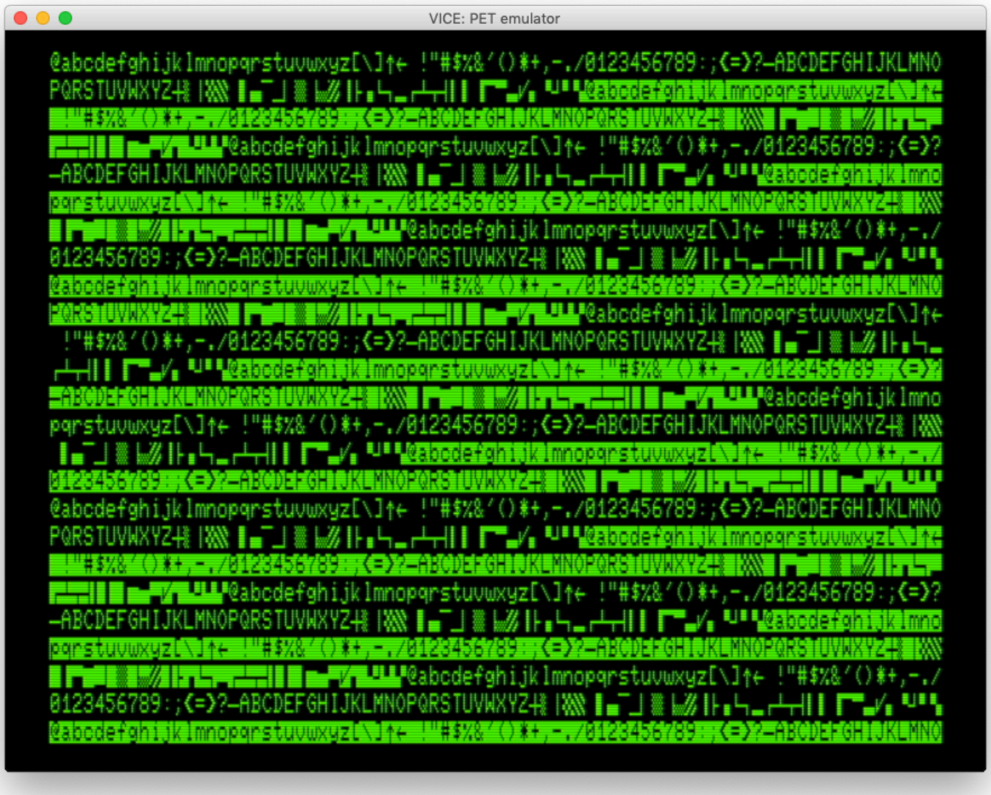


Figure 1 – A 'PASS' display for the VDU test (80-columns).

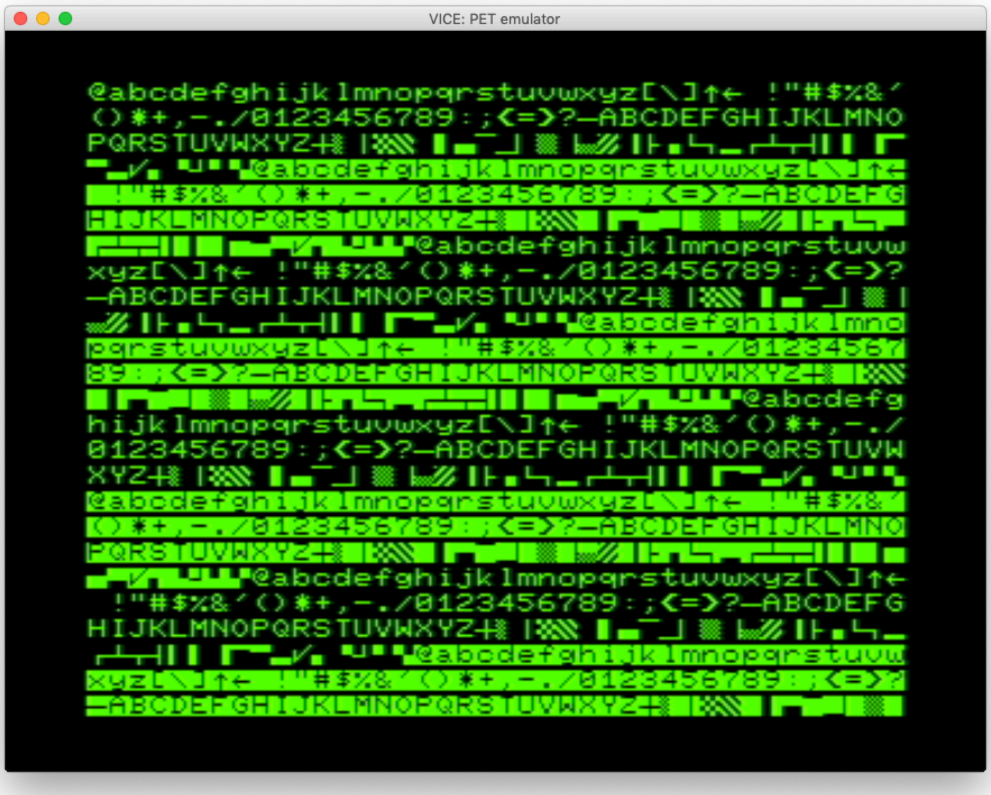


Figure 2 - A 'PASS' display for the VDU test (40-columns).

Page 0/1 test with data byte values ranging from $00 to $FF.

Assuming the VDU test completed without errors, the diagnostic ROM moves on to perform a basic test on pages 0 and 1 of the RAM (Random Access Memory). Page 0 is a special area of memory that is used by a particular addressing mode of the 6502 CPU. Page 1 (or part of page 1) is generally reserved for the stack. Each page consists of 256 bytes of RAM. A failure in either page 0 or page 1 of RAM could result in a catastrophic failure of the PET BASIC firmware stored in ROM.

The diagnostic firmware writes the data values from $00 to $FF into consecutive bytes of Page 0 and Page 1. The value $00 gets written into addresses $0000 and $0100. The value $01 gets written into addresses $0001 and $0101. … . The value $FE gets written into addresses $00FE and $01FE. And finally, the value $FF gets written into addresses $00FF and $01FF.

The diagnostic firmware then verifies that the correct values that should have been written can be correctly read back. The status of the basic memory test is stored on the screen in a coded form as follows:

The screen is divided into four (4) sections, each section containing 256 consecutive character locations on the screen.

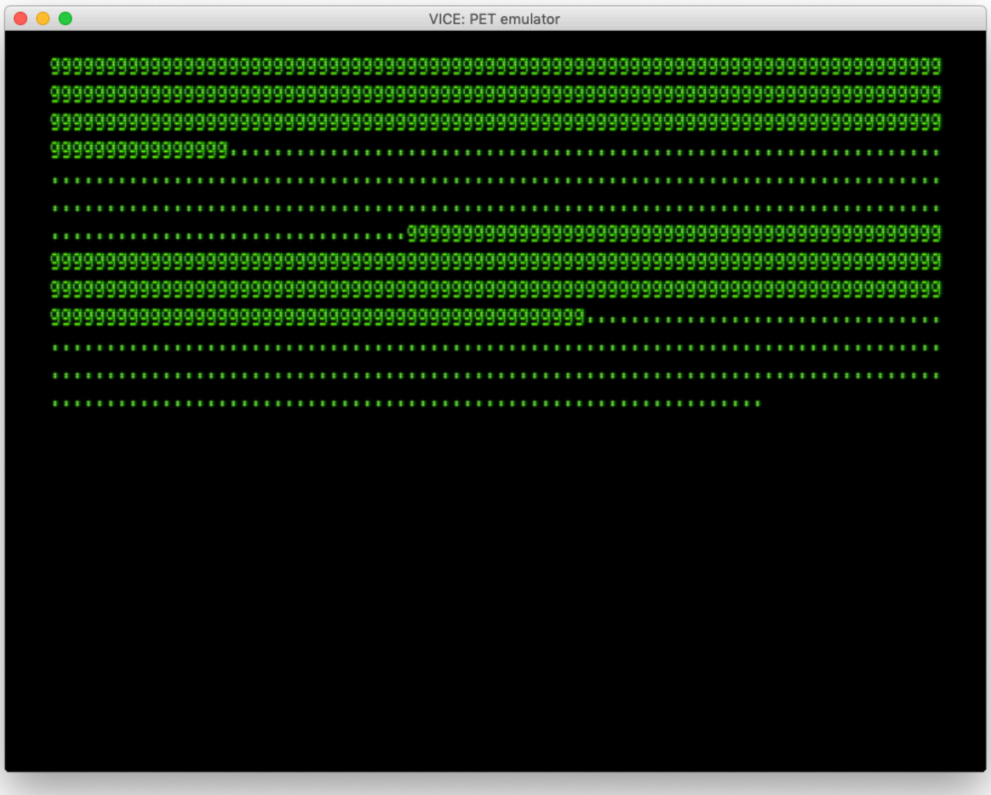
The first 256 characters indicate a ‘G’ or ‘B’ (or a ‘g’ or ‘b’) character (depending upon which character generator ROM is fitted and whether the Page 0 memory location corresponding to that address contains the correct data when read). A ‘G’ or ‘g’ character indicates a ‘Good’ memory location; whereas a ‘B’ or ‘b’ character indicates a ‘Bad’ memory location.

The second 256 characters indicate either a ‘.’ character (if the memory location in page 0 was determined to be ‘Good’) or the character corresponding to the data that was found (if the memory location in page 0 was determined to be ‘Bad’).

The next two lots of 256 characters are a repeat of the above but for page 1 RAM.

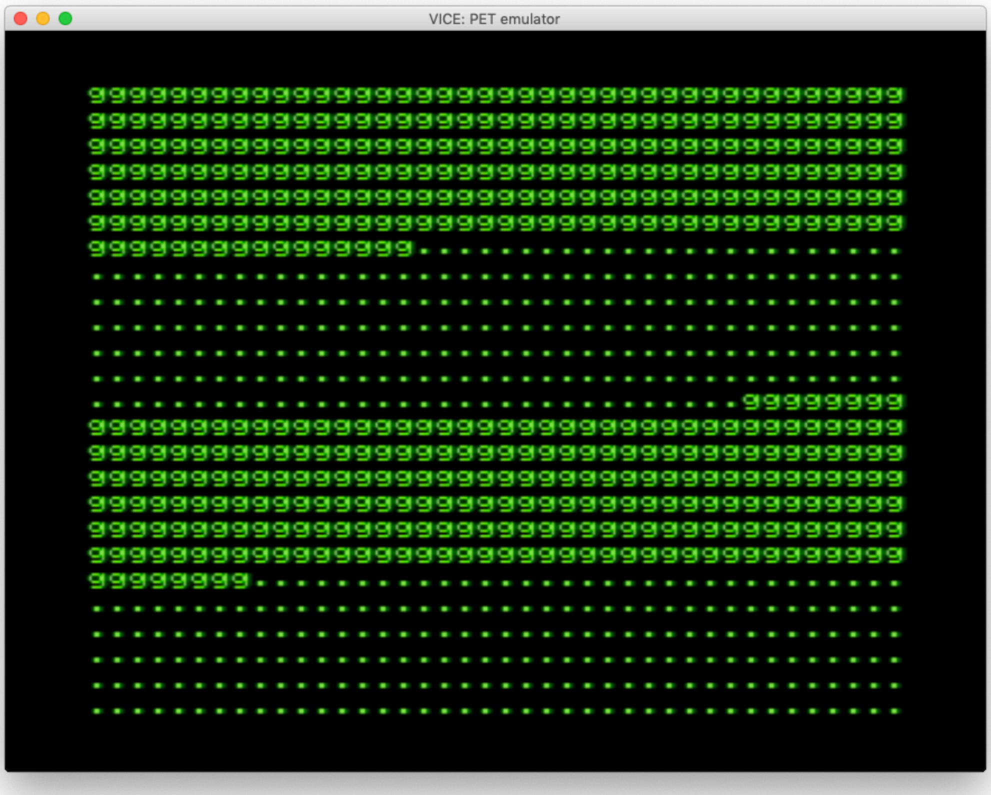
Note that on an 80 column PET you will see all of the characters corresponding to the full contents of pages 0 and 1 under test. On a 40 column PET you will be missing a few characters as only 40 [columns] * 25 [rows] = 1,000 [characters] (out of the 1,024 characters required for the test) is displayable.

If the test works fine, a delay will be performed (to permit the test operator to briefly see the results) and the firmware will move on automatically to the next test.



```
gggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggg
gggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggg
gggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggg
gggggggggggggggg................................................................
................................................................................
................................................................................
................................gggggggggggggggggggggggggggggggggggggggggggggggg
gggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggg
gggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggg
gggggggggggggggggggggggggggggggggggggggggggggggg................................
................................................................................
................................................................................
................................................................
```

Figure 3 - A 'PASS' display for the page 0/1 $00 to $FF test (80-columns).



```
gggggggggggggggggggggggggggggggggggggggg
gggggggggggggggggggggggggggggggggggggggg
gggggggggggggggggggggggggggggggggggggggg
gggggggggggggggggggggggggggggggggggggggg
gggggggggggggggggggggggggggggggggggggggg
gggggggggggggggggggggggggggggggggggggggg
gggggggggggggggg........................
........................................
........................................
........................................
........................................
........................................
................................gggggggg
gggggggggggggggggggggggggggggggggggggggg
gggggggggggggggggggggggggggggggggggggggg
gggggggggggggggggggggggggggggggggggggggg
gggggggggggggggggggggggggggggggggggggggg
gggggggggggggggggggggggggggggggggggggggg
gggggggggggggggggggggggggggggggggggggggg
gggggggg................................
........................................
........................................
........................................
........................................
........................................
```

Figure 4 - A 'PASS' display for the page 0/1 $00 to $FF test (40-columns).

Page 0/1 test with data bytes of $55 and $AA.

Assuming the VDU test completed without errors, the diagnostic ROM moves on to perform a basic 'stuck bit' test of page 0 and page 1 RAM

This test first stores the pattern $55 (binary 01010101) into pages 0 and 1 of RAM and checks for the correct value being stored. The test next stores the pattern $AA (binary 10101010) into pages 0 and 1 of RAM and checks once again for the correct value being stored. The status of the testing (for each byte of page 0 and 1) is shown on the VDU. A 'G' or 'g' character is displayed for each byte that is GOOD/good whilst a 'B' or 'b' character is displayed for each byte that is BAD/bad respectively. Note that an upper or lower case character may be displayed – depending upon which variation of the character generator is actually fitted to your PET.

As a 1K (40-column) PET screen can display 25*40 = 1,000 characters; I can almost use that to indicate the state of each byte of the test (although I do, unfortunately, loose the last 24 bytes). There is no problem on a 2K (80-column) PET screen though.

The first 256 bytes of the display indicate the state of storing and testing the memory in page 0 with a value of $55.

The second 256 bytes of the display indicate storing and testing the memory in page 1 with a value of $55.

The third 256 bytes of the display indicate storing and testing the memory in page 0 with a value of $AA.

The fourth 256 bytes of the display indicate storing and testing the memory in page 1 with a value of $AA.

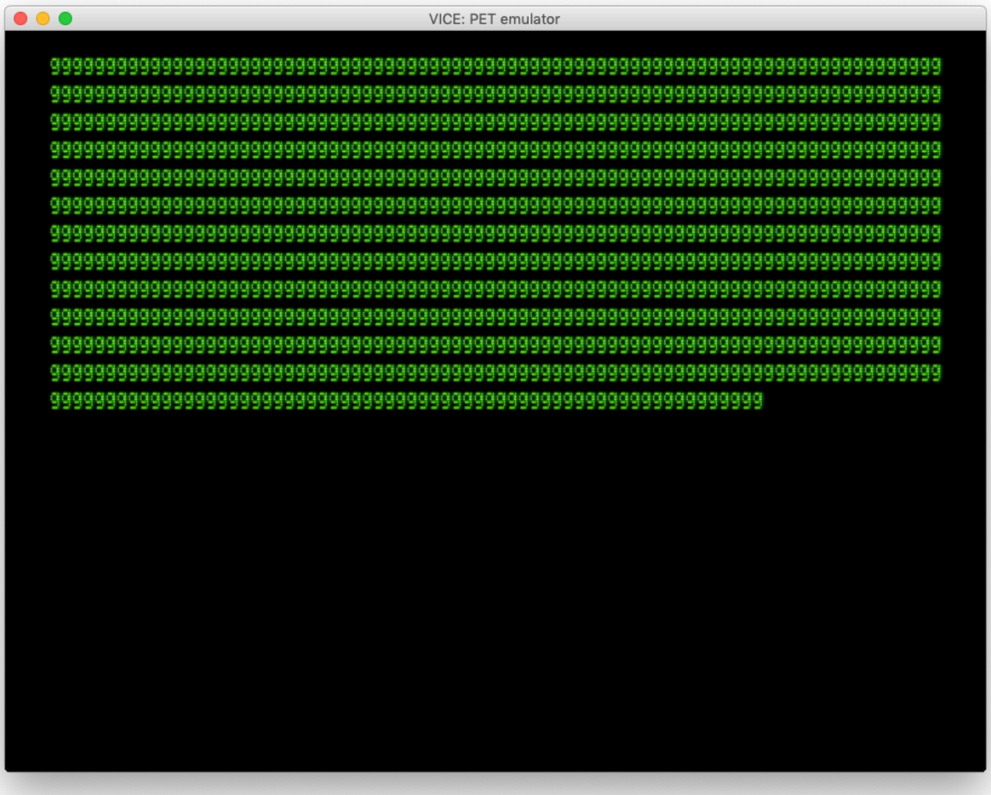
As stated previously, the displayed character indicates if the test passed (character = 'G' or 'g') or the test failed (character = 'B' or 'b'). The location of the character on the screen (i.e. the index into the 256 byte display block) indicates which byte of the memory page failed the test.

Unfortunately, using this means of testing and displaying the results of the test, it is not possible to display what the actual value was when the memory was read. This is why I perform a simple test prior to this one first so that most of the common errors would (hopefully) be trapped and the errant value displayed prior to this test.

If the test fails, the diagnostics keep looping on this test.

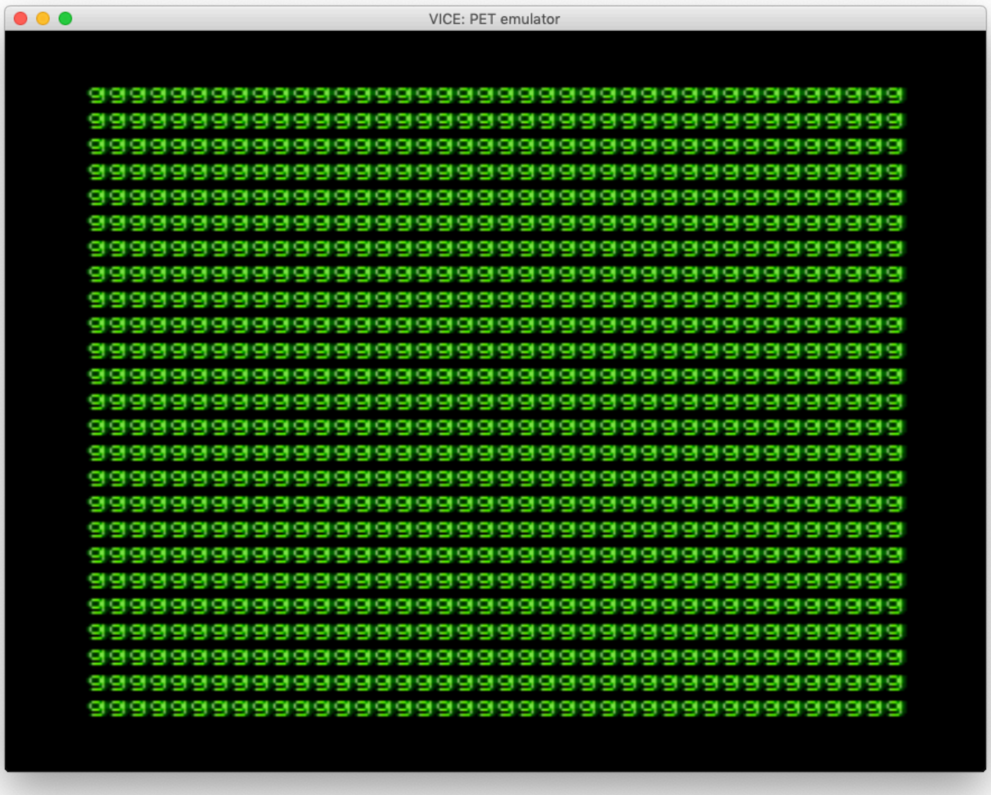
Again, it may be possible to detect 'random' memory faults by observing the displayed status character alternating from good to bad and vice-versa.

If the test works fine, a delay will be performed (to permit the test operator to briefly see the results) and the firmware will move on automatically to the next test.



```
99999999999999999999999999999999999999999999999999999999999999999999999999999999
99999999999999999999999999999999999999999999999999999999999999999999999999999999
99999999999999999999999999999999999999999999999999999999999999999999999999999999
99999999999999999999999999999999999999999999999999999999999999999999999999999999
99999999999999999999999999999999999999999999999999999999999999999999999999999999
99999999999999999999999999999999999999999999999999999999999999999999999999999999
99999999999999999999999999999999999999999999999999999999999999999999999999999999
99999999999999999999999999999999999999999999999999999999999999999999999999999999
99999999999999999999999999999999999999999999999999999999999999999999999999999999
99999999999999999999999999999999999999999999999999999999999999999999999999999999
99999999999999999999999999999999999999999999999999999999999999999999999999999999
99999999999999999999999999999999999999999999999999999999999999999999999999999999
9999999999999999999999999999999999999999999999999999999999999999
```

Figure 5 - A 'PASS' display for the page 0/1 $55 and $AA test (80-columns).



```
gggggggggggggggggggggggggggggggggggggggg
gggggggggggggggggggggggggggggggggggggggg
gggggggggggggggggggggggggggggggggggggggg
gggggggggggggggggggggggggggggggggggggggg
gggggggggggggggggggggggggggggggggggggggg
gggggggggggggggggggggggggggggggggggggggg
gggggggggggggggggggggggggggggggggggggggg
gggggggggggggggggggggggggggggggggggggggg
gggggggggggggggggggggggggggggggggggggggg
gggggggggggggggggggggggggggggggggggggggg
gggggggggggggggggggggggggggggggggggggggg
gggggggggggggggggggggggggggggggggggggggg
gggggggggggggggggggggggggggggggggggggggg
gggggggggggggggggggggggggggggggggggggggg
gggggggggggggggggggggggggggggggggggggggg
gggggggggggggggggggggggggggggggggggggggg
gggggggggggggggggggggggggggggggggggggggg
gggggggggggggggggggggggggggggggggggggggg
gggggggggggggggggggggggggggggggggggggggg
gggggggggggggggggggggggggggggggggggggggg
gggggggggggggggggggggggggggggggggggggggg
gggggggggggggggggggggggggggggggggggggggg
gggggggggggggggggggggggggggggggggggggggg
gggggggggggggggggggggggggggggggggggggggg
gggggggggggggggggggggggggggggggggggggggg
```

Figure 6 – A 'PASS' display for the page 0/1 \$55 and \$AA test (40-columns).

<u>ROM/KBD TEST</u>

Assuming all of the previous tests have passed successfully, the screen is cleared and the ROM checksums and keyboard circuitry is tested next.

The first few lines of the screen are populated with all of the possible characters available within the character generator.

The diagnostic firmware then performs a 16-bit checksum of the code in the following ROMS:

- \$Bxxx.
- \$Cxxx.
- \$Dxxx.
- \$Fxxx.

Check summing the \$Exxx ROM does not make sense, as the Commodore original EDIT ROM has been replaced by the PETTEST diagnostic firmware!

There is a single line of text displayed on the VDU as follows:

```
rom b=xxxx c=xxxx d=xxxx f=xxxx
```

Where 'xxxx' is replaced by the computed checksum. The 4-character hexadecimal 16-bit checksum will depend upon the particular version of BASIC (ROM set) installed within the PET.

Known variants are:

BASIC version	**Computed 16-bit checksum**			
	ROM $Bxxx	**ROM $Cxxx**	**ROM $Dxxx**	**ROM $Fxxx**
1 [note 1]	7800	ccc9	5045	0cd4
2	7800	3838	506a	7c98
4	4168	5960	a425	cf19

Table 1 - ROM Checksums.

[Note 1]: It is just possible that the $Cxxx ROM checksum identified here is incorrect for BASIC 1. I believe there was a fault within the code associated with the IEEE488 interface of BASIC 1 and a 'patch' was issued. I see that VICE automatically applies the patch if it finds a copy of BASIC 1. If you have an unmodified version of BASIC 1, you may find that the $Cxxx ROM checksum differs from the one identified (ccc9). It may end in $xx5d.

The next integrated test to run concurrently with the ROM checksums is a test of the keyboard key matrix and associated circuitry. The key matrix is scanned and the results displayed on a single line of text on the VDU as follows:

```
kbd 00 00 00 00 00 00 00 00 00 00
```

Without any keys depressed, the results should be all zeros. Any other value signifies a 'stuck' or 'shorted' key or a fault with the scanning circuitry.

As each key on the keyboard is alternately pressed and released, you should see a single bit flip from '0' to '1'. The results are indicated in hexadecimal, so each digit could indicate '0', '1', '2', '4' or '8' depending upon which bit (if any) is set…

Each key on the keyboard should result in one bit (and one bit only) becoming set. When the key is released, all of the bits should be cleared once again.

Unfortunately, Commodore (in their infinite wisdom) decided to change the key matrix layout, not only based on the 'type' of keyboard (e.g. chicklet, business or graphics) but also depending upon which country the PET was designed for. All of the 'magic' for the key matrix arrangement took place within the firmware of the EDIT ROM (which was unique to the keyboard type and country). Hence, there is no 'simple' way of easily mapping a generic keyboard to the associated key matrix displayed.

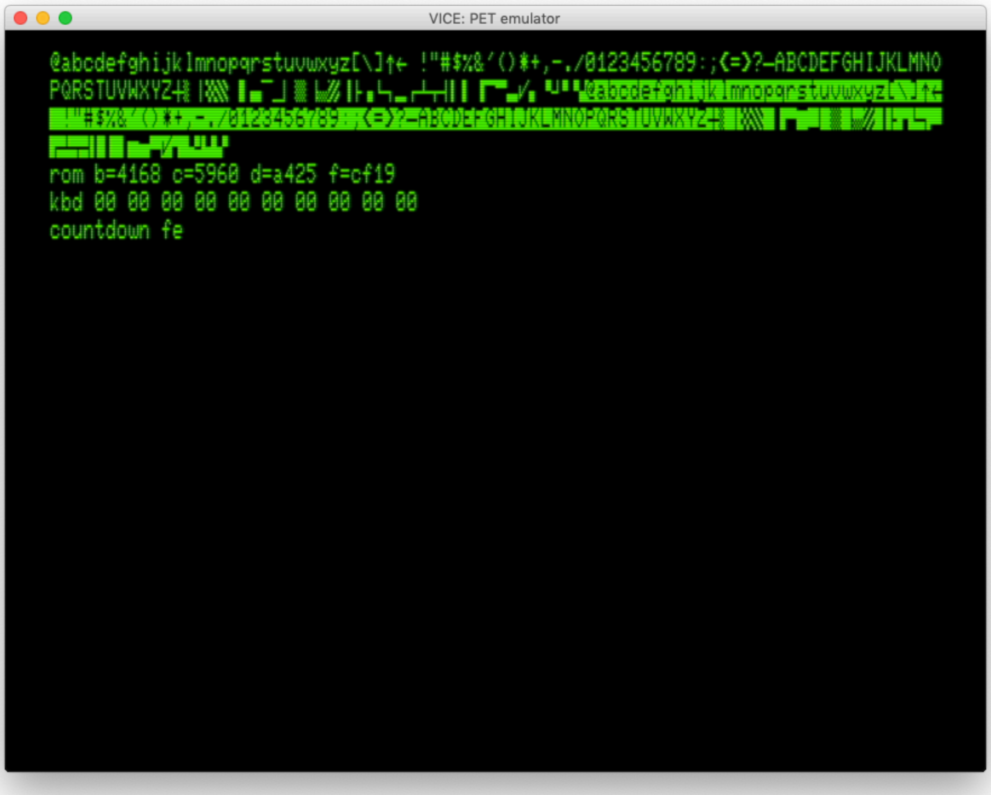
You will find some of the PET keyboard matrices described on the website http://www.6502.org/users/andre/petindex/keyboards.html.

If you look at this website, you will notice that for each different keyboard layout there are exactly ten ‘rows’ (numbered 0 to 9) and eight ‘columns’ (numbered 7 to 0). On my diagnostic screen display there are ten 2-digit hexadecimal numbers. Each of the ten separate bytes on the VDU display map directly onto the ‘rows’ of the keyboard matrix whilst the 8 bits of the displayed hexadecimal byte map directly onto the ‘columns’ of the keyboard matrix. Given this detail, it should be possible to deduce which row and column correspond to each keyboard key – and which of the 10 numbers and bit of my display should correspond. However, it is simpler just to press each key on the keyboard and ensure that you can see a single bit change…

The last line of the display consists of a counter counting down in hexadecimal from $FF to $00. The diagnostic firmware continuously calculates the ROM checksums and displays the key matrix whilst the counter is counting down. Once the counter reaches $00 the test finishes and the full memory test automatically starts.

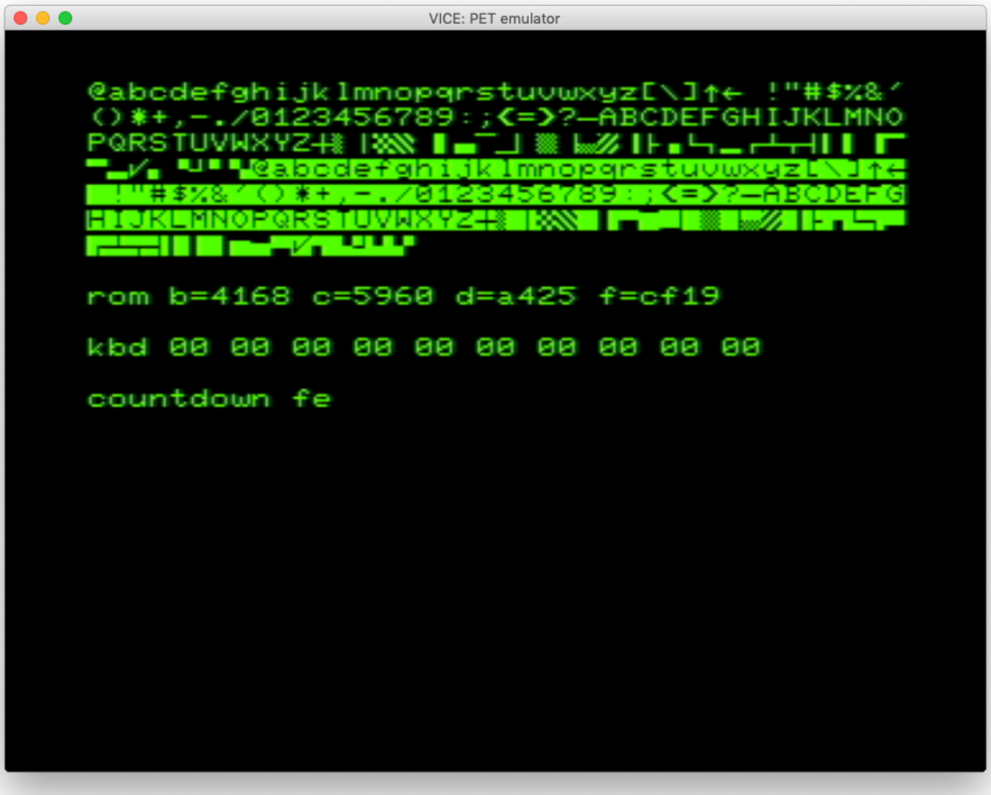
ROMs that occasionally misread can result in the checksum display changing as the test proceeds. Look for signs of the checksums changing.

Technically, this test has no PASS/FAIL criteria. It is up to the user to interpret the results appropriately.



```
@abcdefghijklmnopqrstuvwxyz[\]↑← !"#$%&'()*+,-./0123456789:;<=>?—ABCDEFGHIJKLMNO
PQRSTUVWXYZ
rom b=4168 c=5960 d=a425 f=cf19
kbd 00 00 00 00 00 00 00 00 00 00
countdown fe
```

Figure 7 – The display for the ROM/KBD test (80-columns).



```
@abcdefghijklmnopqrstuvwxyz[\]↑← !"#$%&'
()*+,-./0123456789:;<=>?—ABCDEFGHIJKLMNO
PQRSTUVWXYZ

rom b=4168 c=5960 d=a425 f=cf19

kbd 00 00 00 00 00 00 00 00 00 00

countdown fe
```

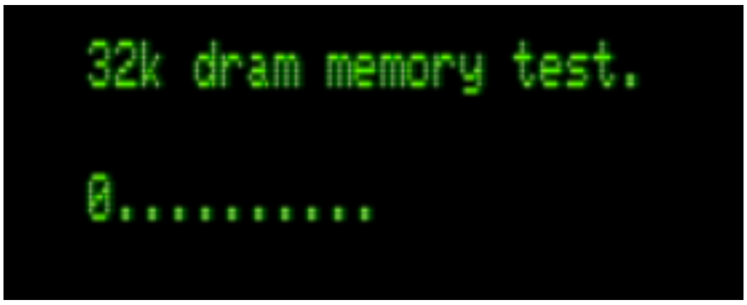
Figure 8 – The display for the ROM/KBD test (40-columns).

MEMORY TEST

Assuming all the above tests worked OK, the firmware then attempts a comprehensive test of the Dynamic RAM (DRAM) within the machine. The firmware has already tested out pages 0 and 1 of the DRAM (addresses $0000 to $01FF inclusively) – albeit in a relatively simple manner – but will now test memory from $0200 upwards fairly exhaustively.

Note that in order to address memory using 16-bit pointers, some memory locations within page 0 ($0000 to $00FF) have to be used as an indirect pointer. In addition, to make the memory test code easier to write, subroutines are utilised. The stack is stored in page 1 ($0100 to $01FF). As a result of the previous simple memory tests on pages 0 and 1, it is just possible that the memory test will 'crash' during operation. This would be a 'revealed failure', as the memory test program outputs status information to the screen as the tests proceed – and everything will appear to 'freeze' if the CPU crashes. On a 32K 8032, each individual test or sub-test should take no longer than 30 seconds; so no activity for a period of 1 minute would indicate that the CPU has crashed (most likely due to a memory fault in either pages 0 or 1 (i.e. the lower bank of memory fitted to the PET).

The first thing the memory test program does is to test how much memory is in the machine. It does this in a very, very simple way by testing one memory byte at the 4K ($0FFF), 8K ($1FFF), 16K ($3FFF) and 32K ($7FFF) memory boundaries and displays the result at the top of the memory test screen at the conclusion:



```
32k dram memory test.

0..........
```

Figure 9 - Starting the DRAM Memory test.

The firmware uses the test values of $55 and $AA to ascertain if RAM memory is present or not.

Commodore use 1 bank of 4K DRAM chips in a 4K PET; 2 banks of 4K DRAM chips in an 8K PET; 1 bank of 16K DRAM chips in a 16K PET and 2 banks of 16K DRAM chips in a 32K PET. Look at what DRAM devices are fitted to the PET and how many banks are physically fitted to understand what the maximum complement of memory should be.

If the indicated amount of memory is not correct, take this as a test failure!

The next thing the firmware does is to perform the memory tests themselves. The tests range from relatively simple to fairly complex – with a corresponding increase in the time taken to execute each test. Obviously, the larger the PET memory is to test, and the more complex each test is to perform, this can lead to a significant time taken to execute the tests themselves.

As can be seen in the image above, the tests to be undertaken in one pass of the memory are indicated by a number and a series of dots. The number indicates the test that is currently being performed, whilst the dots indicate the tests yet to be performed. The last test has sub-tests. I will explain this indication later.

Note that the memory is treated as a series of bits not bytes. Obviously, the bits are packed as 8 bits to the byte. There is nothing 'magic' about this; it is just how the memory tests work.

<u>Test 0 (0..........)</u>

This test first fills all of the memory bits with a '0' and then tests all of the memory bits for the presence of a '0'. This test will detect any memory bit stuck at '1'.

<u>Test 1 (01.........)</u>

This test first fills all of the memory bits with a ‘1’ and then tests all of the memory bits for the presence of a ‘1’. This test will detect any memory bit stuck at ‘0’.

<u>Test 2 (012........)</u>

This test first fills all of the memory bits with an alternating pattern of ‘0’ and ‘1’ bits and then tests all of the memory bits for the presence of the alternating pattern of ‘0’ and ‘1’ bits.

<u>Test 3 (0123.......)</u>

This test implements a memory test known as MARCH-C. It consists of seven (7) sub-tests numbered 0 through 6. These sub-tests are indicated by turning the dots from 0 through 6 in sequence.

I am not going to bore you with the details of the MARCH-C algorithm, but this can be looked up on-line if you wish. Note that there are a number of ‘flavours’ of the MARCH-C algorithm.

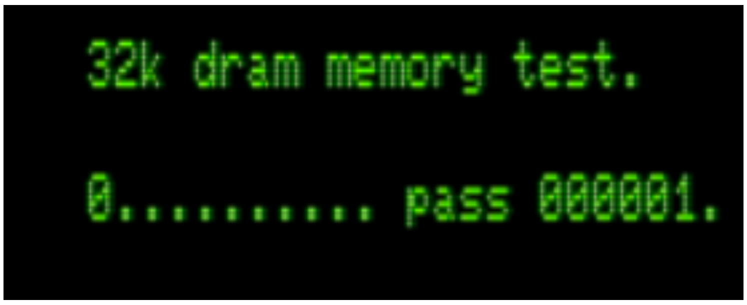
For further reference see the papers at:

http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.461.3754&rep=rep1&type=pdf

http://www.ee.ncu.edu.tw/~jfli/test1/lecture/ch07

The question you are thinking is “what does a memory test PASS and FAIL look like” isn’t it!

Let’s look at the display for a memory test PASS:



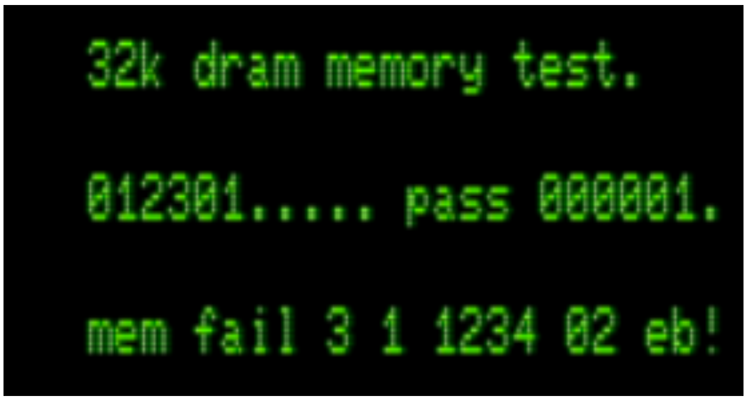
```
32k dram memory test.

0.......... pass 000001.
```

Figure 10 – The first good memory pass.

A count of the total number of successful passes is displayed. Note that this is a 24-bit (3 byte) count – but in hexadecimal not decimal (sorry, I was feeling lazy). The memory tests will keep cycling round and round forever… My suggestion is to leave the test running for a significant period of time (otherwise intermittent faults may be missed).

What about a test FAIL:



```
32k dram memory test.

012301..... pass 000001.

mem fail 3 1 1234 02 eb!
```

Figure 11 - A DRAM Failure report.

This is not a real failure (I bodged it for documentation purposes) but the principle of the error message diagnosis is the same!

After detecting a single memory failure, the memory testing stops. The message consists of the following parts:

- 3 – This is the test number that failed. You can safely ignore this.
- 1 – This is the sub-test number that failed. You can safely ignore this.
- 1234 – This is the hexadecimal address of the memory byte that failed the test.
- 02 – This is the hexadecimal value of a bitmask. One (and only one) bit should be set in this bitmask indicating the bit within the memory byte that failed.
- eb – This is the memory byte value (in hexadecimal) that was read from the memory address that failed.

How do we use these codes to identify the potentially faulty memory device?

My example will be for a 'bog-standard' 32K 8032 PET. See the example schematic at:

http://www.zimmers.net/anonftp/pub/cbm/schematics/computers/pet/8032/8032029-05.gif

From the schematics for the 8032 PET; DRAM devices UA5, UA7, UA9, UA11, UA13, UA15, UA17 and UA19 form the first 16K memory bank (addresses $0000 to $3FFF) whilst DRAM devices UA4, UA6, UA8, UA10, UA12, UA14, UA16 and UA18 form the second 16K memory bank (addresses $4000 to $7FFF).

You can differentiate the two banks from each other by the fact that the first (lower) bank has a signal /CAS0 wired to the DRAM devices, whilst the second (higher) bank has a signal /CAS1 wired to the DRAM devices.

You should also notice which of the DRAM devices are wired to the data bus lines. One DRAM device from each bank is wired to each CPU data line:

In tabular form:

	D7	D6	D5	D4	D3	D2	D1	D0
UPPER	UA4	UA6	UA8	UA10	UA12	UA14	UA16	UA18
LOWER	UA5	UA7	UA9	UA11	UA13	UA15	UA17	UA19

Table 2 - DRAM Allocation for a 'standard' 8032 PET.

So, from our example above:

- The faulting address is $1234.
- The faulting bitmask is $02.
- The faulting data byte is $eb.

The faulting address of $1234 tells us that the fault is in the first (**lower**) bank of 16K memory (as this covers the address range from $0000 to $3FFF).

The faulting bitmask is $02 which (in binary) is 0000_0010 indicating that the faulting bit is **D1** (the leftmost bit being D7 and the rightmost bit being D0).

The potentially faulty DRAM IC is therefore UA17 (from the above table at the intersection between 'LOWER' and 'D1'). Simples ☺!

You can create the table above for any PET from the appropriate schematic.

However, I wouldn't immediately change this device… I would RESET the PET and perform a number of memory tests to further confirm whether it is one specific DRAM device or potentially spread across more devices – either indicating multiple DRAM device failures OR the data bus buffers etc.

We haven't used the faulting data byte ($eb) yet though? Technically, we don't need to! You would, however, use it as follows:

The faulting data byte is $eb which (in binary) is 1110_1011.

The faulting bitmask (in binary) was 0000_0010.

Extracting the indicated bit from the byte value (1110_10**1**1) gives us a value of '1'. We know this is wrong (as the test failed) so the bit value **should have been** a '0' for the test to pass (the opposite of what was actually found).

I hope you find this test utility useful. Enjoy ☺!

END