# Test RAM For Bad Bits, Nondestructively

Leo J. Scanlon
Inverness, FL

In a recent article in this magazine (**COMPUTE!**, April, 1981 #23) I presented a 6502 assembly language program that tests the integrity of a selected portion of RAM. That program was designed to detect "dead" bits or bytes, pattern sensitivity, crosstalk, and a variety of other error conditions. It could also be used to detect soft errors, in which the memory accepts the test data, but reverts back to its previous state after some period of time.

As useful as it is, that program has one possible shortcoming: it clobbers the contents of the portion of memory being tested. Clearly, that doesn't matter if you are just verifying a newly installed memory board, but is unacceptable if a program or some data is sitting within the test area. In this article, I present another kind of program, one that performs a *nondestructive* test on RAM memory. That is, a program that alters memory, but subsequently restores all locations to their previous (pretest) values.

## The Test Algorithm

Essentially, the test program described here validates RAM by comparing the actual contents of memory to the known data that should be contained within it. To make this comparison, the program uses a method that is often employed for testing punched paper tape and read only memories (ROMs) – the *checksum*. A checksum is that value produced by taking the exclusive-OR of all bytes in test memory (see box).

Briefly, here is the sequence of operations for the test program:

1. Calculate a checksum value for the entire range of test memory, by exclusive-ORing all bytes.
2. Invert the state of the first bit in test memory – Bit 7 of the "start" location – but leave all other bits unchanged.
3. Calculate a new checksum value.
4. Invert the state of the altered bit position in the new checksum.
5. Compare the new (altered) checksum with the initial checksum.
6. The result of this comparison can cause either of two things to take place:

   *If the checksums are different,* the program jumps to an error routine, to print out the bit position and address of the bad bit.

   *If the checksums are identical,* the program restores the state of the test bit – by reinverting it – then branches back to Step 2, to test the next bit (Bit 6 of the "start" location).

This process continues until all bits have been tested, or until a mismatch is detected.

Will this nondestructive test program catch all of the fault conditions that can be detected by the previously published destructive test program? Probably not all of them. The nondestructive test program will not detect pattern sensitivity or soft errors (unless you modify the program to include a time delay), but it should be able to detect most other types of errors.

## Program Flowchart

Now that you understand *what* the test program must do, and know *how* the program will do it, it's time to look at the structure of the program itself. This program is comprised of three parts: a main program loop, a checksum calculating subroutine and an error printout routine.

A flowchart for the main program loop is shown in Figure 1. As you can see, this flowchart is nothing more than a detailed version of the algorithm we defined in preceding Steps 1 through 6. The program begins by calculating the byte count, then calls the checksum subroutine (CHKSUM) to generate the initial value of the checksum. This done, the base address and byte index are initialized to reference the first byte in test memory.

Next, the bit mask index is initialized to reference the most significant bit, Bit 7. With this initialization out of the way, the program inverts the current test bit. The first time through the loop, this will be Bit 7 of the Start location. Now the

**Figure 1: Nondestructive Memory Test Program**

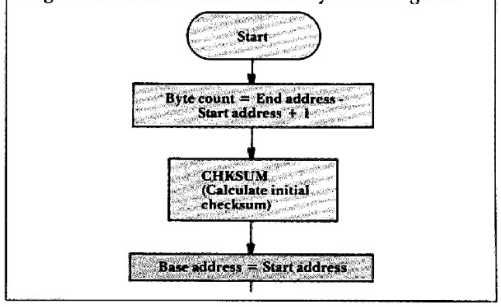
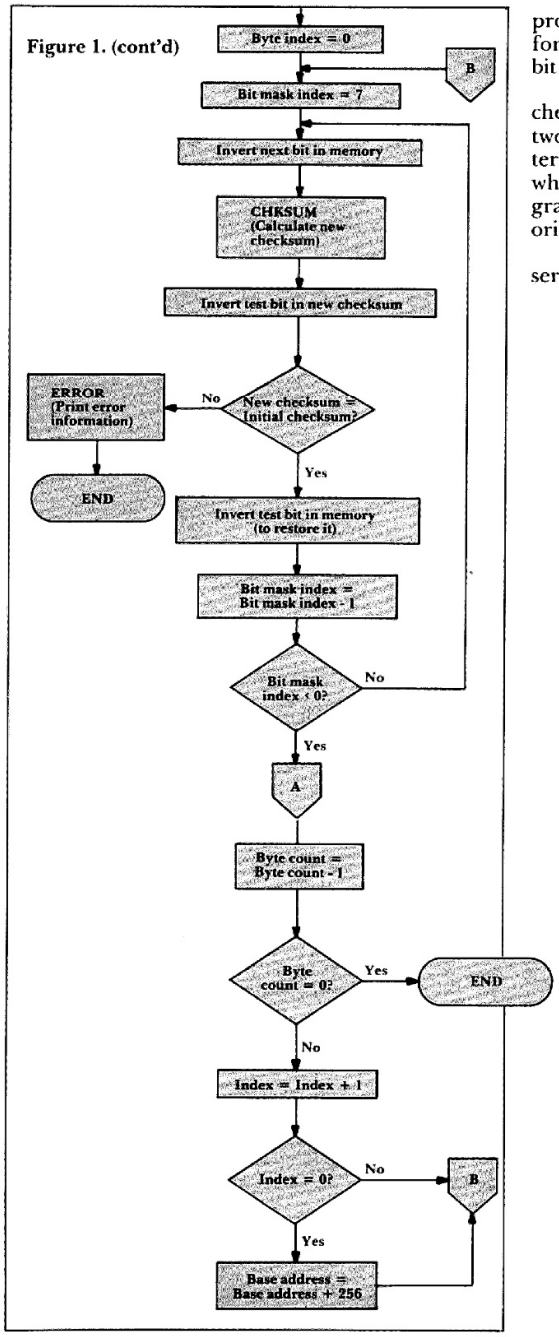


Figure 1. (cont'd)

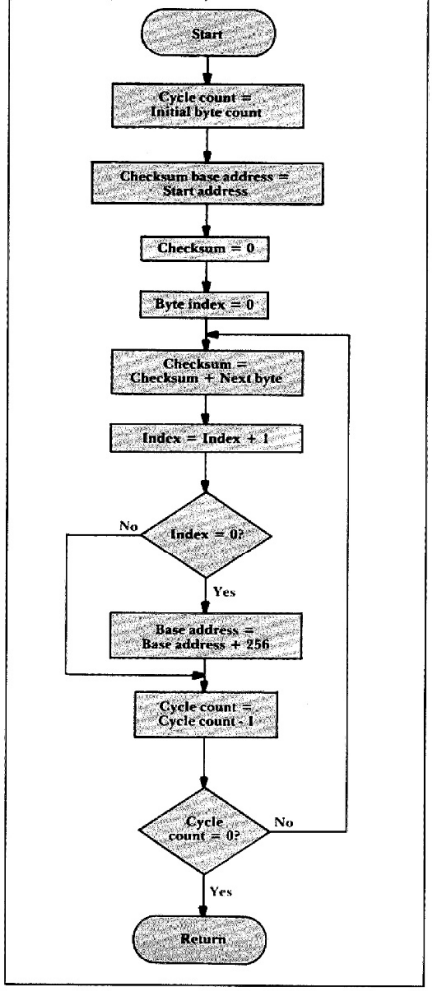



program calls CHKSUM again, to get the checksum for memory with one bit inverted, and inverts that bit position in the checksum.

This invert operation should make the new checksum identical to the initial checksum. If the two checksums are not identical, the program terminates by printing the bit position and address where the error was detected. Otherwise, the program reinverts the current test bit, to restore its original state.

The remainder of the program involves a series of three counter/index adjustment opera-

**Figure 2: Checksum-Calculating Subroutine (CHKSUM)**



tions, with each followed by a branch/no-branch decision. In the first of these operations, the bit mask index is decremented; if it is nonnegative, the program branches back to invert the next bit. Otherwise, the byte count is decremented; if all bytes have been tested, the program terminates, error free. Otherwise, the byte index is incremented. The byte index is eight bits long, and can hold values from 0 to 255 (decimal). If the incrementation caused the byte index to overflow to zero, the program increments the high order byte of the base address, then branches back to reinitialize the bit mask index. Otherwise, the branch takes place with no change to the base address.

Figure 2 shows the flowchart for the checksum subroutine, CHKSUM. This subroutine is called from two places in the program: (A) it is called at the beginning of the program, to calculate the initial checksum, and (B) it is called from within the main loop, to calculate a new checksum after a test bit has been inverted. This second source of call requires the subroutine to maintain its own, separate byte count and base address, so as not to disturb the current values of these parameters in the main program. In the flowchart, these "working" parameters are labeled *cycle count* and *checksum base address*, respectively.

To start, cycle count is set equal to initial byte count, checksum base address is set equal to test start address, and the checksum and byte index are initialized to zero. The rest of the subroutine is just one big loop. In this loop, the checksum is accumulated, byte by byte, with intervening index and cycle count adjustments. The loop is terminated when all bytes have been processed; that is, when cycle count has been decremented to zero.

### The Test Program

Now that you understand the criteria of the program and its sequences, we can look at the program itself. Program 1 shows the source code for the nondestructive test program, which was flowcharted in Figure 1. Note that before executing the program, the starting address must be stored in locations 00 and 01 (00 holds low byte) and the ending address must be stored in locations 02 and 03 (02 holds low byte).

Besides these four locations, the program uses 13 other zero page locations, as working storage. These include six parameters that are used in the main program – initial byte count (IBYTES), byte count (BYTES), base address (BADDR), initial checksum (CSUM) and temporary storage for the X and Y registers (SAVEX and SAVEY), and two parameters that are used in the checksum subroutine, a working copy of the byte count (CYCLES) and a checksum base address (CBADDR). Of these parameters, only IBYTES and CSUM remain unchanged throughout the program; all six other parameters will change during execution.

Following these reserve equates come three equates that reference subroutines in the AIM 65 monitor: CRLOW initializes the display and printer to their START positions; NUMA prints the contents of the accumulator, as two ASCII digits; OUTPRI sends one character to the print buffer. Other 6502-based computers have equivalent subroutines.

The actual code that follows is straightforward, so you should have no problem following it if you studied the flowchart in Figure 1. Some readers may wonder why I chose to save X and Y in zero page (locations SAVEX and SAVEY), rather than on the stack, during the call to CHKSUM in the main loop. There are two reasons why this was done:

1. The instructions used to save X and Y in zero page execute eight cycles faster than those to save X and Y on the stack (12 cycles versus 20 cycles). If you consider that for each byte tested, CHKSUM is called eight times – once for each bit position – saving X and Y in zero page saves 64N microseconds for an N-byte test run.

2. We need to use the checksum contents of the accumulator upon return from CHKSUM, and a pull from the stack (PLA) always loads the stack information into the accumulator. If the 6502 had the instructions PHX, PHY, PLX and PLY, the stack would have been the likely place to hold X and Y, but unfortunately it has no such instructions.

Programmers may also be interested in the way the bit masks are accessed by the EOR BMASK,X instructions that follow the labels INVERT and NXTBIT. The bit mask table, BMASK (shown at the end of Program 2), is arranged by *ascending* bit position. That is, the mask for Bit 0 comes first, followed by the mask for Bit 1, and so on. However, this table is accessed in *descending* order; Bit 7 is tested first and Bit 0 is tested last. This allows us to initialize the bit mask index to 7 (LDX #7 at label IBMSK), then decrement this index until it goes negative. Otherwise, working with a descending table and an incrementing index, the program would have to include a CPX #8 instruction to make the done/not done branch decision. By using the ascending table and decrementing index approach we've eliminated that compare instruction. Since the CPX #8 instruction executes in just two cycles, the difference in approaches is not significant, but the backwards access is a handy gimmick for your programming bag of tricks.

Program 2 shows the code for the checksum calculating subroutine, CHKSUM, which was flowcharted in Figure 2. It follows the flowchart closely, and needs no additional explanation. Program 2 also includes the previously mentioned bit mask table, BMASK, and the text for the error message.

This program will produce one of two messages. If the test memory is error free, the message *OKAY!* will be printed, otherwise an error message of the form *BIT n OF LOC. aaaa* will be printed. In the error message, the bit position and address that are printed identify the bit that was being tested when the checksum mismatch occurred. It's possible, of course, that inverting that bit actually caused some other bit in the memory to be inverted, due to crosstalk, so the printout position may not be the actual culprit. One way of finding out is to run a second test, starting at the location following the printout location; that is, rerun the test starting at "aaaa + 1."

## Execution Times For The Test Program

As you can see from the listings, the program occupies slightly less than a page of memory; to be exact, it occupies 245 bytes. Of even greater significance, however, is the amount of time it takes to execute. That is, the amount of time it takes to test a selected portion of memory. In a test that I ran, the program took just over four minutes to check out a 1K portion of memory (1024 bytes).

At first I suspected that something was wrong with the program, but after a few calculations I became convinced that this was indeed a respectable time, in light of what the program was doing. First, consider that in a 1K byte test, the CHKSUM subroutine is called 8193 times; once to get the initial checksum, then once more for each of the 8192 bit positions in the 1024 byte test memory. The CHKSUM subroutine takes 28 + (29 x N) cycles to calculate the checksum for an N-byte memory, so it takes 29,724 cycles (microseconds) for a 1024 byte memory. Cranking out the math, we find that with

## Exclusive-ORs And Checksums

An exclusive-OR is a logical operation in which two byte operands are combined to produce a result byte with these characteristics:

- For each bit position in which the operands are different (one is logic 0, the other is logic 1), the result will contain a logic 1.
- For each bit position in which the operands are the same (both logic 0 or both logic 1), the result will contain a logic 0.

These rules can be summarized as follows:

| Bit Operand #1 | Bit Operand #2 | Result Bit |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

All of the popular 8-bit microprocessors have an exclusive-OR instruction. In the 6502, it has the mnemonic *EOR*. The EOR instruction operates on the contents of the accumulator with an immediate value or a value in memory, and leaves the result in the accumulator.

For example, if the accumulator contains the value $AB (where $ denotes hexadecimal) and location $40 contains the value $0F, the instruction *EOR $40* will produce a value of $A4 in the accumulator. The binary arithmetic looks like this:

```
  0000 1111    Contents of location $40 = $0F
⊕ 1010 1011    Contents of accumulator = $AB
  1010 0100    Result in accumulator = $A4
```

Note what has happened here. The value $0F in location $40 has caused the four low order bits (0 through 3) to be *inverted*, but has left the four high order bits (4 through 7) intact.

This shows one of the primary uses for the EOR instruction: to invert some selected bits, but leave all other bits unchanged. In fact, the test program in this article uses the EOR instruction to invert a single bit in memory, by reading the appropriate memory byte into the accumulator, then exclusive-ORing it with a "mask" value that has just one bit set to logic 1. To invert Bit 7, the program applies a mask value of $10000000_2$ ($80); to invert Bit 6, the program applies a mask value of $01000000_2$ ($40); and so on.

The program in this article also uses a series of EOR instructions to calculate a *checksum* value. As mentioned in the article, the checksum is the exclusive-OR of all bytes being tested. For example, if locations $0400, $0401 and $0402 are being tested, the program will perform this type of operation:

```
  0010 1101    ($0400) = $2D
  1010 0011    ($0401) = $A3
⊕ 0001 1000    ($0402) = $18
  1001 0110    Checksum = $96
```

8193 calls, the program spends about 4.06 minutes in the CHKSUM subroutine!

Since the program is spending virtually all of its time in the CHKSUM subroutine, the total execution time of the program is directly dependent on the efficiency of this subroutine. If any readers have suggestions on how to streamline CHKSUM, I'd be happy to hear from them.

**Program 1: Source Code for Nondestructive Test Program**

```
 LINE#   ADDR   OBJECT      LABEL  SOURCE            PAGE 0001

01-0010  2000               ; THIS PROGRAM PERFORMS A NONDESTRUCTIVE TEST
01-0020  2000               ; ON RAM MEMORY, BY CALCULATING A SERIES OF CHECKSUMS.
01-0030  2000               ; BEFORE EXECUTING, STORE THE STARTING ADDRESS
01-0040  2000               ;  AT LOCS. 00 AND 01, AND THE ENDING ADDRESS
01-0050  2000               ;  AT LOCS. 02 AND 03.
01-0060  2000               ; IF THE TEST IS SUCCESSFUL, AN 'OKAY!' MESSAGE
01-0070  2000               ; IS PRINTED.  OTHERWISE, THE BAD BIT POSITION
01-0080  2000               ; AND ADDRESS ARE PRINTED.

01-0100  2000               ; USER-SUPPLIED PARAMTERS

01-0120  2000                      *=0
01-0130  0000               START  *=*+2             ; STARTING ADDRESS
01-0140  0002               END    *=*+2             ; ENDING ADDRESS

01-0160  0004               ; EQUATES FOR WORKING STORAGE IN ZERO PAGE

01-0180  0004               IBYTES *=*+2             ; INITIAL BYTE COUNT
01-0190  0006               BYTES  *=*+2             ; BYTE COUNT
01-0200  0008               CYCLES *=*+2             ; WORKING COPY OF BYTES
01-0210  000A               BADDR  *=*+2             ; BASE ADDRESS
01-0220  000C               CBADDR *=*+2             ; BASE ADDRESS FOR CHECKSUM SUBR.
01-0230  000E               CSUM   *=*+1             ; INITIAL CHECKSUM
01-0240  000F               SAVEX  *=*+1             ; TEMP. STORAGE FOR X REGISTER
01-0250  0010               SAVEY  *=*+1             ; TEMP. STORAGE FOR Y REGISTER

01-0270  0011               ; AIM 65 MONITOR SUBROUTINES

01-0290  0011               CRLOW  =$EA13            ; RESET DISPLAY & PRINTER
01-0300  0011               NUMA   =$EA46            ; PRINT A, AS TWO ASCII CHARS.
01-0310  0011               OUTPRI =$F000            ; OUTPUT A TO PRINT BUFFER

01-0330  0011                      *=$200
01-0340  0200  38                  SEC              ; BYTE COUNT = END ADDR. - START
                                                      ADDR. + 1
01-0350  0201  A5 02               LDA END
01-0360  0203  E5 00               SBC START
01-0370  0205  85 04               STA IBYTES
01-0380  0207  85 06               STA BYTES
01-0390  0209  A5 03               LDA END+1
01-0400  020B  E5 01               SBC START+1
01-0410  020D  85 05               STA IBYTES+1
01-0420  020F  85 07               STA BYTES+1
01-0430  0211  E6 04               INC IBYTES
01-0440  0213  E6 06               INC BYTES
01-0450  0215  D0 04               BNE GETSUM
01-0460  0217  E6 05               INC IBYTES+1
01-0470  0219  E6 07               INC BYTES+1
01-0480  021B  20 B1 02    GETSUM  JSR CHKSUM       ; CALCULATE INITIAL CHECKSUM
01-0490  021E  85 0E               STA CSUM         ;  AND SAVE IT IN MEMORY
01-0500  0220  A5 00               LDA START        ; BASE ADDRESS = START ADDRESS
01-0510  0222  85 0A               STA BADDR
01-0520  0224  A5 01               LDA START+1
01-0530  0226  85 0B               STA BADDR+1
01-0540  0228  A0 00               LDY #0           ; BYTE INDEX = 0
01-0550  022A  A2 07       IBMSK   LDX #7           ; BIT MASK INDEX = 7
```

```
01-0560  022C  B1 0A       INVERT LDA (BADDR),Y    ; INVERT NEXT BIT IN MEMORY
01-0570  022E  5D DF 02           EOR BMASK,X
01-0580  0231  91 0A              STA (BADDR),Y
01-0590  0233  86 0F              STX SAVEX         ; SAVE X AND Y IN MEMORY
01-0600  0235  84 10              STY SAVEY
01-0610  0237  20 B1 02           JSR CHKSUM        ; CALCULATE NEW CHECKSUM
01-0620  023A  A6 0F              LDX SAVEX         ; RETRIEVE X AND Y
01-0630  023C  A4 10              LDY SAVEY
01-0640  023E  5D DF 02           EOR BMASK,X       ; INVERT TEST BIT IN NEW CHECKSUM
01-0650  0241  C5 0E              CMP CSUM          ; NEW CHECKSUM = INITIAL CHECKSUM?
01-0660  0243  D0 39              BNE ERROR         ;  NO.  PRINT ERROR INFO.
01-0670  0245  B1 0A       NXTBIT LDA (BADDR),Y    ;  YES.  INVERT TEST BIT IN MEMORY
01-0680  0247  5D DF 02           EOR BMASK,X
01-0690  024A  91 0A              STA (BADDR),Y
01-0700  024C  CA                 DEX               ;  NO.  DECREMENT BIT MASK INDEX
01-0710  024D  10 DD              BPL INVERT        ; BIT MASK INDEX NEGATIVE?
01-0720  024F  C6 06              DEC BYTES         ;  YES.  DECREMENT BYTE COUNT
01-0730  0251  E4 06              CPX BYTES
01-0740  0253  D0 02              BNE BCNT0
01-0750  0255  C6 07              DEC BYTES+1
01-0760  0257  A6 06       BCNT0  LDX BYTES         ; BYTE COUNT = 0?
01-0770  0259  D0 1B              BNE INCIDX
01-0780  025B  A6 07              LDX BYTES+1
01-0790  025D  D0 17              BNE INCIDX
01-0800  025F  A0 00              LDY #0            ;  YES.  ALL DONE, WITH NO ERRORS
01-0810  0261  B9 70 02    OKLOOP LDA OKMSG,Y
01-0820  0264  20 00 F0           JSR OUTPRI
01-0830  0267  C8                 INY
01-0840  0268  C0 06              CPY #6
01-0850  026A  D0 F5              BNE OKLOOP
01-0860  026C  20 13 EA           JSR CRLOW
01-0870  026F  00                 BRK

01-0890  0270  20 4F       OKMSG  .BYT ' OKAY!'

01-0910  0276  C8          INCIDX INY               ;  NO.  INCREMENT BYTE INDEX
01-0920  0277  D0 B1              BNE IBMSK         ; BYTE INDEX=0?
01-0930  0279  E6 0B              INC BADDR+1       ;  YES.  ADD 256 TO BASE ADDRESS
01-0940  027B  4C 2A 02           JMP IBMSK
01-0950  027E

01-0970  027E              ; THIS ROUTINE PRINTS OUT THE BIT POSITION
01-0980  027E              ; AND ADDRESS AT WHICH THE MISMATCH OCCURRED

01-1000  027E  20 13 EA    ERROR  JSR CRLOW         ; RESET DISPLAY & PRINTER
01-1010  0281  A0 00              LDY #0            ; PRINT FIRST PART OF TEXT
01-1020  0283  B9 E7 02    LOOP1  LDA EMSG,Y
01-1030  0286  20 00 F0           JSR OUTPRI
01-1040  0289  C8                 INY
01-1050  028A  C0 05              CPY #5
01-1060  028C  D0 F5              BNE LOOP1
01-1070  028E  8A                 TXA               ; PRINT BIT PATTERN
01-1080  028F  09 30              ORA #$30
01-1090  0291  20 00 F0           JSR OUTPRI
01-1100  0294  B9 E7 02    LOOP2  LDA EMSG,Y        ; PRINT SECOND PART OF TEXT
01-1110  0297  20 00 F0           JSR OUTPRI
01-1120  029A  C8                 INY
01-1130  029B  C0 0E              CPY #14
01-1140  029D  D0 F5              BNE LOOP2
01-1150  029F  A5 10              LDA SAVEY         ; ERROR ADDRESS = BASE ADDRESS +
                                                      INDEX
01-1160  02A1  18                 CLC
01-1170  02A2  65 0A              ADC BADDR
01-1180  02A4  48                 PHA
01-1190  02A5  A9 00              LDA #0
01-1200  02A7  65 0B              ADC BADDR+1
01-1210  02A9  20 46 EA           JSR NUMA          ; PRINT ERROR ADDRESS
```

```
01-1220  02AC  68              PLA
01-1230  02AD  20 46 EA        JSR NUMA
01-1240  02B0  00              BRK              ; RETURN TO MONITOR
```

**Program 2: Source Code for CHKSUM Subroutine**

```
 LINE#   ADDR   OBJECT     LABEL   SOURCE          PAGE 0004

01-1260  02B1             ; THIS SUBROUTINE ACCUMULATES THE CHECKSUM,
01-1270  02B1             ; BY EXCLUSIVE-ORING ALL BYTES

01-1290  02B1  A5 04      CHKSUM LDA IBYTES       ; CYCLE COUNT = BYTE COUNT
01-1300  02B3  85 08             STA CYCLES
01-1310  02B5  A5 05             LDA IBYTES+1
01-1320  02B7  85 09             STA CYCLES+1
01-1330  02B9  A5 00             LDA START        ; BASE ADDRESS = START ADDRESS
01-1340  02BB  85 0C             STA CBADDR
01-1350  02BD  A5 01             LDA START+1
01-1360  02BF  85 0D             STA CBADDR+1
01-1370  02C1  A9 00             LDA #0           ; CHECKSUM = 0
01-1380  02C3  A0 00             LDY #0           ; BYTE INDEX = 0
01-1390  02C5  51 0C      ACCUM  EOR (CBADDR),Y   ; CHECKSUM = CHECKSUM EOR NEXT BYTE
01-1400  02C7  C8                INY              ; INCREMENT INDEX
01-1410  02C8  D0 02             BNE DECCYC       ; INDEX = 0?
01-1420  02CA  E6 0D             INC CBADDR+1     ;  YES.  ADD 256 TO BASE ADDRESS
01-1430  02CC  A2 FF      DECCYC LDX #$FF         ;  NO.  DECREMENT CYCLE COUNT
01-1440  02CE  C6 08             DEC CYCLES
01-1450  02D0  E4 08             CPX CYCLES
01-1460  02D2  D0 02             BNE CYCZ
01-1470  02D4  C6 09             DEC CYCLES+1
01-1480  02D6  A6 08      CYCZ   LDX CYCLES       ; CYCLE COUNT = 0?
01-1490  02D8  D0 EB             BNE ACCUM        ;  NO.  GO PROCESS NEXT BYTE
01-1500  02DA  A6 09             LDX CYCLES+1
01-1510  02DC  D0 E7             BNE ACCUM
01-1520  02DE  60                RTS              ;  YES.  RETURN WITH CHECKSUM IN A


01-1550  02DF             ; MASKS USED TO INVERT BITS IN MEMORY

01-1570  02DF  01         BMASK  .BYT 1,2,4,8,$10,$20,$40,$80
01-1570  02E0  02
01-1570  02E1  04
01-1570  02E2  08
01-1570  02E3  10
01-1570  02E4  20
01-1570  02E5  40
01-1570  02E6  80

01-1590  02E7             ; ERROR MESSAGE TEXT

01-1610  02E7  20 42      EMSG   .BYT ' BIT '
01-1620  02EC  20 4F             .BYT ' OF LOC. '
01-1630  02F5                    .END

ERRORS = 0000

END OF ASSEMBLY = 02F4
:
```

# Universal 6502 Memory Test

Carl W. Moser

This article contains a memory test program which tests RAM memory in various 6502 based systems. This test was developed after using several tests which did not perform a complete test. The problem areas were untested chip selects and address line inputs.

The program performs two tests:

Test 1: Tests memory cells for storage retention, and open, shorted, or non-functioning data and Ao-An address lines. This is done by writing 00 01 1 ... FF 00 01 1 ... FF continually throughout the memory range for the first pass. When this has been written, it is checked to validate the data. On the next pass 01 02 ... FF 00 01 1 ... FF is written and checked. This continues for 256 (hex FF) passes until all possible combinations of bit patterns have been used.

Test 2: Tests the RAM chip select inputs. This is the same as test 1 except data 00 01 ... F2 00 01 ... F2 is used. The purpose of this test is to test the remaining $A_8$-$A_{15}$ address lines. Listings 1 (originating at memory address $0002) and 2 (originating at $0800) contain the source of the memory test program. The reason for these two listings is that not all 6502 microcomputers have RAM at a common address from which the memory test program can execute. To determine which listing is appropriate for your system, consult table A. Next enter the object code from the appropriate listing, and then configure the I/O for your system, also from table A.

Enter the start address and end address of the memory range to be tested as described in table B. Execution begins with test 1 at $0002 for Listing 1 and $0800 for Listing 2.

If an error occurs, it will be outputted in the following format:

| Address | Test Pattern | Error |
|---|---|---|
| xxxx | yy | zz |

Note: This program performs a lengthy but exhaustive test of RAM memory. It takes approximately 38 seconds per 1K of memory for each test 1 and test 2.

When test 1 runs to completion, a break instruction will be executed to enter your systems monitor program. Register A will contain E1 indicating end of test 1. To execute test 2, simply continue execution by typing G to your monitor.

If errors occur, they will be of the same form as described above. When test 2 has run to completion, a break instruction will again transfer control to your monitor and register A will contain E2 signifying the end. To continue execution again at test 1, simply type G. The start and end address range is not altered by the memory test program.

If errors occurred in test 2 but not in test 1, you can safely assume a chip select malfunction (possible stuck in enable state or malfunction with circuitry which generates the chip select) or an address line other than $A_9$-$A_7$. Usually a number of errors will occur in test 1 when the fault is a single defective address input, data input, or data output.

If a continuous sequence of addresses with errors occur, the problem is likely to be an open data input or a data output stuck at '1' or '0.'

If errors occur every 2nd, 4th, 8th, 16th or some power of 2 address sequence, check for defective address inputs as follows:

| Data bit with error | Check address input | Data bit with error | Check address input |
|---|---|---|---|
| $D_0$ | $A_0$ or $A_8$ | $D_4$ | $A_4$ or $A_{12}$ |
| $D_1$ | $A_1$ or $A_9$ | $D_5$ | $A_5$ or $A_{13}$ |
| $D_2$ | $A_2$ or $A_{10}$ | $D_6$ | $A_6$ or $A_{14}$ |
| $D_3$ | $A_3$ or $A_{11}$ | $D_7$ | $A_7$ or $A_{15}$ |

If, for example, you are checking 2102's (1x1K) and are specifying a 4K range of memory and an error common to the whole range occurs, the problem is likely to be in the power leads, defective data or address buffers, stuck at '0' address inputs, stuck at '0' data inputs, or stuck at '0' data outputs.

In all of the above, you may have to examine the various memory error patterns for some similarity in order to isolate the defective component. This is especially true of the 1x1K 2102, and 1x16K 4116 memory chips where each chip is devoted to a particular data lead ($D_0$-$D_7$).

## TEST 2

```
0800- A2 00     0802- A2 00
0802- 8E E2 08  0804- 8E E4 00
0805- 20 1B 08  0807- 20 1D 00
0808- A9 E1     080A- A9 E1
080A- 00        080C- 00
080B- EA        080D- EA
080C- EA        080E- EA
080D- EE E2 08  080F- EE E4 00
0810- 20 1B 08  0812- 20 1D 00
0813- A9 E2     0815- A9 E2
0815- 00        0817- 00
0816- EA        0818- EA
0817- EA        0819- EA
0818- 4C 00 08  081A- 4C 02 00
081B- 20 C9 08  081D- 20 CB 00
081E- A0 00     0820- A0 00
0820- A2 00     0822- A2 00
0822- 8E E1 08  0824- 8E E3 00
0825- 4C 2E 08  0827- 4C 30 00
0828- EE E1 08  082A- EE E3 00
082B- D0 01     082D- D0 01
082D- 60        082F- 60
082E- AC E1 08  0830- AC E3 00
0831- 20 9F 08  0833- 20 A1 00
0834- 98        0836- 98
0835- 81 00     0837- 81 00
0837- C1 00     0839- C1 00
0839- F0 03     083B- F0 03
083B- 20 81 08  083D- 20 83 00
083E- 20 6E 08  0840- 20 70 00
0841- F0 06     0843- F0 06
0843- 20 61 08  0845- 20 63 00
0846- 4C 34 08  0848- 4C 36 00
0849- AC E1 08  084B- AC E3 00
084C- 20 9F 08  084E- 20 A1 00
084F- 98        0851- 98
0850- C1 00     0852- C1 00
```

## TEST 1

```
0010 ;       MCS 6502 MEMORY TEST
0080 ;ZERO PAGE LOCATIONS
0090 ADDRS      .DE 0 ;2 BYTES - ADDRESS OF
0100                                  MEMORY
0140            .BA $0002 OR  .BA $0800
0160 MEM<TEST   LDX #$00
0170            STX TEST<TYPE         ; TEST 1
0180            JSR TEST<PGM
0190            LDA #$E1
0200            BRK
0210            NOP
0220            NOP
0230            INC TEST<TYPE         ; TEST 2
0240            JSR TEST<PGM
0250            LDA #$E2
0260            BRK
0270            NOP
0280            NOP
0290            JMP MEM<TEST
0300 ;
0310 TEST<PGM   JSR CRLF
0320            LDY #$00      ; PATTERN REGISTER
0330            LDX #$00
0340            STX TEST<PATRN
0350            JMP NX<PASS
0360 ;
0370 NX<PATRN   INC TEST<PATRN
0380            BNE NX<PASS
0390            RTS
0400 NX<PASS    LDY TEST<PATRN
0410            JSR INI<ADDRS
0420 LOOP1      TYA
0430            STA (ADDRS,X)    ; STORE PATTERN
0440            CMP (ADDRS,X)    ; CHECK
0450            BEQ NO<ERR1
0460            JSR ERROR ; ADDRS, R(A),(ADDRS,X)
0470 NO<ERR1    JSR INC<ADDRSC
0480            BEQ CK<PATRN
0490            JSR INC<RY
0500            JMP LOOP1
0510 ;
0520 CK<PATRN   LDY TEST<PATRN
0530            JSR INI<ADDRS   ; INITIALIZE ADDRS
0540 LOOP2      TYA
0550            CMP (ADDRS,X)
```

```
0852- F0 03     0854- F0 03     0560            BEQ NO<ERR2
0854- 20 81 08  0856- 20 83 00  0570            JSR ERROR   ; ADDRS,R(A),(ADDRS,X)
0857- 20 61 08  0859- 20 63 00  0580 NO<ERR2    JSR INC<RY
085A- 20 6E 08  085C- 20 70 00  0590            JSR INC<ADDRSC
085D- D0 F8     085F- D0 F8     0600            BNE LOOP2
085F- F0 C7     0861- F0 C7     0610            BEQ NX<PATRN
0861- C8        0863- C8        0640 INC<RY     INY
0862- AD E2 08  0864- AD E4 00  0650            LDA TEST<TYPE
0865- F0 06     0867- F0 06     0660            BEQ EXIT1
0867- C0 F3     0869- C0 F3     0670            CPY #$F3  ; RESET R(Y) TO CHECK
0869- 90 02     086B- 90 02     0680            BCC EXIT1           CHIP SELECTS
086B- A0 00     086D- A0 00     0690            LDY #$00
086D- 60        086F- 60        0700 EXIT1      RTS
                                0710 ;
                                0720 ;
086E- E6 00     0870- E6 00     0730 INC<ADDRSC INC *ADDRS
0870- D0 02     0872- D0 02     0740            BNE SKIP<HI
0872- E6 01     0874- E6 01     0750            INC *ADDRS+$01
0874- AD DF 08  0876- AD E1 00  0760 SKIP<HI    LDA END
0877- C5 00     0879- C5 00     0770            CMP *ADDRS
0879- D0 05     087B- D0 05     0780            BNE EXIT2
087B- AD E0 08  087D- AD E2 00  0790            LDA END+$01
087E- C5 01     0880- C5 01     0800            CMP *ADDRS+$01
0880- 60        0882- 60        0810 EXIT2      RTS
                                0840 ;OUTPUT THE ERROR; ADDRESS, PATTERN, ERROR
0881- 48        0883- 48        0850 ERROR      PHA
0882- A5 01     0884- A5 01     0860            LDA *ADDRS+$01
0884- 20 AA 08  0886- 20 AC 00  0870            JSR TBYT     ; OUTPUT ADDRS HI
0887- A5 00     0889- A5 00     0880            LDA *ADDRS
0889- 20 AA 08  088B- 20 AC 00  0890            JSR TBYT     ; OUTPUT ADDRS LO
088C- 20 D4 08  088E- 20 D6 00  0900            JSR SPACE2
088F- 68        0891- 68        0910            PLA
0890- 20 AA 08  0892- 20 AC 00  0920            JSR TBYT     ; OUTPUT PATTERN
0893- 20 D4 08  0895- 20 D6 00  0930            JSR SPACE2
0896- A1 00     0898- A1 00     0940            LDA (ADDRS,X)
0898- 20 AA 08  089A- 20 AC 00  0950            JSR TBYT     ; OUTPUT ERROR IN MEMORY
089B- 20 C9 08  089D- 20 CB 00  0960            JSR CRLF
089E- 60        00A0- 60        0970            RTS
                                0980 ;
                                0990 ;
                                1000 ;INITIALIZE ADDRS WITH START
089F- AD DD 08  00A1- AD DF 00  1010 INI<ADDRS  LDA START
08A2- 85 00     00A4- 85 00     1020            STA *ADDRS
08A4- AD DE 08  00A6- AD E0 00  1030            LDA START+$01
08A7- 85 01     00A9- 85 01     1040            STA *ADDRS+$01
08A9- 60        00AB- 60        1050            RTS
                                1060 ;
                                1070 ;
                                1080 ;ROUTINE TO OUTPUT A BYTE
08AA- 48        00AC- 48        1090 TBYT       PHA
08AB- 4A        00AD- 4A        1100            LSR A
08AC- 4A        00AE- 4A        1110            LSR A
08AD- 4A        00AF- 4A        1120            LSR A
08AE- 4A        00B0- 4A        1130            LSR A
08AF- 20 B3 08  00B1- 20 B5 00  1140            JSR NIBBLE
08B2- 68        00B4- 68        1150            PLA
08B3- 29 0F     00B5- 29 0F     1160 NIBBLE     AND #$0F
08B5- 09 30     00B7- 09 30     1170            ORA #$30
08B7- C9 3A     00B9- C9 3A     1180            CMP #$3A
08B9- 90 02     00BB- 90 02     1190            BCC WRITE
08BB- 69 06     00BD- 69 06     1200            ADC #$06
                                1210
                                1220 ;ROUTINE TO WRITE AN ASCII CHAR.
08BD- 8C E3 08  00BF- 8C E5 00  1230 WRITE      STY SAVEY
08C0- EA        00C2- EA        1240 ROM.LINK   NOP
08C1- EA        00C3- EA        1250            NOP
08C2- EA        00C4- EA        1260            NOP
08C3- EA        00C5- EA        1270            NOP
08C4- EA        00C6- EA        1280            NOP
08C5- AC E3 08  00C7- AC E5 00  1290            LDY SAVEY
08C8- 60        00CA- 60        1300            RTS
                                1310
                                1320 ;ROUTINE TO OUTPUT CRLF
08C9- A9 0D     00CB- A9 0D     1330 CRLF       LDA #$0D
08CB- 20 BD 08  00CD- 20 BF 00  1340            JSR WRITE
08CE- A9 0A     00D0- A9 0A     1350            LDA #$0A
08D0- 20 BD 08  00D2- 20 BF 00  1360            JSR WRITE
08D3- 60        00D5- 60        1370            RTS
                                1380
                                1390 ;SPACE2 = OUTPUT 2 SPACES
                                1400 ;SPACE  = OUTPUT 1 SPACE
08D4- 20 D7 08  00D6- 20 D9 00  1410 SPACE2     JSR SPACE
08D7- A9 20     00D9- A9 20     1420 SPACE      LDA #'
08D9- 20 BD 08  00DB- 20 BF 00  1430            JSR WRITE
08DC- 60        00DE- 60        1440            RTS
                                1450
                                1460
08DD-           00DF-           1470 START      .DS 2 ;USER ENTERS START OF MEMORY RANGE
08DF-           00E1-           1480 END        .DS 2 ;USER ENTERS END OF MEMORY RANGE
                                1490
08E1-           00E3-           1500 TEST<PATRN .DS 1         ;CURRENT TEST PATTERN
08E2-           00E4-           1510 TEST<TYPE  .DS 1         ;=1,2 FOR TEST TYPE
08E3-           00E5-           1520 SAVEY      .DS 1         ;SAVE R(Y)
                                1530
                                1540
                                1550 END.PGM    .EN
```

**TABLE A**

| Computer | Use Listing | Enter at ROM LINK: 00C2 for Listing 1 08C0 for Listing 2 |
|---|---|---|
| PET | 2 | 20 D2 FF |
| APPLE II | 2 | 09 80 20 ED FD |
| SYM | 1 or 2 | 20 63 A6 |
| KIM | 1 | 20 A0 1E |
| TIM | 1 or 2 | 20 C6 72 |
| OSI 65D | 2 | 20 0B FE |
| Western Data Systems | 2 | 20 A5 FC |
| ATARI | ? | ? |
| AIM | ? | ? |
| Super KIM | ? | ? |

**TABLE B**

| | Listing 1 | Listing 2 |
|---|---|---|
| Start Address lo | 00DF | 08DD |
| Start Address hi | 00E0 | 08DE |
| End Address lo | 00E1 | 08DF |
| End Address hi | 00E2 | 08E0 |
| Execution Address | 0002 | 0800 |

**Statement 140: $0002 For Test 1**
**$0800 For Test 2**

**Universal 6502 Memory Test**

EASTERN HOUSE SOFTWARE
Carl W. Moser
3239 Linda Drive
Winston-Salem, NC 27106