The Practical Side Of Assembly Language

Part II: Loops And Arrays

Bruce D. Carbrey, Raleigh, NC

In the last installment (**COMPUTE!** #14) I presented some ideas on how to represent and use flags in 6502 assembly language programs. This time I will discuss methods for programming loop control structures for the manipulation of arrays of data. Let's start by writing a loop which simply initializes all elements of an array to zero. In BASIC, you might write:

```
100 DIM AR(50)
110 FOR I=1 TO 50
120 AR(I)=0
130 NEXT I
```

If you are a neophyte assembly language programmer and try to translate this program segment on a line-by-line basis, you might wind up with something like this:

```
AR      *=*+ 50         ;SPACE FOR ARRAY AR (50 BYTES)
I       *=*+ 1          ;LOOP COUNTER VARIABLE I
        ...
        LDA  #1
        STA  I          ;INITIALIZE LOOP COUNTER
LOOP    LDA  #0
        LDX  I          ;RECALL CURRENT LOOP
                         COUNTER
        STA  AR,X       ;SET ELEMENT OF ARRAY TO 0
        INX             ;ADVANCE TO NEXT ELEMENT
        STX  I          ;STORE INDEX REGISTER
        CPX  #50        ;CHECK AGAINST LIMIT
        BNE  LOOP       ;REPEAT UNTIL DONE
```

If you run this program you'll be dismayed to find out that it only sets the last 49 elements of the array to 0 and skips the first element, because the first element of the array should be indexed with a zero, not a one.

Rule #7. To access the first element of an assembly language array, you should use an index of 0, not 1. The last element of an array of size N is indexed by N–1.

You may also recognize that it is not necessary to allocate space or save the variable I for the loop. Since it is only needed to control the loop, it can be simply kept in the index register (I chose the X register; the Y register will serve equally well). We can correct and improve the loop as follows:

```
AR      *=*+ 50         ;SPACE FOR ARRAY AR (50 BYTES
        ...
        LDA  #0         ;CONSTANT TO FILL ARRAY
                         WITH
        TAX             ;X=0=INITIAL INDEX TO
                         ARRAY AR
LOOP    STA  AR,X       ;ZERO OUT ONE ARRAY
                         ELEMENT
        INX             ;ADVANCE TO NEXT ELEMENT
        CPX  #50        ;CHECK INDEX AGAINST LIMIT
        BNE  LOOP       ;REPEAT UNTIL DONE
```

You may notice some other subtle improvements in this program segment. The A register is only loaded with 0 once, outside the loop, since it does not change inside the loop. This will make the program run faster by eliminating 49 unneeded repetitions of the LDA #0 instruction.

Rule #8. Move code which does not need to be repeated out of the loop if possible.

Also note that one byte of code was saved by using TAX to initialize the X register instead of LDX #0. Naturally if we were filling our array with something other than 0, this trick won't work. We now have a correctly-functioning loop which is equivalent of our BASIC program (strictly speaking, is not exactly equivalent because the BASIC interpreter uses floating point arithmetic which uses four or five bytes for each array element instead of one).

Can our loop be further improved in terms of efficiency? Consider this alternative:

```
AR      *=*+ 50         ;SPACE FOR ARRAY (50 BYTES
        ...
        LDA  #0         ;CONSTANT TO FILL THE ARRAY
                         WITH
        LDX  #49        ;INDEX TO LAST ELEMENT OF
                         THE ARRAY
LOOP    STA  AR,X       ;SET AN ELEMENT OF THE
                         ARRAY TO 0
        DEX             ;BACKUP TO PREVIOUS
                         ELEMENT
        BPL  LOOP       ;REPEAT UNTIL DONE
```

This code segment fills the loop backwards, filling the last element first and the first element last. Once the 0th element has been filled, the index register is decremented to -1 ($FF) and the BPL LOOP instruction will exit the loop. Notice that we have eliminated the CMP instruction from the loop, saving two cycles.

Rule #9. Moving backwards through an array will usually be more efficient.

If you try to make the array bigger than 128 elements you will be in trouble! Suppose you increase the dimension of AR to 200. In this case your loop will be executed only once because on the first pass, the DEX instruction will change the index from 199 to 198. But 198 has the hexadecimal representation $C6, which has bit 7 (the sign bit) set to 1. Therefore the 6502 will consider this a negative number (-58 decimal) and the BPL instruction will let control "fall through." Therefore, our BPL instruction will only work right up to +127 decimal, which is the largest *signed* 8-bit number. We can remedy this problem for index values up to 255 with a slightly more "tricky," but equivalent, method:

```
AR      *=*+ 200        ;SPACE FOR ARRAY (200 BYTES)
        ...
        LDA  #0         ;CONSTANT TO FILL ARRAY
                         WITH
        LDX  #200       ;INDEX TO LAST ELEMENT OF
                         THE ARRAY+1
```

```
LOOP  STA   AR-1,X     ;SET AN ELEMENT OF THE
                        ARRAY TO 0
      DEX               ;BACKUP TO PREVIOUS
                        ELEMENT
      BNE   LOOP        ;REPEAT UNTIL DONE
```

Here we have replaced the BPL instruction with a BNE instruction so that the loop will terminate one pass earlier, but will not be stymied by index values greater than 127. Since our last pass through the array will now have an index of 1 instead of 0, we must compensate by changing the destination for our indexed STA instruction to AR-1. Therefore the last element set will be AR-1+1 = AR+0. Finally, the starting index must be bumped from 199 to 200 for the same reason. Note that a starting index of 0 will clear a full 256 byte array.

The same technique can be used to move a block of data of up to 256 bytes from one known location to another:

```
ARA   *=*+  200       ;ARRAY A CONTAINING 200
                        ELEMENTS
ARB   *=*+  200       ;ARRAY B CONTAINING 200
                        ELEMENTS
      ...
      LDX   #200      ;NUMBER OF ELEMENTS TO
                        MOVE
LOOP  LDA   ARA-1,X   ;FETCH ELEMENT OF ARRAY A
      STA   ARB-1,X   ;INSTALL IN ARRAY B
      DEX             ;DECREMENT TO PREVIOUS
                        ELEMENT
      BNE   LOOP      ;REPEAT UNTIL DONE
      ...
```

What happens if you have more than 256 bytes? Throw away your 6502 and get a processor with a 16-bit index register? Nope. The indirect,X and indirect,Y addressing modes will solve this problem.

Rule #10. To use arrays of more than 256 bytes or arrays whose location is not known at assembly time, plan on using indirect,X or indirect,Y addressing.

Unlike the absolute, indexed addressing modes, indirect,X and indirect,Y are not equivalent. You may remember that indirect,X addressing uses pre-indexing and indirect,Y uses post-indexing. As a practical matter, indirect,X addressing will almost always be used with a permanent index of 0, simulating simple indirect addressing. This mode lends itself to manipulating large data arrays in non-time-critical portions of a program. For example, the following loop initializes a 1000-element array to 0:

```
ARRAY *=*+  1000      ;ROOM ARRAY OF 1000
                        ELEMENTS
PTR   *=*+  2         ;POINTER TO AN ARRAY
                        ELEMENT
      ...
CLR1K LDA   #ARRAY&$FF  ;LOW 8 BITS OF ADDRESS
                        OF START OF ARRAY
      STA   PTR         ;INITIALIZE POINTER
      LDA   #ARRAY/256  ;HIGH 8 BITS OF AD-
                        DRESS OF START OF
                        ARRAY
      STA   PTR+1       ;INITIALIZE HIGH BYTE
                        OF POINTER
      LDX   #0          ;PERMANENTLY LOAD
                        X WITH 0
LOOP  LDA   #0
      STA   (PTR),X     ;ZERO BYTE POINTED
                        TO BY PTR
      INC   PTR         ;BUMP POINTER UP TO
                        NEXT ELEMENT
      BNE   CHECK       ;BRANCH IF NOT
                        CROSSING PAGE
                        BOUNDARY
      INC   PTR+1       ;ELSE BUMP HI-ORDER
                        BYTE OF POINTER
CHECK LDA   PTR
      CMP   #ARRAY+1000&$FF ;CHECK POINTER
                        AGAINST LIMIT
      BNE   LOOP        ;REPEAT IF NOT DONE
      LDA   PTR+1
      CMP   #ARRAY+1000/256 ;ELSE CHECK HI BYTE
                        OF POINTER
      BNE   LOOP        ;REPEAT IF NOT DONE
```

Some assemblers use the notation #<ARRAY to mean the low byte of the address of ARRAY and #>ARRAY for the high byte instead of #ARRAY&$FF and #ARRAY/256. Clearly this program segment is quite a bit "messier" than the one for arrays of less than 256 bytes. When planning sizes for arrays, you should remember this and try to limit arrays to 256 bytes or less whenever practical.

Luckily, the indirect,Y addressing mode is considerably more powerful than indirect,X. For our final problem, let's use the indirect,Y mode to build a subroutine to move a large block of data from one place to another in memory as fast as possible. The source address, destination address, and number of bytes to be moved are to be specified as input to the routine as three 16-bit variables in page 0:

```
FROM  *=*+  2     ;POINTER TO STARTING ADDRESS OF
                    ARRAY TO MOVE
TO    *=*+  2     ;POINTER TO STARTING ADDRESS
                    OF DESTINATION
COUNT *=*+  2     ;NUMBER OF BYTES TO COPY
```

In an earlier example we saved execution time by removing the need for a compare inside the loop. We can apply the same principle to speed up our block move by sub-dividing the routine into two loops, one which moves entire pages (1 page = 256 bytes), and one which moves the final fractional page. This allows us to avoid any compares in the part which moves entire pages(which is part of the routine executed the most when copying large blocks). This will also let us use both 8-bit index registers to maximum effectiveness by allocating one for counting pages and index registers to maximum effectiveness by allocating one for counting pages and the other for indexing bytes within the page. The resulting routine (shown in Program 4) can easily be converted into a block-fill routine instead by removing FROM and all lines that refer to it, and presetting A to 0 (or the value with which to fill the array).

Rule #11. To deal with large arrays, split your program into two loops, one to operate on entire pages and one to operate on the "leftover"

fractional page.

The routine in Program 4 moves data at about 16.1 machine cycles per byte for large blocks, which means a 16K byte array can be moved in 0.26 seconds using a 6502 with a 1MHz clock. In certain applications where speed is of paramount importance, you may wish to improve even this super-fast copy routine. Can it be done? Yes, if you are willing to trade some increased program size for increased execution speed. Again, we employ the same gen-

Program 2: Keyboard Driver with Alpha Lock Flag Using 0 = False and Non-0 = True

Editor's Note: Part of this program was not printed in **COMPUTE!** *#14, we reprint it entirely here. — RTM*

```
                ;
                ;       SUBROUTINE INCH: KEYBOARD DRIVER FOR ASCII-ENCODED
                ;       KEYBOARD WITH PARALLEL INTERFACE.
                ;
                ;       ADDRESSES SHOWN ARE FOR 6530 ON KIM-1 COMPUTER.
                ;       KEYBOARD DATA LINES TO PORT A BITS 0 TO 6,
                ;       NEGATIVE-GOING STROBE TO BIT 7.
                ;
                ;       ON ENTRY: IF ALFALK IS NON-0, THEN ALL LOWERCASE LETTERS WILL
                ;       BE RETURNED AS THE EQUIVALENT UPPERCASE ALPHA.
                ;       ON RETURN: REGISTER A = ASCII CODE FOR KEY PRESSED;
                ;       X AND Y PRESERVED.
                ;
1700            PAD     =       $1700       ;KIM PORT A DATA REGISTER ON 6530
1701            PADD    =       $1701       ;KIM PORT A DATA DIRECTION REGISTER
                ;
0000                    *=      $1780       ;PROGRAM ORIGIN
                ;
1780 A900       INCH    LDA     #$00
1782 8D0117             STA     PADD        ;SET PORT DIRECTION = INPUTS
1785 AD0017     INCH1   LDA     PAD         ;TEST PORT
1788 30FB               BMI     INCH1       ;WAIT FOR STROBE PULSE
178A 2C0017     INCH2   BIT     PAD
178D 10FB               BPL     INCH2       ;WAIT FOR END OF STROBE
                ;
                ;       IF ALPHA-LOCK FLAG IS SET, FOLD ANY LOWERCASE LETTERS TO
                ;       EQUIVALENT UPPERCASE LETTERS.
                ;
178F 48         FOLD    PHA                 ;SAVE CHARACTER TEMPORARILY
1790 ADA317             LDA     ALFALK      ;RECALL "ALPHA LOCK" FLAG
1793 F00C               BEQ     FOLD2       ;BRANCH IF NO FOLDING DESIRED
1795 68                 PLA                 ;ELSE RECALL CHARACTER
1796 C97B               CMP     #$7B        ;LOWER CASE "Z" + 1
1798 B006               BCS     FOLD1       ;BRANCH IF PUNCTUATION
179A C961               CMP     #$61        ;LOWER CASE "A"
179C 9002               BCC     FOLD1       ;BRANCH IF NOT LOWER CASE ALPHA
179E E920               SBC     #$20        ;ELSE FOLD TO EQUIVALENT UPPERCASE
17A0 60         FOLD1   RTS
                ;
17A1 68         FOLD2   PLA                 ;RECALL CHARACTER
17A2 60                 RTS
                ;
                ;       ALPHA LOCK FLAG (DEFAULT = ALLOW LOWER CASE)...
                ;
17A3            ALFALK  .BYTE   0           ;"ALPHA LOCK" FLAG; NON-0=UPPERCASE ONLY.
                ;
0000                    .END

NO ERROR LINES
```

Program 4: Block-Move Memory Routine

```
GENERAL BLOCK-MOVE SUBROUTINE                          MTU 6502 ASSEMBLER 1.0

0002 0000                .PAGE   'GENERAL BLOCK-MOVE SUBROUTINE'
0003 0000                *=      0       ;ZERO PAGE ORIGIN
0004 0000        FROM    *=*+    2       ;STARTING ADDRESS OF BLOCK TO BE COPIED
0005 0002        TO      *=*+    2       ;STARTING ADDRESS OF DESTINATION
0006 0004        COUNT   *=*+    2       ;NUMBER OF BYTES TO BE MOVED
0007 0006        ;
0008 0006                *=      $2000   ;ORIGIN FOR PROGRAM
0009 2000        ;
0010 2000        ;       THIS ROUTINE COPIES A BLOCK OF ANY SIZE FROM ONE
0011 2000        ;       LOCATION TO ANOTHER.
0012 2000        ;
0013 2000        ;       ON ENTRY: FROM (2 BYTES) IS THE STARTING ADDRESS OF
0014 2000        ;       THE BLOCK TO BE COPIED; TO (2 BYTES) IS THE DESIRED
0015 2000        ;       STARTING DESTINATION ADDRESS FOR THE COPY; COUNT
0016 2000        ;       (2 BYTES) IS THE NUMBER OF BYTES TO COPY.
0017 2000        ;
0018 2000        ;       ON RETURN: NO REGISTERS PRESERVED; FROM, TO AND COUNT
0019 2000        ;       ARE CLOBBERED.
0020 2000        ;
0021 2000        ;       NOTE: THE DESTINATION BLOCK MAY OVERLAP THE SOURCE
0022 2000        ;       BLOCK ONLY IF "TO" IS AT A LOWER ADDRESS THAN "FROM".
0023 2000        ;
0024 2000 A000   BLKMOV  LDY     #0      ;INITIAL INDEX WITHIN A PAGE
0025 2002 A605           LDX     COUNT+1 ;NUMBER OF PAGES TO BE MOVED
0026 2004 F00E           BEQ     FRCMOV  ;BRANCH IF ONLY A FRACTIONAL PAGE
0027 2006        ;
0028 2006        ;       THIS LOOP COPIES ENTIRE PAGES...
0029 2006        ;
0030 2006 B100   PAGMOV  LDA     (FROM),Y ;FETCH A BYTE FROM SOURCE
0031 2008 9102           STA     (TO),Y  ;COPY TO DESTINATION
0032 200A C8             INY             ;BUMP POINTER
0033 200B D0F9           BNE     PAGMOV  ;REPEAT TILL ENTIRE PAGE MOVED
0034 200D E601           INC     FROM+1  ;BUMP HI BYTE OF POINTERS
0035 200F E603           INC     TO+1
0036 2011 CA             DEX             ;DECREMENT COUNT OF PAGES TO COPY
0037 2012 D0F2           BNE     PAGMOV  ;REPEAT TILL ALL WHOLE PAGES COPIED
0038 2014        ;
0039 2014        ;       THIS LOOP COPIES THE FINAL FRACTION OF A PAGE...
0040 2014        ;
0041 2014 A604   FRCMOV  LDX     COUNT   ;RECALL NUMBER OF BYTES LEFT TO COPY
0042 2016 F008           BEQ     DONEMV  ;BRANCH IF COUNT IS EXACT PAGE MULTIPLE
0043 2018 B100   FRLOOP  LDA     (FROM),Y ;FETCH A BYTE FROM SOURCE
0044 201A 9102           STA     (TO),Y  ;COPY TO DESTINATION
0045 201C C8             INY             ;BUMP INDEX
0046 201D CA             DEX             ;DECREMENT COUNT OF BYTES LEFT
0047 201E D0F8           BNE     FRLOOP  ;REPEAT UNTIL DONE
0048 2020        ;
0049 2020 60     DONEMV  RTS
0050 2021        ;
0051 2021                .END

   0 ERRORS IN PASS 2
```

eral principle of loop optimization:

Rule #12. To optimize loop execution speed, try to remove unnecessary compares from within the loop.

About the only way we can remove more compares from Program 4 is to "unwind" part of the loop and, instead, write some of the loop code "inline." Since we know the first loop will always move exactly 256 bytes, we can move two bytes at a time instead of one before checking for a page crossing:

```
        ...
PAGMOV  LDA   (FROM),Y  ;FETCH A BYTE FROM SOURCE
        STA   (TO),Y    ;COPY TO DESTINATION
        INY             ;BUMP POINTER
        LDA   (FROM),Y  ;FETCH A BYTE FROM SOURCE
        STA   (TO),Y    ;COPY TO DESTINATION
        INY             ;BUMP POINTER
        BNE   PAGMOV    ;REPEAT TILL ENTIRE PAGE
                         MOVED
        ...
```

This loop now takes 14.5 cycles per byte moved versus 16 cycles for the equivalent loop of Program 4, because the three cycle BNE instruction is only executed for every other byte moved. The loop can be unwound still further to move four, eight or more bytes per pass, but the speed improvement gained drops off rapidly as more code is written inline.

In the next installment I will explore some techniques for optimizing jumps and subroutine calls. ©

```
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