SWEETS for KIM

A Low Calorie Text Editor

Dan Fylstra
22 Weitz St #3
Boston MA 02134

```
ENERG   LDY   #10     SET UP FOR 10 MSEC DELAY
        JSR   WAIT    LOOP FOR THAT LONG
        LDY   #0      SEND 0'S TO OUTPUT PORT
        STY   PORT    TO TURN OFF MAGNET CURRENT
        RTS           RETURN TO CALLER
WAIT    LDX   #200    NO. TIMES THRU INNER LOOP
LOOP    DEX           DECREMENT INNER LOOP COUNT
        BNE   LOOP    LOOP UNTIL COUNT IS 0
        DEY           DECREMENT OUTER LOOP COUNT
        BNE   WAIT    LOOP UNTIL COUNT IS 0
        RTS           RETURN TO CALLER
```

Listing 1a: A segment of 6502 assembly language code used to demonstrate SWEETS, a Simple Way to Enter, Edit and Test Software. SWEETS is a small text editor and assembler which operates on hexadecimal code and which is designed to fit in the KIM-1's 1 K byte small memory while leaving room for the user's programs. The key sequence for editing is shown in table 1b.

(AD) F F 0 1 0 0	F	F	0	1	0	0
(+) (AD) A 0 0 A	A	0	0	A		
(+) (AD) 2 0 0 2 0 0	2	0	0	2	0	0
(+) (AD) A 0 0 0	A	0	0	0		
(+) (AD) 8 C 0 0 1 7	8	C	0	0	1	7
(+) (AD) 6 0	6	0				
(+) (AD) F F 0 2 0 0	F	F	0	2	0	0
(+) (AD) A 2 C 8	A	2	C	8		
(+) (AD) F F 0 3 0 0	F	F	0	3	0	0
(+) (AD) C A	C	A				
(+) (AD) D 0 0 3	D	0	0	3		
(+) (AD) 8 8	8	8				
(+) (AD) D 0 0 2	D	0	0	2		
(+) (AD) 6 0	6	0				

Table 1a: The sequence of keys used to enter the program in listing 1a when using the SWEETS editor and assembler. The right side of the table shows the resulting LED readout seen at each step. Notice that an entire instruction is entered and displayed at one time.

If you would like to experiment with microcomputers on a limited budget, the MOS Technology KIM-1 is an excellent choice. For $245, it comes preassembled with, among other things, a 6502 microprocessor, a read only memory monitor, an audio cassette interface, 1 K bytes of programmable memory, and its own special peripheral: a 23 key keyboard plus a 6 digit LED display. The monitor lets you load a machine language program byte by byte from the keyboard, and once loaded the program can be saved on tape via the audio cassette interface. The KIM-1 manual shows how you can "hand-translate" an assembly language program into the absolute hexadecimal form required for keyboard entry.

This is fine for very small programs, but the process of hand translation gets rather tedious after you've assembled a few hundred bytes of code. And, worse, once you've painstakingly worked out all the subroutine call addresses and branch displacements and keyed the whole program in, you invariably find that you've forgotten something. Often, instructions must be inserted or deleted in the middle of the program, which throws everything off by a few bytes.

The obvious solution to this problem is to obtain a text editor and assembler program for the 6502. But, alas, such a program probably needs more than the 1 K bytes of memory provided on the KIM-1, and, more seriously, it requires an alphabetic character terminal device such as a Teletype. What if you can't afford the extra peripherals and memory? Are you doomed to spend most of your microcomputing hours keying in the same program over and over again?

Maybe not. Perhaps we can avoid most

of the tedium by concentrating on those features of a text editor and assembler which we really need. Although we'll be limited by the KIM-1 keyboard to hexadecimal instruction entry, perhaps we can provide an automatic way to insert and delete instructions and to fix up all those subroutine call addresses and branch displacements. And perhaps by limiting ourselves to these features, we'll be able to cram the "editor and assembler" into some fraction of the KIM's 1 K of memory.

This is the purpose of SWEETS. SWEETS is an example of a program invented to fit an acronym: It stands for Simple Way to Enter, Edit and Test Software. If you own a KIM-1 and have grown tired of absolute machine language programming, now you can step up to "symbolic hex"! While it's not as convenient as a real text editor and assembler, SWEETS can save you a lot of time and index finger soreness.

SWEETS Functions

Under the control of the KIM-1 monitor, the 6 digit LED display normally shows you the address and data of a single byte of memory. You can enter data using the hexadecimal keys, but this causes the data previously in the displayed byte of memory to be destroyed.

Under the control of SWEETS, however, an entire instruction of one, two or three bytes in length is displayed on the LEDs at any given time. An instruction can be inserted just before the displayed instruction by pressing the AD key followed by from 2 to 6 hexadecimal keys. When this is done, the instruction just entered appears on the display; the old instruction and everything following it in the program area have been moved down to make room. Similarly, pressing the DA key causes the currently displayed instruction to be deleted, and everything following this instruction in the program area is moved up to eliminate the slack space.

Successive instructions can be examined by pressing the + key, which advances to and displays the next complete instruction. And to go back to a previous point, or to find an arbitrary point in the instruction sequence, you can press the GO key followed by a two byte (four hexadecimal digit) search pattern. SWEETS will search for the first instruction(s) whose initial two bytes match the search pattern, and then will display this as the current instruction.

This much of SWEETS can be used by itself; but so far we're still burdened by the need to calculate and adjust subroutine call addresses and branch displacements. To lift this burden, we can use hexadecimal "labels." A label is a 3 byte "pseudo-instruction" with an opcode of hexadecimal FF. The second byte is the "label number," any hexadecimal value, and the third byte is ignored. A label is inserted in the hexadecimal instruction sequence at each point where an alphabetic label appears in a normal assembly listing. When we key in a subroutine call, jump, or relative branch instruction, we enter the destination label number as the second byte of the instruction, in place of a branch displacement or absolute address. As we insert and delete instructions, the "label" pseudo-instructions move up and down in memory along with the rest of the code.

When we're ready for a test run of the edited program, we can use the KIM-1 monitor to execute the SWEETS "assembler." This program removes the label

Key	Keys entered				Display					
(GO)	A	0	0	0	A	0	0	0		
(DA)					8	C	0	0	1	7

Table 1b: The procedure used in SWEETS to locate and delete an instruction, in this case the superfluous instruction LDY #0 (A000 in hexadecimal code). The rest of the program is moved up in memory and the next instruction is then displayed, as shown.

```
0200   A0   0A          ENERG   LDY   =10
0202   20   09    02            JSR   WAIT
0205   8C   00    17            STY   PORT
0208   60                       RTS
0209   A2   C8          WAIT    LDX   =200
020B   CA               LOOP    DEX
020C   D0   FD                  BNE   LOOP
020E   88                       DEY
020F   D0   F8                  BNE   WAIT
0211   60                       RTS
```

Listing 1b: The absolute hexadecimal form of the program segment shown in listing 1a after removal of the LDY #0 instruction (see table 1b) and execution of the SWEETS assembler (shown for purposes of comparison in the format of an ordinary assembler output listing).

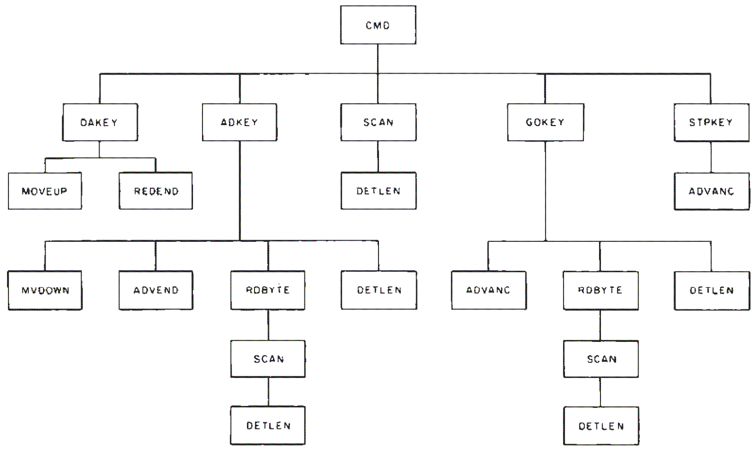


Figure 1: The subroutine calling tree structure of SWEETS. CMD, the control routine, maintains the LED display and scans the keyboard for a command key (by means of SCAN) and transfers to one of the four command processing subroutines, ADKEY, DAKEY, GOKEY or STPKEY. These routines perform the editing functions with the aid of three other subroutines: DETLEN (which determines instruction lengths), MVDOWN, and MOVEUP (which move portions of edited program down and up in memory, respectively).

pseudo-instructions from the instruction sequence, and replaces label references in branch, jump and subroutine call instructions with the proper branch displacements or absolute addresses. Then the edited program is ready for a test execution. (Since the test is likely to fail, leading to further changes in the edited program, we should always dump the program on the audio cassette in "symbolic hexadecimal" form before executing the SWEETS assembler. Then we can reload it later, replacing the program in memory which has been converted to absolute machine language.)

As an example, suppose that you wished to enter the program segment shown in listing 1a, which is taken from an earlier BYTE article of mine (see "Selectric Keyboard Printer Interface," June 1977 BYTE, page 46). Table 1a shows the keys you would press and the resulting instructions displayed on the LEDs by SWEETS. You might then notice that the instruction LDY #0 is superfluous after the call to subroutine WAIT, so you would search for and delete this instruction as shown in table 1b. Finally you would execute the SWEETS assembler, leaving the contents of the program area as shown in listing 1b.

Of course, we will pay some penalty for use of these features of SWEETS, since we will have less memory available for the program to be debugged while SWEETS itself is loaded and running. But larger programs usually can be divided into segments, and loaded, "assembled," and debugged that way. Also, since the SWEETS hexadecimal editor and assembler run separately, we can conserve memory space by loading the assembler from tape whenever we want to use it, overlaying the editor in memory and reloading it from tape in a similar way when we need it again.

Although SWEETS is a useful tool in its present form, you will undoubtedly want to customize it for your own purposes. But to customize SWEETS you've got to understand exactly how it works, so let's take a look at the overall design of SWEETS before puzzling over its realization in 6502 assembly language.

The SWEETS Editor

The subroutine calling tree in figure 1 gives you a quick, "top-down" overall look at the SWEETS editor. CMD, the control routine, maintains the LED display and scans the keyboard for a command key (using SCAN) and then transfers to one of the command processing routines: ADKEY, DAKEY, GOKEY and STPKEY. These routines perform the editing functions with the aid of three critical subroutines: DETLEN, which determines the length of an instruction in bytes based on its opcode; MVDOWN, which moves a portion of the edited program down in memory to make room for an inserted instruction;

and MOVEUP, which moves a portion of the program up in memory to eliminate the empty space created when an instruction is deleted.

The edited program area is managed with the aid of three 16 bit pointers: BEGAD, which points to the beginning of the program area; ENDAD, which points just beyond the end of the program area; and CURAD, which points to the currently displayed instruction. This layout is shown in figure 2. Whenever a new instruction becomes the "current" one, subroutine DETLEN is called to determine its length in bytes, and this value is saved in the variable BYTES.

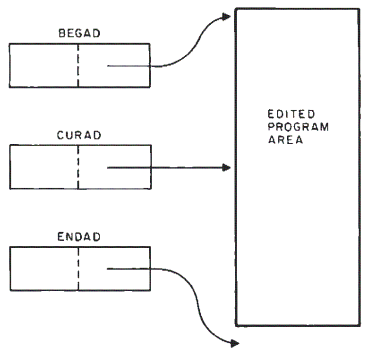
The most basic functions we need in SWEETS are some utility routines to manipulate these 16 bit pointers on an 8 bit machine such as the 6502. The routines we need are shown in listing 2. The most important one is ADVANC, which advances the current instruction pointer CURAD to the next instruction, and tests to see if the end of the program area has been reached. As we shall see later, STPKEY, the command processing routine for the + key, is basically just a call to ADVANC.

Another basic function is the subroutine DETLEN, which we've already mentioned. It is shown in listing 3. The logic of this routine clearly depends on the system of encoding opcodes on the 6502: in most cases (DETLEN tests for the exceptions), the low order hexadecimal digit of the opcode tells us the instruction length. For example, all opcodes of the form x5 represent two byte instructions, while all opcodes of the form xC represent three byte instructions.

The heart of the SWEETS editor lies in the subroutines MOVEUP and MVDOWN, which are shown in listings 4a and 4b. The main concern in these routines is that we must be careful not to move a byte up or down to a location which contains another byte that will be moved later. For MOVEUP, we must move bytes starting at CURAD and proceeding down to ENDAD, while for MVDOWN, we must move bytes in the opposite direction, as shown in figure 3.

So far we haven't faced the issue of how to control our one and only peripheral, the KIM-1 keyboard and LED display.

Figure 2: Three 16 bit pointers are used to manage the edited program area. BEGAD points to the beginning of the program area; ENDAD points to the location immediately beyond the end of the program area, and CURAD points to the currently displayed instruction.




```
                  ; SET CURAD = BEGAD
1780  A5  E0      BEGIN    LDA  BEGAD      LOW-ORDER BYTE
1782  85  E4               STA  CURAD
1784  A5  E1               LDA  BEGAD+1    HIGH-ORDER BYTE
1786  85  E5               STA  CURAD+1
1788  60                   RTS             RETURN TO CALLER
                  ; CURAD = CURAD + BYTES, COMPARE TO ENDAD
1789  18          ADVANC   CLC             CLEAR CARRY
178A  A5  E4               LDA  CURAD      LOW-ORDER BYTE
178C  65  E8               ADC  BYTES
178E  85  E4               STA  CURAD
1790  A5  E5               LDA  CURAD+1    HIGH-ORDER BYTE
1792  69  00               ADC  #0
1794  85  E5               STA  CURAD+1
1796  C5  E3               CMP  ENDAD+1    COMPARE HI-ORDER
1798  30  04               BMI  ADRET
179A  A5  E4               LDA  CURAD
179C  C5  E2               CMP  ENDAD      COMPARE LO-ORDER
179E  60          ADRET    RTS             RETURN TO CALLER
                  ; ENDAD = ENDAD + BYTES
179F  18          ADVEND   CLC             CLEAR CARRY
17A0  A5  E2               LDA  ENDAD      LOW-ORDER BYTE
17A2  65  E8               ADC  BYTES
17A4  85  E2               STA  ENDAD
17A6  90  02               BCC  ADRET1     CHECK CARRY
17A8  E6  E3               INC  ENDAD+1    INCREMENT HI-ORDER
17AA  60          ADRET1   RTS             RETURN TO CALLER
                  ; ENDAD = ENDAD - BYTES
17AB  38          REDEND   SEC             SET CARRY
17AC  A5  E2               LDA  ENDAD      LOW-ORDER BYTE
17AE  E5  E8               SBC  BYTES
17B0  85  E2               STA  ENDAD
17B2  B0  02               BCS  REDRET     CHECK CARRY
17B4  C6  E3               DEC  ENDAD+1    DECREMENT HI-ORDER
17B6  60          REDRET   RTS             RETURN TO CALLER
```

Listing 2: Four utility subroutines used by SWEETS to manipulate three 16 bit pointers which point to the beginning of the program area, the location just beyond the end of the program area, and the currently displayed instruction.

Listing 3: DETLEN, a subroutine which determines instruction length based on op code.

```
0080  A0  00        DETLEN  LDY   #0
0082  B1  E4                LDA   (CURAD),Y   PICK UP OPCODE
0084  A0  01        DETLN1  LDY   #1          ASSUME LENGTH IS 1
0086  C9  00                CMP   #0          TEST FOR 'BRK'
0088  F0  19                BEQ   DETERM
008A  C9  40                CMP   #$40        TEST FOR 'RTI'
008C  F0  15                BEQ   DETERM
008E  C9  60                CMP   #$60        TEST FOR 'RTS'
0090  F0  11                BEQ   DETERM
0092  A0  03                LDY   #3          ASSUME LENGTH IS 3
0094  C9  20                CMP   #$20        TEST FOR 'JSR'
0096  F0  0B                BEQ   DETERM
0098  29  1F                AND   #$1F        STRIP TO 5 BITS
009A  C9  19                CMP   #$19        TEST FOR ABS, Y
009C  F0  05                BEQ   DETERM
009E  29  0F                AND   #$0F        STRIP TO 4 BITS
00A0  AA                    TAX               TO TABLE INDEX
00A1  B4  A6                LDY   LENTB,X     LENGTH FROM TABLE
00A3  84  E8        DETERM  STY   BYTES       SAVE IN 'BYTES'
00A5  60                    RTS               RETURN TO CALLER
00A6  02  02  02    LENTB   .BYTE 2,2,2,1,2,2,2,1
00A9  01  02  02
00AC  02  01
00AE  01  02  01            .BYTE 1,2,1,1,3,3,3,3
00B1  01  03  03
00B4  03  03
```

```
17B7  A5  E4   MOVEUP  LDA  CURAD       START MOVE FROM
17B9  85  E6           STA  MOVAD       BEGIN OF PROGRAM
17BB  A5  E5           LDA  CURAD+1     SEGMENT (CURAD)
17BD  85  E7           STA  MOVAD+1
17BF  A4  E8   UPLOOP  LDY  BYTES       AMOUNT TO MOVE
17C1  B1  E6           LDA  (MOVAD),Y   FETCH BYTE
17C3  A0  00           LDY  #0
17C5  91  E6           STA  (MOVAD),Y   STORE BYTE
17C7  A5  E6           LDA  MOVAD       CHECK FOR
17C9  A6  E7           LDX  MOVAD+1     END OF MOVE
17CB  C5  E2           CMP  ENDAD       LOW-ORDER BYTE
17CD  D0  04           BNE  INCMOV
17CF  E4  E3           CPX  ENDAD+1     HIGH-ORDER BYTE
17D1  F0  09           BEQ  MVURET
17D3  E6  E6   INCMOV  INC  MOVAD       INCREMENT LO-ORDER
17D5  D0  E8           BNE  UPLOOP
17D7  E6  E7           INC  MOVAD+1     INCREMENT HI-ORDER
17D9  B8               CLV
17DA  50  E3           BVC  UPLOOP      BACK TO MOVE MORE
17DC  60       MVURET  RTS              RETURN TO CALLER
```

```
00B6  A5  E2   MVDOWN  LDA  ENDAD       START MOVE FROM
00B8  85  E6           STA  MOVAD       END OF PROGRAM
00BA  A5  E3           LDA  ENDAD+1     SEGMENT (ENDAD)
00BC  85  E7           STA  MOVAD+1
00BE  A0  00   MVLOOP  LDY  #0
00C0  B1  E6           LDA  (MOVAD),Y   FETCH BYTE
00C2  A4  E8           LDY  BYTES       AMOUNT TO MOVE
00C4  91  E6           STA  (MOVAD),Y   STORE BYTE
00C6  A5  E6           LDA  MOVAD       CHECK FOR
00C8  A6  E7           LDX  MOVAD+1     END OF MOVE
00CA  C5  E4           CMP  CURAD       LOW-ORDER BYTE
00CC  D0  04           BNE  DECMOV
00CE  E4  E5           CPX  CURAD+1     HIGH-ORDER BYTE
00D0  F0  0D           BEQ  MVDRET
00D2  38       DECMOV  SEC              SET CARRY
00D3  E9  01           SBC  #1          DECREMENT LO-ORDER
00D5  85  E6           STA  MOVAD
00D7  8A               TXA
00D8  E9  00           SBC  #0          DECREMENT HI-ORDER
00DA  85  E7           STA  MOVAD+1
00DC  B8               CLV
00DD  50  DF           BVC  MVLOOP      BACK TO MOVE MORE
00DF  60       MVDRET  RTS              RETURN TO CALLER
```

Listings 4a and 4b: Subroutines MOVEUP and MVDOWN, which form the heart of the SWEETS editor. MOVEUP moves a given program segment starting at address CURAD and ending at address ENDAD upward in memory (toward decreasing addresses) by the amount stored in BYTES. MVDOWN performs the same operation downward by the amount stored in BYTES.

Fortunately, several routines are provided for this purpose in the KIM-1 monitor; the source listings for these routines are available on request from MOS Technology. In the SWEETS assembly code listings, we have underlined references to KIM-1 monitor subroutines and variables for easy identification. We will use the KIM-1 subroutine SCAND1, which lights up the LEDs momentarily and checks to see if a key is pressed, and the subroutine GETKEY, which returns a numeric value in the accumulator telling us which particular key has been pressed.

The six LED digits display the contents of three successive bytes in memory, denoted POINTH, POINTL and INH in the KIM-1 monitor. Unfortunately, the order of these bytes is the opposite of the normal order of the bytes in an instruction in memory, so we must reverse the order as the first step of our subroutine SCAN (listing 5). The main additional complication in this routine is the need to "debounce" the keyboard's bare contact switches in software. Since SWEETS performs its operations so quickly relative to a mechanical event, the key from the last operation invariably is *still pressed* when we come back to the keyboard looking for the next command. Also shown in listing 5 is subroutine RDBYTE, which calls SCAN to read two successive hexadecimal digits from the keyboard.

With all of this machinery in place, the top level logic is straightforward. The control routine, CMD routine, and the command processing routines are shown in listings 6a, 6b and 6c. The most complicated of the processing routines is ADKEY. It determines how many bytes to read for the inserted instruction, and displays each byte as it is entered; then it copies (in reverse

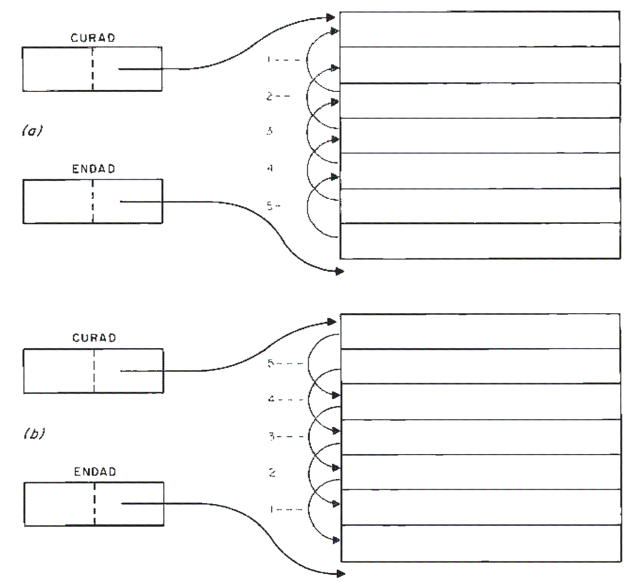


Figure 3: Correct procedures for moving programs in SWEETS. Figure 3a shows that the uppermost location must be moved first when transferring a section of program upward. Otherwise, some locations could be inadvertently destroyed. Figure 3b shows the analogous situation for a downward movement of code.

```
0100  20  80  00    SCAN     JSR   DETLEN      DETERMINE LENGTH
                    ; COPY INSTRUCTION TO DISPLAY AREA,
                    ; REVERSING ORDER OF INSTRUC.  BYTES
0103  A0  00                 LDY   #0
0105  A6  E8                 LDX   BYTES
0107  B1  E4        SCOPY    LDA   (CURAD),Y   INSTRUCTION BYTE
0109  95  F8                 STA   INH-1,X     TO DISPLAY AREA
010B  C8                     INY
010C  CA                     DEX
010D  D0  F8                 BNE   SCOPY
                    ; LOGIC TO 'DEBOUNCE' KEYBOARD CONTACT
010F  20  22  01    SCAN1    JSR   SCAN3       WAIT UNTIL LAST
0112  D0  FB                 BNE   SCAN1       KEY IS RELEASED
0114  20  22  01    SCAN2    JSR   SCAN3
0117  F0  FB                 BEQ   SCAN2       WAIT FOR NEW KEY
0119  20  22  01             JSR   SCAN3
011C  F0  F6                 BEQ   SCAN2       BUT REJECT JITTER
011E  20  6A  1F             JSR   GETKEY      GET CODE FOR KEY
0121  60                     RTS               RETURN TO CALLER
                    ; SET UP PARMS AND CALL KIM-1 DISPLAY SCAN
0122  A4  E8        SCAN3    LDY   BYTES
0124  A2  09                 LDX   #9
0126  A9  7F                 LDA   #$7F
0128  8D  41  17             STA   PADD        SET UP DATA DIRECT
012B  20  28  1F             JSR   SCAND1      CALL KIM-1 ROUTINE
012E  60                     RTS               RETURN TO CALLER

                    ; RDBYTE READS TWO HEX DIGITS, RETURNS BYTE
                    ; VALUE IN ACCUMULATOR.  IF A NON-HEX DIGIT
                    ; KEY IS PRESSED, IT RETURNS THE KEY CODE
                    ; IN THE ACCUMULATOR AND N FLAG = 0

012F  20  0F  01    RDBYTE   JSR   SCAN1       GET FIRST KEY
0132  C9  10                 CMP   #$10        IS IT A HEX DIGIT?
0134  10  11                 BPL   RDRET       NO, RETURN
0136  0A                     ASL   A           SHIFT OVER 4 BITS
0137  0A                     ASL   A
0138  0A                     ASL   A
0139  0A                     ASL   A
013A  85  E9                 STA   TEMP        SAVE FIRST DIGIT
013C  20  0F  01             JSR   SCAN1       GET SECOND KEY
013F  C9  10                 CMP   #$10        IS IT A HEX DIGIT?
0141  10  04                 BPL   RDRET       NO, RETURN
0143  05  E9                 ORA   TEMP
0145  A2  FF                 LDX   #$FF        SET N FLAG = 1
0147  60            RDRET    RTS               RETURN TO CALLER
```

Listing 5: Subroutines SCAN and RDBYTE. SCAN displays the instruction at location CURAD, scans the keyboard for a depressed key, and places the code for that key in the accumulator. RDBYTE calls SCAN to read two successive hexadecimal digits from the keyboard.

order) the new instruction bytes from the display to the program area. If you've understood everything so far, you should have little trouble following the code for these top level functions. More important, once you're familiar with the basic SWEETS design, you can easily add customized top level routines of your own.

The SWEETS Assembler

None of the editor routines just discussed were concerned with the processing of the hexadecimal "labels" described earlier as one of the features of SWEETS. This is because, as far as the editor is concerned, a label is just another 3 byte instruction. Labels take on a special meaning only when the SWEETS assembler is invoked.

The assembler operates in two passes over the program area. On the first pass, the assembler searches for "instructions" with an opcode of hexadecimal FF (the labels). When one is found, the second byte of the instruction (the label number) is moved to the end of the program area, and the current instruction address is also deposited there (figure 4a). The label instruction is then deleted using MOVEUP to take up the slack space. This process continues until all of the labels have been removed and stored in the "symbol table" at the end of the program area (figure 4b). Since the labels are (by design) three bytes long, we gain the space for the symbol table when

(a)

```
0148  20  2F  01    GOKEY   JSR  RDBYTE     GET FIRST BYTE
014B  10  28                BPL  GCMD       OF SEARCH PATTERN
014D  85  FB                STA  POINTH     SAVE IN DISPLAY
014F  20  2F  01            JSR  RDBYTE     GET SECOND BYTE
0152  10  21                BPL  GCMD       OF SEARCH PATTERN
0154  85  FA                STA  POINTL     SAVE IN DISPLAY
0156  20  80  17            JSR  BEGIN      CURAD := BEGAD
                    ; LOOP SEARCHING FOR 2-BYTE MATCH
0159  A0  00        GOLOOP  LDY  =0
015B  B1  E4                LDA  (CURAD),Y  COMPARE 1ST BYTE
015D  C5  FB                CMP  POINTH     AGAINST PATTERN
015F  D0  07                BNE  GONEXT
0161  C8                    INY
0162  B1  E4                LDA  (CURAD),Y  COMPARE 2ND BYTE
0164  C5  FA                CMP  POINTL     AGAINST PATTERN
0166  F0  0A                BEQ  CMD        MATCH, NEXT CMD
0168  20  80  00    GONEXT  JSR  DETLEN     DETERMINE LENGTH
016B  20  89  17            JSR  ADVANC     ADVANCE TO NEXT
016E  F0  15                BEQ  ERROR      MATCH NOT FOUND?
0170  D0  E7                BNE  GOLOOP     CONTINUE SEARCH
```

(b)

```
0172  20  00  01    CMD     JSR  SCAN       WAIT FOR A KEY
0175  C9  10        GCMD    CMP  =$10       TEST FOR VARIOUS
0177  F0  2B                BEQ  ADKEY      COMMAND KEY CODES
0179  C9  11                CMP  =$11
017B  F0  1E                BEQ  DAKEY
017D  C9  12                CMP  =$12
017F  F0  13                BEQ  STPKEY
0181  C9  13                CMP  =$13
0183  F0  C3                BEQ  GOKEY
0185  A9  EE        ERROR   LDA  =$EE       OPERATOR ERROR.
0187  85  F9                STA  INH        SET UP HEX 'EE'
0189  85  FA                STA  POINTL     IN DISPLAY AREA
018B  85  FB                STA  POINTH
018D  20  1F  1F    ERR1    JSR  SCANDS     CALL KIM-1 ROUTINE
0190  D0  FB                BNE  ERR1       UNTIL KEY RELEASED
0192  F0  DE                BEQ  CMD

                    ; STPKEY ADVANCES TO THE NEXT INSTRUCTION

0194  20  89  17    STPKEY  JSR  ADVANC     ADVANCE TO NEXT
0197  10  EC                BPL  ERROR      CHECK FOR ADVANCING
0199  30  D7                BMI  CMD        PAST END OF PROGRAM

                    ; DAKEY DELETES THE CURRENT INSTRUCTION

019B  20  B7  17    DAKEY   JSR  MOVEUP     MOVE UP REST OF PROG
019E  20  AB  17            JSR  REDEND     ADJUST ENDAD UPWARDS
01A1  B8                    CLV
01A2  50  CE                BVC  CMD
```

(c)

```
                    ; READ OPCODE, DETERMINE INSTRUCTION LENGTH
01A4  20  2F  01    ADKEY   JSR  RDBYTE     ACCEPT OPCODE UNLESS
01A7  10  CC                BPL  GCMD       NON-HEX KEY PRESSED
01A9  85  FB                STA  POINTH     SAVE IN DISPLAY
01AB  20  84  00            JSR  DETLN1     DETERMINE LENGTH
                    ; READ REST OF INSTRUCTION INTO DISPLAY
01AE  84  EA                STY  COUNT      SAVE LENGTH
01B0  C6  EA                DEC  COUNT
01B2  F0  12                BEQ  ADSET      1-BYTE INSTRUCTION
01B4  20  2F  01            JSR  RDBYTE     READ SECOND BYTE
01B7  10  BC                BPL  GCMD       NON-HEX KEY PRESSED
01B9  85  FA                STA  POINTL
01BB  C6  EA                DEC  COUNT
01BD  F0  07                BEQ  ADSET      2-BYTE INSTRUCTION
01BF  20  2F  01            JSR  RDBYTE     READ THIRD BYTE
01C2  10  B1                BPL  GCMD       NON-HEX KEY PRESSED
01C4  85  F9                STA  INH
                    ; MOVE CODE DOWN TO MAKE ROOM
01C6  20  86  00    ADSET   JSR  MVDOWN     MOVE CODE DOWNWARD
01C9  20  9F  17            JSR  ADVEND     ADJUST ENDAD DOWN
                    ; INSERT INSTRUCTION INTO NEW SPACE
01CC  A0  00                LDY  =0
01CE  A2  02                LDX  =2
01D0  B5  F9        INSERT  LDA  INH,X      FETCH FROM DISPLAY
01D2  91  E4                STA  (CURAD),Y  STORE INTO PROGRAM
01D4  CA                    DEX
01D5  C8                    INY
01D6  C4  E8                CPY  BYTES      UNTIL ENTIRE INSTRUCTION
01D8  D0  F6                BNE  INSERT     IS INSERTED
01DA  F0  96                BEQ  CMD
```

Listing 6: Processing routines used in the SWEETS editor. Listing 6a shows GOKEY, which searches the program for a given 2 byte pattern and makes this the current instruction. It can also search for labels. The CMD (for "command") routine, listing 6b, waits for a command key to be pressed and transfers to the processing routine for that key. If an invalid key is pressed, "EEEEEE" is displayed. ADKEY (listing 6c) accepts a new instruction, inserts it, and shifts the code following it downward to make room.

we delete the labels from the instruction sequence.

On its second pass through the program area, the assembler searches for subroutine call, jump and relative branch instructions. When one of these instructions is found, its second byte, normally a label number, is used to search for a matching label in the symbol table. Assuming that the label is found in the table, the corresponding actual address is inserted into the second and third instruction bytes for jump or subroutine call instructions, or a branch displacement is calculated and inserted for relative branch instructions (figure 4c). Since at times we may wish to enter instructions with an actual address or displacement rather than a label number, no substitution is made if the label is not found in the symbol table.

The assembly source code for the SWEETS assembler is presented in listings 7a, 7b and 7c. The subroutine FINDLB is used by pass 2 of the assembler to look up labels in the symbol table. Note, too, that the assembler uses some of the editor's subroutines: DETLEN, ADVANC, REDEND, and MOVEUP. The addresses shown in the assembly code listing are designed to allow the assembler to overlay the main part of the editor without destroying those editor subroutines which the assembler must use.

Some Operating Hints

Except for subroutine call addresses, each SWEETS routine is relocatable: it will execute properly no matter where it is loaded in memory. The assembled code shown here is designed to provide the largest possible contiguous area (512 bytes at hexadecimal addresses 200 to 3FF) for editing and assembling programs. This has the disadvantage of breaking up SWEETS into four pieces: one in page zero, two in page one, and one starting at address 1780 (which makes it a bit cumbersome to load piece by piece from audio cassette). The SWEETS routines could be consolidated, however, to provide two or more noncontiguous areas for program editing.

In general, when starting up SWEETS, or after reloading a "symbolic hexadecimal" program from tape, you must store the proper values in BEGAD, CURAD and ENDAD. Then, of course, you merely key in the CMD routine starting address and press GO. The assembler, which can be started up in the same way, automatically returns control to the KIM-1 monitor; the editor can be interrupted at any point by pressing RS (reset). Avoid using the ST

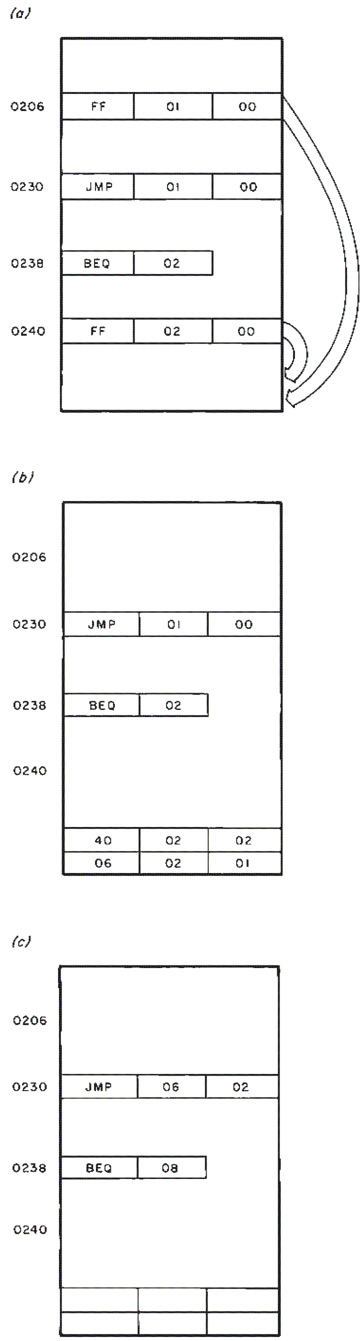


Figure 4: Mechanics of pass 1 of the SWEETS assembler are shown in figure 4a. The assembler first searches for "instructions" having an op code of hexadecimal FF (the labels). When one is found, the second byte of the instruction, which is the label number, is moved to the end of the program area and the current instruction address is also deposited there. The label instruction is then deleted using subroutine MOVEUP. Figure 4b is a continuation of the process shown in figure 4a, showing that all of the labels have been arranged in a symbol table at the end of the program area. A typical result of pass 2 of the SWEETS assembler is shown in figure 4c. Here a jump instruction has been modified so that the actual address of the destination appears in bytes 2 and 3 of the instruction, and the actual branch displacement has been calculated and inserted for a relative branch instruction. In general, this pass takes care of all jump, subroutine call, and relative branch instructions.

BEGAD	00E0, 00E1
ENDAD	00E2, 00E3
CURAD	00E4, 00E5
CMD	0172
ASSEM	011C

Table 2: Locations of the variables BEGAD, ENDAD, CURAD, CMD and ASSEM. BEGAD, CURAD and ENDAD must be set up by the user to point to the area of memory which will hold the edited program. CMD is the entry point to the SWEETS editor, and ASSEM is the entry point to the SWEETS assembler.

(stop) key repeatedly, since this may cause the stack to grow in length to the point where it could destroy one of the SWEETS routines. The special address information you need is summarized in table 2.

Once you have SWEETS up and running, you can use it to develop improvements to SWEETS itself. In order to do this, you will have to edit code in the program area which is designed to run in another area of memory. One way to facilitate this is to add a 16 bit offset to jump and subroutine call addresses as they are resolved in pass 2 of the assembler. Another addition to SWEETS would be a small routine to save ENDAD at the end of the program area, set up the starting and ending addresses for the KIM-1 audio cassette dump routine, and then transfer control directly to this read only memory routine to carry out the tape dump operation.

One of the peculiarities of SWEETS is that it tends to make itself obsolete. This is because of our insatiable desire to do more with our personal computers. As soon as you find that writing a 512 byte program isn't so tedious anymore, you'll immediately want to write a 1024 byte program (at least), and then you'll be stretching the capabilities of SWEETS and the KIM-1. In a sense, SWEETS, as its name suggests, is an enticement: It helps develop the market for assemblers. But why not give it a try? It's a lot sweeter than absolute hex.

(a)

```
0100  B1  E4      FINDLB   LDA  (CURAD),Y   PICK UP LABEL
0102  A0  FF               LDY  # $FF       SYMBOL TABLE INDEX
0104  C4  EB      FDLOOP   CPY  LABELS
0106  F0  0D               BEQ  FDRET       NO LABELS IN TABLE
0108  D1  EC               CMP  (TABLE),Y   DOES LABEL MATCH?
010A  D0  0A               BNE  FDNEXT
010C  88                   DEY              WE HAVE A MATCH
010D  B1  EC               LDA  (TABLE),Y   GET HI-ORDER ADDR
010F  AA                   TAX              INTO X REGISTER
0110  88                   DEY
0111  B1  EC               LDA  (TABLE),Y   GET LO-ORDER ADDR
0113  A0  01               LDY  # 1         INTO A REG., Y=1
0115  60          FDRET    RTS              RETURN TO CALLER
0116  88          FDNEXT   DEY
0117  88                   DEY              ADVANCE TO NEXT
0118  88                   DEY              SYMBOL TABLE ENTRY
0119  D0  E9               BNE  FDLOOP
011B  60                   RTS              UNLESS END OF TBL
```

(b)

```
011C  20  80  17  ASSEM   JSR  BEGIN      CURAD := BEGAD
011F  18                  CLC
0120  A5  E2              LDA  ENDAD      ENDAD + 6 IS JUST
0122  69  06              ADC  # 6        BEYOND UPPERMOST
0124  85  EC              STA  TABLE      LABEL IN TABLE
0126  A9  FF              LDA  # $FF
0128  85  EB              STA  LABELS     BEGINNING TBL INDEX
012A  65  E3              ADC  ENDAD+1    ADJUST TABLE DOWN BY
012C  85  ED              STA  TABLE+1    256 FOR INDEX BASE
012E  20  80  00  ASLOOP  JSR  DETLEN     DETERMINE LENGTH
0131  A0  00              LDY  # 0
0133  B1  E4              LDA  (CURAD),Y  PICK UP OPCODE
0135  C9  FF              CMP  # $FF      IS IT A LABEL?
0137  D0  1D              BNE  ASNEXT
0139  C8                  INY
013A  B1  E4              LDA  (CURAD),Y  YES, GET LABEL NO
013C  A4  EB              LDY  LABELS     GET TABLE INDEX
013E  91  EC              STA  (TABLE),Y  DEPOSIT LABEL IN TBL
0140  88                  DEY
0141  A5  E5              LDA  CURAD+1    HI-ORDER ADDRESS
0143  91  EC              STA  (TABLE),Y  DEPOSIT IN TABLE
0145  88                  DEY
0146  A5  E4              LDA  CURAD      LO-ORDER ADDRESS
0148  91  EC              STA  (TABLE),Y  DEPOSIT IN TABLE
014A  88                  DEY
014B  84  EB              STY  LABELS     SAVE NEW TBL INDEX
014D  20  B7  17          JSR  MOVEUP     MOVE UP PROGRAM
0150  20  AB  17          JSR  REDEND     ADJUST ENDAD UPWARD
0153  B8                  CLV
0154  50  D8              BVC  ASLOOP     BACK FOR NEW LABEL
0156  20  89  17  ASNEXT  JSR  ADVANC     TO NEXT INSTRUCTION
0159  30  D3              BMI  ASLOOP     UNTIL ENDAD REACHED
```

(c)

```
015B  20  80  17          JSR  BEGIN      CURAD := BEGAD
015E  20  80  00  RSLOOP  JSR  DETLEN     DETERMINE LENGTH
0161  A0  00              LDY  # 0
0163  B1  E4              LDA  (CURAD),Y  PICK UP OPCODE
0165  C9  20              CMP  # $20      JSR INSTRUCTION?
0167  F0  04              BEQ  JMPJSR
0169  C9  4C              CMP  # $4C      JMP INSTRUCTION?
016B  D0  0E              BNE  CHKBR
016D  C8          JMPJSR  INY             ADVANCE TO LABEL
016E  20  00  01          JSR  FINDLB     LOOKUP IN TABLE
0171  F0  1C              BEQ  RSNEXT     LABEL NOT FOUND
0173  91  E4              STA  (CURAD),Y  LO-ORDER ADDRESS
0175  8A                  TXA
0176  C8                  INY
0177  91  E4              STA  (CURAD),Y  HI-ORDER ADDRESS
0179  D0  14              BNE  RSNEXT     TO NEXT INSTRUC
017B  29  1F      CHKBR   AND  # $1F
017D  C9  10              CMP  # $10      BRANCH INSTRUC?
017F  D0  0E              BNE  RSNEXT
0181  C8                  INY             ADVANCE TO LABEL
0182  20  00  01          JSR  FINDLB     LOOKUP IN TABLE
0185  F0  08              BEQ  RSNEXT     LABEL NOT FOUND
0187  38                  SEC
0188  E5  E4              SBC  CURAD      DEST. – SOURCE
018A  38                  SEC
018B  E9  02              SBC  # 2        DEST. – SOURCE – 2
018D  91  E4              STA  (CURAD),Y  = DISPLACEMENT
018F  20  89  17  RSNEXT  JSR  ADVANC     TO NEXT INSTRUC
0192  30  CA              BMI  RSLOOP     BACK TO EXAMINE IT
0194  4C  4F  1C          JMP  START      TO KIM-1 MONITOR ■
```

Listing 7: The assembly source code for SWEETS. Subroutine FINDLB (listing 7a) is used during pass 2 of the assembler to look up labels in the symbol table. FINDLB looks up the label at CURAD, Y and returns with Y=1, X=the high order part of the address, A = the lower part of the address, and Z=0. Z is set equal to 1 if the label is not found. Listing 7b shows pass 1 of the assembler during which labels are collected and stored with their addresses at the end of the program. Listing 7c is pass 2. During this pass, the operands of the branch, jump and JSR instructions are converted from label references to displacements or actual addresses. Note that jump indirect operands are not converted.