Apple Assembly Line

Volume 1 -- Issue 5 February, 1981

The number of subscribers keeps climbing! From 0 in September, to 45 in October, 85 in November, 179 on Cristmas Eve, and now 242 in late January!

In This Issue...

Bug Reports

1. Several readers have reported a problem with the COPY program in the December issue. As written, if you try to copy a block of lines to a point before the first line of the program, the block is inserted between the first and second bytes of the first line. Ouch! To fix it, insert lines 2221-2225 and change line 2250:

```
2221         LDA A2L
2222         CMP A1L
2223         LDA A2H
2224         SBC A1H
2225         BCC .5

2250 .5      LDA SS        MOVE IN SOURCE BLOCK
```

2. When I typed up Lee Meador's article for the January issue, I inadvertently changed one address to a crazy value. The address $2746 in the 4th paragraph on page 9 should be $1246.

(Bug Reports continued on page 12.)

Apple Noises and Other Sounds

The Apple's built-in speaker is one of its most delightful features. To be sure, it is very limited; but I have used it for everything from sound effects in games to music in six parts (weird-sounding guitar chords) and even speech. Too many ways to put all in one AAL article! I will describe some of the sound effects I have used, and maybe you can go on from there.

The speaker hardware is very simple. A flip-flop controls the current through the speaker coil. Everytime you address $C030, the flip-flop changes state. This in turn reverses the current through the speaker coil. If the speaker cone was pulled in, it pops out; if it was out, it pulls in. If we "toggle" the state at just the right rate, we can make a square-wave sound. By changing the time between reversals dynamically, we can make very complex sounds. We have no control over the amplitude of the speaker motions, only the frequency.

Simple Tone: This program generates a tone burst of 128 cycles (or 256 half-cycles, or 256 pulses), with each half-cycle being 1288 Apple clocks. Just to make it easy, let's call Apple's clock 1MHz. It is really a little faster, but that will be close enough. So the tone will be about 388 Hertz (cycles per second, if you are as old as me!).

How did I figure out those numbers? To get the time for a half-cycle (which I am going to start calling a pulse), I added up the Apple 6502 cycles for each instruction in the loop. LDA SPEAKER takes 4 cycles. DEX is 2 cycles, and BNE is 3 cycles when it branches. The DEX-BNE pair will be executed 256 times for each pulse, but the last time BNE does not branch; BNE only takes 2 cycles when it does not branch. The DEY-BNE pair will branch during each pulse, so we use 5 cycles there. So the total is 4+256*5-1+5=1288 cycles. I got the frequency by the formula f=1/T; T is the time for a whole cycle, or 2576 microseconds.

```
                    1000 *---------------------------------
                    1010 *      SIMPLE TONE
                    1020 *---------------------------------
C030-               1030 SPEAKER    .EQ $C030
                    1040 *---------------------------------
0800- A0 00         1050 TONE   LDY #0          START CYCLE COUNTER
0802- A2 00         1060        LDX #0          START DELAY COUNTER
0804- AD 30 C0      1070 .1     LDA SPEAKER     TOGGLE SPEAKER
0807- CA            1080 .2     DEX             DELAY LOOP
0808- D0 FD         1090        BNE .2
080A- 88            1100        DEY             QUIT AFTER 128 CYCLES
080B- D0 F7         1110        BNE .1
080D- 60            1120        RTS
```

Apple "Bell" Subroutine: Inside your monitor ROM there is a subroutine at $FBE2 which uses the speaker to make a bell-like sound. Here is a copy of that code. Notice that the pulse width is controlled by calling another monitor subroutine, WAIT.

```
                  *------------------------------
                  *      APPLE "BELL" ROUTINE
                  *------------------------------
                         .OR $FBE2     IN MONITOR ROM
                         .TA $800
                  *------------------------------
FCA8-             WAIT        .EQ $FCA8     MONITOR DELAY ROUTINE
C030-             SPEAKER     .EQ $C030
                  *------------------------------
FBE2- A0 C0       M.FBE2 LDY #192      # OF HALF-CYCLES
FBE4- A9 0C       BELL2  LDA #12       SET UP DELAY OF 500 MICROSECONDS
FBE6- 20 A8 FC           JSR WAIT      FOR A HALF CYCLE OF 1000 HERTZ
FBE9- AD 30 C0           LDA SPEAKER   TOGGLE SPEAKER
FBEC- 88                 DEY           COUNT THE HALF CYCLE
FBED- D0 F5              BNE BELL2     NOT FINISHED
FBEF- 60                 RTS
```

Machine-Gun Noise: What if we use a random pulse width? Then we get something called noise, instead of a tone. We can create a burst of pulses of random-sounding width by using values from some arbitrary place in the Apple's memory as loop counts. The program uses the 256 values starting at $BA00 (which is inside DOS). If you make just one burst like that, it doesn't sound like much. But if you make ten in a row, you get a pattern of repetitious random noise bursts that in this case sounds like machine-gun fire. Doesn't it? Well, close enough....

```
                  *------------------------------
                  *      MACHINE-GUN NOISE
                  *------------------------------
C030-             SPEAKER    .EQ $C030
0000-             CNTR       .EQ $00
                  *------------------------------
0800- A2 40       NOISE  LDX #64        LENGTH OF NOISE BURST
                  *------------------------------
0802- A9 0A              LDA #10        NUMBER OF NOISE BURSTS
0804- 85 00              STA CNTR
0806- AD 30 C0    .2     LDA SPEAKER    TOGGLE SPEAKER
0809- BC 00 BA           LDY $BA00,X    GET PULSE WIDTH PSEUDO-RANDOMLY
080C- 88          .1     DEY            DELAY LOOP FOR PULSE WIDTH
080D- D0 FD              BNE .1
080F- CA                 DEX            GET NEXT PULSE OF THIS NOISE BURST
0810- D0 F4              BNE .2
0812- C6 00              DEC CNTR       GET NEXT NOISE BURST
0814- D0 F0              BNE .2
0816- 60                 RTS            RETURN
```

Laser "SWOOP" Sound: We can change the pulse width by making it go from wide to narrow in steps of 5 microseconds. It sounds like a low tone that gradually slides higher and higher until it is beyond the range of the human ear (or the Apple speaker). I used this program in a "space war" game to go with the laser fire. Even though the sound was entirely generated before the laser even appeared on the screen, it looks and sounds like the light beam and sound are simultaneous.

I have indicated in line 1110 that you should try experimenting with some other values for the maximum pulse width count. I have included a separate entry point at SWOOP2 to make ten swoops in a row. Try the various values for the maximum width and run each one from SWOOP2. You might also experiment with running the pulse width in the opposite direction (from narrow to wide) by changing line 1200 to INC PULSE.WIDTH.

```
                        1000 *--------------------------------
                        1010 *      LASER "SWOOP" SOUND
                        1020 *--------------------------------
C030-                   1030 SPEAKER     .EQ $C030
0000-                   1040 PULSE.COUNT .EQ $00
0001-                   1050 PULSE.WIDTH .EQ $01
0002-                   1060 SWOOP.COUNT .EQ $02
                        1070 *--------------------------------
0800- A9 01             1080 SWOOP  LDA #1         ONE PULSE AT EACH WIDTH
0802- 85 00             1090        STA PULSE.COUNT
0804- A9 A0             1100        LDA #160       START WITH MAXIMUM WIDTH
                        1110 *  (ALSO TRY VALUES OF 40, 80, 128, AND 160.)
0806- 85 01             1120        STA PULSE.WIDTH
0808- A4 00             1130 .1     LDY PULSE.COUNT
080A- AD 30 C0          1140 .2     LDA SPEAKER  TOGGLE SPEAKER
080D- A6 01             1150        LDX PULSE.WIDTH
080F- CA                1160 .3     DEX            DELAY LOOP FOR ONE PULSE
0810- D0 FD             1170        BNE .3
0812- 88                1180        DEY            LOOP FOR NUMBER OF PULSES
0813- D0 F5             1190        BNE .2         AT EACH PULSE WIDTH
0815- C6 01             1200        DEC PULSE.WIDTH  SHRINK PULSE WIDTH
0817- D0 EF             1210        BNE .1         TO LIMIT OF ZERO
0819- 60                1220        RTS
                        1230 *--------------------------------
                        1240 *      MULTI-SWOOPER
                        1250 *--------------------------------
081A- A9 0A             1260 SWOOP2 LDA #10        NUMBER OF SWOOPS
081C- 85 02             1270        STA SWOOP.COUNT
081E- 20 00 08          1280 .1     JSR SWOOP
0821- C6 02             1290        DEC SWOOP.COUNT
0823- D0 F9             1300        BNE .1
0825- 60                1310        RTS
```

<u>Another Laser Blast</u>: This one sounds very much the same as the swoop of the previous program, but it uses less memory. You should try experimenting with the pulse widths of the first and last pulses in lines 1060 and 1130. You could also try changing the direction by substituting a DEX in line 1120.

```
                        1000 *--------------------------------
                        1010 *      ANOTHER LASER BLAST
                        1020 *--------------------------------
C030-                   1030 SPEAKER     .EQ $C030
                        1040 *--------------------------------
0800- A0 0A             1050 BLAST  LDY #10        NUMBER OF SHOTS
0802- A2 40             1060 .1     LDX #64        PULSE WIDTH OF FIRST PULSE
0804- 8A                1070 .2     TXA            START A PULSE WITHIN A SHOT
0805- CA                1090 .3     DEX            DELAY FOR ONE PULSE
0806- D0 FD             1100        BNE .3
0808- AA                1105        TAX
0809- AD 30 C0          1110        LDA SPEAKER    TOGGLE SPEAKER
080C- E8                1120        INX
080D- E0 C0             1130        CPX #192       PULSE WIDTH OF LAST PULSE
080F- D0 F3             1140        BNE .2
0811- 88                1150        DEY            FINISHED SHOOTING?
0812- D0 EE             1160        BNE .1         NO
0814- 60                1170        RTS
```

<u>Inch-Worm Sounds</u>: I stumbled onto this one by accident, while looking for some sound effects for a lo-res graphics demo. The demo shows what is supposed to be an inch-worm, inching itself across the screen. By plugging various values (as indicated in lines 1100 and 1130), I got some sounds that synchronized beautifully with the animation. Complete with an exhausted sigh at the end!

```
                 1000 *-------------------------------
                 1010 *      INCH-WORM SOUNDS
                 1020 *-------------------------------
 C030-           1030 SPEAKER     .EQ $C030
 0000-           1040 PULSE.WIDTH .EQ $00
 0001-           1050 PULSE.STEP  .EQ $01
 0002-           1060 PULSE.LIMIT .EQ $02
                 1070 *-------------------------------
                 1080 INCH.WORM
 0800- A9 01     1090        LDA #1       SET STEP TO 1
                 1100 *   (ALSO TRY 77, 129, 179)
 0802- 85 01     1110        STA PULSE.STEP
 0804- A9 B0     1120        LDA #176     SET PULSE.WIDTH AND LIMIT TO 176
                 1130 *   (ALSO TRY 88)
 0806- 85 00     1140        STA PULSE.WIDTH
 0808- 85 02     1150        STA PULSE.LIMIT
 080A- AD 30 C0  1160 .1     LDA SPEAKER  TOGGLE SPEAKER
 080D- A6 00     1170        LDX PULSE.WIDTH  DELAY LOOP FOR PULSE WIDTH
 080F- 48        1180 .2     PHA          LONGER DELAY LOOP
 0810- 68        1190        PLA
 0811- CA        1200        DEX          END OF PULSE?
 0812- D0 FB     1210        BNE .2       NO
 0814- 18        1220        CLC          CHANGE PULSE WIDTH BY STEP
 0815- A5 00     1230        LDA PULSE.WIDTH
 0817- 65 01     1240        ADC PULSE.STEP
 0819- 85 00     1250        STA PULSE.WIDTH
 081B- C5 02     1260        CMP PULSE.LIMIT  UNTIL IT REACHES THE LIMIT
 081D- D0 EB     1270        BNE .1
 081F- 60        1280        RTS
```

<u>Touch-Tones Simulator</u>: I used this one with a telephone demo program. The screen shows a touch tone pad. As you press digits on the keyboard, the corresponding button on the screen lights up (displays in inverse mode). Then the demo program CALLs this machine language code to produce the twin-tone sound that your telephone makes. It isn't perfect, you can't fool the Bell System. But it makes a good demo!

I will describe the program from the top down. The four variables in page zero are kept in a "safe" area, inside Applesoft's floating point accumulator. Applesoft doesn't use these locations while executing a CALLed machine language routine.

The Applesoft demo program stores the button number (0-9) in location $E7. This could be done with "POKE 231,DGT", but I had more fun using "SCALE=DGT". SCALE= is a hi-res graphics command, but all it really does is store the value as a one-byte integer in $E7. Since we aren't using hi-res graphics, the location is perfectly safe to use.

CALL 768 gets us to line 1150, TWO.TONES. This is the main routine. It uses the button number to select the two tone numbers from LOW.TONES and HIGH.TONES. ONE.TONE is called to play first the low tone, then the high tone, back and forth, for ten times each. This is my attempt to fool the ear, to make it sound like both are being played at once.

```
                       1000 *-----------------------------------
                       1010 *      TOUCH TONES SIMULATOR
                       1020 *-----------------------------------
C030-                  1030 SPEAKER     .EQ $C030
                       1040 *-----------------------------------
009D-                  1050 DOWNTIME    .EQ $9D
009E-                  1060 UPTIME      .EQ $9E
009F-                  1070 LENGTH      .EQ $9F
00A0-                  1080 CHORD.TIME  .EQ $A0
                       1090 *-----------------------------------
00E7-                  1100 BUTTON      .EQ $E7   SET BY "SCALE= # "
                       1110 *                     USE VALUES FROM 0 THRU 9
                       1120 *-----------------------------------
                       1130        .OR $300
                       1140 *-----------------------------------
                       1150 TWO.TONES
0300- A9 0A            1160        LDA #10
0302- 85 A0            1170        STA CHORD.TIME
0304- A6 E7            1180 .3     LDX BUTTON
0306- BD 59 03         1190        LDA LOW.TONES,X
0309- 20 17 03         1200        JSR ONE.TONE
030C- BD 63 03         1210        LDA HIGH.TONES,X
030F- 20 17 03         1220        JSR ONE.TONE
0312- C6 A0            1230        DEC CHORD.TIME
0314- D0 EE            1240        BNE .3
0316- 60               1250        RTS
                       1260 *-----------------------------------
                       1270 ONE.TONE
0317- A8               1280        TAY
0318- B9 44 03         1290        LDA DOWNTIME.TABLE,Y
031B- 85 9D            1300        STA DOWNTIME
031D- B9 4B 03         1310        LDA UPTIME.TABLE,Y
0320- 85 9E            1320        STA UPTIME
0322- B9 52 03         1330        LDA LENGTH.TABLE,Y
0325- 85 9F            1340        STA LENGTH
                       1350 *-----------------------------------
0327- A4 9E            1360 PLAY   LDY UPTIME
0329- AD 30 C0         1370        LDA SPEAKER
032C- C6 9F            1380        DEC LENGTH
032E- F0 13            1390        BEQ .4         FINISHED
0330- 88               1400 .1     DEY
0331- D0 FD            1410        BNE .1
0333- F0 00            1420        BEQ .2
0335- A4 9D            1430 .2     LDY DOWNTIME
0337- AD 30 C0         1440        LDA SPEAKER
033A- C6 9F            1450        DEC LENGTH
033C- F0 05            1460        BEQ .4
033E- 88               1470 .3     DEY
033F- D0 FD            1480        BNE .3
0341- F0 E4            1490        BEQ PLAY
0343- 60               1500 .4     RTS
                       1510 *-----------------------------------
                       1520 DOWNTIME.TABLE
0344- 8E 80 74
0347- 68 51 49
034A- 42               1530        .HS 8E807468514942
                       1540 *-----------------------------------
                       1550 UPTIME.TABLE
034B- 8E 80 74
034E- 69 51 49
0351- 42               1560        .HS 8E807469514942
                       1570 *-----------------------------------
                       1580 LENGTH.TABLE
0352- 14 12 10
0355- 0F 20 1D
0358- 1A               1590        .HS 1412100F201D1A
                       1600 *-----------------------------------
                       1610 LOW.TONES
0359- 03 00 00
035C- 00 01 01
035F- 01 02 02
0362- 02               1620        .HS 03000000010101020202
                       1630 HIGH.TONES
0363- 05 04 05
0366- 06 04 05
0369- 06 04 05
036C- 06               1640        .HS 05040506040506040506
```

ONE.TONE wiggles the speaker for LENGTH half-cycles. Each half-cycle is controlled by either the UPTIME or DOWNTIME counts. These three parameters are selected from three tables, according to the tone number selected by TWO.TONES. Lines 1270-1340 pick up the values from the three tables and load the page zero variables. Lines 1360-1500 do the actual speaker motions and time everything. The purpose of having two routines, one for uptime and one for downtime, is to be able to more closely approximate the frequency. For example, if the loop count we ought to use is 104.5, we could use an uptime of 104 and a down time of 105; this makes the total time for the full cycle correct. The redundant BEQ in line 1420 is there to make the loop times for UPTIME and DOWNTIME exactly the same.

Since you do not have my Applesoft program, which drives this, I wrote a simulated drive to just "push" the buttons 0-9. Lines 1650-1790 do this. I separated each button push by a call to the monitor WAIT subroutine, to make them easier to distinguish.

```
                   1650 *-------------------------------
                   1660 *      SIMULATED DRIVER
                   1670 *-------------------------------
FCA8-              1680 MON.WAIT    .EQ $FCA8
                   1690 PUNCH.ALL
036D- A9 00        1700        LDA #0
036F- 85 E7        1710        STA BUTTON
0371- 20 00 03     1720 .1     JSR TWO.TONES
0374- A9 00        1730        LDA #0
0376- 20 A8 FC     1740        JSR MON.WAIT
0379- E6 E7        1750        INC BUTTON
037B- A5 E7        1760        LDA BUTTON
037D- C9 0A        1770        CMP #10
037F- 90 F0        1780        BCC .1
0381- 60           1790        RTS
```

Morse Code Output: I have always thought that computers really only need one output line and one input line for communicating with humans. I could talk to my Apple with a code key, and it could beep back at me. One of the first programs I attempted in 6502 language was a routine to echo characters in Morse code. I looked it up about two hours ago, and shuddered at my sloppy, inefficient, hard to follow code. So, I wrote a new one.

I broke the problem down into three littler ones: 1) getting the characters which are to be output; 2) converting the ASCII codes to the right number of dots and dashes; and 3) making tones and spaces of the right length.

SETUP.MORSE (lines 1190-1240) links my output routine through the monitor output vector. Line 1240 JMPs to $3EA to re-hook DOS after me.

MORSE (lines 1260-1310) are an output filter. If the character code is less than $B0, I don't know how to send it in Morse code; therefore, I just go to $FDF0 to finish the output on the screen. Codes exist for these other characters, but I did not look them up. If you want a complete routine, you should modify line 1260 to CMP #$A0 and add the extra codes to the code table (lines 1130-1170).

SEND.CHAR looks up the Morse code for the character in the code table, and splits it into the number of code elements (low-order three bits) and the code elements themselves (high-order five bits). If a code element is zero, a short beep (dot) is sounded. If an element is one, three calls to the short beep routine make one long beep (dash). Between elements, a silence equal to the length of a short beep intervenes. After the last beep of a character, a longer silence, equal to three short silences, is produced. A 00 code from the code table makes a silent gap of three times the inter-character gap.

EL.SPACE and EL.DIT are nearly identical. The only difference is that EL.DIT makes a sound by addressing the speaker, while EL.SPACE does not. The value of EL.PITCH determines the pulse width, and EL.SPEED determines the number of pulses for an inter-element-space or a short beep. If the code stream is too fast for you, you can slow it down by increasing either or both of these two numbers.

```
                  *--------------------------------
                  *      MORSE CODE OUTPUT
                  *--------------------------------
C030-             SPEAKER    .EQ $C030
C000-             DUMMY      .EQ $C000
                  *--------------------------------
0800-             SAVEX   .BS 1
0801-             SAVEY   .BS 1
0802-             EL.COUNT .BS 1
0803-             EL.CODE  .BS 1
0078-             EL.SPEED .EQ 120
0050-             EL.PITCH .EQ 80
                  *--------------------------------
0804- FD 7D 3D
0807- 1D 0D 05
080A- 85 C5 E5
080D- F5          CODES    .HS FD7D3D1D0D0585C5E5F5   0, 1-9
080E- 00 00 00
0811- 00 00 00             .HS 000000000000
0814- 00 42 84
0817- A4 83 01
081A- 24 C3 04
081D- 02 74 A3
0820- 44 C2                .HS 004284A4830124C3040274A344C2  @, A-M
0822- 82 E3 64
0825- D4 43 03
0828- 81 23 14
082B- 63 94 B4
082E- C4                   .HS 82E364D443038123146394B4C4    N-Z
082F- 00 00 00
0832- 00 00 00             .HS 000000000000
                  *--------------------------------
                  SETUP.MORSE
0835- A9 40                LDA #MORSE
0837- 85 36                STA $36
0839- A9 08                LDA /MORSE
083B- 85 37                STA $37
083D- 4C EA 03             JMP $3EA
                  *--------------------------------
0840- C9 B0       MORSE    CMP #$B0     SEE IF PRINTING CHAR
0842- 90 05                BCC .1       NO
0844- 48                   PHA          SAVE CHAR ON STACK
0845- 20 4C 08             JSR SEND.CHAR
0848- 68                   PLA          GET CHAR OFF STACK
0849- 4C F0 FD    .1       JMP $FDF0
```

```
                *---------------------------------
                SEND.CHAR
084C- 8E 00 08             STX SAVEX
084F- 8C 01 08             STY SAVEY
0852- 38                   SEC
0853- E9 B0                SBC #$B0
0855- AA                   TAX
0856- BD 04 08             LDA CODES,X
0859- 8D 03 08             STA EL.CODE
085C- 29 07                AND #7        GET ELEMENT COUNT
085E- F0 23                BEQ .4        NO CODE
0860- 8D 02 08             STA EL.COUNT
0863- 0E 03 08  .1         ASL EL.CODE    PUT NEXT ELEMENT INTO CARRY
0866- 90 06                BCC .2        MAKE 'DIT'
0868- 20 A0 08             JSR EL.DIT    MAKE 'DAH' FROM 3 DITS
086B- 20 A0 08             JSR EL.DIT
086E- 20 A0 08  .2         JSR EL.DIT    MAKE 'DIT'
0871- 20 92 08             JSR EL.SPACE
0874- CE 02 08             DEC EL.COUNT
0877- D0 EA                BNE .1
0879- 20 8C 08  .3         JSR CH.SPACE
087C- AE 00 08             LDX SAVEX
087F- AC 01 08             LDY SAVEY
0882- 60                   RTS
0883- 20 8C 08  .4         JSR CH.SPACE
0886- 20 8C 08             JSR CH.SPACE
0889- 4C 79 08             JMP .3
                *---------------------------------
                CH.SPACE
088C- 20 92 08             JSR EL.SPACE
088F- 20 92 08             JSR EL.SPACE
                EL.SPACE
0892- A0 78                LDY #EL.SPEED
0894- A2 50     .1         LDX #EL.PITCH
0896- AD 00 C0             LDA DUMMY
0899- CA        .2         DEX
089A- D0 FD                BNE .2
089C- 88                   DEY
089D- D0 F5                BNE .1
089F- 60                   RTS
                *---------------------------------
08A0- A0 78     EL.DIT     LDY #EL.SPEED
08A2- A2 50     .1         LDX #EL.PITCH
08A4- AD 30 C0             LDA SPEAKER
08A7- CA        .2         DEX
08A8- D0 FD                BNE .2
08AA- 88                   DEY
08AB- D0 F5                BNE .1
08AD- 60                   RTS
```

Stuffing Object Code in Protected Places

Several users of Version 4.0 have asked for a way to defeat the protection mechanism, so that they can store object code directly into the language card. One customer has a EPROM burner which accepts code at $D000. He wants to let the assembler write it out there directly, even though he could use the .TA directive and later a monitor move command. Or, he could use the .TF directive, and a BLOAD into his EPROM.

For whatever reason, if you really want to do it, all you have to do is type the following patch just before you assemble: $1A25:EA EA. In case you want to put it back, or check before you patch, what should be there is B0 28.

Multiplying on the 6502

Brooke Boering wrote an excellent article, "Multiplying on the 6502", in MICRO--The 6502 Journal, December, 1980, pages 71-74. If you are wondering how to do it, or you want a faster routine for a special application, look up that article.

Brooke begins by explaining and timing the multiply subroutine found in the old Apple Monitor ROM. The time to multiply two 16-bit values and get a 32-bit result varies from 935 to 1511 microseconds, depending on how many "1" bits are in the multiplier. He proceeds to modify that subroutine to cut the execution time by 40%!

Finally, he presents two limited versions which are still quite useful in some applications. His 8x16 multiply averages only 383 microseconds, and his 8x8 version averages 192 microseconds.

Here is the code for his 16x16 version, which averages 726 microseconds. It has the same setup as the routine in the Apple ROM. On entry, the multiplicand should be in AUXL,AUXH ($54,55); the multiplier should be in ACL,ACH ($50,51); whatever is in XTNDL,XTNDH ($52,53) will be added to the product. Normally, XTNDL and XTNDH should be cleared to zero before starting to multiply. However, I have used this routine to convert from decimal to binary; I put the next digit in XTNDL and clear XTNDH, and then multiply the previous result by ten. The "next digit" is automatically added to the product that way. (I have corrected the typographical error in the listing as published in MICRO.)

```
                *-------------------------------
                *       FASTER 16X16 MULTIPLY
                *       BY BROOKE W. BOERING
                *       NEARLY AS PUBLISHED IN MICRO--THE 6502 JOURNAL
                *       PAGE 72, DECEMBER, 1980.
                *-------------------------------
0050-           ACL     .EQ $50
0051-           ACH     .EQ $51
0052-           XTNDL   .EQ $52
0053-           XTNDH   .EQ $53
0054-           AUXL    .EQ $54
0055-           AUXH    .EQ $55
                *-------------------------------
0800- A0 10     RMUL    LDY #16         16-BIT MULTIPLIER
0802- A5 50     .1      LDA ACL         (AC * AUX) + XTND
0804- 4A                LSR             CHECK NEXT BIT OF MULTIPLIER
0805- 90 0D             BCC .2          IF ZERO, DON'T ADD MULTIPLICAND
0807- 18                CLC             ADD MULTIPLICAND TO PARTIAL PRODUCT
0808- A5 52             LDA XTNDL
080A- 65 54             ADC AUXL
080C- 85 52             STA XTNDL
080E- A5 53             LDA XTNDH
0810- 65 55             ADC AUXH
0812- 85 53             STA XTNDH
0814- 66 53     .2      ROR XTNDH       SHIFT PARTIAL PRODUCT
0816- 66 52             ROR XTNDL
0818- 66 51             ROR ACH
081A- 66 50             ROR ACL
081C- 88                DEY             NEXT BIT
081D- D0 E3             BNE .1          UNTIL ALL 16
081F- 60                RTS
```

```
                1310 *-------------------------------
                1320 *      TEST ROUTINE FOR MULTIPLY
                1330 *-------------------------------
                1340 SETUP.Y
0820- A9 4C     1350        LDA #$4C      PUT "JMP TESTMPY" IN $358-35A
0822- 8D F8 03  1360        STA $3F8
0825- A9 30     1370        LDA #TESTMPY
0827- 8D F9 03  1380        STA $3F9
082A- A9 08     1390        LDA /TESTMPY
082C- 8D FA 03  1400        STA $3FA
082F- 60        1410        RTS
                1420 *-------------------------------
                1430 TESTMPY
0830- A5 3C     1440        LDA $3C       MOVE A1L,A1H TO ACL,ACH
0832- 85 50     1450        STA ACL
0834- A5 3D     1460        LDA $3D
0836- 85 51     1470        STA ACH
0838- A5 3E     1480        LDA $3E       MOVE A2L,A2H TO AUXL,AUXH
083A- 85 54     1490        STA AUXL
083C- A5 3F     1500        LDA $3F
083E- 85 55     1510        STA AUXH
0840- A5 42     1520        LDA $42       MOVE A4L,A4H TO XTNDL,XTNDH
0842- 85 52     1530        STA XTNDL
0844- A5 43     1540        LDA $43
0846- 85 53     1550        STA XTNDH
0848- 20 00 08  1560        JSR RMUL      MULTIPLY
084B- A5 53     1570        LDA XTNDH     PRINT 32-BIT RESULT
084D- 20 DA FD  1580        JSR $FDDA
0850- A5 52     1590        LDA XTNDL
0852- 20 DA FD  1600        JSR $FDDA
0855- A5 51     1610        LDA ACH
0857- 20 DA FD  1620        JSR $FDDA
085A- A5 50     1630        LDA ACL
085C- 4C DA FD  1640        JMP $FDDA
```

I wrote a test routine for the multiply, so that I could check it out. After assembling the whole program, I typed "MGO SETUP.Y" to link the control-Y Monitor Command to my test routine. Control-Y will parse three 16-bit hexadecimal values this way: val1<val2.val3cY stores val1 in $42,$43; val2 in $3C,$3D; and val3 in $3E,$3F. ("cY" stands for control-Y.)

I define val1 to be the initial value for XTNDL,XTNDH; this should normally be zero. The two values to be multiplied are val2 and val3. After TESTMPY receives control from the control-Y processor, it moves the three values into the right locations for the multiply subroutine. Then JSR RMUL calls the multiply routine. The following lines (1570-1640) print the 32-bit result by calling a routine in the monitor ROM which prints a byte in hex from the A-register.

<u>Bug Reports</u> (continued from page 1)

3. The Variable Cross Reference program for Applesoft from the November issue leaves something behind after it has run. If you LIST the Applesoft program after running VCR, the line number of the first line will come out garbage. This only happens the first time you use the LIST command. For some reason, typing CALL 1002 before the LIST will fix it. I haven't found out the cause or cure yet. If you find it first, let me know!

APPLE II™ OWNERS!!

Tired of the Autostart ROM? Using it only to have the I,J,K,M cursor movement...or the Stop-List feature? What about the LACK of all of the other routines available in the original Monitor? Oh well, you can't have those TOO, right? WRONG!

Enter XMON™ (eXpanded MONitor)...an EPROM version of the good 'ole original APPLE II™ Monitor, designed for the disk-based APPLE II™ user. Features include:

*** All ESCape functions work from upper OR lower case, including the added I,J,K,M functions.

*** Lowercase input STAYS lowercase. NO CONVERSION to uppercase before storing in memory.

*** SINGLE-STEP and TRACE functions are restored.

*** 16-bit MULTIPLY & DIVIDE functions are restored.

*** STOP-LIST <Ctrl S> IMMEDIATELY halts output.

*** <Ctrl C> terminates the output from ANY type of listing (e.g., APPLESOFT™, Integer or Monitor)!

*** Two BRAND-NEW ESCape functions, <ESC P> & <ESC T>, provide extra editing power.

*** Plugs directly* into the Language Card, in place of the Autostart ROM. User MUST provide copies of DOS 3.3 and "BASICS", which will be updated with an image of XMON™ at no extra charge.

*** For the non-Language Card user, XMON™ plugs into the main board with an adapter (supplied). User removes Autostart ROM from the APPLESOFT™ or Integer Card (if so equipped).

So...What's missing? TAPE READ & TAPE WRITE ROUTINES. Remember, XMON™ is for the disk-based user; when was the last time YOU used a tape recorder for I/O?

Introductory Price: $50.00. Please add $1.00 for shipping.

* Requires conversion to 2716 EPROM use, provided for on the card by APPLE™. Voids Card Warranty.

APPLE, APPLE II & APPLESOFT are trademarks of Apple Computer, Inc.
XMON is a trademark of C. J. WELMAN.

C. J. WELMAN, P.O. BOX 5114, ANAHEIM CA 92804 (714) 540-1411

A String Swapper for Applesoft

Practically every program rearranges data in some way. Many times you must sort alphanumeric data, and Applesoft makes this relatively easy. At the heart of most sort algorithms you will have to swap two items.

If the items are numbers, you might do it like this: T=A(I) : A(I)=A(J) : A(J)=T. If the items are in string variables, you might use this: T$=A$(I) : A$(I)=A$(J) : A$(J)=T$.

Before long, Applesoft's wonderful string processor eats up all available memory and your program screeches to a halt with no warning. You think your computer died. Just about the time you reach for the power switch, it comes to life again (if you aren't too impatient!); the garbage collection procedure has found enough memory to continue processing. If only Applesoft had a command to swap the pointers of two strings, this wouldn't happen.

What are pointers? Look on page 137 of your Applesoft Reference Manual. The third column shows how string variables are stored in memory. Each string, whether a simple variable or an element of an array, is represented by three bytes: the first byte tells how many bytes are in the string value at this time; the other two bytes are the address of the first byte of the string value. The actual string value may be anywhere in memory. I am calling the three bytes which define a string a "pointer".

All right, how can we add a string swap command? The authors of Applesoft thoughtfully provided us with the "&" command; it allows us to add as many new commands to the language as we want. (Last month I showed you how to add a computed GOSUB command using the &.) We could make up our own swap command; perhaps something like &SWAP A$(I) WITH A$(J). However, to keep it a little simpler, I wrote it this way: &A$(I),A$(J).

The program is in two sections. The first part, called SETUP, simply sets up the &-vector at $3F5, $3F6, and $3F7. It stores a "JMP SWAP" instruction there. When Applesoft finds an ampersand (&) during execution, it will jump to $3F5; our JMP SWAP will start up the second section.

SWAP calls on two routines inside the Applesoft ROMs: PTRGET ($DFE3) and SCAN.COMMA ($DEBE). I found the addresses for these routines in the article "Applesoft Internal Entry Points", by John Crossley, pages 12-18 of the March/April 1980 issue of _The Apple Orchard_. I also have disassembled and commented the Applesoft ROMs, so I checked to see if there were any bad side effects. Both routines assume that Applesoft is about to read the next character of your program. PTRGET assumes you are sitting on the first character of a variable name. SCAN.COMMA hopes you are sitting on a comma.

SWAP merely calls PTRGET to get the address of the pointer for the first variable, check for an intervening comma, and then calls PTRGET again to get the pointer address for the second variable. Then lines 1350-1430 exchange the three bytes for the two pointers.

```
                  1000 *---------------------------------
                  1010 *      STRING SWAP FOR APPLESOFT
                  1020 *      "BRUN B.STRING.SWAP" TO SET IT UP;
                  1030 *      THEN "&A$,B$" MEANS SWAP A$ AND B$.
                  1040 *---------------------------------
                  1050        .OR $300
                  1060        .TF B.STRING.SWAP
                  1070 *---------------------------------
03F5-             1080 AMPERSAND.VECTOR    .EQ $3F5
                  1090 *---------------------------------
DFE3-             1100 PTRGET              .EQ $DFE3    SCAN FOR VARIABLE NAME,
                  1110 *                                SEARCH FOR ITS ADDRESS,
                  1120 *                                LEAVE ADDRESS IN $83,$84
                  1130 *                                AND A,Y
                  1140 *---------------------------------
DEBE-             1150 SCAN.COMMA          .EQ $DEBE    IF NEXT CHARACTER IS
                  1160 *                                IS A COMMA, SCAN OVER
                  1170 *                                IT; IF NOT, SYNTAX ERROR.
                  1180 *---------------------------------
0085-             1190 A.PNTR .EQ $85,86
0083-             1200 B.PNTR .EQ $83,84
                  1210 *---------------------------------
0300- A9 10       1220 SETUP  LDA #SWAP    SET UP AMPERSAND VECTOR
0302- 8D F6 03    1230        STA AMPERSAND.VECTOR+1
0305- A9 03       1240        LDA /SWAP
0307- 8D F7 03    1250        STA AMPERSAND.VECTOR+2
030A- A9 4C       1260        LDA #$4C     JMP OPCODE
030C- 8D F5 03    1270        STA AMPERSAND.VECTOR
030F- 60          1280        RTS
                  1290 *---------------------------------
0310- 20 E3 DF    1300 SWAP   JSR PTRGET   GET POINTER TO FIRST STRING
0313- 85 85       1310        STA A.PNTR
0315- 84 86       1320        STY A.PNTR+1
0317- 20 BE DE    1330        JSR SCAN.COMMA  CHECK FOR COMMA
031A- 20 E3 DF    1340        JSR PTRGET
031D- A0 02       1350        LDY #2       PREPARE TO SWAP 3 BYTES
031F- B1 85       1360 .1     LDA (A.PNTR),Y
0321- 48          1370        PHA
0322- B1 83       1380        LDA (B.PNTR),Y
0324- 91 85       1390        STA (A.PNTR),Y
0326- 68          1400        PLA
0327- 91 83       1410        STA (B.PNTR),Y
0329- 88          1420        DEY          NEXT BYTE
032A- 10 F3       1430        BPL .1
032C- 60          1440        RTS          RETURN
```

How about a demonstration? I have a list of 20 names (all are subscribers to the Apple Assembly Line), and I want to sort them into alphabetical order. Since I am just writing this to demonstrate using the swap command, I will use one of the WORST sort algorithms: the bubble sort.

Line 100 clears the screen and prints a title line. Line 110 loads the swap program and calls SETUP at 768 ($0300). Line 120 reads in the 20 names from the DATA statement in line 130, and calls a subroutine at line 200 to print the names in a column.

Lines 150-170 are the bubble sort algorithm. If two names are out of order, they are swapped at the end of line 160. Line 180 prints the sorted list of names in a second column.

```
100  TEXT : HOME : PRINT "DEMO USE OF 'STRING SWAP' ROUTINE"
110  DIM A$(20): PRINT  CHR$ (4)"BLOAD B.STRING.SWAP": CALL 768
120  FOR I = 1 TO 20: READ A$(I): NEXT :P = 1: GOSUB 200
130  DATA AMES,BURKE,PUTNEY,LEE,LEVY,RAMSDELL,BISHOP,RANDALL,LAND
     SMAN,LEIPER,OSLISLO,KOVACS,MEADOR,KRIEGSMAN,MERCIER,WHITE,LE
     VY,BLACK,SCHORNAK,STITT
140  REM BUBBLE SORT
150 M = 20
160 M = M - 1:SW = 0: FOR I = 1 TO M: IF A$(I + 1) < A$(I) THEN S
     W = 1: & A$(I + 1),A$(I): REM SWAP
170  NEXT : IF SW THEN 160
180 P = 20: GOSUB 200: END
200  VTAB 3: FOR I = 1 TO 20: HTAB P: PRINT A$(I): NEXT : RETURN
```

A Third Disassembler for the S-C Assembler II?......Lee Meador

A couple of months back, I was reading the most recent copy of the Apple Assembly Line. I was horrified! For some time I had been working on a disassembler that would work with the S-C Assembler II. There before my very eyes were not one, but two, advertisements for disassemblers that would do the same things mine would do, and in some areas they would do a better job. Sure, my disassembler would form the source statements, put in the labels, take care of non-6502 machine language bytes and it would even provide a cross-reference listing as it went along. But, my program forms the source directly in memory. So, you can't disassemble much more than around 1.5K at a time. Secondly, I hadn't gotten it to the point that you could specify various commands and disassemble code, tables, strings, 16-bit data, etc. all in one pass. I had a separate program to do each one.

After thinking about it for some time I decided to sell what I've got. But how could I compete with the other two disassemblers with more features. They are even relatively low priced.

I did come up with a solution. (As if you hadn't guessed that by now.) I am offering my disassembler with the S-C source code and comments. In fact, you will need to assemble the program yourself to get it to work. If you want to join the various modules to create the program I was working toward, you can do that. If you don't like the way I did some part of the program you are free to make whatever changes you like. If the code you want to disassemble lies in the same place in memory as the disassembler, then you just reassemble the program somewhere else. You ARE limited to the size of memory. The disassembler, the table of labels, the program you are disassembling and the source version will have to fit in memory. For large programs you should disassemble in 1K pieces. But, if you choose too large a piece, be prepared to re-boot your system.

Let me make it clear that this disassembler works correctly. I have used it on thousands of bytes of machine language files. It does not die without cause. The source code it produces can be saved and when reassembled, the result matchs the original program.

If you choose to order this from me you will get a 13-sector disk with the S-C Assembler II source code for the various programs. You will not get a well written manual and you will not be able to boot and go. But, where else can you get a working version of a disassembler for only $25.00 with the source code.

Ask for Lee Meador's Disassembler. Enclose check or money order for $25.00 (add $5.00 if outside North America.) Your disassembler source code will be mailed within a day or two of when we receive your order.

Lee Meador, Box 3621, Arlington, TX 76010 (817) 469-6019

(Paid Advertisement)

Apple Assembly Line is published monthly by S-C SOFTWARE, P. O. Box 5537, Richardson, TX 75080. Subscription rate is $12/year, in the U.S.A., Canada, and Mexico. Other countries add $6/year for extra postage.