Display File Organization

(by Alvin Albrecht - http://www.geocities.com/aralbrec/spritepack/)

(Note: info related to ts2068 microcomputer, the american version of the Sinclair Spectrum. All info below also applies to our beloved 8 bit rubber-key machine :)

The ts2068's display file is where all the screen information is stored. The SCLD chip constructs the TV display by reading the information stored there. The display file is "memory-mapped" because the storage exists in the z80's memory space, from address 16384 to 22527. If you poke values into those addresses you will see the display change. In the ts2068's other display modes (dual screen, hi-colour, hi-res) more areas of memory are used to hold the display. In this article, we'll only concern ourselves with the default 256x192 mode.

A pixel display occupying 16384 to 22527 reserves 6144 bytes to store all the screen information. The ts2068 has a resolution of 256*192 = 49152 pixels. How do we cram information about 49152 pixels into 6144 bytes? Well, each pixel can be represented by one bit - either one or zero, on or off. Cramming 8 pixels into a byte, we'd need 256*192/8 = 6144 bytes. Problem solved!

A simple way to organize the display might have pixels 0..7 for the top line of the display stored at address 16384, pixels 8..15 at address 16385, … pixels 248-255 stored at address 16415. The next pixel line would follow with pixels 0..7 of line 1 at address 16416, and so forth for all 192 lines on the screen. This is indeed how the TV draws its display, left to right, top to bottom. But the display organization was chosen to optimize the printing of characters so it's not done in this simple manner. To see evidence of this, try this short program:

```
10 FOR z=16384 TO 22527
20 POKE z,255
30 NEXT z
```

On the largest scale you will notice that the display is divided into three parts called blocks. First the top block is filled, then the second and finally the third. Each block is further divided into eight character lines. Each of these lines is divided into eight scan lines. The first scan line for all character lines in a block is filled, followed by the second scan line for all character lines, and so on to the final eighth scan line. Each scan line itself is composed of 32 horizontal bytes with each byte holding eight pixels.

This organization sounds complicated but it really isn't that bad if some thought is applied to it. By paying attention to how the display is built up in increasing byte order, we can construct a screen address given block, character line, scan line and column as follows:

FIGURE 1. Screen Address Organization in Binary

0	1	0	B	B	S	S	S	L	L	L	C	C	C	C	C

Where:

BB = screen block, 0..2
SSS = scan line, 0..7
LLL = character line, 0..7
CCCCC = horizontal byte / character, 0..31

While observing the Basic program in action, you'll notice that the horizontal column changes the fastest. There are 32 columns, requiring 5 bits to represent those. They increase the fastest so they appear in the bottom 5 bits of the 16-bit address. The next fastest thing that changes is the character line. There are 8 lines in each block, requiring 3 bits to represent them. These 3 bits appear next to the column bits. Next, in order of fastest changing, are the scan lines (8 of them requiring 3 bits) followed by the block (3 of them requiring 2 bits). The display starts at address 16384 (0x4000) so we add that to our 16-bit address. This is responsible for the lone '1' you see in figure 1.

The character position row = 10, column = 12 is located in block 1 (the second block since it holds the second third of the display, rows 8..15), line 2 (the third character line in this block -- rows 8, 9, **10**), scan lines 0 (top) through 7 (bottom) for the full character square, and column 12. This leads to a screen address that looks like:

0	1	0	0	1	S	S	S	0	1	0	0	1	1	0	0

With various values of SSS:

SSS	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
Screen	484C	494C	4A4C	4B4C	4C4C	4D4C	4E4C	4F4C
Address	18508	18764	19020	19276	19532	19788	20044	20300

To print a letter 'A' at (10,12), poke the appropriate values into memory at these addresses:

```
POKE 18508,BIN 00000000
POKE 18764,BIN 00111100
POKE 19020,BIN 01000010
POKE 19276,BIN 01000010
POKE 19532,BIN 01111110
POKE 19788,BIN 01000010
POKE 20044,BIN 01000010
POKE 20300,BIN 00000000
```

At this point, you may realize why UDGs and printed characters are 8x8 pixels in size. There are 8 vertical scan lines in each character line and there are 8 pixels packed into a byte. But you may

not realize why this particular display file organization speeds up character printing. If you back up and look at the screen addresses computed above, you'll notice that each scan line is separated by exactly 256 bytes. In assembly language, an address is held in a register pair, like HL. Adding 256 to an address to move to the next scan line is a simple matter of incrementing the most significant register, in this case H with the "INC H" instruction. That's all it takes! Moving horizontally to the right one character position involves adding one to the screen address (ie adding one to "CCCCC" in figure 1), which can be done just as quickly with "INC L". You can't get any faster than that. In fact, this display file organization was patented by Sinclair's Richard Altwasser back in 1982 (visit http://wearmouth.demon.co.uk/ to see the patent).

That's all fine and good but we still haven't managed to easily map a pixel coordinate to a screen address. Here's how we do it:

FIGURE 2. Mapping Pixel Coordinates to Screen Address Units

X								Y							
C	C	C	C	C	T	T	T	B	B	L	L	L	S	S	S

The thought process that led to figure 2 is similar to the previous one. The X coordinate is more or less obvious: there are 32 columns horizontally (5 bits) with each column containing 8 pixels (requiring 3 bits). The pixel position within a byte (0..7) changes fastest as we move horizontally so it appears as "TTT" in the least significant bits of X. For the Y coordinate, the fastest changing items as we move from the top of the screen to the bottom are the scan line, followed by the character line, followed by the block.

Given an X coordinate in the range 0-255 and a Y coordinate in the range 0-192, convert them to binary as in figure 2 and reassemble the bits as in figure 1. For example, pixel coordinate (x,y) = (133,67) in binary is (1000 0101, 0100 0011) with CCCCC=10000, BB=01, LLL=000, SSS=011 according to figure 2. Moving the bits around to the form in figure 1 gives an address of "0100 1011 0001 0000" or 19216 in decimal. The bits "TTT" in the X coordinate do not appear in figure 1. They identify which bit within the screen byte corresponds to the individual pixel. "0" corresponds to the leftmost bit and "7" corresponds to the rightmost; in this case it's 5. To plot the pixel (133,67) we could simply "POKE 19216,BIN 00000100" where the single '1' in the BIN statement sits in bit 5 from the left. Keep in mind that the pixel coordinates I am using have the screen's origin located at the top left corner of the screen. This is different from TS2068 Basic which places the origin 16 pixels above the bottom left corner of the screen.

If this procedure had to be done by hand for each pixel, it would get tedious quickly. Here's a short machine code routine that does it for us:

```
; Get Screen Address
;
; Returns the screen address and pixel mask corresponding
; to a given pixel coordinate.
;
; enter: a = h = y coord
;        l = x coord
; exit : de = screen address, b = pixel mask
; uses : af, b, de, hl

.SPGetScrnAddr
   and $07    ; A = 00000SSS
   or $40     ; A = 01000SSS
   ld d,a     ; D = 01000SSS
   ld a,h     ; A = Y coord = BBLLLSSS
   rra
   rra
   rra        ; A = ???BBLLL
   and $18    ; A = 000BB000
   or d       ; A = 010BBSSS
   ld d,a     ; D = 010BBSSS top 8 bits of address done

   ld a,l     ; A = X coord = CCCCCTTT
   and $07    ; A = 00000TTT
   ld b,a     ; B = 00000TTT = which pixel?
   ld a,$80   ; A = 10000000
   jr z, norotate  ; if B=0, A is the right pixel so skip

.rotloop
   rra        ; rotate the pixel right one place B times
   djnz rotloop

.norotate
   ld b,a     ; B = pixel mask
   srl l
   srl l
   srl l      ; L = 000CCCCC
   ld a,h     ; A = Y coord = BBLLLSSS
   rla
   rla        ; A = LLLSSS??
   and $e0    ; A = LLL00000
   or l       ; A = LLLCCCCC
   ld e,a     ; E = LLLCCCCC
   ret        ; DE = 010BBSSS LLLCCCCC, the screen address!
```

The subroutine is called with A=H=Y coordinate and L=X coordinate and we get the screen address in DE and the pixel mask in B on the way out. If we ORed B into (DE), we could plot the pixel. If we ANDed the complement of B into (DE), we could unplot the pixel and if we ANDed B with (DE) we could test whether the pixel was set.

This subroutine is great for calculating a screen address corresponding to a pixel position from scratch, but you'll notice that it is rather lengthy and therefore slow, in a relative sense. Frequently you'll be plotting a pixel and then plotting many more nearby, possibly a single pixel

away. For example, in the process of drawing a line, the initial point is plotted and then succeeding points above, below, to the left or right are plotted. We could handle the drawing of the line as plotting many individual pixel points, calling the above subroutine to compute the screen address for every pixel, but that would be much slower than working directly on the screen address to move up, down, left and right from a current pixel position.

Let's investigate further to substantiate that claim. Given a screen address in HL and a pixel mask in B, how would one move left one pixel? Here's the necessary code:

```
; hl = screen address, B = pixel mask
.left
   rlc b
   ret nc
   dec l
   ret
```

The pixel mask is rotated left one bit. This will be a valid pixel position unless B was already at the leftmost pixel position in the screen byte (ie B=1000 0000). The "RLC B" instruction will set the carry flag in that case and leave B=0000 0001. We use the no carry flag to return early if the new mask is valid, otherwise we update the column position one character to the left by decreasing the "CCCCC" portion of the screen address. The value of B at this point is 0000 0001, correctly masking the rightmost pixel in the new screen byte to the left of the old one. These four instructions are clearly quicker than rerunning the screen address subroutine. Notice that this subroutine doesn't check if it runs off the edge of the screen. The right pixel movement is similar, substituting "rrc b" for "rlc b" and "inc l" for "dec l".

To move up a pixel we need to decrement the Y coordinate, as pictured in figure 2. Given a screen address, this means first decreasing SSS followed by LLL (if necessary) and finally BB (if necessary). These bits are scattered about in the screen address pictured in figure 1 so a little care must be taken. The necessary code is shown here:

```
; hl = screen address

.SPPixelUp
   ld a,h          ; A=H=010BBSSS
   dec h           ; decrease SSS
   and $07         ; if SSS was not originally 000
   ret nz          ; we're done
   ld a,$08        ; otherwise SSS=111 (correct)
   add a,h         ; and we fix BB in H (one was subtracted)
   ld h,a
   ld a,l          ; A=X coord=LLLCCCCC
   sub $20         ; decrease LLL
   ld l,a
   ret nc          ; if no carry, LLL was not originally 000, ok
   ld a,h          ; otherwise LLL=111 now, that's okay
   sub $08         ; but need to decrease screen block
   ld h,a
   ret
```

This subroutine derives a lot of speed by minimizing the number of instructions executed in the most common cases. For example, 7 out of 8 times, only the first four instructions will be executed. 7 out 64 times, the first 11 instructions will execute and the rest of the time (1 out of 64) all the instructions will execute. This makes the subroutine much quicker than one would initially guess by looking at the size of the code. The PixelDown subroutine is similar but is not shown here. All these pixel movement routines are reprinted in full in the floodfill listings elsewhere in this article.