Don Lancaster
Synergetics

Lowercase for Your Apple II (Part II)

The first part of this article introduced an inexpensive hardware mod to transform the Apple II into an uppercase/lowercase machine. Now for the software to complete the mod.

In part 1, we introduced a low-cost hardware modification to turn your Apple II into a combined uppercase/lowercase computer. In part 2 of this article, we will examine the software needed to complete the modification.

How much software you need depends on what you want to do with your new lowercase ability. If you are only going to use it to annotate an occasional game, very little new software will be needed. Most likely, your lowercase software will have to interact with any floppy disks or printers you have on-line, and you'll want an extensive editing capability. So, let's look at three different levels of software involvement.

First, we'll use the absolute minimum to display lowercase on the screen. Then, we'll show you software that lets you fill the screen with mixed uppercase and lowercase, with a working carriage return, scroll and so on. Finally, we'll check into a "heavyweight" full lowercase editing program that lets you put any character you want anywhere on the screen without the prompts and with full and easy editing. From here on, you'll be on your own to interact with what you really want to do with your new lowercase ability.

We will use integer BASIC for our software. This is easy but risky. Cursor and entry programs are fast and efficient when written in machine language. Integer BASIC *may* end up too slow for some things, particularly for repeatedly inserting and deleting characters. But integer BASIC is flexible and easy to use. It's also easy to change. So, we'll use the integer BASIC route. If things turn out a bit slow, we can pull some of the stunts in the green Apple book to speed things up. Once you know exactly what you want, you can go the machine-language route.

We will note in passing that there are simple and elegant machine-language cursor and entry manipulations already in the Apple monitor. These are available for call to an integer BASIC program. But, many of these sequences *demand* uppercase only and are restrictive in how you access them. So, we will avoid using what is already on hand—unless these sequences clearly and simply speed things up for us without creating more hassles than they solve.

Direct Entry

The minimum software route for displaying lowercase is to simply POKE the value of the character into the place you want it to go on the screen. This is very limited if you want to put down more than a few characters at once.

We'll shortly see what the decimal memory locations of every point on the display are. For instance, we'll find out that the bottom line of the screen goes from decimal 2000 at the left to decimal 2039 at the right.

Fig. 1 shows you the correct character codes for all the characters *as they are to be stored in memory*. For instance, say you want to put a character on the bottom line, third from the left. For an uppercase A, use POKE 2002,129. For a lowercase a, use POKE 2002, 33, and so one.

The missing numbers in Fig. 1 are repeats of the characters already shown. A POKE in the range of 64 to 127 will flash an uppercase character or letter. I haven't found a good hardware way to flash lowercase, so we will use software for flashing or winking cursors. More on this later.

Lowercase

32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
\	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
p	q	r	s	t	u	v	w	x	y	z	<	¦	>	~	■

Uppercase

128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143
@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159
P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	↑	_

Numerals

160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175
spc	!	"	#	$	%	&	'	(	)	*	+	,	–	.	/
176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191
0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?

Fig. 1. Decimal character codes needed for direct POKEing into display memory. Use software only to flash lowercase. To flash uppercase or numerals, subtract 64 from the decimal value or use software. Decimal numbers now shown are redundant.

A. To read the keyboard:

```
200  CHAR = PEEK( – 16384): IF CHAR<127 THEN 200: POKE ( – 16368),0
```

This sequence stays at 200 till a key is pressed. Key value before strobe reset appears as CHAR.

B. To print the decimal value of a pressed key:

```
200  CHAR = PEEK( – 16384): IF CHAR<127 THEN 200: POKE ( – 16368),0 :
        PRINT CHAR : GOTO 200
```

This sequence stays at 200 till a key is pressed. Key decimal value is displayed for each new key pressed. CTRL C stops the action.

C. To stop a program without scrolling or prompting:

```
        600 GOTO 600
```

This trap holds the screen and prevents scrolling or prompting. To get out of the trap, use CTRL C.

D. To measure the speed of an integer BASIC sequence:

```
100  FOR N = 1 TO 10000
200  (((((SEQUENCE GOES HERE)))))
300  NEXT N
```

The execution time in *milliseconds* equals *one-tenth* the number of *seconds* from RUN till the speaker beeps, minus the time (about 1 millisecond) to run without step 200.

Fig. 2. Apple II integer BASIC utility sequences.

Four Utility Sequences

It's far more desirable to get your characters from the keyboard than extracting them from memory or using POKE commands. Before we look at the lowercase keyboard entry data, let's pick up some integer BASIC utility sequences that may be very handy for us. Four of these sequences are shown in Fig. 2.

First and most important, we have to be able to read the keyboard without using a carriage return for every character. Fig. 2a shows us how to do this. The Apple II keyboard is located at decimal – 16384. If a key is pressed, the number at this location will exceed decimal 127, and the value will correspond to the selected key.

We'll call the look at the keyboard CHAR, short for character. We'll keep looking at the keyboard with the PEEK command. A CHAR that is more than 127 means a key has been pressed, so we save the value of CHAR. Then we reset the keyboard strobe with the POKE (– 16368),0 command shown. Be sure to always reset the keyboard after you read it. Your value for CHAR is the decimal equivalent of the pressed key. It can be used in the next step of your program or saved till needed. After you are done with this particular key, jump to 200 to await a new closure.

You can print the decimal values of all the keys simply by adding a PRINT CHAR command. This will display the value of each key as it is pressed (see Fig. 2b). The results of this for all the keys are shown in Fig. 3. You'll find this chart convenient to decode the various control functions. We see that the Apple II keyboard has no apparent way to provide lowercase characters, as well as the uppercase \ and [. Control characters NUL, FS, GS, RS and US are also not immediately available. Uppercase] is hidden as a shifted M and is used as the AppleSoft prompt.

One of the more infuriating things that happens when you are building a display editing program is that you put something somewhere, and then the BASIC throws in a scroll and a prompt, moving everything up the screen. To temporarily defeat the return to BASIC, just use a trap such as the 600 GOTO 600 shown in Fig. 2c. Your program will stick in the trap till you release it. This allows you to watch part of a program to make sure it is doing what you want it to. To release your trap, use CTRL C. You must, of course, eliminate all traps from your final program.

Suppose something we do turns out too slow. How can we find out how fast our BASIC is working for us? Fig. 2d shows us the way to measure the execution time of any BASIC sequence. What you do is repeat the sequence over and over again for 10,000 times in a loop. The number of *tens* of seconds it takes to execute the sequence will equal the number of milliseconds the sequence actually took. This is easily timed with a kitchen clock or a stopwatch. Be sure to subtract the millisecond it takes for the timer loop to cycle with nothing inside the loop.

With luck, you will never need this speed measurer. But, if ever you have characters being ignored or have things taking far too long in your particular program, this how-fast-is-it program can often show you what is holding up the works.

NORMAL		SHIFT		CTRL	
1	(177)	!	(161)	1	(177)
2	(178)	"	(162)	2	(178)
3	(179)	#	(163)	3	(179)
4	(180)	$	(164)	4	(180)
5	(181)	%	(165)	5	(181)
6	(182)	&	(166)	6	(182)
7	(183)	'	(167)	7	(183)
8	(184)	(	(168)	8	(184)
9	(185)	)	(169)	9	(185)
0	(176)	0	(176)	0	(176)
:	(186)	*	(170)	:	(186)
–	(173)	=	(189)	–	(173)
ESC	(155)	ESC	(155)	ESC	(155)
Q	(209)	Q	(209)	DC1	(145)
W	(215)	W	(215)	ETB	(151)
E	(197)	E	(197)	ENQ	(133)
R	(21)	R	(210)	DC2	(146)
T	(212)	T	(212)	DC4	(148)
Y	(217)	Y	(217)	EM	(153)
U	(213)	U	(213)	NAK	(149)
I	(201)	I	(201)	HT	(137)
O	(207)	O	(207)	SI	(143)
P	(208)	@	(192)	DLE	(144)
RETURN	(141)	CR	(141)	CR	(141)

NORMAL		SHIFT		CTRL	
A	(193)	A	(193)	SOH	(129)
S	(211)	S	(211)	DC3	(147)
D	(196)	D	(196)	EOT	(132)
F	(198)	F	(198)	ACK	(134)
G	(199)	G	(199)	BEL	(135)
H	(200)	H	(200)	BS	(136)
J	(202)	J	(202)	LF	(138)
K	(203)	K	(203)	VT	(139)
L	(204)	L	(204)	FF	(140)
'	(187)	+	(171)	;	(187)
←	(136)	BS	(136)	BS	(136)
→	(149)	NAK	(149)	NAK	(149)
Z	(218)	Z	(218)	SUB	(154)
X	(216)	X	(216)	CAN	(152)
C	(195)	C	(195)	ETX	(131)
V	(214)	V	(214)	SYN	(150)
B	(194)	B	(194)	STX	(130
N	(206)	↑	(222)	SO	(142)
M	(205)	]	(221)	CR	(141)
,	(172)	<	(188)	,	(172)
.	(174)	>	(190)	.	(174)
/	(175)	?	(191)	/	(191)
SPACE	(160)	SPACE	(160)	SPACE	(160)

Fig. 3. Decimal codes for the Apple II keyboard. REPEAT, SHIFT and CTRL act only on other keys. RESET is direct acting. Bordered values = control commands. Values shown are before strobe reset. For ASCII equivalent, subtract decimal 128.

A Lowercase Tester

Program A shows us a simple program that reads the keyboard and puts lowercase characters on the bottom line of the display for us. The program has only one feature—it is short. This makes it convenient for initial tests. But since it lacks a cursor and a way to print uppercase and prints all machine commands on the screen, we'll really need a better program for anything but checkout.

The program is a simple loop that progresses across the bottom line addresses 2000 to 2039. We read the keyboard in 110, until a key is pressed. Then we reset the keyboard. If the character has a value greater than decimal 192, we subtract 160 from it to convert it to lowercase. For instance, an uppercase A will have a CHAR value of 193, per Fig. 3. *Subtract 160* from this to get 33, the lowercase a needed in Fig. 2. We then load the character onto the display in the cursed position. Incrementing the loop with the NEXT CURS instruction in 150 moves us across the screen while the GOTO 100 in line 160 resets us to the beginning of the line.

A Useful Display Program

Let's add some statements to Program A to make it more useful. We can scroll at the end of the line to move the statements progressively up on the display. We can decode a RETURN to do the same thing. And, if we can only figure out some way to get both uppercase and lowercase characters out of an uppercase keyboard, we will be home free toward a simple way to get continuous uppercase and lowercase messages displayed.

To trick the keyboard into being something it is not, we'll use the ESCAPE key. We'll set the program up so that under "normal" conditions you get all lowercase characters. If you hit ESCAPE once, only the next character will be capitalized. This is just like hitting SHIFT momentarily on a regular typewriter.

If you hit ESCAPE twice in a row, the keyboard will lock into an uppercase-only mode. This is just like using the LOCK on a regular typewriter. If you are locked into uppercase, hitting ESCAPE one more time gets you into lowercase once again, just as hitting SHIFT after LOCK on a typewriter puts you back into lowercase. However, since we are using software, our ESCAPE commands will only apply to the alphabet—everything else stays the same.

This may sound complicated, but it's simple to use. When and if your Apple II is to have mixed uppercase and lowercase, just use ESCAPE instead of SHIFT to shift the alphabet. Everything else stays the same.

The software behind this is simple enough. We have a variable called SHIFT and a variable called LOCK. Every time a character is entered, it attempts to reset SHIFT to zero and is allowed to do so if LOCK is also a zero.

When an ESCAPE key is sensed:

1. First you check to see if LOCK is a 1. If so, this means you want to *release* all caps, so you simply make LOCK a 0 and SHIFT a 0 and go on to the next key.

2. Then you check to see if the previous key is also an ESCAPE. If it is, SHIFT must be a 1, since no intervening character has a chance to reset SHIFT back to 0. We then make LOCK a 1 and go on to the next key.

3. If you got this far, SHIFT and LOCK must both be 0. This means you either want to capitalize only one word, or else another ESCAPE will follow to lock. So, make SHIFT a 1 and then go on to the next key.

```
100 FOR CURS = 2000 TO 2039
110       CHAR = PEEK ( - 16384): IF CHAR<127 THEN 110
120       POKE ( - 16368),0
130       IF CHAR>192 THEN CHAR = CHAR - 160
140       POKE CURS,CHAR
150       NEXT CURS
160 GOTO 100
```

Program A. Lowercase test program. Puts lowercase characters on the bottom display line. Numerals and punctuation appear normally. Use this program only for hardware checkout. CTRL-C restores normal BASIC operation.

The new, improved program is shown in Program B. This enters full alphabet characters sequentially on the bottom line for us, with working scroll and carriage return. SHIFT is used for everything already on the keycaps, while ESCAPE is used to pick upper, lower and mixed cases. Once again, one ESCAPE capitalizes only the next character. Two ESCAPEs capitalize everything, until a third ESCAPE resets back to lowercase.

The program works the same way as Program A does. Line 100 indexes us across the bottom of the screen, while 110 reads the keyboard for us.

Line 120 tests for carriage return and calls for a scroll if one is needed. Line 130 tests for ESCAPE and then does the shift lock processing in lines 190-210. If shift is not locked, line 140 converts to lowercase. Line 150 enters the characters on the screen.

Your turn: What does line 160 do in Program B ? Why is it needed.

Line 170 tells us to pick the next character location to the right. If this happens to be off the screen to the right, we drop out of the loop, do a scroll and start over.

A Full-Performance Lowercase Editor

The previous "gee-whiz" programs are handy to put lowercase on an Apple II. But, what we really may want is some full editing system that lets us:

- Put any character anywhere on the screen.
- Move around anywhere we like.
- Insert and delete characters and lines.
- Justify, ragged or flush right.
- Have lines longer than 40 characters.
- Transfer into and out of floppy.
- Provide hard-copy output.
- Have an attractive cursor for all characters.
- Have no BASIC prompts or unwanted scrolls messing up the screen.

Let's look at some of the bits and pieces that will be helpful to build an editor and display system. Then we'll show you a medium-complexity display editor that lets you wander around the screen with a vengeance. From there, you should be able to pick up just about as fancy a text editor as you care to.

Apple Display Memory Locations

The Apple people were among the first to recognize the incredible power and economy of using main memory also as a display memory. They do this by sharing each clock cycle so that the computer gets the memory for half a microsecond and the dedicated system timing gets the memory for display uses on the other half.

As you find out fast when you try to stuff things onto the screen, the memory locations are *not* sequential and are not all in one piece. How can we find what goes where?

The Apple II has two display pages, one residing between decimal 1024 and 2047 and a second page immediately above. Only the first page is normally used. Fig. 4 shows us a hex map of the Apple II display memory locations. Their mapping is somewhat similar to the memory repacking done in the KIM systems in *The Cheap Video Cookbook* (Sams 21524). Apple chose to stuff three lines per each half of a 6502's page of 256 words.

Apple uses a 40-character horizontal line numbered left to right from 0 to 39. They use a 24-row vertical field numbered top to bottom from 0 to 23.

Fig. 4 is fine for all us machine-language freaks. But integer BASIC works in decimal numbers, and it's not at all obvious what goes where. Fig. 5 is a remapping of the Apple II screen showing us what portion of the memory goes where on the screen, in decimal numbers. For instance, decimal character location 1706 is the third character from the left on the fourteenth line down from the top.

These sure are strange numbers. They were picked to simplify the internal Apple II system timing. As you can see, if you just try to sequentially put stuff on the screen, you'll put down the top line, then the ninth line down, then the seventeenth. Then you'll lose eight characters *down the drain* somewhere. Then onto the second, tenth, eighteenth lines. Then lose eight more characters. Messy, yes, but a great hardware simplification.

As a general rule, if you can't use hardware to simplify software, then you use software to simplify hardware. One or the other. Works every time.

Suppose a programmer would like to have a variable H for the horizontal position with a 0 to 39 range and a variable V for the vertical position ranging from 0

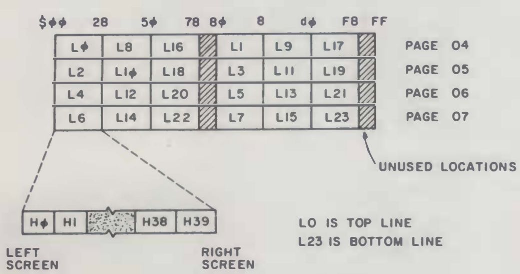

Fig. 4. Display memory locations of Apple II shown as hex map.

	H0	H39
V0	1024	1063
	1152	1191
V2	1280	1319
	1408	1447
V4	1536	1575
	1664	1703
V6	1792	1831
	1920	1959
V8	1064	1103
	1192	1231
V10	1320	1359
	1448	1487
V12	1576	1615
	1704	1743
V14	1832	1871
	1960	1999
V16	1104	1143
	1232	1271
V18	1360	1399
	1488	1527
V20	1616	1655
	1744	1783
V22	1872	1911
	2000	2039

Locations *not* on screen
1144-1151
1272-1279
1400-1407
1528-1535
1656-1663
1784-1791
1912-1919
2040-2047
Each horizontal row is numbered sequentially from left to right

Fig. 5. Display memory locations of Apple II shown as decimal locations.

```
10 REM THIS APPLE INTEGER BASIC PROGRAM DISPLAYS LOWER CASE CHARACTERS. USE ESC
      TWICE FOR SHIFT LOCK. USE ESC ONCE FOR SHIFT OR RELEASE.
100 FOR CURS = 2000 TO 2039
110      CHAR = PEEK ( - 16384): IF CHAR<127 THEN 110: POKE ( - 16368),0
120           IF CHAR = 141 THEN 180 .: REM CR
130           IF CHAR = 155 THEN 190 : REM ESC
140           IF CHAR> 192 AND SHIFT = 0 THEN CHAR = CHAR - 192
150      POKE, CURS, CHAR
160           IF LOCK = 0 THEN SHIFT = 0
170 NEXT CURS
180 CALL -912: GOTO 100: REM SCROLL
190 IF LOCK = 0 THEN 200: LOCK = 0: SHIFT = 0: GOTO 110: REM RELEASE LOCK
200 IF SHIFT = 0 THEN 210: LOCK = 1: GOTO 110: REM SET LOCK ESC #2
210 SHIFT = 1: GOTO 110: REM SHIFT ON ESC #1
```

Program B. Lowercase display program to fill the screen with combined upper and lowercase text via bottom line entry. SCROLL and RETURN work. No visible cursor or upper screen access.

H = Horizontal position 0(left) to 39(right)
V = Vertical position 0(top) to 23(bottom)
For lines 0-7:
Address = 1024 + (128•V) + H
For lines 8-15:
Address = 1064 + (128•(V − 8)) + H
For lines 16-23:
Address = 1104 + (128•(V − 16)) + H

Fig. 6. One method of calculating Apple II display addresses.

to 23. Obviously, we need a way to go from the H and V locations to the magic display memory addresses.

The Apple II monitor does this in the firmware with a disgustingly elegant sequence, BASCALC, starting at hex $FBC1. BASCALC takes the H value in $24 and the V value in $25 and puts the result BASL in $28 and BASH in $29. Thus, the programmer uses H and V, while the machine hardware uses BASL and BASH, and everybody is happy.

Unfortunately, quite a bit of PEEKing, POKEing, pushing and shoving is involved to call this sequence from integer BASIC. Instead, let's find a BASIC way to generate the right addresses.

Fig. 6 shows the math needed to find a particular address on the screen. The formulas are in three parts, depending on what third of the screen you happen to be on. To find a screen location, just use one of these formulas, and the results should agree with Fig. 5.

You can, of course, program these formulas into integer BASIC—and it's fun to do—but we need something faster and simpler. Fig. 7 shows us a look-up table to do the same thing. We store the leftmost address for the 24 lines as an array of values called B(V), meaning "Base address for line #V." To this, we add the horizontal value and get a result, CURS, that has the correct display address for a given H and V.

Note that there are two ways to enter the program. The first time you enter, you have to set up the B(V) array and initialize all the values. It's recommended you do this every time you clear the screen to make sure this table is intact. After we are sure the table is properly stashed, we can enter at 2000, and do the simple one-line CURS calculation shown in 2020. It is very important to be sure that the V and H values are, in fact, on the screen. Otherwise, you might end up POKEing a character into memory somewhere off the screen, plowing up a program or some operating system. This is why you should check H and V (lines 2000 and 2010) immediately before you use them.

A Software Cursor

There doesn't seem to be an obvious way to keep Apple II compatibility and be able to use the hardware cursor to wink lowercase. So, a software cursor can be used instead. Fig. 8 shows us how to combine your keyboard scanning with a cursor routine that winks any character on the screen by replacing the character with a solid box, repeating a few times a second.

A single loop is used to both provide a cursor and test for pressed keys. If no key is pressed, the loop will continue.

On the first trip through the loop, the cursed character is temporarily saved and is then replaced with a box cursor. On the 12th trip through the loop, the box cursor is removed and replaced with the saved character. On the 24th trip through the loop, the sequence repeats.

```
initial
enter    1000 DIM B(64)
         1010 B(0) = 1024:B(1) = 1152:B(2) = 1280:B(3) = 1408:
                B(4) = 1536:B(5) = 1664:B(6) = 1792:B(7) = 1920
         1020 B( 8) = 1064:B( 9) = 1192:B(10) = 1320:B(11) = 1448:
                B(12) = 1576:B(13) = 1704:B(14) = 1832:B(15) = 1960
         1030 B(16) = 1104:B(17) = 1232:B(18) = 1360:B(19) = 1488
                B(20) = 1616:B(21) = 1744:B(22) = 1872:B(23) = 2000
usual
enter    2000 IF V>23 THEN V = 23: IF V<0 THEN V = 0
         2010 IF H>39 THEN H = 39: IF H<0 THEN H = 0
         2020 CURS = B(V) + H
```

Fig. 7. An integer BASIC sequence to find Apple II display memory locations. For cold start, enter and initialize sequence at 1000. To find a location after initialization, enter at 2000. CURS will carry the correct display location to the instruction following 2020.

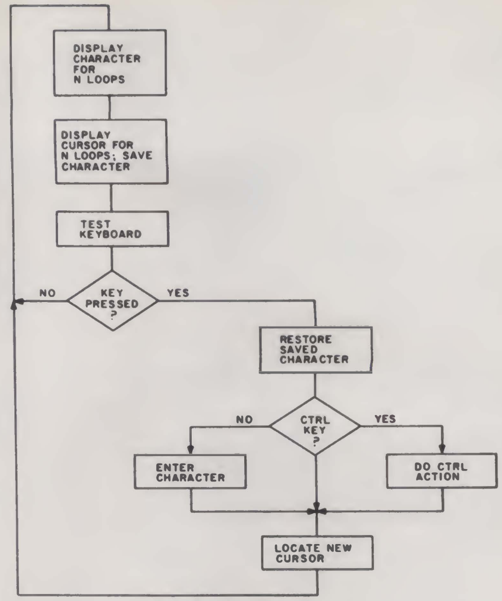

Fig. 8. Flowchart for an editing display that combines a winking software cursor within the keyboard testing loop.

So long as no key is pressed, a winking cursor appears on the screen. When a key finally is hit, the cursor is immediately erased and replaced with the correct character. If things happen to be on the second half of the loop, the character simply replaces itself. At any rate, when we are sure we have a pressed key, we make sure the cursor goes away.

The key is then tested to see if it is a character or a machine command. If it is a character, it is entered. If it's a machine command, the command is acted on if valid and ignored if not.

The new cursor location is found only after character entry or machine command actions are complete. The program then jumps back to the main loop, testing for pressed keys and winking the cursor. Cursor winking speed is software adjustable.

One interesting feature of the combined cursor and key-check loop is that the cursor always goes on the instant after a new location appears. This is much cleaner looking and easier to follow than the "aliasing" that sometimes takes place with rapid motions of a hardware-blinked, asynchronous cursor.

A Full Dual-Case Editing System

Program C shows a medium-complexity full-editing system that puts uppercase and lowercase characters anywhere you want on the screen, with full cursor motions. Features included are uppercase and lowercase, clearing, normal entry, cursor right-left-up-down, carriage return, scrolling, erase to end of line, erase to end of paragraph, lowercase shift and shift lock. Four "hooks" are provided to interact with your disk or hard-copy system or to add other features. It's a simple matter to add all the extras you want.

In lines 100 through 200, we set up the base address file for our screen address finder. These values are rechecked every time the screen is erased. Line 140 gives us a clear screen on startup and when called for. It uses the clearing sequence already in the monitor. Lines 160 through 180 find valid cursor locations for us, starting with H and V positions.

Our combination cursor loop

and keyboard test appears in lines 200 through 280. A cursor-counting variable, CCNT, counts from 0 to 24 for us. On count #1, the character being cursed is stored temporarily as CSTR. The cursor box (an ASCII 63, DEL) is loaded in its place. On CCNT count #12, the original character is replaced. On CCNT count #24, the cycle repeats. Meanwhile, the keyboard has been checked for a pressed key 24 times. You can think of CCNT as a divide-by-24 counter that is clocked by the keyboard testing. By changing the numbers, you can change the winking rate and the ratio of cursor to character time.

Once a key is pressed, we reset the keyboard strobe and make sure the cursed character has been put back where it belongs. Line 270 does this for us. Then, 280, we test for CTRL keys.

If the pressed key happens to be a character, line 300 decides whether lowercase or uppercase is to be displayed. Line 310 releases shift after a capital letter unless the shift is locked.

Actual character entry takes place in 400, while the cursor is adjusted in 410 and 420. If we go off-screen to the right, H is reset to 0 and V is incremented down-screen by one. If V goes off-screen, we call for a scroll, using the firmware scroll sequence in the monitor. After repositioning the cursor, the program returns to the main cursor and keycheck loop by jumping to 160. At this time, the cursor starts winking in the new location.

CTRL keys are processed in lines 1000 to 1040. Most are obvious. Line 1000 is needed so you can use the firmware erase-to-end-of-screen in the monitor; this step transfers the BASIC H and V values to the slots in the monitor where they are needed. Unfortunately, the monitor's erase-to-end-of-line firmware sequence doesn't seem to be as useful (it doesn't calculate its own base address), so this shorter erase sequence is done on our own in line 1080.

The spare hooks are shown in 1100 through 1130. Simply replace 160 (return-to-keyboard-loop) with the location you need for access to your disk, printer or other program. About a dozen other hooks can be added, just by picking new CTRL commands from Fig. 3. Remember that CTRL-C is excluded as this returns you to the integer BASIC operating system.

Should no valid CTRL key be found, the jump in 1140 puts us back into the keyboard checking business.

Lines 2000 through 2020 do the now familiar ESCAPE processing for the lowercase shift lock. As before, a single ESCAPE gives one capital letter. Two in a row locks us into capitals only. Should we be locked into capitals only, the next ESCAPE unlocks back to lowercase.

Some Extras

You can add just about anything you like to this editor program. For super-easy editing, you might like to add an additional keypad that generates all the motion commands with a single keystroke each. This heavyweight modification would be handy for word processing, typesetting and so on.

It's fairly obvious how you would add diagonal and cursor home motions, cursor OFF-ON, tabs, etc. To do really fancy editing, you have to be able to add and delete characters. How you do this depends on the rules you choose to set up for your particular system. Several full editors are available as software packages that may be of help to you.

A simple example of a delete-character subroutine is shown in Program D. Starting at the cursor plus one, every character on the line is moved one to the left. When this is finished, the last character will be repeated twice. The duplicate end character is then erased. The repeated moves take place in the 4000 to 4030 loop, while the end-character erasure happens in step 4040. This particular delete-character sequence

```
  10  REM  EDITING DUAL CASE DISPLAY SYSTEM FOR APPLE II
  20  REM  CLEAR = CTRL X             CURSOR RIGHT = RIGHT ARROW
           SHIFT = ESCAPE             CURSOR LEFT = LEFT ARROW
           LOCK = ESCAPE X2           CURSOR UP = CTRL A
  30  REM  UNLOCK = ESCAPE            CURSOR DOWN = CTRL B
           RETURN = RETURN            ERASE EOL = CTRL D
           HOOKS = CTRL Q,R,S,T       ERASE EOP = CTRL W
 100  DIM B(64): REM SET UP BASE ADDRESS TABLE
 110  B(0) = 1024:B(1) = 1152:B(2) = 1280:B(3) = 1408:
      B(4) = 1536:B(5) = 1664:B(6) = 1792:B(7) = 1920
      B(8) = 1064:B(9) = 1192:B(10) = 1320:B(11) = 1448
 120  B(12) = 1576:B(13) = 1704:B(14) = 1832:B(15) = 1960
      B(16) = 1104:B(17) = 1232:B(18) = 1360:B(19) = 1488
      B(20) = 1616:B(21) = 1744:B(22) = 1872:B(23) = 2000
 140  CALL -936: H = 0: V = 0: REM CLEAR SCREEN; HOME CURSOR
 160  IF V>23 THEN V = 23: IF V<0 THEN V = 0
 170  IF H>39 THEN H = 39: IF H<0 THEN H = 0
 180  CURS = B(V) + H: REM FIND CURS ADDRESS AFTER VALID V,H
 200  CCNT = 0
 210  CCNT = CCNT + 1
 220  IF CCNT>1 THEN 240: CSTR = PEEK (CURS)
 230       POKE (CURS),63: REM SAVE CHAR; WRITE CURSOR
 240  IF CCNT = 12 THEN POKE CURS,CSTR
 250  IF CCNT>23 THEN CCNT = 0: REM UNWINK CURSOR
 260  CHAR = PEEK ( - 16384): IF CHAR<127 THEN 210
 270       POKE ( - 16368),0: POKE CURS,CSTR
 280       IF CHAR<160 THEN 1000: REM CTRL KEY TEST
 300  IF (CHAR>192 AND SHIFT = 0) THEN CHAR = CHAR - 160: REM LOWER CASE ONLY IF
      UNSHIFTED CAPITAL LETTER
 310  IF LOCK = 0 THEN SHIFT = 0: REM RETURN TO LOWER CASE IF UNLOCKED
 400  POKE CURS, CHAR: REM ENTER CHAR
 410       H = H + 1: IF H<40 THEN 160: H = 0: REM ADJ H POS
 420  V = V + 1; IF V>23 THEN CALL -912: GOTO 160: REM ADJ V POS; SCROLL IF OFF SCREEN
1000  POKE 36,H: POKE 37,V: REM TRANSFER HV TO MONITOR FOR EOS
1010  IF CHAR = 152 THEN 100: REM CLEAR AND HOME ON CTRL X
1020  IF CHAR #141 THEN 1030: H = 0: V = V + 1: IF V>23 THEN CALL -912: REM CARRIAGE
      RETURN. SCROLL IF OFF SCREEN.
1030  IF CHAR = 136 THEN H = H - 1: REM BACKSPACE ON ARROW
1040  IF CHAR = 139 THEN H = H + 1: REM ADVANCE ON ARROW
1050  IF CHAR = 129 THEN V = V - 1: REM CURSOR UP ON CTRL A
1060  IF CHAR = 130 THEN V = V + 1: REM CURSOR DOWN ON CTRL B
1070  IF CHAR = 155 THEN 2000: REM ESCAPE SHIFT SEQUENCE
1080  IF CHAR #132 THEN 1090: FOR H1 = H TO 39: POKE (B(V) + H),63: NEXT H1: REM ERASE TO
      END OF LINE ON CTRL D
1090  IF CHAR = 151 THEN CALL -958: REM MONITOR ERASE EOS ON CTRL W
1100  IF CHAR = 145 THEN 160: REM SPARE HOOK ON CTRL Q DC1
1110  IF CHAR = 146 THEN 160: REM SPARE HOOK ON CTRL R DC2
1120  IF CHAR = 147 THEN 160: REM SPARE HOOK ON CTRL S DC3
1130  IF CHAR = 148 THEN 160: REM SPARE HOOK ON CTRL T DC4
1140  GOTO 160: REM RESUME KEYBOARD SCAN ON UNUSED CTRL COMMAND
2000  IF LOCK = 0 THEN 2010: LOCK = 0: SHIFT = 0: GOTO 160: REM RELEASE LOCK
2010  IF SHIFT = 0 THEN 2020: LOCK = 1: GOTO 160: REM SET LOCK ON SECOND ESCAPE
2020  SHIFT = 1: GOTO 160: REM SHIFT ON FIRST ESCAPE
```

Program C. Full lowercase Apple II editing display system.

operates only on a single line. Lines further down the screen are not affected.

Inserting extra characters is a more difficult problem, since everything has to be shoved around the screen to make enough room. Once again, you have to pick the shoving rules you want to use for your particular editing needs.

One possibility, insert-a-character subroutine, is shown in Program E. This uses a rule that says it will keep bumping characters until it finds a line whose last character is a space. Usually, this will be the line you are working on, but if not, characters will keep getting bumped till a space at the end of a line is found. Then the bumping stops and the rest of the screen stays the way it was.

Here are the steps involved in this insert-a-character sequence:

1. A check is made to find out how many lines are involved, till one is found with a space at the end (lines 3000 to 3040).
2. Everything on the bottommost line to be bumped shifts one to the right. Remember that at least the rightmost character is a space on this line.
3. There will be a double character at the left of the line, provided it's not the one that had the cursor on it. This double character is replaced with the last character on the previous line (3160, 3170).
4. The process repeats as often as needed for all but the top line to be bumped. The loop is done with line 3100.
5. The line with the cursor on it gets characters bumped only from the cursor to the end of the line and has no need to borrow a character from a previous line. The change in policy for the cursed line is handled by line 3110.
6. Finally, everything will be bumped, but a duplicate character will remain at the cursed location. This dupe is erased in line 3190.

This is a fairly simple inserter that works fairly well and reasonably fast. If you don't like its rules, change them to suit yourself. The sequence is rather slow if you use it over and over again, as you might while justifying a whole page of text. You should be able to speed it up considerably if you want. The rule selected does have one possible bug in it—repeated insertions can swallow end spaces and run words together, since the next line bumping takes place with a character in the last slot and does not if a space is there. Requiring two spaces at line end may help. There are all sorts of other options, depending on what you want your particular editor to do.

```
4000  FOR H1 = H TO 38
4010        CURM = B(V) + H1
4020        POKE CURM,PEEK (CURM + 1): REM MOVE ONE LEFT
4030  NEXT H1
4040  POKE (B(V) + 39),160; REM BLANK END CHAR
4050  RETURN
```

Program D. BASIC subroutine to delete a single character on the Apple II display. It starts at the cursed location and moves everything on its own line left one character. The last character is erased.

```
3000 V2 = 0: H2 = 0
3010 FOR V1 = V TO 23: REM FIND FIRST END SPACE
3020       CEND = PEEK(B(V1)  +  39)
3030       IF CEND = 160 THEN 3100
3040       V2 = V2 + 1: NEXT V1
3100 FOR V1 = (V2 + V) TO V STEP  – 1: REM: NEXT LINE
3110       IF V1 = V THEN H2 = H
3120       FOR H1 = 38 TO H2 STEP  – 1: REM SHIFT A LINE
3130             CURM = B(V1) + H1
3140             POKE (CURM + 1), PEEK (CURM)
3150             NEXT H1
3160       IF V1 = V THEN 3180: REM MOVE (V1 – 1),39 TO V1,0
3170             POKE B(V1),PEEK (B(V1 – 1) + 39)
3180       NEXT V1
3190 POKE ( B(V) + H), 160: REM: DELETE CHARACTER
3200 RETURN
```

Program E. BASIC subroutine to insert a single character on the Apple II display. It starts at the cursed location. It finds the first available characters as needed. The cursed character is then erased.

Your turn: Add the following extras to your editing program:

- *Ragged justify right—in which whole words are never broken on the right side of the screen and you can continuously type without carriage returns.*
- *Flush justify right—in which everything ends up square on the right side of the screen as needed for typesetting. What hyphening and short-line rules will you use for this?*
- *Variable character lines—in which you can go as long as 80 characters for text and form letter editing.*

As a hint to longer lines, just select *pairs* of lines when they are needed and act on these line pairs. Thus you should be able to output up to 80 characters for a business letter or a manuscript to your hard copy, while still viewing the results on a normal 40-character Apple up to you since the results are application specific. Have fun with all this.

Further Hardware Mods

Some of the more popular Apple II software uses the screen-reversal feature. This software may not be reasonably displayed with the hardware mods we've shown you so far. The checkbook program is one example, where deposits are shown reversed as black on white numerals. Is there some way we can still run these programs *and* have lowercase?

One obvious way is to use a switch to select *either* screen reversal *or* lowercase. Fig. 9 shows where this switch goes. Only an SPST switch and a resistor need be added to the existing modifications. This switch can be mounted along the right side of the circuit board far enough to the rear that it is easily reached. A miniature slide switch held in place with double-stick foam should do the trick.

The switchover works by providing a DL6 signal to A11 and A13 for uppercase and a logic 1 for screen reversal. If we provide DL6, we get lowercase since A11 forces the lowercase ASCII bit 6 output, and A13 inhibits screen reversal. If we provide a logic 1, lowercase is inhibited and reversal is allowed when it is called for.

You put the switch in the reverse position for programs that need reverse video continuously displayed. You put the switch in the lowercase position when you must display lowercase.

Your turn: The character generator in module A also will display CTRL characters if you make DL5 and ASCII bit 6 both zeros. When would you want to display control characters? How can you do this? Can you eliminate the changeover switch and replace it with a series of software flags that gives you everything at once—reversal, full case blinking, lowercase, CTRL displayed on command and invisibility on existing software?

Note that you can also use other character generators by suitably changing the pins around. There's also a lowercase 2513 you can piggyback onto the exisiting uppercase one.

You can also use your own character generator by burning your own 2716 EPROM. The advantages of the EPROM are that you can get any character and lots of graphics symbols that you like on a hardware basis. For instance, instead of the awkward treatment of the descenders on the lowercase g, p, and so on, you could use 5×7 uppercase for caps and 5×5 uppercase for lowercase. This can be both legible and attractive.

There is one limitation to the 2716 when you use it with an only slightly modified Apple II. With the Apple II, only five output lines are used, with the remaining three being permanent blanks. Unless you rework the output video, your 2716 would be more suited to new characters than to graphics symbols that have to butt against each other.■

Apple II conversion kits, TVT 6-5/8 Module As, *Cheap Video Cookbooks* and other cheap video stuff are available from:
PAIA Electronics
1020 West Whilshire
Box 14359
Oklahoma City OK 73114

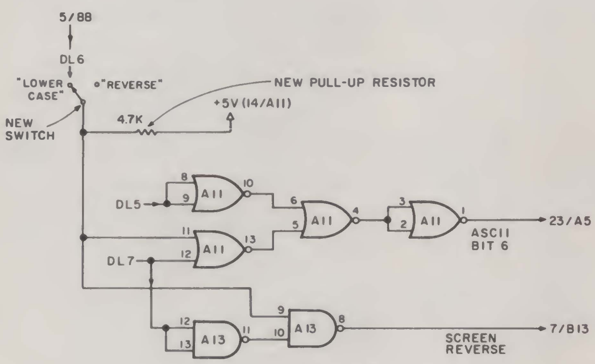

Fig. 9. A changeover switch and pullup resistor may be added to give an option of lowercase or reverse video displays.