Apple Assembly Line

Volume 1 -- Issue 3 December, 1980

As I write this, there are 85 paid subscribers! I sent out about 140 flyers in the last two weeks, so maybe the number will double again next month! Pass the word to your friends and local Apple clubs...and let me know how you like the content, style, et cetera.

In This Issue...

Intelligent Disassemblers 2
Integer BASIC Pretty Lister 3
Listed Expressions with .DA Directive 9
Block MOVE and COPY for Version 4.0 11
Handling 16-bit Comparisons 16

Quarterly Disk #1

If you find there just isn't enough time to type in all the source programs in the Apple Assembly Line, I will be happy to save you the trouble. Every three months I will put together a "Disk of the Quarter" which contains all the source in the format of the S-C ASSEMBLER II Version 4.0. The price is only $15, and I will pay the postage.

The first such disk is ready now, covering October, November, and December of 1980. The disks and the programs are for subscribers only. Save your fingers, get yours now!

Help for beginners

I will write some beginner's material from time to time for this newsletter, but I cannot cover every base at once. Meanwhile, many of the magazines and club newsletters are beginning to publish articles for beginners who want to learn assembly language. One of the best and most accessible is Creative Computing. Chuck Carpenter's "Apple-Cart", a monthly feature, in the November, 1980 issue, was great! He actually began the subject of machine language in the October issue, but in the November one he covered indexing, indirect addressing, and interrupts. By the way, Chuck is also a subscriber to the Apple Assembly Line.

There have also been some good beginner articles in recent copies of Nibble and Softalk. Nibble has been printing a lot of assembly language programs, which are good to study.

Intelligent Disassemblers

Not one, but two! In this issue of AAL you find two ads for intelligent disassemblers. Dr. Robert F. Zant, of Decision Systems, and Bob Kovacs, of RAK-WARE, have each written one. After all these years, two of them pop up in the same week!

Dr. Zant's reads a binary file and writes a text file which can be EXECed into either the S-C ASSEMBLER II Version 4.0 or the Apple assembler from the DOS Tool Kit. He writes an intermediate text file during pass one of the disassembly, and then reads in back in, formats it for the desired assembler, and writes it back out. His disassembler is a combination of machine language code and Applesoft code; you have to have Applesoft in ROM and at least 32K RAM. He includes a couple of handy utility programs on the diskette.

Bob Kovac's disassembler works from a binary program already in memory. Both passes are performed in memory, and then the text file is written. Since everything is done in memory, it is very fast. The resulting text file is EXECed into the S-C ASSEMBLER II Version 4.0.

Both disassemblers create labels for all branch addresses inside the block being disassembled. Bob Kovac's version also makes labels for all external branch addresses, putting .EQ lines at the beginning to define them. The RAK-WARE version also makes symbols for all page-zero references. They also are set up with .EQ lines at the beginning of the text file.

Both disassemblers output a control-I at the beginning of each line rather than a line number. This causes the assembler to generate its own line number when the file is EXECed, and allows you to set your own increment and starting line number just before typing the EXEC command. Set the increment by using the INC command; and set the starting line number by typing the number you want less the increment, followed by a space and return.

I forgot to mention, Bob Kovac's disassembler works with either Integer BASIC or Applesoft. He has driver programs written in both languages included on the diskette.

They both are excellent tools, which have long been needed. They both cost the same, $25. What can I say? Buy them both! Do it before the end of 1980, and get a tax deduction before Reagan and our new Congress lower the income tax rate!

Advertising in AAL

For the first time, there are some ads in your newsletter. I think you will find them almost as useful as the non-ad material, because so many of you have asked me for compatible two-pass disassemblers to go along with the S-C ASSEMBLER II. If you have written some programs that you want to sell, which you think other readers of the Apple Assembly Line would be interested in, you can advertise here, too. The cost is quite low...$20 for a full page, $10 for ½ page.

Integer BASIC Pretty Lister

About 2½ years ago Mike Laumer, of Carrollton, Texas, wrote a program to make pretty listings of Integer BASIC programs. He gave me a copy to look at, and then we both forgot about it. A few days ago I found it again, dusted it off, typed it in, and tried it out. After a little debugging, here is the result.

Which is neater?

```
100 FOR I=1 TO 40: A(I)=I: A(I+41)=I*I: NEXT I
```

or?

```
100 FOR I=1 TO 40
    : A(I)=I
    : A(I+41)=I*I
    : NEXT I
```

Mike and I happen to like the latter format, especially for printing in newsletters. It is a lot easier to read. And why print it if no one is going to read it?

If you are in Integer BASIC, and you have a program in memory ready to list, here are the steps to get a "pretty listing".

1. BLOAD B.PRETTY.LISTER
2. POKE 0,40 (or whatever number of characters per line
3. CALL 2048 you wish it to use)

If you want it to print on your printer, be sure to turn it on in the way you usually do before the CALL 2048. For example, if you have a standard Apple interface in slot 1, type "PR#1" just before the CALL 2048.

If you check it out, you will find a lot of similarity between the code in this program and what is stored in the Integer BASIC ROMs around locations $E00C through E0F9. The routines are not in the same order, and there are a few significant changes to make the listing "pretty" and to control the line length. As I was typing in Mike's program, I took the liberty of "modularizing" it a little more, so that I could understand it. The PRINT.DECIMAL routine in lines 2500-2810 is almost identical to the one at $E51B in the BASIC ROMs. The changes are for the purpose of counting the number digits actually printed; this allows a closer control over line length.

Since one of the promised features of the Apple Assembly Line was commented disassemblies of some of Apple's ROM code, I will try to explain how PRETTY.LIST works in some detail, module by module. You can then apply my explanation to the code which resides in ROM at $E00C-$E0F9.

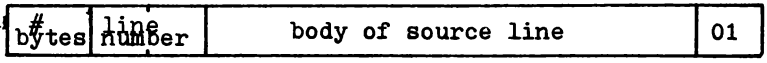
PRETTY.LIST: This module is the overall control for the listing process. Since PP points to the beginning of the BASIC source program, lines 1270-1300 transfer this pointer into SRCP. Then SRCP is compared with HIMEM, to see if we are finished listing. The check is made before even listing one line, because it is possible that there is no source program to list! If the value in SRCP is greater than or equal to the value in HIMEM, then the listing is finished, and PRETTY.LIST returns to BASIC by JMP to DOS.REENTRY (3D0). If the listing is not finished, I call PRINT.ONE.LINE to format and print out one line of the source program. "One line" may be several statements separated by colons. Then I jump back to the test to see whether we are through yet, and so on and on and on.

```
]ASM

                        1000         .TF B.PRETTY.LISTER

                        1020 *-------------------------------
                        1030 *       INTEGER BASIC PRETTY-LIST
                        1040 *-------------------------------
0000-                   1050 LINE.LENGTH          .EQ $00
0001-                   1060 LINE.POSITION        .EQ $01
0024-                   1070 MON.CH               .EQ $24
00CA-                   1080 PP       .EQ $CA,CB
004C-                   1090 HIMEM    .EQ $4C,4D
00E2-                   1100 SRCP     .EQ $E2,E3
00CE-                   1110 TKNP     .EQ $CE,CF
00EA-                   1120 IB.FLAG      .EQ $EA
00FA-                   1130 IB.FILL      .EQ $FA
                        1140 *-------------------------------
03D0-                   1150 DOS.REENTRY          .EQ $3D0
E02A-                   1160 GET.NEXT.BYTE        .EQ $E02A
ED00-                   1170 TOKEN.TABLE          .EQ $ED00
FDED-                   1180 MON.COUT             .EQ $FDED
FD8E-                   1190 MON.CROUT            .EQ $FD8E
                        1200 *-------------------------------
0001-                   1210 TOKEN.EOL            .EQ $01
0003-                   1220 TOKEN.COLON          .EQ $03
005D-                   1230 TOKEN.REM            .EQ $5D
0028-                   1240 TOKEN.QUOTE          .EQ $28
                        1250 *-------------------------------
                        1260 PRETTY.LIST
0800- A5 CA             1270         LDA PP
0802- 85 E2             1280         STA SRCP
0804- A5 CB             1290         LDA PP+1
0806- 85 E3             1300         STA SRCP+1
0808- A5 E2             1310 .1      LDA SRCP       SEE IF AT END
080A- C5 4C             1320         CMP HIMEM
080C- A5 E3             1330         LDA SRCP+1
080E- E5 4D             1340         SBC HIMEM+1
0810- B0 06             1350         BCS .2         FINISHED
0812- 20 1B 08          1360         JSR PRINT.ONE.LINE
0815- 4C 08 08          1370         JMP .1
0818- 4C D0 03          1380 .2      JMP DOS.REENTRY
                        1390 *-------------------------------
                        1400 PRINT.ONE.LINE
081B- A9 00             1410         LDA #0
081D- 85 01             1420         STA LINE.POSITION
081F- A9 A0             1430         LDA #$A0       SET UP PRINT.DECIMAL FOR RJBF
0821- 85 FA             1440         STA IB.FILL
0823- 20 2A E0          1450         JSR GET.NEXT.BYTE SKIP OVER BYTE COUNT
0826- 98                1460         TYA            (A)=0
0827- 85 EA             1470 .1      STA IB.FLAG
0829- 20 2A E0          1480         JSR GET.NEXT.BYTE GET LINE NUMBER
082C- AA                1490         TAX            LOW BYTE
082D- 20 2A E0          1500         JSR GET.NEXT.BYTE HIGH BYTE
0830- 20 DE 08          1510         JSR PRINT.DECIMAL  PRINT THE LINE NUMBER
0833- A9 F9             1520 .2      LDA #-7        WITHIN 7 OF END OF LINE
0835- 20 BD 08          1530         JSR CHECK.EOL.GET.NEXT.BYTE
0838- 84 FA             1540         STY IB.FILL    CLEAR RJBF
083A- AA                1550         TAX            TEST BYTE AND SAVE IN X-REG
083B- 10 1C             1560         BPL .6         TOKEN
083D- 0A                1570         ASL
083E- 10 E7             1580         BPL .1         CONSTANT, GO PRINT IT
0840- A5 EA             1590         LDA IB.FLAG
0842- D0 07             1600         BNE .3         DO NOT NEED A BLANK
0844- A9 A0             1610         LDA #$A0
0846- 85 EA             1620         STA IB.FLAG
0848- 20 D9 08          1630         JSR CHAR.OUT
084B- 8A                1640 .3      TXA            RETRIEVE BYTE
084C- 20 D9 08          1650 .4      JSR CHAR.OUT   AND PRINT IT
084F- A9 FD             1660 .5      LDA #-3        WITHIN 3 OF EOL
0851- 20 BD 08          1670         JSR CHECK.EOL.GET.NEXT.BYTE
0854- AA                1680         TAX            TEST BYTE, SAVE IN X-REG
0855- 30 F5             1690         BMI .4         NORMAL CHAR
0857- 85 EA             1700         STA IB.FLAG
                        1710 *-------------------------------
0859- C9 01             1720 .6      CMP #TOKEN.EOL
085B- D0 03             1730         BNE .7         NOT END OF LINE
085D- 4C 8E FD          1740         JMP MON.CROUT  END OF LINE
0860- C9 03             1750 .7      CMP #TOKEN.COLON
0862- D0 05             1760         BNE .8
```

PRINT.ONE.LINE: A source line in Integer BASIC is encoded in token form, and this routine has to convert it back to the original form to list it. First, let's look at how a coded line is laid out.



The first byte of a line is the line length; we will ignore it in this program, because we do not need it. The last byte of each line is the hex value $01, which is the token for end-of-line. That is all we need to signal the end of a line, and the start of another one. The second and third bytes of each line are the line number, in binary, with the low byte first. The body of the line is made up of a combination of tokens and ascii characters.

For the most part, tokens have a hex value less than $80, while the ascii characters have a hex value greater than $80. One important exception is the token for a decimal constant. These are flagged by a pseudo-token consisting of the first digit of the constant in ascii (hex $B0 through B9); after the token, two bytes follow which contain the binary form of the constant with the low byte first. For example, the decimal constant 1234 would be stored in three bytes as: B1 D2 04.

The task of PRINT.ONE.LINE is to scan through the coded form of a line, printing each ascii character, and converting each token to its printing form. In addition, the routine must count line position as it goes, so that a new line can be started when one fills up. Furthermore, we want it to start a new line whenever the ":" indicates a new statement has begun within a line. We have to look out for REM statements and quoted strings, because the ":" might appear in them without signalling a new statement.

Lines 1400-1460 start the ball rolling. The line position is set to zero, and the fill flag for the PRINT.DECIMAL routine is set to produce a right-justified-blank-filled number. Then GET.NEXT.BYTE is called to advance the SRCP past the byte count in the first byte of the line. GET.NEXT.BYTE returns the value of the byte in A, and with Y=0. This time we ignore the value in A, and use the fact that Y=0 to clear A.

Lines 1470-1510 pick up the two bytes of the line number and call PRINT.DECIMAL to print it out. These same lines will be used later to print out any constants which are in the line. These lines are entered this time with A=0 and with IB.FILL set for the RJBF mode (right-justified-blank-filled). Later for constants they will be entered with IB.FILL set for printing with no leading blanks, and with A≠0. The value in A is used to set IB.FLAG, which determines whether a trailing blank will be printed. One will be printed after the line number, but not after a constant inside a line. (For a character that uses so little ink, blanks can sure eat up a lot of code!)

At line 1520 the main body of the PRINT.ONE.LINE routine begins. CHECK.EOL.GET.NEXT.BYTE decides whether we are getting too close to the end of the line. This prevents splitting token-words

in half, with a few characters dangling off the end of one line, and the rest starting a new one. (At least, on the screen it would look like that; on a printer it might just print out into a margin.) The routine will start a new line before returning if the end is too near. When it finally does return, the next byte will be in A, and Y will be zero. If the next byte is a token (less than $80), control branches to line 1720. If the first bit of the byte is 1, and the second bit is 0, the code at lines 1550-1580 assumes the pseudo-token for a constant has appeared. If the second bit is also 1, the byte is an ascii character. Before printing the character, lines 1590-1630 may print a blank. This would be a trailing blank after printing a token or a line number. The character is then printed at lines 1640-1650, and another end-of-line check is made. This time "too near the end" is defined as within 3 spaces. The next byte must either be a token or yet another ascii character, so a determination is made in lines 1680-1700.

Tokens are harder to handle, because we have to test for several special cases, and if not a special case the token table must be searched to find the token's name. Lines 1720-1740 test for the end-of-line token; if this is it, a carriage return is printed and PRINT.ONE.LINE returns back to its caller.

If the token is the new-statement-token, used for ":", a new line is started. Then the fun begins: we have to search the token table. This table is the most recondite portion of the whole Apple computer! I have only scratched its surface. The table is located between $EC00 and $EDFF, but it is not in that order. It goes like this: first $ED00, then $EDFF-$ED01 (yes, backwards!), then $EC00, then $ECFF-$EC01. The names for all the tokens are stored in the table, along with various bits of information about precedence and syntax. If you print out the table, you will not see any names... Steve Wozniak subtracted $20 from each byte before putting it into the table. Well, there is a lot more to it than that, but I am getting lost, side-tracked.

After finding the token's name string inside the token table, we have to print it out. This is done in lines 1840-1940. The name is terminated either by the last character having a value greater than $BF, or by the next character in the table having a value less than $80. The routine at $E00C decides whether or not to print a trailing blank, I think.

After printing the token's name, lines 1960-2010 test for REM or a quoted string. Either of these would be followed by a bunch of ascii characters terminated by a token, so control branches to line 1660 to print them out. If neither, we go back to line1520, to get the next token, or whatever.

Somehow I skipped over line 1830. I believe the JSR $EFF8 determines whether or not to print a space in front of the token name.

```
0864- 20 CA 08 1770          JSR CR.7.BLANKS
0867- A9 03    1780          LDA #TOKEN.COLON
0869- 48       1790 .8       PHA             SAVE TOKEN
086A- 20 96 08 1800          JSR FIND.TOKEN
086D- 24 EA    1810          BIT IB.FLAG
086F- 30 03    1820          BMI .9
0871- 20 F8 EF 1830          JSR $EFF8
0874- B1 CE    1840 .9       LDA (TKNP),Y  GET CHAR OF TOKEN NAME
0876- 10 10    1850          BPL .10
0878- AA       1860          TAX             SAVE CHAR IN X
0879- 29 3F    1870          AND #$3F
087B- 85 EA    1880          STA IB.FLAG
087D- 18       1890          CLC
087E- 69 A0    1900          ADC #$A0
0880- 20 D9 08 1910          JSR CHAR.OUT
0883- 88       1920          DEY
0884- E0 C0    1930          CPX #$C0
0886- 90 EC    1940          BCC .9
0888- 20 0C E0 1950 .10      JSR $E00C
088B- 68       1960          PLA             GET ORIGINAL UNMOLESTED TOKEN
088C- C9 5D    1970          CMP #TOKEN.REM
088E- F0 BF    1980          BEQ .5          REM
0890- C9 28    1990          CMP #TOKEN.QUOTE
0892- F0 BB    2000          BEQ .5          QUOTATION
0894- D0 9D    2010          BNE .2          NEITHER
               2020 *-------------------------------
               2030 FIND.TOKEN
0896- A2 00    2040          LDX #TOKEN.TABLE
0898- 86 CE    2050          STX TKNP
089A- A2 ED    2060          LDX /TOKEN.TABLE
089C- C9 51    2070          CMP #$51        SEE IF NEED OTHER HALF
089E- 90 03    2080          BCC .1          NO
08A0- CA       2090          DEX             YES             TOKEN.TABLE
08A1- E9 50    2100          SBC #$50
08A3- 86 CF    2110 .1       STX TKNP+1
08A5- 48       2120 .2       PHA             SAVE MODIFIED TOKEN ON STACK
08A6- B1 CE    2130          LDA (TKNP),Y  Y GOES 0,FF,FE,...
08A8- AA       2140 .3       TAX
08A9- 88       2150          DEY
08AA- B1 CE    2160          LDA (TKNP),Y LOOK FOR NEGATIVE BYTE
08AC- 10 FA    2170          BPL .3
08AE- E0 C0    2180          CPX #$C0        IF BYTE BEFORE NEGATIVE BYTE
08B0- B0 04    2190          BCS .4          BTWN $C0 AND $FF, THEN
08B2- E0 00    2200          CPX #$00        KEEP LOOKING
08B4- 30 F2    2210          BMI .3
08B6- AA       2220 .4       TAX
08B7- 68       2230          PLA
08B8- E9 01    2240          SBC #1          DECREMENT TOKEN
08BA- D0 E9    2250          BNE .2          NOT THERE YET
08BC- 60       2260          RTS
               2270 *-------------------------------
               2280 CHECK.EOL.GET.NEXT.BYTE
08BD- 18       2290          CLC
08BE- 65 00    2300          ADC LINE.LENGTH
08C0- C5 01    2310          CMP LINE.POSITION
08C2- B0 03    2320          BCS .1
08C4- 20 CA 08 2330          JSR CR.7.BLANKS
08C7- 4C 2A E0 2340 .1       JMP GET.NEXT.BYTE
               2350 *-------------------------------
               2360 CR.7.BLANKS
08CA- A9 8D    2370          LDA #$8D
08CC- A0 07    2380          LDY #7
08CE- 84 01    2390          STY LINE.POSITION
08D0- 20 D9 08 2400 .1       JSR CHAR.OUT
08D3- A9 A0    2410          LDA #$A0
08D5- 88       2420          DEY
08D6- D0 F8    2430          BNE .1
08D8- 60       2440          RTS
               2450 *-------------------------------
               2460 CHAR.OUT
08D9- E6 01    2470          INC LINE.POSITION
08DB- 4C ED FD 2480          JMP MON.COUT
               2490 *-------------------------------
               2500 PRINT.DECIMAL
08DE- 85 F3    2510          STA $F3
08E0- 86 F2    2520          STX $F2
08E2- A2 04    2530          LDX #4
08E4- 85 C9    2540          STA $C9         LEADING ZERO FLAG
08E6- A9 B0    2550 .7       LDA #$B0
08E8- 85 F9    2560          STA $F9
```

FIND.TOKEN: Lines 2040-2110 set up a pointer to the half of the token table which contains the name string for the token we wnat. Tokens 00 through 50 are in the first half, and 51 through 7F are in the second half.

Lines 2120-2250 scan through the table, counting token names as they are passed. When the nth one is found, where n is the token value, the routine returns. It returns with A=0, and Y = offset in the half of the token table we have been scanning.

CHECK.EOL.GET.NEXT.BYTE: Enter this routine with A containing the number of bytes short of the end of the line you want to test for, as a negative number. If too near the end, CR.7.BLANKS will be called to start a new line. In any case the routine exits by transfering to GET.NEXT.BYTE to get the next byte from the source line.

CR.7.BLANKS: Prints a carriage return and 7 blanks to start a new line.

CHAR.OUT: Simply counts columns and then calls on the Apple monitor to print out a character. We need to count columns for CHECK.EOL.GET.NEXT.BYTE.

PRINT.DECIMAL: Lifted out of Integer BASIC from $E51B, and modified to eliminate the ability to store the converted number in the input buffer, and to add the ability to count output characters.

Additions to this program: You might like to add some more features to this program. For example, it would be nice to have it request the line length and printer slot number itself, and turn the printer on and off. Also it would be helpful to add indentation for FOR...NEXT loops and IF...THEN statements. The same program could be merged with a cross reference program to build and print a variable and line number cross reference.

If you decide to try any of these, or any other enhancements, why not write them up and send them to me for publication?

```
08EA- A5 F2    2570 .1    LDA $F2
08EC- DD 63 E5 2580       CMP $E563,X
08EF- A5 F3    2590       LDA $F3
08F1- FD 68 E5 2600       SBC $E568,X
08F4- 90 0D    2610       BCC .2
08F6- 85 F3    2620       STA $F3
08F8- A5 F2    2630       LDA $F2
08FA- FD 63 E5 2640       SBC $E563,X
08FD- 85 F2    2650       STA $F2
08FF- E6 F9    2660       INC $F9
0901- D0 E7    2670       BNE .1        ...ALWAYS
0903- A5 F9    2680 .2    LDA $F9
0905- E0 00    2690       CPX #0        SEE IF LAST DIGIT
0907- F0 0E    2700       BEQ .4        YES
0909- C9 B0    2710       CMP #$B0      NO, SEE IF LEADING ZERO
090B- F0 02    2720       BEQ .3        MAYBE
090D- 85 C9    2730       STA $C9       NO
090F- 24 C9    2740 .3    BIT $C9       STILL PLUS IF LEADING ZERO
0911- 30 04    2750       BMI .4        NOT LEADING ZERO
0913- A5 FA    2760       LDA IB.FILL   SEE IF BLANK FILL
0915- F0 03    2770       BEQ .5        NO
0917- 20 D9 08 2780 .4    JSR CHAR.OUT  PRINT CHAR
091A- CA       2790 .5    DEX
091B- 10 C9    2800       BPL .7
091D- 60       2810       RTS
```

Allow List of Expressions with .DA Directive

Some customers have said they wished the .DA directive in the S-C ASSEMBLER II allowed more than one expression per line. For example, ".DA 1000,100,10,1" would then produce 8 bytes of code just as though there were four separate .DA lines. (Once and a while I wish it worked this way too!)

The following little patch will transform your .DA in just that way. Because of the .OR and .TF directives, assembling these 42 lines will produce two binary files that are ready to BLOAD. When you BLOAD them, the copy of the assembler in memory will be patched. You can then BSAVE the assembler (use a different name!), and you have the new capability.

If you do not have version 4.0 of the assembler, then this patch will not work. If you have one of the very earliest copies of version 4.0, it may have some different addresses. Check it out before you type in the code: at $20D4 you should find three JMP instructions, as indicated in the comments here in lines 1210 through 1230. If you find those JMP's, go right ahead and make the patches. Of course, if you have already added some code at 24B0, then you will have to put this patch somewhere else.

If you do not find those JMP instructions at 20D4, but you do find them at $20B1, then you need to change a few addresses in the patch code. Change the following lines as indicated:

```
1170 PSDA    .EQ $2092
1190         .OR $20B1
```

```
                        1000 *--------------------------------
                        1010 *       PATCH FOR .DA WITH COMMA LIST
                        1020 *--------------------------------
                        1030 *
                        1040 *       TO INSTALL THIS PATCH:
                        1050 *
                        1060 *       1.  BRUN ASMDISK 4.0
                        1070 *       2.  BLOAD PATCH.DA.1
                        1080 *       3.  BLOAD PATCH.DA.2
                        1090 *       4.  BSAVE ASMDISK 4.1,A$1000,L$14FB
                        1100 *
                        1110 *--------------------------------
00DB-                   1120 EXP.VALUE .EQ $DB
                        1130 *--------------------------------
128B-                   1140 GNC     .EQ $128B
19FA-                   1150 EMIT    .EQ $19FA
188E-                   1160 CMNT    .EQ $188E
20B5-                   1170 PSDA    .EQ $20B5
                        1180 *--------------------------------
                        1190         .OR $20D4          REPLACES:
                        1200         .TF PATCH.DA.1
20D4- 4C B0 24          1210         JMP BOTH.BYTES     (JMP $19B2)
20D7- 4C C7 24          1220         JMP LOW.BYTE       (JMP $194D)
20DA- 4C B5 24          1230         JMP HIGH.BYTE      (JMP $19D7)
                        1240 *--------------------------------
                        1250         .OR $24B0     PATCH AREA
                        1260         .TF PATCH.DA.2
                        1270 BOTH.BYTES
24B0- A5 DB             1280         LDA EXP.VALUE
24B2- 20 FA 19          1290         JSR EMIT
                        1300 HIGH.BYTE
24B5- A5 DC             1310         LDA EXP.VALUE+1
24B7- 20 FA 19          1320 ALL     JSR EMIT
24BA- 20 8B 12          1330         JSR GNC
24BD- C9 2C             1340         CMP #',       COMMA?
24BF- F0 03             1350         BEQ MORE
24C1- 4C 8E 18          1360         JMP CMNT      FINISHED
24C4- 4C B5 20          1370 MORE    JMP PSDA
                        1380 LOW.BYTE
24C7- A5 DB             1390         LDA EXP.VALUE
24C9- 18                1400         CLC
24CA- 90 EB             1410         BCC ALL       ...ALWAYS
```

Block MOVE and COPY for Version 4.0

How many times have you wished there was an easy way to move a bunch of lines of your source program to some other place? I know it happens to me, an I frequently wish the assembler had this capability. Now, at last, it is possible. I no longer have to use DELETE, SAVE, HIDE, MERGE, LOAD in a very complicated sequence just to move that 20 line subroutine from the middle to the end of my source program!

The program as written assumes you have set up the USR command vector to jump to $800. You do this by stuffing a 0 into $1007 and an 8 into $1008 (type $1007:00 08 as a command). Then if you type, for example,"USR 1100,1190,1800", a copy of lines 1100 through 1190 will be inserted before line 1800. A word of caution: the lines in their new location will still have the old line numbers, until you RENUMBER. You can LIST, SAVE, and LOAD while the lines are out of sequence like this, but beware of doing any further editing! First, use the USR command to make the new copy of the lines; second, RENUMBER the program; third, DELETE the lines from their old location. Voila! You have moved them.

I just know someone (maybe everyone) is going to think that I should have made this program do its own renumbering. The reason I am confident of this is that I feel the same way. But the program as it stands is useful, and I will refine it later. My plan is to add one more parameter which specifies the increment for the line numbers in their new location. Then let the third parameter be the new line number for the first line of the block being copied. The program will check whether making the copy will clobber any existing lines, and error out if so. If not, the copy will be made with its new line numbers. Then a question will be asked of the form "DO YOU WISH TO DELETE THE OLD LINES? (Y/N)". But for now, I will live with the more tedious but still very useful version you see here.

I would suggest that you put the object code of this program on a binary file, and then create an EXEC text file that contains the patch line to set up the USR command and a BLOAD command for the COPY program. The quarterly AAL diskette contains just such a file.

Now let me describe how the COPY program works. Notice that lines 1000-1060 are a summary of the operating syntax. Line 1070, together with lines 2390 and 2400, make the last three symbols in the symbol table listing tell me the start, end, and length of the object code. These are very useful for writing the object code out to a binary file. (Of course, I could use the .TF directive and write it automatically.)

Lines 1090-1220 define the page-zero locations the program uses. SS, SE, SL, and NEWPP are peculiar to this program; the rest of them are used by the monitor and the assembler. PP points to the beginning of the first source line in memory, and LOMEM is the lowest PP can go. A0, A1, A2, and A4 are used to pass addresses to the Apple Monitor Memory Move subroutine.

Lines 1240-1280 define some addresses of routines inside the S-C ASSEMBLER II Version 4.0. SYNX is the Syntax Error routine. You will get a syntax error message if you type in less than three parameters with the USR command, if the first two parameters are backwards or the same, if the block specified to be copied is empty, or if the target location is inside the block to be copied. MFER is the routine to print MEM FULL ERR, and you will get this error message if there is not room to make a copy; that is, the space between PP and LOMEM is less than the size of the block you want to copy.

SCND is the assembler routine to scan an input line from the current position and look for a decimal number. If it finds a decimal number, it will convert the number to binary and store it in A2L and A2H. As explained on page 10 of the Upgrade manual for version 4.0, the first two parameters will have already been stored in A0 and A1.

SERTXT is the assembler routine to find a line in your source program, given the line number. It is called with the X-register containing the address of the first byte in page-zero of the byte-pair containing the line number you are looking for. When SERTXT is finished, $E4,E5 points at the first byte of the line found, and $E6,$E7 points at the first byte of the next line. (Of course, if your line number could not be found, both pointers will point at the next larger line.)

MON.MOVE is a program inside the Apple Monitor ROM. It will copy a block of memory whose first byte address is in A1, last byte address in A2, to a new place in memory starting at the byte address in A4. This is the routine used when you use the monitor "M" command. It works fine as long as the target is not inside the source block.

Now to the COPY program itself. Briefly, the three parameters are checked for presence and consistency, and pointers are set up defining the area to be copied. A new value of PP is computed based on the length of this block, and I check to see if there is room in memory. Next I search for the target location, and check to make sure it is not inside the source block. (We don't want any infinite loops!) If the target is higher in memory than the source block, I adjust the source block pointers by subtracting the block length from them. Then I move all source lines below the insertion point down in memory far enough to make a hole in the text into which the source block can be copied. Finally, I copy in the source block, and return.

Some final comments.... The COPY program is very fast, so play with a little on a scratch program to convince yourself it is working. If you don't want to type in the source, you can just enter the hex codes from the monitor, and BSAVE it. Or, you can order the Quarterly AAL diskette, which will have the source, object, and a textfile to EXEC for BLOADing and patching the USR vector. Or, if you are very patient, you can wait till next August for version 5.0 of the S-C ASSEMBLER II!

```
:ASM

                        1000 *-------------------------------
                        1010 *       COPY L1,L2,L3
                        1020 *            L1 = FIRST LINE OF RANGE TO COPY
                        1030 *            L2 = LAST LINE OF RANGE TO COPY
                        1040 *            L3 = LINE NUMBER BEFORE WHICH TO INSERT
                        1050 *                 THE COPIED LINES
                        1060 *-------------------------------
0800-                   1070 ZZ.BGN .EQ *
                        1080 *-------------------------------
0000-                   1090 SS     .EQ $00,01    START OF SOURCE BLOCK
0002-                   1100 SE     .EQ $02,03    END OF SOURCE BLOCK
0004-                   1110 SL     .EQ $04,05    LENGTH OF SOURCE BLOCK
0006-                   1120 NEWPP  .EQ $06,07    NEW PROGRAM POINTER
003A-                   1130 A0L    .EQ $3A
003B-                   1140 A0H    .EQ $3B
003C-                   1150 A1L    .EQ $3C
003D-                   1160 A1H    .EQ $3D
003E-                   1170 A2L    .EQ $3E
003F-                   1180 A2H    .EQ $3F
0042-                   1190 A4L    .EQ $42
0043-                   1200 A4H    .EQ $43
004A-                   1210 LOMEM  .EQ $4A,4B
00CA-                   1220 PP     .EQ $CA,CB
                        1230 *-------------------------------
105E-                   1240 SYNX   .EQ $105E
1128-                   1250 MFER   .EQ $1128
112D-                   1260 SCND   .EQ $112D
14F6-                   1270 SERTXT .EQ $14F6
FE2C-                   1280 MON.MOVE   .EQ $FE2C
                        1290 *-------------------------------
0800- 4C 09 08          1300        JMP COPY
                        1310 *-------------------------------
0803- 4C 5E 10          1320 ERR1   JMP SYNX
0803-                   1330 ERR2   .EQ ERR1
0806- 4C 28 11          1340 ERR3   JMP MFER
0803-                   1350 ERR4   .EQ ERR1
                        1360 *-------------------------------
                        1370 COPY
0809- 20 2D 11          1380        JSR SCND      GET THIRD PARAMETER
080C- E0 06             1390        CPX #6        BE SURE WE GOT THREE
080E- 90 F3             1400        BCC ERR1      NOT ENOUGH PARAMETERS
0810- A2 3A             1410        LDX #A0L      FIND BEGINNING OF SOURCE
0812- 20 F6 14          1420        JSR SERTXT
0815- A5 E4             1430        LDA $E4       SAVE POINTER
0817- 85 00             1440        STA SS
0819- A5 E5             1450        LDA $E5
081B- 85 01             1460        STA SS+1
081D- A2 3C             1470        LDX #A1L      FIND END OF SOURCE BLOCK
081F- 20 F6 14          1480        JSR SERTXT
0822- 38                1490        SEC           SAVE POINTER AND COMPUTE
0823- A5 E6             1500        LDA $E6                     LENGTH
0825- 85 02             1510        STA SE
0827- E5 00             1520        SBC SS
0829- 85 04             1530        STA SL        SOURCE LENGTH
082B- A5 E7             1540        LDA $E7
082D- 85 03             1550        STA SE+1
082F- E5 01             1560        SBC SS+1
0831- 85 05             1570        STA SL+1
0833- 90 CE             1580        BCC ERR2      RANGE BACKWARD
0835- D0 04             1590        BNE .4
0837- A5 04             1600        LDA SL
0839- F0 C8             1610        BEQ ERR2      NOTHING TO MOVE
                        1620 *-------------------------------
083B- A5 CA             1630 .4     LDA PP        COMPUTE NEW PP POINTER
083D- E5 04             1640        SBC SL
083F- 85 06             1650        STA NEWPP
0841- A5 CB             1660        LDA PP+1
0843- E5 05             1670        SBC SL+1
0845- 85 07             1680        STA NEWPP+1
                        1690 *-------------------------------
0847- A5 06             1700        LDA NEWPP     SEE IF ROOM FOR THIS
0849- C5 4A             1710        CMP LOMEM
084B- A5 07             1720        LDA NEWPP+1
084D- E5 4B             1730        SBC LOMEM+1
084F- 90 B5             1740        BCC ERR3      MEM FULL ERR
                        1750 *-------------------------------
```

```
0851- A2 3E      1760          LDX $A2L       FIND TARGET LOCATION
0853- 20 F6 14   1770          JSR SERTXT
0856- A5 00      1780          LDA SS         BE SURE NOT INSIDE SOURCE
0858- C5 E4      1790          CMP $E4
085A- A5 01      1800          LDA SS+1
085C- E5 E5      1810          SBC $E5
085E- B0 24      1820          BCS .1         BELOW SOURCE BLOCK
0860- A5 E4      1830          LDA $E4
0862- C5 02      1840          CMP SE
0864- A5 E5      1850          LDA $E5
0866- E5 03      1860          SBC SE+1
0868- 90 99      1870          BCC ERR4       INSIDE SOURCE BLOCK
                 1880 *    TARGET IS ABOVE SOURCE BLOCK, SO WE HAVE TO
                 1890 *    ADJUST SOURCE BLOCK POINTERS.
086A- 38         1900          SEC
086B- A5 00      1910          LDA SS
086D- E5 04      1920          SBC SL         SS = SS - SL
086F- 85 00      1930          STA SS
0871- A5 01      1940          LDA SS+1
0873- E5 05      1950          SBC SL+1
0875- 85 01      1960          STA SS+1
0877- 38         1970          SEC
0878- A5 02      1980          LDA SE
087A- E5 04      1990          SBC SL         SE = SE - SL
087C- 85 02      2000          STA SE
087E- A5 03      2010          LDA SE+1
0880- E5 05      2020          SBC SL+1
0882- 85 03      2030          STA SE+1
                 2040 *-------------------------------
0884- A5 CA      2050 .1       LDA PP         SET UP MOVE TO MAKE HOLE
0886- 85 3C      2060          STA A1L
0888- A5 CB      2070          LDA PP+1
088A- 85 3D      2080          STA A1H
088C- A5 06      2090          LDA NEWPP
088E- 85 CA      2100          STA PP
0890- 85 42      2110          STA A4L
0892- A5 07      2120          LDA NEWPP+1
0894- 85 CB      2130          STA PP+1
0896- 85 43      2140          STA A4H
0898- A5 E5      2150          LDA $E5
089A- 85 3F      2160          STA A2H
089C- A5 E4      2170          LDA $E4
089E- 85 3E      2180          STA A2L
08A0- D0 02      2190          BNE .2
08A2- C6 3F      2200          DEC A2H
08A4- C6 3E      2210 .2       DEC A2L        A2 = A2 - 1
08A6- A0 00      2220          LDY #0
08A8- 20 2C FE   2230          JSR MON.MOVE   A4<A1.A2M
                 2240 *-------------------------------
08AB- A5 00      2250 .5       LDA SS         MOVE IN SOURCE BLOCK
08AD- 85 3C      2260          STA A1L           (MON.MOVE left A4
08AF- A5 01      2270          LDA SS+1           pointing at first
08B1- 85 3D      2280          STA A1H            byte of the hole.)
08B3- A5 03      2290          LDA SE+1
08B5- 85 3F      2300          STA A2H
08B7- A5 02      2310          LDA SE
08B9- 85 3E      2320          STA A2L
08BB- D0 02      2330          BNE .3
08BD- C6 3F      2340          DEC A2H          A2 = A2 - 1
08BF- C6 3E      2350 .3       DEC A2L
08C1- 20 2C FE   2360          JSR MON.MOVE     A4<A1.A2M
08C4- 60         2370          RTS
                 2380 *-------------------------------
08C4-            2390 ZZ.END   .EQ *-1
00C5-            2400 ZZ.SIZ   .EQ ZZ.END-ZZ.BGN+1
```

LDA A2L
CMP A1L
LDA A2H
SBC A1H
BCC .5

SYMBOL TABLE

```
003B- A0H
003A- A0L
003D- A1H
003C- A1L
003F- A2H
003E- A2L
0043- A4H
0042- A4L
0809- COPY
.04=083B, .01=0884, .02=08A4, .03=08BF
0803- ERR1
0803- ERR2
0806- ERR3
0803- ERR4
004A- LOMEM
1128- MFER
FE2C- MON.MOVE
0006- NEWPP
00CA- PP
112D- SCND
0002- SE
14F6- SERTXT
0004- SL
0000- SS
105E- SYNX
0800- ZZ.BGN
08C4- ZZ.END
00C5- ZZ.SIZ
```

Keeping Printer On After Error Message

One customer wanted this, and maybe you would too. He needed the printer to stay enabled even if an editor or assembler error message was generated. S-C ASSEMBLER II Version 4.0 shuts off any printer after any error occurs, so he couldn't get his printer to stay on long enough to get a listing.

Here is a patch that will leave a printer "hooked in".

```
:$1756:F0 24     (address of patch area)
:$24F0:A9 FF 85 D9 20 80 1F 4C 26 10
```

After making the patch, you can BSAVE using A$1000, L$14FB.

The patch is put at 24F0; if you have already put some other patch there, be sure to put this one somewhere else! Be sure you TEST it before you clobber or delete the original! Be sure you really WANT it before you even bother to type it in!

Handling 16-bit Comparisons

It can be confusing enough in the 6502 to compare two single-byte values. Trying to remember that BCC means "branch if less than" (assuming that the values were considered to be unsigned values from 0-255), and that BCS means "branch if greater than or equal to" is enough to saturate my memory banks. I finally made a note on a card and tacked it up over my computer. Of course, if the values are considered to be signed values, in the range of -128 through +127, the problem is compounded, to say the least.

But what about comparing two values of two-bytes each? Like comparing to address pointers, for instance? A last resort would be to subtract one from the other, in two-byte arithmetic and then compare the difference to zero. At least that would be understandable! But let's try to do it a little better than that. There is an example of this kind of comparison in lines 1310 through 1350 of the PRETTY.LIST program elsewhere in this issue of the Apple Assembly Line. Here is the segment:

```
1310 .1     LDA SRCP
1320        CMP HIMEM
1330        LDA SRCP+1
1340        SBC HIMEM+1
1350        BCS .2
```

The object is to determine whether the value in PP,PP+1 is still less than the value in HIMEM,HIMEM+1 or not. The low-order byte of each value is stored in the first byte of each byte-pair, and the high-order byte is stored in the second byte. If all we needed to compare was the low-order bytes, we could do it with lines 1310 and 1320 above. Carry would be cleared by the CMP instruction if (SRCP) was less than (HIMEM). (I have just started using "(" and ")" to mean "the value stored in".)

Now let's use that carry bit and continue the comparison by actually subtracing the two high-order bytes. If the result of the subtraction leaves carry clear, we know that (SRCP) is indeed less than (HIMEM), all 16 bits of it.

If you need to extend this to more than two bytes per value, you may. Just insert a pair of LDA-SBC instructions for each extra byte of precision, before the BCS instruction.

For another example of this kind of comparison, you might look up the NXTA1 routine in the Apple Monitor listing, at $FCBA. This routine is used by the Monitor MOVE command, and several other routines.

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.