PRESHIFT-TABLE GRAPHICS ON YOUR APPLE

This fast assembly-language routine allows you to move rectangular images quickly to any point on the Apple high-resolution graphics screen

BY BILL BUDGE, WITH GREGG WILLIAMS AND ROB MOORE

[Editor's note: *This article was a collaborative effort. Bill Budge shared his implementation of preshift-table graphics with us and helped Rob Moore and me with the article itself. Rob wrote the assembly-language subroutine, and I wrote the article text and* BASIC *program. . . .*G.W.]

In microcomputer graphics, speed is always of the essence—we are always trying to get the same processor to do more work than it's done before. In creating Mousepaint (a drawing program that uses icons, windows, and Apple's new mouse), Bill Budge devised a technique that simplifies moving rectangular blocks of pixels on the Apple II high-resolution graphics screen. The cost is a slight loss in speed and an overhead of about 3K bytes of memory, but the versatility of this routine certainly justifies the expense.

Expressed simply, Budge's method (called *preshift-table lookup*) uses a short assembly-language routine to access a large table that lists all the possible shifted results for all possible byte values. This saves time by exchanging a certain amount of calculation, looping, and shifting with a single table lookup, a logical OR operation, and some occasional set-up instructions.

APPLE II HI-RES GRAPHICS

The Apple high-resolution graphics page is organized as 192 lines of 280 dots each; you will have this resolution available to you if you view the page on a monochrome display. If you use a color display, certain two-dot patterns will appear as a single color dot; this means that your effective resolution for color graphics is 140 by 192 dots. The high-resolution page can display six colors: black, white, violet, green, orange, and blue. Each line of graphics occupies 40 bytes in memory; given 280 dots per line, this means that each byte converts to 280/40 (or 7) dots per byte.

The fact that each byte of memory translates to 7 (not 8) dots—3½ if you're talking about high-resolution color—is one of the many subtleties that characterize Apple graphics. Speaking in terms of the color display of dots within a byte, you can display black, white, and one of two sets of two colors each—green/violet or orange/blue. The computer interprets the most significant bit of a byte (the only one that doesn't translate to a dot) as a color bit; when this bit is off, you can get green/violet, and when it is on, you can get orange/blue. Two adjacent bits on—anywhere in the byte—make a white dot appear. Two adjacent bits off make a black dot. A

(continued)

Bill Budge is well-known for his graphics work on the Apple II; *his best-known products are* Raster Blaster *(the first pinball game of its caliber),* Pinball Construction Kit, *and* MousePaint *(supplied with the* Apple II *mouse). Gregg Williams is a senior technical editor at* BYTE. *Rob Moore is a hardware designer and a frequent contributor to* BYTE. *They can be reached at* POB 372, *Hancock,* NH 03449.

single even bit on (with both adjacent bits off) makes a violet or blue dot, depending on the color bit. Similarly, a single odd bit on makes a green or orange dot. See figure 1 for details. (Because the even/odd position of a bit within a byte determines its color, images to be viewed on a color display must move an even number of bits at a time. If they don't, they will alternate between the two color sets every move.)

One final detail: if you consider the byte as a binary number, you must strip off the most significant bit and reverse the order of the remaining bits before you put them on the high-resolution screen—in other words, the rightmost bit in the byte becomes the leftmost dot on screen, and vice versa. Figure 2 shows this relationship. In this article, we will be shifting rectangular blocks of dots to the right. Because of the above relationship, this means we will be shifting bits to the *left*. Keep the following sentence in mind: *to shift dots right, shift bits left.*

Finally, we get to the overall makeup of the high-resolution screen. With seven dots per byte, 40 consecutive bytes become 140 color dots (or 280 monochrome dots). But do the next 40 bytes make up the next row of dots? Unfortunately, no—they are the 65th row. As you can see from figure 3, consecutive rows of dots are not 40 bytes apart. The scheme is much more complex than figure 3 shows, but that is of little interest to most programmers. Because speed is of the essence in graphics work, most graphics routines look up the address of the first byte in a row from a table of 192 values instead of having the computer calculate it while it is doing the graphics. Once we create that table, as the program in listing 1 does, the complexity of the high-resolution screen layout becomes irrelevant.

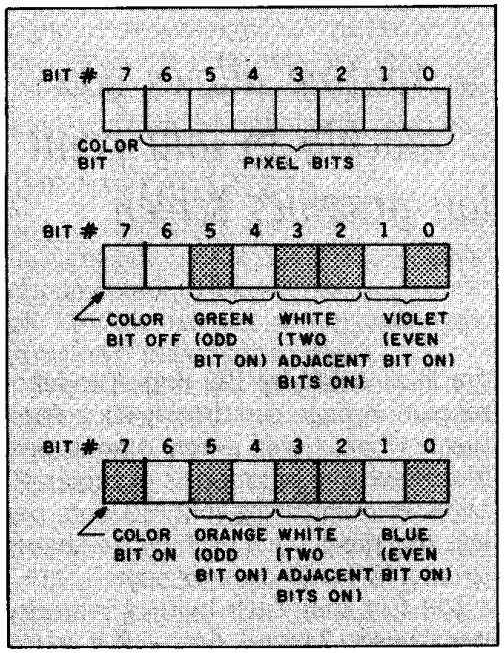

Figure 1: *Translating memory bits to graphics on the Apple high-resolution graphics page. Only 7 of the 8 bits in a byte become dots on the high-resolution screen. Bit 7 determines what colors can appear in that byte.*

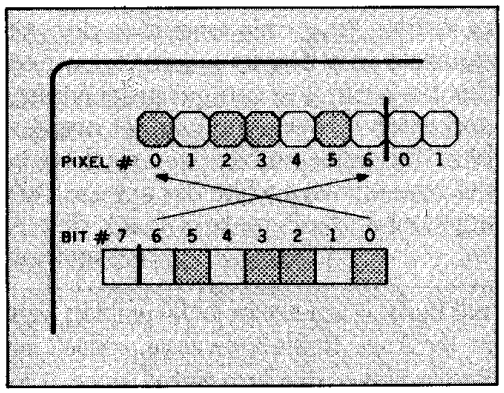

Figure 2: *The relationship between bits and pixels. The Apple II reverses the seven low-order bits of a byte before displaying them on the high-resolution screen—that is, it displays the bits right to left.*

PRESHIFT-TABLE LOOKUP VS. PRESHIFTED SHAPES

Most people who do high-performance work with Apple II graphics know about *preshifted shapes*, a graphics method that stores seven versions of a given image and quickly looks up the appropriate one. (For more details, see the text box "What Are Preshifted Shapes?" on page A26.) Preshifted shapes have one main use: to allow rapid animation of small shapes that move only several dots at a time. Budge's needs were different: he had several windows that he had to be able to move quickly. Such windows are too large to have seven versions of (or modify seven versions of). In addition, their expected movements were large jumps across the screen, not continuous movement.

Shifting an image does not change it visibly, but it does change how that image is represented in byte-sized pieces. For example, the simple two-byte image in figure 4a changes significantly when shifted right three dots—what's more, we now need another byte to store it in! Notice that some dots stay in the same seven-dot group (these are called "shiftstay" dots) and that others move

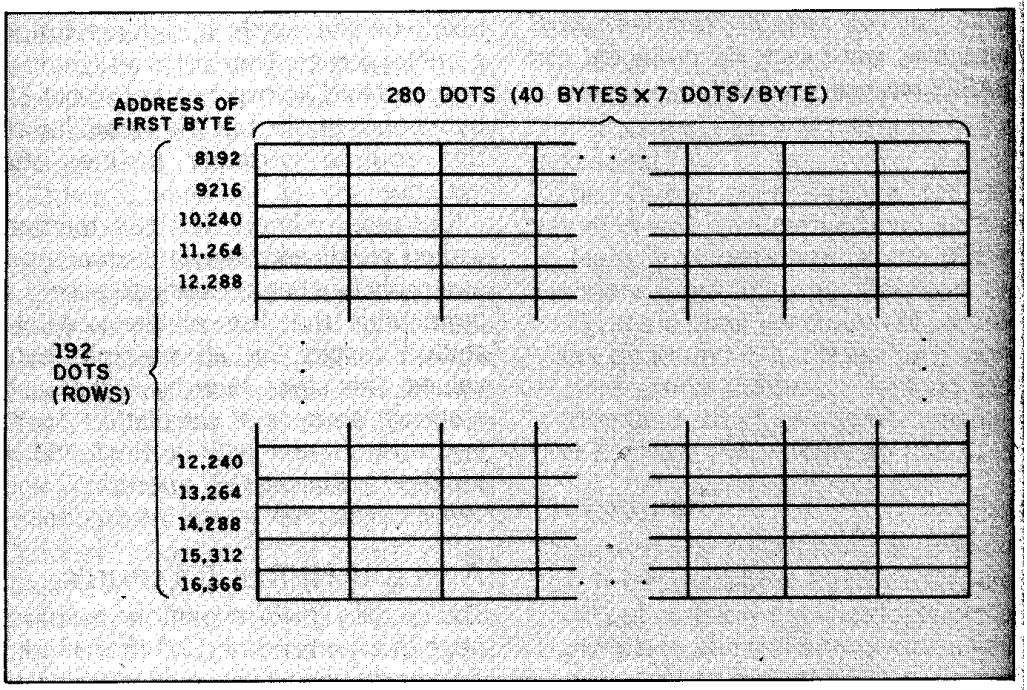

Figure 3: *The Apple II high-resolution graphics page. Each line of dots is given by a contiguous 40-byte area in memory. Successive lines are usually separated by 1024 bytes; every eighth line (lines 8, 16, and so on) has an address 7040 bytes less than the start of the previous line.*

to the next seven-dot group to the right (these are the "shiftout" dots). If you understand this figure and remember that the *bits* corresponding to these dots are stored in reverse within each seven-dot group (byte), then you will understand the essence of this algorithm: to move a single-line image to the right, we split each byte and combine the shift-stay bits of one byte with the shiftout bits of the previous byte. Figure 4a shows how simple it is to shift an image three dots to the right. Unfortunately, when we start looking at manipulating the bits themselves (which, within a byte, are reversed from the way they are displayed on the screen), things get more complicated (see figure 4b). Here is one recipe for manually making the shift (notice that we start with the rightmost byte and work our way left):

1. Shift the second source byte left three bits and put the three overflow bits into the *lower half* of the third destination byte. Hold the right four bits (which have been shifted left three bits) somewhere.
2. Shift the first source byte left three bits. The three overflow bits and the four leftover bits from step 1 join like two adjacent pieces of a jigsaw puzzle. "Fit" them together (using a logical OR instruction) and save them in the second destination byte.
3. Place the remaining bits of the first source byte in the first destination byte.

The 6502 microprocessor inside the Apple can shift only one bit at a time. Therefore, we would have to write a loop of assembly-language code every time we wanted to shift *n* bits. Such a routine would be very slow and would be slower as *n* gets larger.

Enter the Preshift Tables

Bit shifts take too long to calculate, so why don't we precalculate everything and do a simple table lookup instead? We'll need two tables: one for the bits that are shifted out of the byte (which I'll call the *shiftout* value) and another for the bits that remain in the byte, but in their new, shifted position (the *shiftstay* value). A byte has 256 possible values, so each table will take up 256 bytes, for a total of 512 bytes for both tables. We will need these tables for shifts of one through six bits; this takes up a total of 3K bytes of table space, and this overhead is constant whether you have one image to move or a hundred.

(For this article, we've added shiftout and shiftstay tables for a shift of zero bits. This adds an extra 0.5K bytes of tables but greatly simplifies the assembly-language routine needed.)

Figure 5 shows how the shiftout and shiftstay values are created for a given byte, and listings 1 and 2 create the required preshift tables. The tables deal with shifting dots *right*, so the bits in a byte are shifted *left* (remember that bits reverse their positions when they become dots on the high-resolution screen—see figure 2).

The middle line shows the same byte, binary value 11101010, as it is shifted two, three, and four dots (bits). The line above it shows the bits that go into the shiftout byte and the line below it, the

(continued)

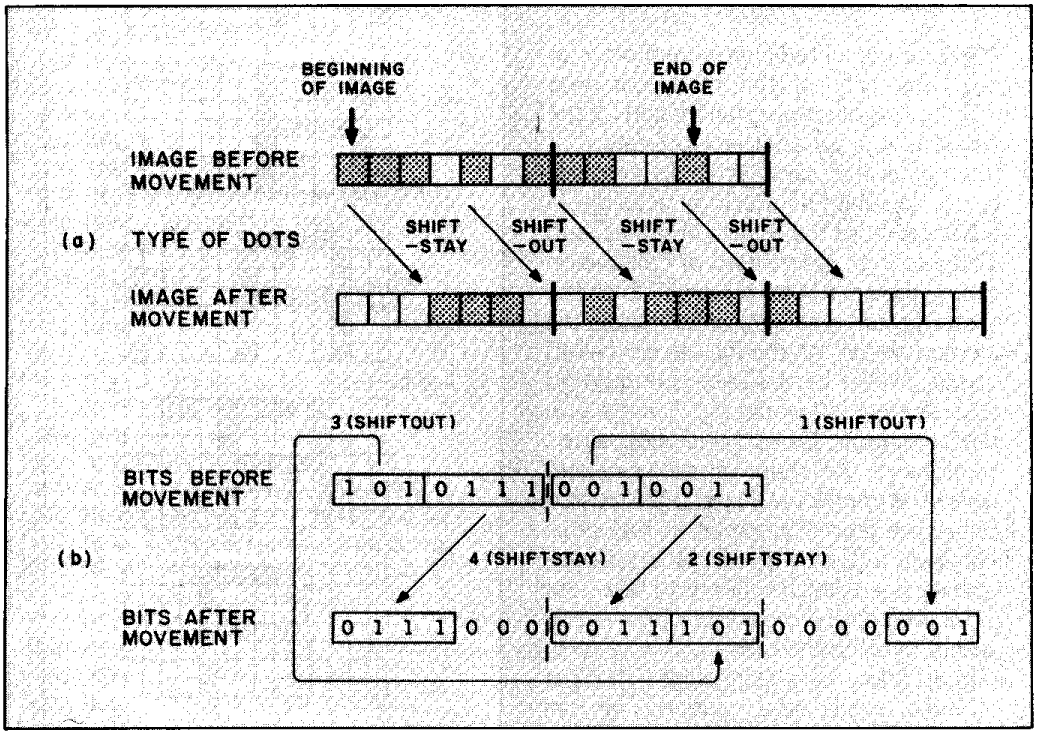

Figure 4: *Moving an image. When a two-byte image is moved three dots to the right (figure 4a), it occupies an extra byte, and the values of the bytes all change in a complex way. Figure 4b shows the bit patterns underlying the original and shifted images (for simplicity, we are ignoring the color bits in each byte). To shift the image right 3 bits, we shift each byte to the left 3 bits and rearrange the pieces in the order shown.*

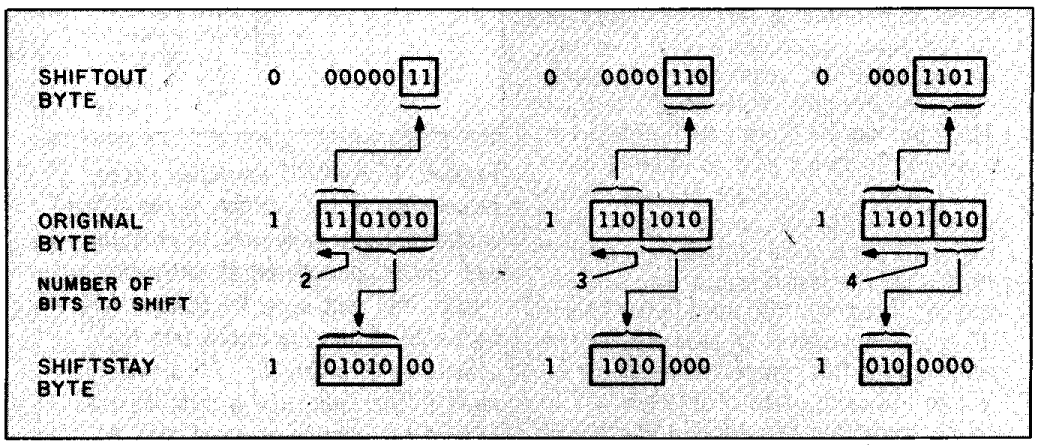

Figure 5: *Creating shiftout bytes (top) and shiftstay bytes (bottom) from an arbitrary byte (center). This figure shows the same byte being shifted two, three, and four bits. Note that the color bit (the leftmost one, outside the boxes) remains with the shiftstay byte.*

shiftstay byte. Notice that the most significant (color) bit, which is not included in the shift, stays with the shiftstay byte. The program in listing 2 makes tables of the shiftout and shiftstay value for each byte (values from 0 to 255) and for each possible value for number of bits shifted (zero to six).

What Are Preshifted Shapes?

The preshifted-shape method is at the root of the most common forms of graphics animation for the Apple II. It is yet another application of the maxim that says, "To make graphics faster, calculate as much as possible in advance, store the results in tables, and do table lookups instead of calculation during the animation process." To shift a shape several dots right or left during the animation (or "on-the-fly," as we call it) would involve lots of byte splitting and recombining and would significantly slow the animation. Instead, we keep seven versions of the shape in memory and select the appropriate one to be "pasted onto" the graphics screen in real time. (We want to deal with whole bytes only, and we need seven versions to take care of all the possible positions of the shape within byte boundaries.)

In addition, by making slight changes to each of the seven shapes, we can achieve "internal animation"—animation of the shape itself as it moves horizontally—at no extra cost. Figure 1 shows a monochrome shape with internal animation in its seven preshifted versions.

Preshifted shapes are the latest word in fast animation. Unfortunately, they take up a lot of room, and it is often inconvenient to maintain a large inventory of such shapes, especially when they are subject to periodic revisions. (Utility packages like Penguin Software's Graphics Magician can help with such tasks.) For example, an image 10 dots square (which is actually 5 dots square in color) must be shifted within an image area 10 lines high by 3 bytes wide (a 10-dot image that begins on the last dot in a byte ends in the second dot of the third byte.) This is 30 bytes per version, or 210 bytes for the entire preshifted shape table—all for an image about the size of a fingernail! In some situations, of course, the method described in this article is a space-saving alternative.

—*Gregg Williams*

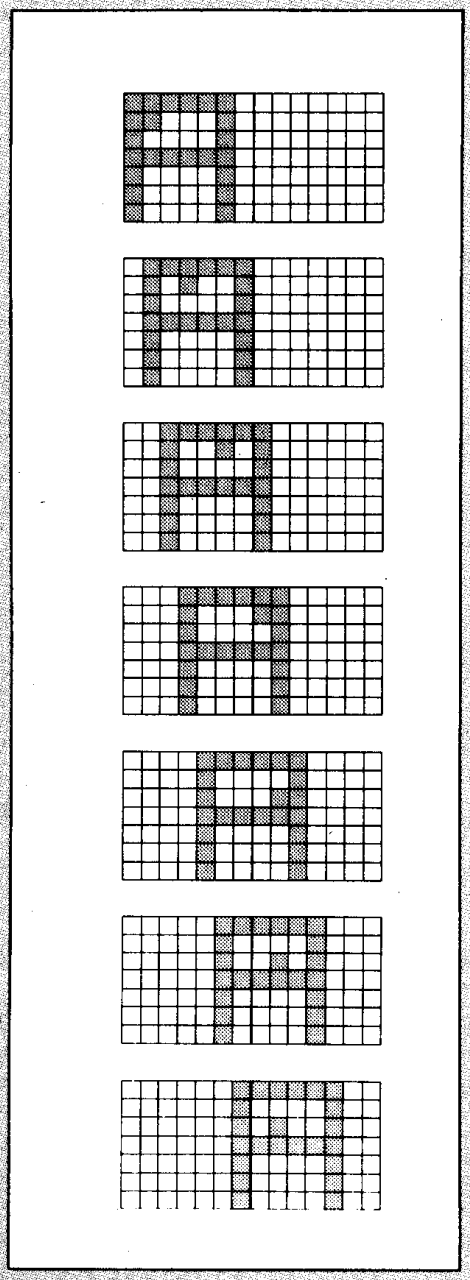

Figure 1: *Apple II preshifted shapes. Depending on the location of the shape within a 2-byte-wide area, a program can "paste" one of the above seven images into that area. In addition, when the seven shapes are drawn into the same area in order, the "A" appears to move to the right, and a little dot rolls around the interior cavity of the "A." This effect is called internal animation.*

The problem of moving a rectangular image is reduced to that of moving a single line of the image, which in turn is reduced to that of moving a single byte of the image between one and six dots (a move of, say, three dots and a move of ten are essentially the same because the extra seven dots—one byte—can be taken care of by adding one to the destination-byte pointer).

Preshifting an Image Line

You may have noticed that the shiftout values are right-justified and the shiftstay values are left-justified within the low 7 bits of the byte. This allows the computer to combine the two image fragments correctly. Figure 6 shows how the core routine (only 23 bytes) accomplishes the task. In particular, figure 6 shows how the shiftstay bits from one byte and the shiftout bits from the byte before it create a new destination byte.

Bill Budge did not get this code density without pulling some tricks. In line 1, the address $*mmmm* is the first byte of the one video line of image that this routine manipulates. The address $*nn*00 is the first byte of the shiftout table being used, and $*pp*00 is the first byte of the shiftstay table. (Remember that the tables must begin on page boundaries—this explains the double zeros in both addresses.) A driver program (discussed below) must change the first address for each new line of image to be processed and must change the last two addresses only at the beginning of a new shift-rectangular-image operation. Yes, this is self-modifying code. It works, and it is necessary to get the speed Budge wanted out of this routine.

The calling program must supply this routine with two other values: the *y* register must contain the width of the image in bytes, and the accumulator (A) must contain zero since there is no previous byte that has left shiftstay bits behind.

Preshifting a Rectangular Image

Given the appropriate shiftout and shiftstay tables (which allow you to split an arbitrary byte by doing two table lookups instead of *n* shift operations), the routine in figure 6 will shift one line of dots and put the result in a buffer area. To make this routine useful, you must surround it with a driver routine that

shifts each line of the rectangular image and does the necessary housekeeping. Many such routines are possible depending on the implementation needed. To illustrate this routine, let's assume that the program has access to an area of memory with image data as specified in table 1. For each line of the image, the driver routine will modify the addresses within the inner loop to point to the right areas, shift a line of the image to the buffer, then transfer the buffer to the correct screen position one byte at a time (see figure 7).

Listing 3 is the final assembly-language subroutine for shifting a block image via the preshift tables and moving it to the screen as described above. The equate statements at the top of the listing show the positions of needed tables and variables. Two tables not yet discussed are the XDIV7 and XMOD7 tables. Because the Apple stores 7 dots per byte, the algorithm needs to divide single-byte numbers by 7 and use either the quotient or remainder. At the expense of 512 bytes (256 bytes per table), we can calculate either quantity as quickly as an indexed load instruction: the *n*th bytes of the XDIV7 and XMOD7 tables are the integer quotient and remainder, respectively, of *n*/7. Listing 4 creates these tables for later use and saves them under the name DIV7 TABLE.

The BASIC program of listing 5 loads the preshift, DIV7, and HIRES1 tables, along with the table for the image to be moved into a single binary file called TABLPAK. The image that the demonstration program is going to move, an arrow pointing diagonally up, takes the form given in table 1 and can be any size. You can change it to any image you wish, as long as the table starts at address 20864 (5180 hexadecimal). When translating a picture of the image to hexadecimal values, remember that the bottom 7 bits of a byte are displayed reversed from the way they are stored; for example, to illuminate the rightmost dot of a seven-dot byte, the byte needed is a 64 (binary 01000000) or a 192 (binary 11000000), not 1 (binary 00000001).

The Demo Program

The BASIC program of listing 6 loads in the TABLPAK package of tables and the

(*text continued on page* A132)

Listing 1: *The* HIRES1 *program creates the binary file* HIRES1 TABLE. *The first* 192 *bytes of the file contain the low bytes of the starting addresses of line n of high-resolution page* 1; *the second* 192 *bytes of the file contain the high bytes of the same addresses.*

```
]LIST

100 REM
110 REM  HIRES1 PROGRAM
120 REM
130 REM  CREATES TABLE OF ADDRESSES
140 REM  OF FIRST BYTE IN EACH LINE
150 REM  OF HI-RES PAGE 1
160 REM
170 REM  BY GREGG WILLIAMS
180 REM  22 APR 84
190 REM
195 REM  ------------------------------------------------
197 REM
200 T2BLBGN = 16384
210 REM --BEGINNING OF TABLE AS
220 REM --STORED IN MEMORY
230 REM
235 TBLWDTH = 192
236 REM --WIDTH OF EACH TABLE
237 REM
240 HI1BGN = 8192
250 REM --ADDRESS OF BEGINNING
260 REM --OF HIRES PAGE 1
263 REM
266 ADDR = T2BLBGN
269 REM --THIS PGM CALCULATES
272 REM --ADDRESSES IN ASCENDING
275 REM --ORDER; ADDR HOLDS THE
277 REM --ADDRESS OF THE NEXT
278 REM --TABLE ELEMENT TO BE
279 REM --FILLED
280 REM  ------------------------------------------------
290 REM
300 REM       MAIN LOOP OF PGM
310 REM
313 HOME : PRINT "CREATING HIRES1 TABLE";
320 FOR I = 0 TO 2
325 : FOR J = 0 TO 7
330 :: FOR K = 0 TO 7
340 :::VL = HI1BGN + 40 * I + 128 * J + 1024 * K
353 :::VHI = INT (VL / 256)
356 :::V2LOW = VL - 256 * VHI
359 ::: POKE ADDR,V2LOW
362 ::: POKE ADDR + TBLWDTH,VHI
366 :::ADDR = ADDR + 1
370 :: NEXT K
375 :: PRINT ".";
380 : NEXT J
384 NEXT I
385 PRINT
386 REM
388 REM --ABOVE ALGORITHM
390 REM --DERIVED FROM TABLES
392 REM --IN APPLE REFERENCE
393 REM --MANUAL
394 REM
400 REM  ------------------------------------------------
410 REM
420 REM --SAVE FILE TO DISK
430 REM
440 PRINT  CHR$ (4);"BSAVE HIRES1 TABLE,A";T2BLBGN;",L384"
450 REM
455 PRINT : PRINT "TABLE SAVED TO DISK"
460 END
```

(*listings continued on page* A28)

Listing 2: *The Preshift program creates fourteen 256-byte tables, which are saved as the binary file* PRESHIFT TABLE. *The first seven are the shiftout tables for 0 through 6 dots, while the last seven are the corresponding shiftstay tables.*

```
]LIST

100 REM
110 REM  CREATE-PRESHIFT-
120 REM  TABLES PROGRAM
125 REM
130 REM  CREATES SHIFTOUT AND
135 REM  SHIFTSTAY TABLES FOR
140 REM  0 THROUGH 6 DOTS
145 REM
150 REM  BY GREGG WILLIAMS,
155 REM  17 APR 84
157 REM
160 REM  ------------------------------------------------
170 REM
180 REM  INITIALIZATION OF
190 REM  CONSTANTS
200 REM
210 BGNTBL = 30720
220 REM --ADDRESS OF START OF
230 REM --TABLE; MUST BE EVENLY
240 REM --DIVISIBLE BY 256
250 REM
260 BININCR = 7 * 256
270 REM --DISTANCE FROM BEGINNING
280 REM --OF SHIFTOUT (SOUT)
283 REM --TO SHIFTSTAY(SSTAY)
286 REM --TABLES
290 REM
300 TBLWDTH = 256
305 REM --DISTANCE BETWEEN
310 REM --ANY TWO TABLES
315 REM
320 C1ZMAXSHF = 6
340 C2MAXBYTEVL = 255
360 REM
380 REM  ------------------------------------------------
400 REM
420 REM  MAIN LOOP
440 REM
450 HOME : PRINT "CREATING PRESHIFT TABLE (THIS WILL TAKE
    SEVERAL MINUTES) ";
460 FOR SHF = 0 TO C1MAXSHF
480 :CURRTBL = BGNTBL + SHF * TBLWDTH
500 : FOR BYTE = 0 TO C2MAXBYTEVL
520 :: GOSUB 1000
540 REM --CALCULATE SHIFTOUT
560 REM --& SHIFTSTAY VALUES
580 REM --FROM SHF AND BYTE
620 REM
640 :: POKE CURRTBL + BYTE,SSTAY
660 :: POKE CURRTBL + BYTE + BIGINCR,SOUT
680 REM --STORE VALUES IN
700 REM --TABLES
720 REM
725 :: PRINT ".";
740 : NEXT BYTE
760 NEXT SHF
770 PRINT
775 REM
780 REM  ------------------------------------------------
800 REM
820 REM  SAVE TABLES AS ONE
840 REM  LARGE DISK FILE
860 REM
880 PRINT  CHR$ (4);"BSAVE PRESHIFT TABLE,A";BGNTBL;",L3584"
890 PRINT : PRINT "TABLE SAVED TO DISK"
900 REM
910 REM  END OF PROGRAM
915 REM
920 END
960 REM  ------------------------------------------------
980 REM
985 REM --SUBROUTINE TO SPLIT
987 REM --AN ARBITRARY 8-BIT
989 REM --VALUE INTO ITS
991 REM --COMPONENTS; SEE
993 REM --TEXT FOR DETAILS.
995 REM
1000 IF SHF = 0 THEN SOUT = 0:SSTAY = BYTE: GOTO 1380

1005 REM
1010 REM --THE FOLLOWING CODE IS
1012 REM --DONE IFF SHF>0
1015 C3SCALEOUT = 2 ^ (7 - SHF)
1020 C4SCALNW = 2 ^ SHF
1030 BSVE = BYTE
1040 REM
1060 IF BYTE > = 128 THEN SIGNBIT = 1:BYTE = BYTE - 128: GOTO
     1140
1080 REM  (ELSE IF BYTE<128)
1100 SIGNBIT = 0
1110 REM  (AND BYTE UNCHANGED
1120 REM
1140 SOUT = INT (BYTE / C3SCALEOUT)
1160 REM --FIND SHIFT-OUT BITS
1180 REM --BY CUTTING OFF RIGHT-
1200 REM --MOST (7-SHF) BITS
1210 REM
1220 SSTAY = 128 * SIGNBIT + (BYTE - SOUT * C3SCALEOUT) *
     C4SCALNW
1240 REM
1260 BYTE = BSVE
1280 REM --RESTORE ORIGINAL
1300 REM --VALUE OF BYTE
1320 REM
1340 REM  END SUBROUTINE
1360 REM
1380 RETURN
```

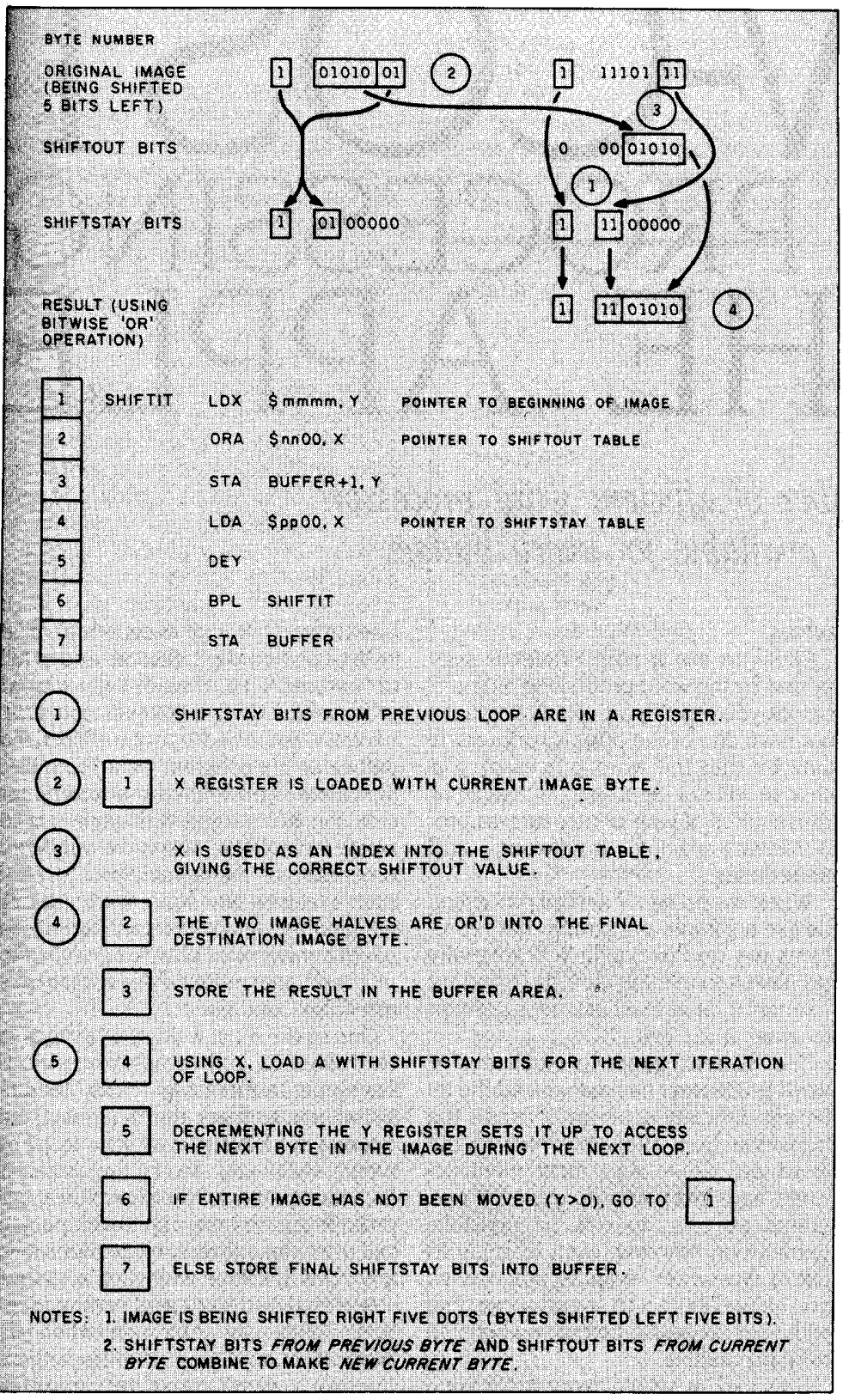

Figure 6: *The basic preshift-table lookup routine. The top area shows how the code creates a single result byte, the middle area is the actual code, and the bottom area is a commentary. The numbers in the circles and squares relate events to lines of code. Hexadecimal mmmm points to the first byte of the image. The hexadecimal addresses nn00 and pp00 point to the proper shiftout and shiftstay tables, respectively. At the beginning of the routine, the accumulator contains zero and the y register contains the number of bytes in the line.*

(listings continued on page A127)

(continued from page A29)

Listing 3: *The Shiftline routine. This 6502 assembly-language routine shifts one line of a video image as described in figure 7. We used the Apple DOS Tool Kit assembler to create the object-code file, which is named* SHIFTLINE ROUTINE.OBJ0.

```
SOURCE FILE: SHIFTLINE ROUTINE
------- NEXT OBJECT FILE NAME IS SHIFTLINE ROUTINE.OBJ0
5210:                1          ORG  $5210
5210:                2 ;
5210:                3 ;                    PRESHIFT GRAPHICS ROUTINE
5210:                4 ;                    BY ROB MOORE & BILL BUDGE
5210:                5 ;

5210:                7 ;
0040:                8 BASE1    EQU  $40          ; BASE PAGE ROW ADDR POINTER
5210:                9 ;
5210:               10 ; LOOKUP TABLES
5210:               11 ;
4000:               12 SHFTSTAY EQU $4000         ; SHIFTOUT TABLES
4700:               13 SHIFTOUT EQU $4700         ; SHIFT TABLES
4E00:               14 XDIV7    EQU  $4E00        ; INDEX DIV 7
4F00:               15 XMOD7    EQU  $4F00        ; INDEX MOD 7
5000:               16 ROWTBL   EQU  $5000        ; SCREEN ROW ADDR LO-BYTES
50C0:               17 ROWTBH   EDU  $50C0        ; SCREEN ROW ADDR HI-BYTES
5210:               18 ;
5210:               19 ; IMAGE DEFINITION PARAMETERS — SET BY USER
5210:               20 ;
5210:               21 IROWS    EQU  $5200        ; # OF ROWS — 1
5210:               22 IDOTS    EQU  $5201        ; DOT WIDTH -1
5202:               23 IBWIDTH  EQU  $5202        ; IMAGE BYTE WIDTH
5203:               24 IBITS    EQU  $5203        ; ADDR OF IMAGE DATA
5205:               25 X1       EQU  $5205        ; IMAGE LEFT X-COORD
5206:               26 Y1       EQU  $5206        ; IMAGE TOP Y-COORD
5210:               27 ;
5210:               28 ;
5210:               29 ; FIRST, SET UP THE VARIOUS PARAMETERS TO PREPARE
5210:               30 ; FOR THE IMAGE DRAW.
5210:               31 ;
5210:8E E9 52       32 DRAWIMAGE STX XSAVE        ; SAVE BASIC X-REG
5213:AD 06 52       33           LDA  Y1          ; IMAGE TOP ROW #
5216:18             34           CLC
5217:6D 00 52       35           ADC  IROWS       ;  + # OF ROWS -1
521A:8D E1 52       36           STA  Y2          ;  = BOTTOM ROW #
521D:               37 ;
521D:AE 05 52       38           LDX  X1          ; IMAGE LEFT X-COORD
5220:BD 00 4E       39           LDA  XDIV7,X     ;  DIVIDED BY 7
5223:8D E2 52       40           STA  LBYTE       ; = IMAGE LEFT BYTE #
5226:               41 ;
5226:BC 00 4F       42           LDY  XMOD7,X     ; IMAGE LEFT BIT #
5229:B9 DA 52       43           LDA  LMASKS,Y    ; INDEXES LMASK TABLE FOR
522C:8D E3 52       44           STA  LMASK       ; IMAGE LEFT BIT MASK
522F:               45 ;
522F:98             46           TYA              ; IMAGE LEFT BIT #
5230:18             47           CLC              ; + SHIFT TABLES ADDRESS
5231:69 40          48           ADC  #<SHFTSTAY  ; * 256
5233:8D 8A 52       49           STA  PATCH3      ; TO SHIFT PATCH
5236:               50 ;
5236:69 07          51           ADC  #7          ; OFFSET TO SHIFTOUT TABLES
5238:8D 84 52       52           STA  PATCH2      ; SETS UP SHIFTOUT PATCH
523B:               53 ;
523B:98             54           TYA              ; IMAGE LEFT BIT #
523C:18             55           CLC
523D:6D 01 52       56           ADC  IDOTS       ;  + IMAGE DOT WIDTH
5240:AA             57           TAX
5241:BD 00 4E       58           LDA  XDIV7,X     ;  DIVIDED BY 7
5244:8D E5 52       59           STA  SBWIDTH     ; = SHIFTED DATA BYTE WIDTH
5247:               60 ;
5247:BC 00 4F       61           LDY  XMOD7,X     ; RIGHT EDGE BIT #
```

(continued)

```
524A:B9 D3 52   62          LDA  RMASKS,Y    : INDEX RMASKS TABLE FOR
524D:8D E4 52   63          STA  RMASK       ; IMAGE RIGHT BIT MASK.
5250:           64 ;
5250:AC 02 52   65          LDY  IBWIDTH     ; IMAGE WIDTH IN BYTES
5253:8C E6 52   66          STY  PITCH       ; IS BITMAP PITCH
5256:88         67          DEY              ; SUBTRACT 1 TO GET
5257:8C E7 52   68          STY  SHFTINDX    ; SHIFT START INDEX
525A:           69 ;
525A:AD 03 52   70          LDA  IBITS       ; COPY IMAGE DATA ADDRESS
525D:8D 80 52   71          STA  PATCH1      ; TO SHIFT ROUTINE PATCH1
5260:AD 04 52   72          LDA  IBITS+1
5263:8D 81 52   73          STA  PATCH1+1
5266:           74 ;
5266:           75 ; MAIN DRAW LOOP STARTS HERE
5266:           76 ;
5266:AE 06 52   77          LDX  Y1          ; IMAGE TOP SCRN ROW #
5269:BD 00 50   78 DRAW     LDA  ROWTBL,X    ; SCREEN ROW ADDR LO-BYTE
526C:18         79          CLC
526D:6D E2 52   80          ADC  LBYTE       ; + LEFT IMAGE BYTE #
5270:85 40      81          STA  BASE1       ; TO SCRN ADDR POINTER
5272:BD C0 50   82          LDA  ROWTBH,X    ; SCREEN ROW ADDR HI-BYTE
5275:85 41      83          STA  BASE1+1     ; TO POINTER HI-BYTE
5277:8E E8 52   84          STX  TEMP        ; SAVE CURRENT ROW #
527A:           85 ;
527A:           86 ; BILL BUDGE'S SHIFT CODE STARTS HERE.
527A:           87 ; IT SHIFTS A ROW OF STORED BITMAP DATA INTO A
527A:           88 ; SINGLE-LINE BUFFER, WHICH CAN THEN BE DRAWN TO
527A:           89 ;        THE SCREEN
527A:           90 ;
527A:A9 00      91          LDA  #0          ; SETUP FOR SHIFTING
527C:AC E7 52   92          LDY  SHFTINDX
527F:           93 ;
5280:           94 PATCH1   EQU  *+1
527F:BE FF FF   95 SHIFTIT  LDX  $FFFF,Y     ; GET A BYTE OF IMAGE DATA
5284:           96 PATCH2   EQU  *+2
5282:1D 00 47   97 ORA      SHIFTOUT,X       ; LOOK UP SHIFTED OUT PART
5285:99 EB 52   98 STA      BUFFER+1,Y       ; AND STORE IN BUFFER
528A:           99 PATCH3   EQU  *+2
5288:BD 00 40  100          LDA  SHFTSTAY,X  ; LOAD THE SHIFTED PART
528B:88        101          DEY  ; DONE WITH ROW YET?
528C:10 F1     102          BPL  SHIFTIT     ; LOOP BACK IF NOT
528E:8D EA 52  103          STA  BUFFER      ; STORE LAST SHIFTED PART
5291:          104 ;
5291:          105 ; ACTUAL DRAW CODE STARTS HERE. NOTE THAT THE
5291:          106 ; RIGHTHAND AND LEFTHAND EDGE BYTES ARE TREATED
5291:          107 ; DIFFERENTLY BECAUSE THEY MAY BE PARTIAL-BYTE
5291:          108 ; WRITE OPERATIONS.
5291:          109 ;
5291:AC E5 52  110          LDY  SBWIDTH     ; SHIFTED DATA BYTE WIDTH
5294:AE E8 52  111          LDX  TEMP        ; GET BACK THE ROW #
5297:          112 ;
5297:          113 ; DO THE RIGHTHAND IMAGE BYTE
5297:          114 ;
5297:B1 40     115          LDA  (BASE1),Y   ; GET A SCREEN BYTE
5299:59 EA 52  116          EOR  BUFFER,Y    ; XOR WITH IMAGE DATA
529C:2D E4 52  117          AND  RMASK       ; MASK UNWRITTEN AREA
529F:51 40     118          EOR  (BASE1),Y   ; RESTORE SCRN AND IMAGE
52A1:4C A7 52  119          JMP  DRAW2
52A4:          120 ;
52A4:          121 ; FAST LOOP TO DRAW IN-BETWEEN BYTES
52A4:          122 ;
52A4:B9 EA 52  123 DRAW1    LDA  BUFFER,Y    ; GET AN IMAGE BYTE
52A7:91 40     124 DRAW2    STA  (BASE1),Y   ; WRITE BYTE TO SCREEN
52A9:88        125          DEY              ; DECREMENT COUNT
52AA:D0 F8     126          BNE  DRAW1       ; LOOP BACK IF NOT DONE
52AC:          127 ;
52AC:          128 ; FINISH UP WITH LEFTHAND BYTE
52AC:          129 ;
52AC:B1 40     130 DRAW3    LDA  (BASE1),Y   ; GET A SCREEN BYTE
52AE:59 EA 52  131          EOR  BUFFER,Y    ; XOR WITH IMAGE BYTE
52B1:2D E3 52  132          AND  LMASK       : MASK UNWANTED PART
```

(continued)

```
52B4:51 40      133          EOR  (BASE1),Y     ; RESTORE SCRN AND IMAGE
52B6:91 40      134          STA  (BASE1),Y     ; AND WRITE TO SCREEN
52B8:           135 ;
52B8:           136 ; TEST FOR LAST ROW COMPLETE
52B8:           137 ;
52B8:EC E1 52   138          CPX  Y2            ; WAS LAST ROW DONE
52BB:F0 12      139          BEQ  EXIT          ; QUIT IF SO.
52BD:           140 ;
52BD:E8         141          INX                ; MOVE TO NEXT ROW
52BE:18         142          CLC                ; ADD THE BITMAP PITCH
52BF:AD 80 52   143          LDA  PATCH1        ; TO THE SHIFT ROUTINE
52C2:6D E6 52   144          ADC  PITCH         ; POINTER TO MOVE TO
52C5:8D 80 52   145          STA  PATCH1        ; THE NEXT BITMAP ROW.
52C8:90 9F      146          BCC  DRAW          ; LOOP AGAIN IF NO CARRY
52CA:EE 81 52   147          INC  PATCH1+1
52CD:B0 9A      148          BCS  DRAW          ; LOOP AGAIN, ALWAYS
52CF:AE E9 52   149 EXIT     LDX  XSAVE
52D2:60         150          RTS                ; EXIT HERE
52D3:           151 ;
52D3:           152 ; RIGHT AND LEFT BIT-MASKS TABLES
52D3:           153 ;
52D3:01 03 07   154 RMASKS   DFB  $01,$03,$07,$0F,$1F,$3F,$7F
52D6:0F 1F 3F
52D9:7F
52DA:7F 7E 7C   155 LMASKS   DFB  $7F,$7E,$7C,$78,$70,$60,$40
52DD:78 70 50
52E0:40
52E1:           156 ;
52E1:           157 ; PROGRAM VARIABLES
52E1:           158 ;
52E1:00         159 Y2       DFB  0             ; BOTTOM Y-COORD
52E2:00         160 LBYTE    DFB  0             ; LEFT BYTE #
52E3:00         161 LMASK    DFB  0             ; LEFT BIT MASK
52E4:00         162 RMASK    DFB  0             ; RIGHT BIT MASK
52E5:00         163 SBWIDTH  DFB  0             ; SHIFTED IMAGE BYTE WIDTH
52E6:00         164 PITCH    DFB  0             ; BITMAP ROW PITCH
52E7:00         165 SHFTINDX DFB  0             ; BITMAP WIDTH
52E8:00         166 TEMP     DFB  0             ; TEMP STORAGE
52E9:00         167 XSAVE    DFB  0             ; FOR SAVED X-REG
52EA:           168 BUFFER   DS   40,0          ; SHIFTED DATA ROW BUFFER

*** SUCCESSFUL ASSEMBLY: NO ERRORS
```

Listing 4: *The* DIV7 *program creates two 256-byte tables that contain the integer quotient and remainder of the value* n/7, *respectively, for* n=0 *to* 255. *The resulting table is saved as* DIV7 TABLE.

```
]LIST

100 REM
110 REM  DIVIDE-BY-7 QUOTIENT
120 REM  AND REMAINDER TABLES
130 REM
140 REM  BY GREGG WILLIAMS,
150 REM  24 APR 84
160 REM
200 QUOTBGN = 16384
210 REM --POINTS TO MEMORY AREA
220 REM --USED TO STORE TABLE
230 REM
240 RMDRBGN = QUOTBGN + 256
250 HOME : PRINT "CREATING DIV7 TABLE...."
300 FOR I = 0 TO 255
310 :QVL = INT (I / 7)
320 :RVL = I - QVL * 7
330 : POKE QUOTBGN + I,QVL
340 : POKE RMDRBGN + I,RVL
350 NEXT I
400 PRINT  CHR$ (4);"BSAVE DIV7 TABLE,A";QUOTBGN;",L512"
420 PRINT : PRINT "TABLE SAVED TO DISK"
500 END
```

(continued)

Listing 5: *The Consolidate program loads the tables created by listings 1, 2, and 4, adds an image table, and stores the composite table in the file* TABLPAK *for later use.*

```
100 REM
110 REM  CONSOLIDATE-
120 REM  TABLES PROGRAM
130 REM
140 REM  BY GREGG WILLIAMS
150 REM  23 APR 84
160 REM
170 REM  ------------------------------------------------
180 REM
190 HOME : PRINT "CREATING TABLPAK...."
200 TBLSIZ = 32
236 REM
240 D$ = CHR$ (4)
250 PRINT D$;"BLOAD PRESHIFT TABLE,A$4000"
260 PRINT D$;"BLOAD DIV7 TABLE,A$4E00"
270 PRINT D$;"BLOAD HIRES1 TABLE,DA$5000"
280 REM
290 REM --LOAD NEEDED TABLES
300 REM --INTO MEMORY
320 REM
330 REM
540 REM
550 QQ$ = "5180:0D 0D 02 00 00 00 00 7C 0F 7C 07 7C 03 7C 01 7C
    03 7C 07 5C 0F 0C 1F 04 0E 00 04 00 00 00 00"
560 GOSUB 63000
570 REM --PUT IMAGE INTO MEMORY
580 REM --USING MONITOR COMMANDS
590 REM
600 REM
610 PRINT D$;"BSAVE TABLPAK,A$4000,L";4480 + TBLSIZ
615 PRINT : PRINT "FILE SAVED TO DISK"
620 REM
630 END
640 REM
62975 REM  ------------------------------------------------
62980 REM
62985 REM --FAMOUS S. H. LAM
62990 REM --MONITOR ROUTINE FOR
62993 REM --EXECUTING MONITOR
62995 REM --COMMANDS FROM BASIC
62997 REM
63000 QQ$ = QQ$ + " ND9C6G"
63010 FOR QQ = 1 TO LEN (QQ$): POKE 511 + QQ,128 + ASC
      ( MID$ (QQ$,QQ,1)): NEXT
63020 POKE 72,0: CALL - 144
63030 RETURN
```

Table 1: *Format for the image table. The* BASIC *demonstration program of listing 6 expects an image table in the format given by this table. Listing 3 expects this table at location* 5180 *hexadecimal* (20864 *decimal*), *but this can be easily changed by changing the value stored at* IBits (*location* 5203 *hexadecimal*).

Byte	Contents
0	(number of rows in image) — 1
1	(number of dots in row) — 1
2	number of bytes in one row of image
3, 4, 5...	successive bytes of image, starting with upper-left corner and proceeding by rows

Listing 6: *A program that demonstrates the preshift-table lookup method. This program uses joystick or paddle input to guide an arrow image across the high-resolution graphics screen. See text for details.*

```
]LIST

100 REM
110 REM  BUDGE PRESHIFT
120 REM  GRAPHICS DEMO
130 REM
140 REM  BY BILL BUDGE,
150 REM  GREGG WILLIAMS,
160 REM  AND ROB MOORE
170 REM
180 REM  ----------------------------------------------------
190 REM
200 REM  INITIALIZATION
210 REM
220 GOSUB 3000
230 REM --LOAD FILES (NEEDS
240 REM --TO BE DONE ONLY ONCE)
250 REM
260 GOSUB 2000
270 REM --INITIALIZE TABLES
280 REM
480 REM
490 REM  ----------------------------------------------------
500 REM
510 REM  MAIN LOOP
520 REM
530 REM  IFPEEK ( - 16287) > 127 THEN
    820
535 REM --WHILE LOOP: LOOP
540 REM --WHILE BUTTON 0
550 REM --NOT PRESSED
560 REM
570 XVLUE = PDL (0)
580 YVLUE = PDL (1)
590 REM --GET JOYSTICK OR
600 REM --PADDLE VALUES
610 REM
620 XINCR = - 1 * DOTSMOVE * (XVLUE < C1THRLO) + DOTS
    MOVE * (XVLUE > C2THRHI)
630 YINCR = - 1 * DOTSMOVE * (YVLUE < C1THRLO) + DOTS
    MOVE * (YVLUE > C2THRHI)
640 REM --CONVERT JOYSTICK
650 REM --INPUT TO -1,0, OR 1
660 REM
670 IF (XPSN + XINCR) > = C3XMIN AND (XPSN + XINCR) < =
    C4XMAX THEN XPSN = XPSN + XINCR
680 IF (YPSN + YINCR) > = C5YMIN AND (YPSN + YINCR) < =
    C6YMAX THEN YPSN = YPSN + YINCR
690 REM --MODIFY X, Y POSITIONS
700 REM --IF WITHIN BOUNDS
703 REM
706 POKE X1,XPSN
708 POKE Y1,YPSN
710 REM --POKE X & Y COORDS
712 REM --OF IMAGE INTO
714 REM --BYTES USED BY THE
716 REM --CODE SUBROUTINE
718 REM
720 CALL CODEADDR
730 REM --CALL MACHINE-LANG.
740 REM --BLOCK-MOVE ROUTINE
750 REM
760 GOTO 530
770 REM --END OF WHILE LOOP
780 REM
790 REM  ----------------------------------------------------
800 REM
810 END
820 REM --END OF PGM
830 REM
1940 REM
1950 REM
1960 REM  ----------------------------------------------------
1970 REM
1980 REM INITIALIZATION
1985 REM SUBROUTINE
1990 REM
2000 DOTSMOVE = 2
2010 REM --MAXIMUM INCREMENT
2020 REM --OF IMAGE
2030 REM
2040 XPSN = 100:YPSN = 100
2050 REM --POSITION OF IMAGE
2060 REM
2070 C1THRLO = 100:C2THRHI = 150
2080 REM --THRESHOLD VALUES
2090 REM --FOR JOYSTICK INPUT
2100 REM
2110 C3XMIN = 5:C4XMAX = 235
2120 C5YMIN = 5:C6YMAX = 175
2130 REM --BOUNDARIES OF IMAGE
2140 REM --MOVEMENT ON SCREEN
2150 REM
2160 CODEADDR = 21008
2170 REM --ADDRESS OF MACHINE-
2180 REM --LANGUAGE SUBROUTINE
2190 REM
2200 IROWS = 20992
2210 IDOTS = IROWS + 1
2220 IWIDTH = IROWS + 2
2230 IBITS = IROWS + 3
2240 X1 = IROWS + 5
2250 Y1 = IROWS + 6
2260 REM --ADDRESSES OF DATA
2270 REM --NEEDED BY ASSBY-
2280 REM --LANGUAGE ROUTINE
2290 REM
2420 POKE IROWS, PEEK (IMAGTBL)
2430 POKE IDOTS, PEEK (IMAGTBL + 1)
2440 POKE IWIDTH, PEEK (IMAGTBL + 2)
2450 IP = IMAGTBL + 3
2460 C8IPLO = INT (IP / 256)
2470 C7IPHI = IP - 256 * C8IPLO
2480 POKE IBITS,C7IPLO
2490 POKE IBITS + 1,C8IPHI
2500 REM --SETUP VALUES NEEDED
2510 REM --BY ASSEMBLY-LANGUAGE
2520 REM --ROUTINE
2530 REM
2540 HGR : POKE - 16302,0
2550 REM --SWITCH TO HIRES PG1
2560 REM
2570 RETURN
2930 REM
2935 REM  ----------------------------------------------------
2940 REM
2950 REM LOAD FILES
2960 REM SUBROUTINE
2970 REM
3000 IMAGTBL = 20864
3010 REM --ADDRESS OF TABLE
3020 REM --CONTAINING IMAGE
3030 REM --TO BE MOVED
3040 REM
3050 D$ = CHR$ (4)
3060 PRINT D$;"BLOAD QD.DEMO.0"
3070 PRINT D$;"BLOAD TABLPAK,A$4000"
3080 PRINT D$;"BLOAD IMAGE,A";IMAGTBL
3090 REM --LOAD ASSBY-LANGUAGE
3100 REM --ROUTINE AND TABLES
3110 REM
3120 RETURN
```

(continued from page A27)

assembly-language routine of listing 3; it then uses joystick (or paddle) input to move the image around the high-resolution graphics screen. This program uses an image that has two rows of unlighted dots on every edge (it is a 10 by 10 image centered in a 14 by 14 box). Because the program moves the image only two dots at a time, the arrow image erases itself as it moves and leaves no trail.

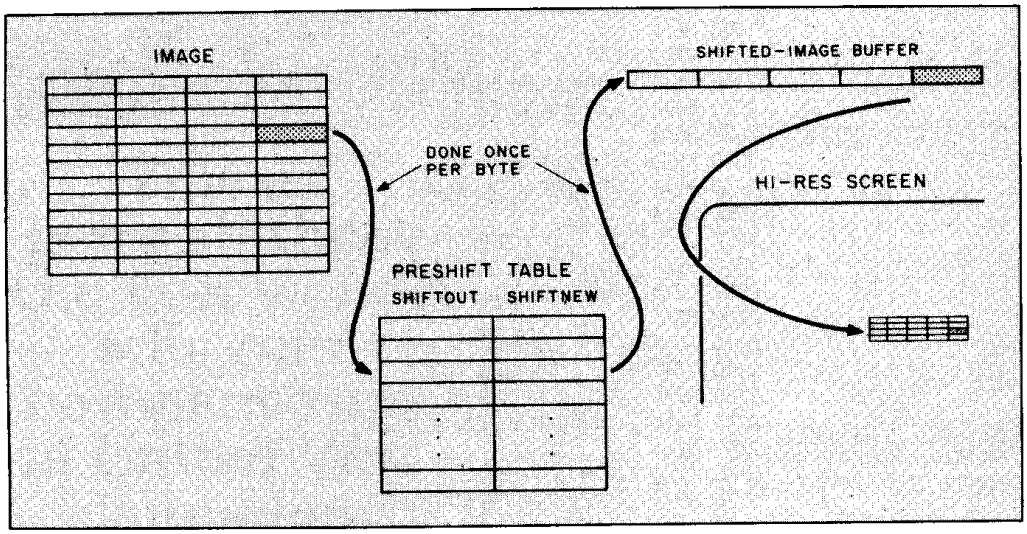

Figure 7: *Moving a rectangular image. In the example subroutine of listing 3, the routine takes the image a line at a time, uses the preshift table to shift it, puts it in a single-line buffer, then transfers it to the high-resolution screen. This is only one possible subroutine that implements the routine in figure 6.*

This program, like the others, was written with simplicity and clarity in mind rather than speed or program features. The fact that the program moves the arrow slowly across the screen is the fault of BASIC, not the assembly-language program. To make this demo run faster, you can "tighten up" the main loop in lines 530–760 or incorporate some of the joystick decoding and boundary checking in another assembly-language program that, in turn, calls listing 3.

The preshift-table lookup method is a compromise between utility and ease of comprehension. Although this technique is not as fast as the preshifted-shapes method often used for animation, it is a general-purpose method that will probably find a number of uses. ■