EDITHA

A REVISION OF THE FYLSTRA KIM-1 EDITOR PROGRAM

BY H.T. GORDON
College of Natural Resources
University of California
Berkeley, CA 94720

Received: 78 April 3

Dear Jim, Tom, et al.,

I am herewith submitting an extensive recoding and enhancement of the editor section of Dan Fylstra's SWEETS program for the KIM-1, published in (and copyrighted by) BYTE. It poses several dilemmas of general interest. First, the problem of program names. SWEETS is Fylstra's acronym for 2 main programs, that share some subroutines. The entry label for the editor (the only one I have worked on) is CMD. I have given the more descriptive name EDITHA to my revision, on the grounds that its coding is mostly quite different and that it includes some major innovations (address information and modification capability, greater error-protection, and a richer command structure with modes not found in CMD). It is clearly an enhancement of the kind that Fylstra himself anticipated would grow out of CMD. I hesitated before adopting the name EDITHA, since renaming can easily become an idiotic game, involving personal vanity, causing confusion, and even trying to disguise software theft. I have restained many of Fylstra's names in EDITHA whenever my recoding did not involve a major functional alteration, both as a courtesy to the originator of the basic concept and as an aid to comprehension by users. One of his subroutines (DETLN1) is an inefficient way of determining the bytecount of a 6502 opcode. In EDITHA I have replaced DTELN1 logic by the decoding logic of my subroutine BYTCNT (published in DDJ *# 22), with minor retailoring to fit the special context. I re-use the name BYTCNT for this trivial modification, because I think it is better to extend the meaning of a subroutine name to include minor variations, instead of endlessly coining new names. There are, alas, no rules defining how great a dissimilarity is needed to justify a new name.*

The second dilemma is the perennial one of software ownership. I assert copyright (qualified by a special non-monopolistic free-diffusion clause) for all components of EDITHA not either previously in the public domain or copyrighted by someone else. Exactly what these are is a moot question. Very little of the original CMD coding is left intact in EDITHA. None of the alterations is anything but functional (i.e., there is no intent to evade existing copyright by mere recoding). None is copied from any work I have had access to (although since there are relatively few ways of doing something efficiently, it is not impossible that some coding is an independent recreation of already existing logic). Here is the qualifying clause: The intent of my copyright is to remove all impediments to free diffusion, including (to whatever extent my copyright gives me leverage to do so) impediments that others may wish to impose on enhanced or different implementations, including systems other than the KIM-1. I therfore authorize any and all uses (including copying in any form, publication in whole or in part, and commercialization) if (and only if) all software associated with the EDITHA coding is declared to be equally available for totally unrestricted use. What I am trying to ensure is not only that my copyright will be no bar to progress but that it shall serve as an aid to it. Perhaps this is Utopian, but there's no harm in trying! It would be a tragicomedy if micro software had to be diffused underground, in the style of Russian samizdats, *as a response to the censorship aspects of copyrighting.*

The third dilemma is: where should revision of a program published in BYTE *be published? First, let me say that I consider* BYTE *a valuable journal. I have had a note (on 6502 opcodes) published in it, and a second one (on a pseudorandom number generator, 8080-coded) is accepted and will (eventually!) get published. There are some advantages (a small remuneration, glossy paper, and a very large readership) and some disadvantages (among them are: loss of all rights to your work, and often long publication delay, and no indication of the date of receipt of articles). On the whole, my relationship with* DDJ *has been a more humanly satisfying one. Friends have told me its title is zany (but to me it's whimsical – I reserve zany [and its synonyms] for a world with tens of thousands of megatons of hydrogen bombs poised for launching). But publication is prompt, always includes date of receipt (a fine ethical touch), and the author retains the rights to his work. These small-is-beautiful qualities compensate for inexpensive paper and no remuneration!*

Sincerely,
H. T. Gordon

The SWEETS machine-language editor program CMD, created by Dan Fylstra (Feb. 1978 *BYTE*) is a major enhancement of the KIM-1 SBC. However, it has some coding inefficiencies, logic errors, and other limitations. The major ones are its inability to allow manipulation of addresses (only program instructions can be seen in the KIM display, and the lack of adequate safeguards against user errors. It is all too easy to ignore the CMD error-warning and cause disastrous upmoves or downmoves of immense blocks of RAM, including the CMD program itself. My revision gradually became a major overhaul, involving drastic alteration of both the main program (whose entry point I have renamed EDITHA) and the subroutines.

Insofar as possible, EDITHA (version 1.0, intended for the basic KIM-1) is located in the same memory areas used by CMD. This conforms with Fylstra's intent of leaving a large block of RAM free for new program construction under editor control. However, the enhancements provided by EDITHA increase its length to 478 bytes (71 more than CMD), and its subroutine READIN had to be located in 0200-0234. This leaves only the 459 locations from 0235-03FF free.

The revision relocates the original CMD pointers in zero-page: MOVAD (to DE-DF), BEGAD (to E0-E1), CURAD (to E2-E3), ENDAD (to E4-E5), and adds a new one, LIMAD (at E6-E7). This allows operation of a new guard subroutine, INLIMT, that protects memory locations outside preset program limits from being moved or overwritten. BEGAD is redefined as one location *below* the program start (lowest) address. LIMAD must be set 2 locations below the highest address that the program can enter. BEGAD and all lower memory locations, and LIMAD+3 and all higher memory locations, cannot be altered or moved by EDITHA. The guard subroutine blocks the action of the DA (delete instruction) key unless BEGAD < CURAD < ENDAD. In addition, it blocks the AD (add instruction) key unless ENDAD < LIMAD. When these conditions are not met, the only effect of pressing either key is to display CURAD in the usual KIM address display, and EE in the data display. CURAD is the address of the instruction currently displayed, that EDITHA is refusing to add to or delete. The same message will be displayed if any hex key is pressed when EDITHA is in the command-entry mode, but in this case the EE does not signify that CURAD is out-of-bounds but that hex keys cannot be interpreted and are being rejected.

Although EDITHA has instructions that allow the user to alter BEGAD or LIMAD, they are normally fixed and define the program limits. As in CMD, CURAD is advanced to the next higher instruction when the + key is pressed. However, while the + key is depressed, EDITHA displays the address of the next instruction (that appears when the key is released). If this address is within limits, 03 appears in the data display, but if it is out-of-bounds, EE is displayed.

During program writing, CURAD does not change, but ENDAD moves upward when AD is used to insert an instruction and downward when DA is used to delete the instruction currently in CURAD. If deletions exceed insertions, ENDAD may move below CURAD, and thereby block the action of the AD and DA keys. This is no problem, since the user can press a hex key to see where CURAD is and then use the pointer-resetting instruction to set ENDAD to a higher location. In fact, it is sometimes useful to set ENDAD to the program start address, so that the user can explore anywhere in memory with no risk of altering programs by accidental pressing of the AD or DA keys.

The operational changes in the + and DA key actions are achieved by quite trivial modification of the original (much less informative) CMD error-message, together with continual supervision by the guard subroutine. Users may notice one more trivial change: one-byte displays by EDITHA are of normal brightness, while those by CMD are extremely bright. CMD refreshes one-byte displays 3 times as often as 3-byte ones; this may or may not shorten the life of the LEDs, but I found it unpleasant. Therefore EDITHA inserts a very brief dark period in every refresh of a one-byte display. Both CMD and EDITHA display two-bytes at about 1.5 times normal brightness.

Implementation of Complex Commands in the AD and GO Operations.

Unlike the + and DA operations, which are simple commands executed by pressing one command key, the AD and GO operations require complex commands. Both call subroutine READIN, since the user is expected to key in more information. Although much more complex than its equivalent in CMD (subroutine RDBYTE), READIN also calls subroutine SCAN1 twice. Unlike RDBYTE (that always error-exits if a command key is pressed), the first call to SCAN1 by READIN will accept either a hex or a command key, the latter causing a return for interpretation by the calling routine. If the first key is hex, the second call to SCAN1 *ignores* all command keys but returns when a hex key is pressed, inserting the completed byte in the leftmost display (all others blank). ADKEY logic interprets this as an opcode; if it is a one-byte opcode, it is inserted in the program and the operation is over. If it is a two- or three-byte opcode, READIN is again called so the user can complete the instruction. At this point, it is possible to press a command key and cause a return. In version 1.0, EDITHA interprets such command-hex-command instructions as errors, simply wiping them out as if they had never been keyed-in at all. In an expanded version, however, more complex interpreting logic could open a cornucopia (or a Pandora's box?) of special commands. Very few of these possibilities are realized in version 1.0. One reason is the memory limitations of the basic KIM-1. Another is that I am not sure how powerful a machine-language editor ought to become, or what operations will prove to be the most useful.

The AD Key Operations

Pressing the AD key has no detectable effect if CURAD is within limits and the instruction currently in the display is one-byte. If it is two- or three-byte, it is left-shifted so that only the previously rightmost byte appears in the leftmost display (all others blank). When the opcode is keyed-in, it replaces the displayed byte. If it is a one-byte opcode the instruction is complete and written into the program (and being displayed by the program-scanning subroutines). Pressing a hex key will cause the usual error-display of CURAD. If the opcode is incorrect, it can be deleted by pressing the DA key.

If the opcode requires 1 or 2 operand bytes, pressing any command key will cause an error wipe-out (a quick way to erase a wrong opcode). If a hex key is pressed, nothing will happen until a second hex key is pressed. If the opcode was two-byte, the second keyed-in byte will appear in the *middle* display (*normal* order, an indication that the whole instruction has been written into the program). It can be deleted by the DA key. However, keying in the first operand of a three-byte opcode will cause the opcode to shift to the middle display and the operand to appear in the leftmost one (*reverse* order, instruction not yet completed). Again, pressing a command key will cause a wipeout. Keying in the third byte (second operand) will cause display in normal order (instruction written-in, deletable by DA).

A peculiar "double-command" operation is also possible, executed if the AD key is pressed *twice* (pressing AD, then any other command key, has the same effect but the AD-AD is safer). The effect is to insert the single-byte in the display (following the first AD) in the program, "completing" the instruction. The AD-AD operation is useful *only* if the instruction in the display before the first AD was a one-byte opcode. It allows *duplication* of this opcode, a convenient way to enter a string of one-byte opcodes (00, 0A, 4A, EA, etc.). The AD-AD was not a planned feature of EDITHA, but an accidental consequence of the planned GO-AD (to be discussed in the next section), retained because its implementation requires very little interpreting logic.

The GO Key Operations

These represent by far the most complex EDITHA enhancements, that still are only a minute fraction of a legion of possibilities. The only "double-command" is the GO-AD (the other 3 possible ones are interpreted as errors). This sequence of 2 command keyings resets CURAD to the program start (one location higher than BEGAD). (Note that, although the command keys are read by the KIM monitor subroutine

GETKEY as hex numbers from $10 to $13, the READIN decoder subtracts 4 from these values and returns command codes from $0C to $0F to the GOKEY routine.)

All other GO operations require keying-in 4 hex bytes for completion, the first 3 appearing in the display one after another, in reverse order. Wipeout can be caused by pressing any command key following entry of any byte into the display. The terminator byte (not displayed) is the command code that will cause execution of a special operation. Since all non-interpreted command codes cause an error-message, it is useful to keep the final key of this code depressed so that it can be seen displayed. If there is no error-message, the command has been executed. Error-messages are always XX 04 EE, where XX is the command code that could not be executed (except for codes 04 and 08, where XX is 00). The following is a list of the 5 valid command modes, with their codes:

Mode I, code 00
This is the ERASE command, that restores the original status before the GO key was pressed. Although the "wipeout" feature of command keys (a later development) makes it unnecessary, it might be useful in later versions.

Mode II, codes 01 to 03
These are IDPR (insert data into program) commands, 01 inserting only the first keyed-in byte (rightmost in the display), 02 the first and second, 03 all three. Before completion, display is in *reverse* order. When the command is completed, the inserted bytes will be displayed in *normal* order (as with the AD operation) but interpreted as data. Thus the sequence of keyings: GO-08-18-28-03, that is displayed as 28 18 08 before the command code is entered, will be displayed as 08 18 28 afterward. This sequence would need 3 separate entries in the AD operation, since all the entries are one-byte opcodes. Also, the *entire* sequence can be deleted by pressing the DA key once (if there has been an error).

In both CMD and EDITHA, to add a new program or data sequence *following* the one in the display, one must first press the + key to advance CURAD. This is necessary to allow the user to see in the display exactly what he has entered. In the EDITHA IDPR Mode II, all bytes that will be entered (and with codes 02 and 01, some that will *not* be) are visible in the display before the command is keyed-in, although in reverse order. Some users may wish to implement an automatic advance, so that more data can be entered by the Mode II GO without having to press the + key. This can be done by replacing the BEQ DACMD at 017B by a BEQ STPKEY (F0 37). The disadvantage is that one will not see displayed what has been entered, and not have a second chance to delete an erroneous entry.

Mode III, codes 05 to 07 and 09 to 0B
These are program SEARCH commands, with some kinship to the GO operation in CMD. There is unnecessary redundancy here (05 = 09, 06 = 0A, 07 = 0B) that saves a little GOKEY interpreting logic, but could be eliminated in expanded versions if codes 09 to 0B can be given another useful interpretation. One major difference from the search GO in CMD is that searching is done in the range from the current CURAD (the address of the instruction that was in the display before the GO) upward in memory to ENDAD. To search the *entire* program, one must first reset CURAD to the program start by a GO-AD. The reason for this modification is that the same instruction may recur several times in a program, and the GO logic in CMD can only find the *first* occurrence. When using the EDITHA search GO, the user should keep the final command key depressed. If a match is found, the display will show the address of this instruction that will be displayed when the key is released (while the data display will show the number of bytes that were searched for). If this is not the one wanted, press the + key and redo the search GO to find the *next* identical instruction, higher in the program. If there is no match, the operation will display the ENDAD address,with EE in the data display.

The second major difference from CMD is that EDITHA can search for only the opcode (code 05), for the opcode and one following byte (code 06, identical with the CMD search GO), or for the opcode and *two* following bytes (code 07). E.g., the key sequence GO-85-FB-60-07 requires that an 85 FB 60 sequence exist in the program. This happens to be the terminal program sequence in the KIM monitor ROM. An interesting test of searching speed is possible if one sets BEGAD to 03FF, CURAD to 1FD4, and ENDAD and LIMAD both to 1FD5. When EDITHA is entered (set address to 019A and press GO) it will display the RTS ($60 at 1FD4. A GO-AD sends it to 0400, and 0404 is displayed (although of course there is nothing there in the basic KIM). The GO-85-FB-60-70 keying sequence will darken the display for less than a second, then (with the final key still depressed) display 1F D2 03. This is the addresss of the last 85 FB instruction in the ROM, that appears when the 7 key is released. Nearly 7K of (mostly non-existent!) memory has been subjected to a complex search in a reasonably short time. If one uses the 06 command code to search for only 85 FB, one can find 5 recurrences of it lower in the KIM ROM than the terminal one.

Mode IV, code OC
This works exactly like the GO-AD double-command. It was the awkwardness of resetting CURAD to the program start in this way (or in another way possible with Mode V) that led me to develop the GO-AD. In the original CMD design (largely retained in this pioneer version of EDITHA), keyings are recognized in 2 different ways. CMD calls the program-scanning routines that maintain display of the instruction at CURAD and detect a depressed key, which causes a return for interpretation by CMD logic. Hex keys are rejected (an EE EE EE error-message being displayed while they are depressed), while each of the 4 command keys causes control to shift to a special routine. With more elaborate logic, one could interpret double-commands here (or even, if one were ambitious, triple-commands). It would be a cleaner way than that in version 1.0, which re-uses the scaffolding of a primitive approach to implement a more sophisticated one (just to avoid one more reconstruction job).

The second kind of key-recognition in CMD is done by the GOKEY and ADKEY routines, via subroutine RDBYTE, which error-exits if a command key is pressed but accepts keyed-in hex bytes. My decision to implement double commands here was based on the fact that I had already included a lot of command-code interpreting logic and it was very simple just to modify READIN slightly. When I started out to revise CMD, I did not have a clear idea of how it worked and never dreamed that it would go so far! The structural flaws in EDITHA (1.0) are the result of its just growing like Topsy, with its growth halted when it started getting too big for the basic KIM-1.

Mode V, even-numbered codes from $10 to $8E
These are IDZP (insert data into zero-page) commands (64 in all), that write the first 2 keyed-in data bytes into contiguous zero-page locations specified by the command code. E.g., the keying sequence GO-12-34-10-10 will cause $34 to be written in location 0010 and $12 in location 0011. Before the keying-in of the command code 10, the first 3 bytes are displayed as 10 34 12 (reverse order); this is wiped out by the command code, since nothing is entered into the program. Since the effect is not visible and one has to be extra careful with zero-page, the third keyed-in (and so displayed) byte

must be *identical* with the final command code that causes execution. If it is not, an error message is displayed while the final command key is depressed to indicate a total wipeout.

Because important subroutines are located in zero page (0080 to 00D4), command codes 80 to 8E skip these locations and write into locations 00D8 to 00E7. E.g., the sequence GO-1C-00-8A-8A resets CURAD to address 1C00 in the KIM ROM, causing display of the instruction there. This program can then be scanned by the + key, since EDITHA does not object to its user going anywhere in memory just to *look* at it. Before wandering around in memory, ENDAD should be reset to the program-start address (easily checked by a GO-AD), using the 8C (reset ENDAD) command code, to "disconnect" the AD and DA keys. Codes 8E (reset LIMAD), 88 (reset BEGAD), and 86 (reset MOVAD) will be less often needed, and the 3 lower pointers (80, 82, 84) are available for future enhancements.

Some General Commands on the Listing

Since EDITHA interacts with molasses-slow human operators, it is not designed for speed but emphasizes coding efficiency, error-protection, and convenience. One major subroutine (ASCNIT, 0100 to 011E) is *not* listed because its coding is basically identical with that of the Fylstra-BYTE copyrighted CMD (located in the original at 0103 to 0121) except that the downward relocation in EDITHA requires that the 3JSRs to SCAN3 be coded as 20 1F 01 instead of the original 20 22 01. The relocation is functional, since the JSR DETLEN that is the first instruction of the CMD subroutine SCAN is now the second instruction in EDITHA (at 019B), so that the third one (JSR SCANIT, labeled DACMD) can be branched to to cause a display *not* interpreted as an instruction. My omission of SCANIT may strike some readers as ridiculous, but the intent is to avert the wrath of whatever gods EDITHA may offend! It also underscores the reasons for the free-diffusion conditions (spelled out in my covering letter) attached to my own copyright.

Listings therefore start at 011F (SCAN3), that is significantly modified in order to reduce the brightness of single-byte displays (the JSR ONE just kills a little time). To be punctilious, I note that the terminal coding of the EDITHA subroutine MVDOWN (00C9-00D4) is identical with that of the CMD subroutine ADVAND. It seems to me that the logic of adding an 8-bit number to a 16-bit number is so commonly used that it is in the public domain.

Since keying in 478 program bytes into a KIM is a headache, I am willing to tape EDITHA for KIM users who would like to try it out if (1) they mail a blank cassette to me, with a stamped self-addressed envelope that I can return it in, and (2) they agree to inform me of any flaws they may discover in operation (*after* checking the RAM to make sure the listing is *correct*). Regrettably, I can not be responsible for any failures of the U.S. Postal Service, and this approach will not be usable by users outside the USA. The taping will be in the standard KIM format (slow but sure) in 2 sections: 0080-0235 (index 01) and 1780-17E3 (index 02). I am not eager to do this hundreds of times, but fortunately this is an unlikely event!

Comment on Enhancements

As I have indicated, EDITHA has many possibilities for restructurings and new operations, and everyone is free to do whatever he or she likes. It is quite possible that someone has already written something as good as or better than EDITHA. More likely, every further revision will have some fine ideas of its own that ought to be in the ultimate version(s). Perhaps *DDJ* can serve as a clearinghouse for pooling these ideas, and thus hasten development of a super-version (that will merit a super-name!).

```
Ø1ØØ                 SCANIT
Ø1ØC                 SCAN1

Ø11F  A4 E8          SCAN3  LDY BYTES
Ø121  CØ Ø1                 CPY # 1 (test for 1-byte)
Ø123  DØ Ø3                 BNE NOTONE (skip if ≠ 1)
Ø125  2Ø 9E 19              JSR ONE  (time delay)
Ø128  A2 Ø9          NOTONE LDX # 9
Ø12A  A9 7F                 LDA # 7F
Ø12C  8D 41 17              STA PADD
Ø12F  2Ø 28 1F              JSR SCAND1
Ø132  6Ø                    RTS

Ø133  A6 EB          SEARCH LDX XINDEX (# bytes sought)
Ø135  AØ ØØ                 LDY # Ø
Ø137  B1 E2          TESBYT LDA (CURAD),Y (get byte)
Ø139  D9 F9 ØØ              CMP INH,Y  (match display?)
Ø13C  DØ Ø6                 BNE GONEXT (≠, new opcode)
Ø13E  C8                    INY        (=, next byte)
Ø13F  CA                    DEX    (matching completed?)
Ø14Ø  DØ F5                 BNE TESBYT (no, continue)
Ø142  FØ 73                 BEQ ADVAL (match found!)
Ø144  2Ø 8Ø ØØ       GONEXT JSR DETLEN
Ø147  2Ø 8D 17              JSR ADVANC
Ø14A  BØ E7                 BCS SEARCH  (more program)
Ø14C  9Ø 69                 BCC ADVAL (no match found)

Ø14E  A2 Ø4          GOKEY  LDX # 4 (user input 4 bytes)
Ø15Ø  86 EA                 STX COUNT (stored in COUNT)
Ø152  2Ø ØØ Ø2              JSR READIN (go get key input)

Ø155  A8                    TAY (move input byte into Y)
Ø156  FØ 5F                 BEQ ADVAL  (an ERASE exit)

Ø158  C9 1Ø                 CMP #$ 1Ø (hi or lo command?)
Ø15A  BØ 25                 BCS HICMD (hi, go to IDZP op)

Ø15C  C9 ØC                 CMP #$ ØC (double-command?)
Ø15E  9Ø ØA                 BCC ANDOUT (no, go to ANDOUT)
Ø16Ø  DØ 5B                 BNE ERROR (exit, only ØC legal)
Ø162  8A                    TXA  (X is ØØ for true double)
Ø163  DØ 58                 BNE ERROR (false, wipe-out)
Ø165  2Ø 8Ø 17              JSR BEGIN (set CURAD to start)
Ø168  BØ 4D                 BCS ADVAL (return to editor)

Ø16A  29 Ø3          ANDOUT AND # 3 (retain bits Ø and 1)
Ø16C  FØ 4F                 BEQ ERROR (Ø4, Ø8, not legal)
Ø16E  85 EB                 STA XINDEX (store for SEARCH)
Ø17Ø  CØ Ø4                 CPY # 4  (SEARCH command?)
Ø172  BØ BF                 BCS SEARCH  (yes, go do it)
```

```
Ø174  85 E8           STA BYTES (store for ops)
Ø176  2Ø DF 17        JSR LIMITS (OK to write?)
Ø179  9Ø 3C           BCC ADVAL  (no, exit)
Ø17B  2Ø EØ Ø1        JSR WRITIN (write data)
Ø17E  FØ 1E           BEQ DACMD (display data)

Ø18Ø  EA              NOP  (just a leftover!)

Ø181  C9 8F     HICMD CMP #$ 8F (command too hi?)
Ø183  BØ 38           BCS ERROR (yes, exit)
Ø185  C5 FB           CMP POINTH  (confirmed?)
Ø187  DØ 34           BNE ERROR (no, exit)
Ø189  4A              LSR A  (even or odd #?)
Ø18A  BØ 31           BCS ERROR (odd, exit)
Ø18C  ØA              ASL A (restore command)
Ø18D  1Ø Ø2           BPL NOADD (< $8Ø, use as is)
Ø18F  69 58           ADC #$ 58 (raise to $D8-up)
Ø191  AA        NOADD TAX (move command to X index)
Ø192  A5 F9           LDA INH (pickup first byte)
Ø194  95 Ø1           STA HIBASE+1,X (IDZP write)
Ø196  A5 FA           LDA POINTL (pickup second)
Ø198  95 ØØ           STA HIBASE,X (IDZP write)

Ø19A  D8        EDITHA CLD  (clear decimal mode)
Ø19B  2Ø 8Ø ØØ         JSR DETLEN (get opcode bytes)
Ø19E  2Ø ØØ Ø1  DACMD  JSR SCANIT

Ø1A1  C9 12      GCMD  CMP #$ 12
Ø1A3  FØ ØF            BEQ STPKEY (+ key = $12)
Ø1A5  BØ A7            BCS GOKEY (GO key = $13)
Ø1A7  C9 1Ø            CMP #$ 1Ø
Ø1A9  9Ø ØC            BCC ADVAL (any hex key)
Ø1AB  FØ 1F            BEQ ADKEY (AD key = $1Ø)

Ø1AD  2Ø B2 17         JSR MOVEUP (DAKEY deletion)
Ø1BØ  BØ E8            BCS EDITHA (OK, done)
Ø1B2  9Ø Ø3            BCC ADVAL (can't be done)

Ø1B4  2Ø 8D 17 STPKEY  JSR ADVANC (to next opcode)
Ø1B7  A6 E2     ADVAL  LDX CURAD (to be displayed)
Ø1B9  A5 E3            LDA CURAD+1  ( "     "   )
Ø1BB  BØ Ø2            BCS INFORM  (not an error)

Ø1BD  AØ EE     ERROR  LDY #$ EE  (display EE)
Ø1BF  84 F9     INFORM STY INH  (display Y in data)
Ø1C1  86 FA            STX POINTL (address low)
Ø1C3  85 FB            STA POINTH (address high)
Ø1C5  2Ø 1F 1F  ERR1   JSR SCANDS (display message)
Ø1C8  DØ FB            BNE ERR1 (while key is held)
Ø1CA  FØ CE            BEQ EDITHA (back to editor)
Ø1CC  2Ø DF 17  ADKEY  JSR LIMITS (write OK?)
Ø1CF  9Ø E6            BCC ADVAL (no, exit)
Ø1D1  86 EA            STX COUNT (sets to ØØ)

Ø1D3  2Ø ØØ Ø2         JSR READIN (get key input)
Ø1D6  FØ Ø3            BEQ ACCEPT (all-hex, OK)
Ø1D8  CA               DEX   (if AD-AD, X was ØØ)
Ø1D9  1Ø E2            BPL ERROR (not AD-AD, exit)
Ø1DB  2Ø EØ Ø1 ACCEPT  JSR WRITIN (write input)
Ø1DE  FØ BA            BEQ EDITHA (back to editor)

Ø1EØ  2Ø A4 ØØ WRITIN  JSR MVDOWN (open up space)
Ø1E3  A6 E8            LDX BYTES (# bytes to write)
Ø1E5  AØ ØØ            LDY # Ø
Ø1E7  B9 F9 ØØ STORIT  LDA INH,Y  (pickup byte)
Ø1EA  91 E2            STA (CURAD),Y (write it in)
Ø1EC  C8               INY   (for next pickup)
Ø1ED  CA               DEX  (if = Ø, all written)
Ø1EE  DØ F7            BNE STORIT (≠ Ø, next byte)
Ø1FØ  6Ø               RTS  (return to caller)

(coding for subroutine READIN)

2ØØØ  A2 Ø1    READIN LDX # 1  (set X to 1)
2ØØ2  86 E8           STX BYTES (display 1 byte)
2ØØ4  CA              DEX   (X now = ØØ)
2ØØ5  86 EB    NEXKEY STX XINDEX  (saves X)
2ØØ7  2Ø ØC Ø1 READHI JSR SCAN1 (read first key)
2ØØA  C9 1Ø           CMP #$ 1Ø  (a hex key?)
2ØØC  9Ø Ø5           BCC SHIFT (yes, use it)
2ØØE  A6 EB           LDX XINDEX (no, restore X)
2Ø1Ø  E9 Ø4           SBC # 4  (subtract, Acc - 4)
2Ø12  6Ø              RTS  (return to caller)
2Ø13  ØA       SHIFT  ASL A  (hex into hi nybble)
2Ø14  ØA              ASL A
2Ø15  ØA              ASL A
2Ø16  ØA              ASL A
2Ø17  85 E9           STA RDTEMP (save hi nybble)
2Ø19  2Ø ØC Ø1 READLO JSR SCAN1 (read second key)
2Ø1C  C9 1Ø           CMP #$ 1Ø (a hex key?)
2Ø1E  BØ F9           BCS READLO (no, try again)
2Ø2Ø  Ø5 E9           ORA RDTEMP (add hi nybble)
2Ø22  A6 EB           LDX XINDEX (restore X)
2Ø24  95 F9           STA INH,X (byte into display)
2Ø26  E8              INX (increment X for next op)
2Ø27  A4 EA           LDY COUNT (=Ø only first AD)
2Ø29  DØ Ø3           BNE OPSKIP (not an opcode)
2Ø2B  2Ø 84 ØØ        JSR BYTCNT (get bytecount)
2Ø2E  86 E8    OPSKIP STX BYTES (BYTES must = X)
2Ø3Ø  E4 EA           CPX COUNT (is X = COUNT?)
2Ø32  DØ D1           BNE NEXKEY (no, continue)
2Ø34  6Ø              RTS (return to caller)
1780  A4 E1    BEGIN  LDY BEGAD+1
1782  A6 EØ           LDX BEGAD
1784  E8              INX (true program start)
1785  86 E2           STX CURAD (resets low)
1787  DØ Ø1           BNE NOINY (high is OK)
```

```
1789  C8           INY  (increment high)
178A  84 E3  NOINY STY CURAD+1 (reset high)
178C  60           RTS (return to caller)

178D  18     ADVANC CLC
178E  A5 E2         LDA CURAD
1790  65 E8         ADC BYTES
1792  85 E2         STA CURAD
1794  90 02         BCC PRELIM
1796  E6 E3         INC CURAD+1
1798  A2 04  PRELIM LDX # 4 (for 2 adchecks)
179A  B5 E1  INLIMT LDA BEGAD+1,X
179C  A8            TAY  (save hi in Y)
179D  D5 DF         CMP MOVAD+1,X
179F  90 0E         BCC LIMRET (error exit)
17A1  D0 08         BNE TESTEX (hi ad is safe)
17A3  B5 E0         LDA BEGAD,X (test lo ad)
17A5  D5 DE         CMP MOVAD,X
17A7  90 06         BCC LIMRET (error exit)
17A9  F0 05         BEQ ERREX (if ads=, error)
17AB  CA     TESTEX DEX
17AC  CA            DEX
17AD  D0 EB         BNE INLIMT (≠0, continue)
17AF  60     LIMRET RTS  (return to caller)
17B0  18      ERREX CLC (clear carry, error)
17B1  60           RTS

17B2  20 98 17 MOVEUP JSR PRELIM (delete OK?)
17B5  90 27         BCC MVURET (no, exit)
17B7  84 DF         STY MOVAD+1
17B9  A6 E2         LDX CURAD
17BB  86 DE  UPLOOP STX MOVAD
17BD  A4 E8         LDY BYTES
17BF  B1 DE         LDA (MOVAD),Y
17C1  A0 00         LDY # 0
17C3  91 DE         STA (MOVAD),Y
17C5  E8            INX
17C6  D0 02         BNE CHKEND
17C8  E6 DF         INC MOVAD+1
17CA  E4 E4  CHKEND CPX ENDAD
17CC  D0 ED         BNE UPLOOP
17CE  A4 DF         LDY MOVAD+1
17D0  C4 E5         CPY ENDAD+1
17D2  90 E7         BCC UPLOOP
17D4  8A            TXA (CURAD in X now=ENDAD)
17D5  E5 E8         SBC BYTES
17D7  85 E4         STA ENDAD
17D9  B0 03         BCS MVURET
17DB  C6 E5         DEC ENDAD+1
17DD  38            SEC
17DE  60     MVURET RTS  (return to caller)
17DF  A2 06  LIMITS LDX # 6 (for 3 adchecks)
17E1  D0 B7         BNE INLIMT

80  A0 00  DETLEN LDY # 0
82  B1 E2         LDA (CURAD),Y
84  A0 01  BYTCNT LDY # 1
86  2C 81 19      BIT TES3 ($08 in 1981)
89  D0 08         BNE HALFOP (bit 3 = 1)
8B  C9 20         CMP #$ 20 (is it JSR?)
8D  F0 0E         BEQ THREE (length is 3)
8F  29 9F         AND #$ 9F (test bits 74321Ø)
91  D0 0B         BNE TWO (if ≠0, length is 2)
93  29 15  HALFOP AND #$ 15 (retain bits 420)
95  C9 01         CMP # 1 (is only bit 0 = 1?)
97  F0 05         BEQ TWO (if yes, length 2)
99  29 05         AND # 5  (retain bits 2,0)
9B  F0 02         BEQ ONE1 (if =0, length one)
9D  C8     THREE  INY   (length 3 exit)
9E  C8       TWO  INY   (length 2 exit)
9F  84 E8    ONE1 STY BYTES (save bytecount)
A1  84 EA         STY COUNT (save it again)
A3  60            RTS  (return to caller)

A4  A5 E5  MVDOWN LDA ENDAD+1  (this will set
A6  85 DF         STA MOVAD+1   MOVAD=ENDAD)
A8  A6 E4         LDX ENDAD
AA  E8            INX
AB  F0 05         BEQ DECEX
AD  8A     MVLOOP TXA           (this will
AE  D0 02         BNE DECEX     decrement
B0  C6 DF         DEC MOVAD+1    MOVAD)
B2  CA      DECEX DEX
B3  86 DE         STX MOVAD
B5  A0 00         LDY # 0       (this will
B7  B1 DE         LDA (MOVAD),Y    move
B9  A4 E8         LDY BYTES      program)
BB  91 DE         STA (MOVAD),Y
BD  E4 E2         CPX CURAD     (this will
BF  D0 EC         BNE MVLOOP     exit when
C1  A4 DF         LDY MOVAD+1    MOVAD=CURAD)
C3  C4 E3         CPY CURAD+1
C5  90 EF         BCC BRKERR (to 00 at 01B6)
C7  D0 E4         BNE MVLOOP
C9  18            CLC  (readjust ENDAD)
CA  A5 E4         LDA ENDAD
CC  65 E8         ADC BYTES
CE  85 E4         STA ENDAD
D0  90 02         BCC MVDRET
D2  E6 E5         INC ENDAD+1
D4  60     MVDRET RTS
```