EPSON
EXCEED YOUR VISION

S1D13705 Embedded Memory LCD Controller

Programming Notes and Examples

Document Number: X27A-G-002-03.1

SEIKO EPSON CORPORATION

Rev. 3.1

Table of Contents

1 Introduction

This guide demonstrates how to program the S1D13705 Embedded Memory Color LCD Controller. The guide presents the basic concepts of the LCD controller and provides methods to directly program the registers. It explains some of the advanced techniques used and the special features of the S1D13705.

The guide also introduces the Hardware Abstraction Layer (HAL), which is designed to make programming the S1D13705 as easy as possible. Future S1D1370x products will support the HAL allowing OEMs the ability to upgrade to future chips with relative ease.

This document is updated as appropriate. Please check for the latest revision of this document before beginning any development. The latest revision can be downloaded at vdc.epson.com.

We appreciate your comments on our documentation. Please contact us via email at vdc-documentation@ea.epson.com.

2 Initialization

Prior to doing anything else with the S1D13705 the controller must be initialized. Initialization is the process of setting up the control registers to a known state in order to generate proper display signals.

2.1 Display Buffer Location

Before we can perform the initialization we have to know where to find the S1D13705 display memory and control registers.

The S1D13705 contains 80 kilobytes of internal display memory. External support logic must be employed to decode the starting address for this display memory in CPU address space. On the S5U13705B00x PC platform evaluation boards the address is usually fixed at F00000h. Alternatively the address can be set to D0000h.

The control registers are located by adding 1FFE0h (128 Kb less 32 bytes) to the base memory address. Thus, on the typical PC platform, we access control register 0 at address F1FFE0h. Control register 5 would be located at address F1FFE5, etc.

2.2 Register Values

This section describes the register settings and sequence of setting the registers. In addition to these setting the Look-Up Table must be programmed with appropriate colors. Look-Up Table setup is not covered here. See Section 4 on page 13 of this manual for Look-Up Table programming details.

The following initialization, presented in table form, shows the sequences and values to set the registers. The notes column comments the reason for the particular value being written.

This example writes to all the necessary registers. Initially, when the S1D13705 is powered up, all registers, unless noted otherwise in the specification, are set to zero. This example programs these registers to zero to establish a known state. In practice, it may be possible to write to only a subset of the registers.

The example initializes a S1D13705 to control a panel with the following specifications:

- 320x240 color single passive LCD panel at 70Hz.
- Color Format 2, 8-bit data interface.
- 8 bit-per-pixel (256 colors).
- 6 MHz input clock (CLKI).

Table 2-1: S1D13705 Initialization Sequence

Register	Value (hex)	Notes	See Also
[01]	0010 0011 (23)	Select a passive, Single, Color panel with an 8-bit data width	
[02]	1100 0000 (C0)	Select 8-bit per pixel color depth	
[03]	0000 0011 (03)	Select normal power operation	
[04]	0010 0111 (27)	Horizontal display size = (Reg[04]+1)*8 = (39+1) * 8 = 320 pixels	
[05]	1110 1111 (EF)	Vertical display size = Reg[06][05] + 1 = 0000 0000 1110 1111 + 1 = 239 +1 = 240 lines	
[06]	0000 0000 (00)		
[07]	0000 0000 (00)	FPLINE start position (only required for TFT configuration)	
[08]	0000 0000 (00)	Horizontal non-display period = (Reg[08] + 4) * 8 = 4 * 8 = 32 pixels	Frame Rate Calculation
[09]	0000 0000 (00)	FPFRAME start position (only required for TFT configuration)	
[0A]	0000 0011 (03)	Vertical non-display period = REG[0A] = 3 lines	Frame Rate Calculation
[0B]	0000 0000 (00)	MOD rate is only required by some monochrome panels	
[0C]	0000 0000 (00)	Screen 1 Start Address - set to 0 for initialization	Split Screen on page 29
[0D]	0000 0000 (00)		
[0E]	0000 0000 (00)	Screen 2 Start Address - set to 0 for initialization	Split Screen on page 29
[0F]	0000 0000 (00)		
[10]	0000 0000 (00)	Screen 1 / Screen 2 Start Address MSB - set to 0	
[11]	0000 0000 (00)	Memory Address offset - not virtual setup - so set to 0	Virtual Display on page 23
[12]	1111 1111 (FF)	Set the vertical size to the maximum value.	Split Screen on page 29
[13]	0000 0011 (03)		
[15]		Leave the LUT alone for now	Look-Up Table (LUT) on page 13
[17]			
[18]	0000 0000 (00)	GPIO control and status registers - set to “0”.	
[19]	0000 0000 (00)		
[1A]	0000 0000 (00)	Set the scratch pad bits to “0”.	
[1B]	0000 0000 (00)	This is not portrait mode so set this register to “0”.	Introduction To Hardware Rotation on page 35
[1C]	0000 0000 (00)	Line Byte Count is only required for portrait mode.	

2.3 Frame Rate Calculation

Frame rate specifies the number of complete frame which are drawn on the display in one second. Configuring a frame rate that is too high or too low adversely effects the quality of the displayed image.

System configuration imposes certain non-variable limitations. For instance the width and height of the display panel are fixed as is, typically, the input clock to the S1D13705. From the following formula it is evident that the two variables the programmer can use to adjust frame rate are horizontal and vertical non-display periods.

The following are the formulae for determining the frame rate of a panel. The formula for a single passive or TFT panel is calculated as follows:

$$\text{FrameRate} = \frac{\text{PCLK}}{(\text{HDP} + \text{HNDP}) \times (\text{VDP} + \text{VNDP})}$$

for a dual passive panel the formula is:

$$\text{FrameRate} = \frac{\text{PCLK}}{2 \times (\text{HDP} + \text{HNDP}) \times \left(\frac{\text{VDP}}{2} + \text{VNDP}\right)}$$

where:	PCLK	= Pixel clock (in Hz)
	HDP	= Horizontal Display Period (in pixels)
	HNDP	= Horizontal Non-Display Period (in pixels)
	VDP	= Vertical Display Period (in lines)
	VNDP	= Vertical Non-Display Period (in lines)

In addition to varying the HNDP and VNDP times we can also select divider values which will reduce CLKi to one half, one quarter up to one eight of the CLKi value. The example below is a portion of a ’C’ routine to calculate HNDP and VNDP from a desired frame rate.

```
for (int loop = 0; loop < 2; loop++)
{
      for (VNDP = 2; VNDP < 0x3F; VNDP += 3)
      {
            // Solve for HNDP
             HNDP = (PCLK / (FrameRate * (VDP + VNDP))) - HDP;
            if ((HNDP >= 32) && (HNDP <= 280))
            {
                  // Solve for VNDP.
                  VNDP = (PCLK / (FrameRate * (HDP + HNDP))) - VDP;
                  // If we have satisfied VNDP then we're done.
                  if ((VNDP >= 0) && (VNDP <= 0x3F))
                        goto DoneCalc;
            }
      }
      // Divide ClkI and try again.
      // (Reg[02] allows us to dived CLKI by 2)
      PCLK /= 2;
}
// If we still can't hit the frame rate - throw an error.
if ((VNDP < 0) || (VNDP > 0x3F) || (HNDP < 32) || (HNDP > 280))
{
      sprintf("ERROR: Unable to set the desired frame rate.\n");
      exit(1);
}
```

This routine first performs a formula rearrangement so that HNDP or VNDP can be solved. Start with VNDP set to a small value. Loop increasing VNDP and solving the equation for HNDP until satisfactory HNDP and VNDP values are found. If no satisfactory values are found then divide CLKI and repeat the process. If a satisfactory frame rate still can't be reached - return an error.

Note

Most passive (STN) panels are tolerant of nearly any combination of HNDP and VNDP values, however panel specifications generally specify only a few lines of vertical non-display period. The S1D13705 is capable of generating a vertical non-display period of up to sixty-three lines. This amount of VNDP is far too great a non-display period and will likely degrade display quality. Similarly, setting a large HNDP value may cause a degrade in image quality.

If possible the system should be designed such that VNDP values of 7 or less lines and HNDP values of 20 or less characters can be selected.

3 Memory Models

The S1D13705 is capable of operating at four different color depths. For each color depth the data format is packed pixel. S1D13705 packed pixel modes can range from one byte containing eight adjacent pixels (1-bpp) to one byte containing just one pixel (8-bpp).

Packed pixel data may be envisioned as a stream of pixels. In this stream, pixels are packed in adjacent to each other. If a pixel requires four bits then it will be located in the four most significant bits of a byte. The pixel to the immediate right on the display will occupy the lower four bits of the same byte. The next two pixels to the immediate right are located in the following byte, etc.

3.1 1 Bit-Per-Pixel (2 Colors/Gray Shades)

1-bit pixels support two color/gray shades. In this memory format each byte of display buffer contains eight adjacent pixels. Setting or resetting any pixel requires reading the entire byte, masking out appropriate bits and, if necessary, setting bits to “1”.

When using a color panel the two colors are derived by indexing into positions 0 and 1 of the Look-Up Table. If the first two LUT elements are set to black (RGB = 0 0 0) and white (RGB = F F F) then each “0” bit of display memory will display as a black pixel and each “1” bit will display as a white pixel. The two LUT entries can be set to any desired colors, for instance red and green or cyan and yellow.

For monochrome panels the two displayed gray shades are generated by indexing into the first two elements of the green component of the Look-Up Table (LUT). Thus, by manipulating the green LUT components we can set either of the two gray shades to any of sixteen possible levels.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Pixel 0	Pixel 1	Pixel 2	Pixel 3	Pixel 4	Pixel 5	Pixel 6	Pixel 7

Figure 3-1: Pixel Storage for 1 Bpp (2 Colors/Gray Shades) in One Byte of Display Buffer

3.2 2 Bit-Per-Pixel (4 Colors/Gray Shades)

2-bit pixels support four color/gray shades. In this memory format each byte of display buffer contains four adjacent pixels. Setting or resetting any pixel requires reading the entire byte, masking out the appropriate bits and, if necessary, setting bits to "1".

Color panels derive their four colors by indexing into positions 0 through 3 of the Look-Up Table. These four LUT entries can be set to any of the 4096 possible color combinations.

Monochrome panels derive four gray shades by indexing into the first four elements of the green component of the Look-Up Table. Any of the four LUT entries can be set to any of the sixteen possible gray shades.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Pixel 0 Bit 1	Pixel 0 Bit 0	Pixel 1 Bit 1	Pixel 1 Bit 0	Pixel 2 Bit 1	Pixel 2 Bit 0	Pixel 3 Bit 1	Pixel 3 Bit 0

Figure 3-2: Pixel Storage for 2 Bpp (4 Colors/Gray Shades) in One Byte of Display Buffer

3.3 4 Bit-Per-Pixel (16 Colors/Gray Shades)

Four bit pixels support 16 color/gray shades. In this memory format each byte of display buffer contains two adjacent pixels. Setting or resetting any pixel requires reading the entire byte, masking out the upper or lower nibble (4 bits) and setting the appropriate bits to "1".

Color panels can display up to sixteen colors simultaneously. These sixteen colors are derived by indexing into the first sixteen elements of the Look-Up Table. Each of these colors may be selected from the 4096 possible available colors.

On a monochrome panel the gray shades are generated by indexing into the first sixteen green components of the LUT. Each of these sixteen possible gray shades can be adjusted to any of the sixteen possible gray shades. For instance, one could program the first eight green LUT entries to be 0 and the second green LUT entries to be FFh. This would result in nibble values of 0 through 7 displaying as black and nibble values 8 through 0Fh displaying as white.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Pixel 0 Bit 3	Pixel 0 Bit 2	Pixel 0 Bit 1	Pixel 0 Bit 0	Pixel 1 Bit 3	Pixel 1 Bit 2	Pixel 1 Bit 1	Pixel 1 Bit 0

Figure 3-3: Pixel Storage for 4 Bpp (16 Colors/Gray Shades) in One Byte of Display Buffer

3.4 Eight Bit-Per-Pixel (256 Colors)

In eight bit-per-pixel mode one byte of display buffer represents one pixel on the display. At this color depth the read-modify-write cycles, required by the lessor pixel depths, are eliminated.

When using a color panel, each byte of display memory acts as and index to one element of the LUT. The displayed color is arrived at by taking the display memory value as an index into the LUT.

Eight bit per pixel is not supported for monochrome display modes. The reason is that each element of the LUT supports a 4-bit (sixteen value) level for red, green and blue. In monochrome display modes on the green value is used to set the gray intensity. Thus we have sixteen possible grey values but, because of the color

.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Red bit 2	Red bit 1	Red bit 0	Green bit 2	Green bit 1	Green bit 0	Blue bit 1	Blue bit 0

Figure 3-4: Pixel Storage for 8 Bpp (256 Colors) in One Byte of Display Buffer

4 Look-Up Table (LUT)

This section is supplemental to the description of the Look-Up Table architecture found in the S1D13705 Hardware Functional Specification. Covered here is a review of the LUT registers, recommendations for the color and gray shade LUT values, and additional programming considerations for the LUT. Refer to the S1D13705 Hardware Functional Specification, document number X27A-A-001-xx for more detail.

The S1D13705 Look-Up Table consists of 256 indexed red/green/blue entries. Each entry is 4 bits wide. Two registers, REG[15h] and REG[17h], control access to the LUT.

Each Look-Up Table entry consists of a red, green, and blue component. Each component consisting of four bits, or sixteen intensity levels. Any Look-Up Table element can be selected from a palette of 4096 (16x16x16) colors.

In color display modes, pixel values are used as an index to an RGB value stored in the Look-Up Table. In monochrome modes, pixel values still index into the LUT, but only the green component is used to determine display intensity.

The selected color depth determines how many index positions are used for image display. For example at one bit-per-pixel (bpp) only index positions 0 and 1 of the Look-Up Table are used. At 4-bpp the first 16 index positions of the Look-Up Table are used and at 8-bpp all 256 Look-Up Table index positions are used.

The Look-Up Table mechanism itself consists of an index register and a data register. The index, or address, register determines which element of the Look-Up Table will be accessed. After setting the index the LUT may be read or written through the data register. The first data element read or written is the red component of the entry. Subsequent read/write operations access the green and then the blue elements of the Look-Up Table.

The S1D13705 LUT architecture is designed to provide a high degree of similarity in operation to a standard VGA RAMDAC. However, there are two considerations which must be kept in mind.

- The S1D13705 Look-Up Table has four bits (16 levels) of intensity per primary color. The standard VGA RAMDAC has six bits (64 levels). This four to one difference must be taken into consideration when converting from a VGA palette to a LUT palette. One suggestion is to divide the VGA intensity level by four to arrive at the LUT intensity.

  However, most applications specify the red, green and blue components as eight bit intensities. To determine the appropriate S1D13705 Look-Up Table value we recommend using the four most significant bits.

4.1 Look-Up Table Registers

REG[15h] Look-Up Table Address Register							Read/Write
LUT Address Bit 7	LUT Address Bit 6	LUT Address Bit 5	LUT Address Bit 4	LUT Address Bit 3	LUT Address Bit 2	LUT Address Bit 1	LUT Address Bit 0

LUT Address

The LUT address register selects which of the 256 LUT entries will be accessed. After three successive reads/writes to the data register this register is automatically incremented to point to the next address.

REG[17h] Look-Up Table Data Register							Read/Write
LUT Data Bit 3	LUT Data Bit 2	LUT Data Bit 1	LUT Data Bit 0	n/a	n/a	n/a	n/a

LUT Data

This register is where the 4-bit red/green/blue data value is written/read. Immediately after setting the LUT index with register [15h] this register accesses the red element of the Look-Up Table. With each successive write/read the internal bank select is incremented. Thus the second access is from the green element and the third is from the blue element.

After the third access the LUT Address is incremented by one, then next access to this register will be the red element of the next Look-Up Table index.

4.2 Look-Up Table Organization

4.2.1 Color Modes

1 bpp color

When the S1D13705 is configured for 1 bpp color mode, the LUT is limited to selecting colors from the first two entries. The two LUT entries can be any two RGB values but are typically set to black-and-white.

Each byte in the display buffer contains eight adjacent pixels. If a bit has a value of “0” then the color in LUT 0 index is displayed. A bit value of “1” results in the color in LUT 1 index being displayed.

The following table shows the recommended values for obtaining a black-and-white mode while in 1 bpp on a color panel.

Table 4-1: Recommended LUT Values for 1 Bpp Color Mode

Index	Red	Green	Blue
00	00	00	00
01	F0	F0	F0
02	00	00	00
...	00	00	00
FF	00	00	00

🟨	unused entries

2 bpp color

When the S1D13705 is configured for 2 bpp color mode, the displayed colors are selected from the first four entries of the Look-Up Table. The LUT entries may be set to any of the 4096 possible colors.

Each byte in the display buffer contains four adjacent pixels. If a bit combination has a value of “00” then the color in LUT index 0 is displayed. A bit value of “01” results in the color in LUT index 1 being displayed. Likewise the bit combination of “10” displays from the third LUT entry and “11” displays a color from the fourth LUT entry.

The following table shows the example values for 2 bit-per-pixel display mode.

Table 4-2: Example LUT Values for 2 Bpp Color Mode

Index	Red	Green	Blue
00	00	00	00
01	70	70	70
02	A0	A0	A0
03	F0	F0	F0
04	00	00	00
...	00	00	00
FF	00	00	00

🟨	indicates unused entries

4 bpp color

When the S1D13705 is configured for 4 bpp color mode, the displayed colors are selected from the first sixteen entries of the Look-Up Table. The LUT entries may be set to any of the 4096 possible colors.

Each byte in the display buffer contains two adjacent pixels. If a nibble has a value of "0000" then the color in LUT index 0 is displayed. A nibble value of "0001" results in the color in LUT index 1 being displayed. The pattern continues to the nibble pattern of "1111" which results in the sixteenth color of the Look-Up Table being displayed.

The following table shows the example values for 4 bit-per-pixel display mode. These colors simulate the colors used by the sixteen color modes of a VGA.

Table 4-3: Suggested LUT Values to Simulate VGA Default 16 Color Palette

Index	Red	Green	Blue
00	00	00	00
01	00	00	A0
02	00	A0	00
03	00	A0	A0
04	A0	00	00
05	A0	00	A0
06	A0	A0	00
07	A0	A0	A0
08	00	00	00
09	00	00	F0
0A	00	F0	00
0B	00	F0	F0
0C	F0	00	00
0D	F0	00	F0
0E	F0	F0	00
0F	F0	F0	F0
10	00	00	00
...	00	00	00
FF	00	00	00

🟨	indicates unused entries

8 bpp color

When the S1D13705 is configured for 8 bpp color mode the entire Look-Up Table is used to display images. Each of the LUT entries may be set to any of the 4096 possible colors.

Each byte in the display buffer represents one pixels. The byte value is used directly as an index into one of the 256 LUT entries. A display memory byte with a value of 00h will display the color contained in the first Look-Up Table entry while a display memory byte of FFh will display a color formed byte the two hundred and fifty sixth Look-Up Table entry.

The following table depicts LUT values which approximate the VGA default 256 color palette.

Table 4-4: Suggested LUT Values to Simulate VGA Default 256 Color Palette

Index	R	G	B	Index	R	G	B	Index	R	G	B	Index	R	G	B
00	00	00	00	40	F0	70	70	80	30	30	70	C0	00	40	00
01	00	00	A0	41	F0	90	70	81	40	30	70	C1	00	40	10
02	00	A0	00	42	F0	B0	70	82	50	30	70	C2	00	40	20
03	00	A0	A0	43	F0	D0	70	83	60	30	70	C3	00	40	30
04	A0	00	00	44	F0	F0	70	84	70	30	70	C4	00	40	40
05	A0	00	A0	45	D0	F0	70	85	70	30	60	C5	00	30	40
06	A0	50	00	46	B0	F0	70	86	70	30	50	C6	00	20	40
07	A0	A0	A0	47	90	F0	70	87	70	30	40	C7	00	10	40
08	50	50	50	48	70	F0	70	88	70	30	30	C8	20	20	40
09	50	50	F0	49	70	F0	90	89	70	40	30	C9	20	20	40
0A	50	F0	50	4A	70	F0	B0	8A	70	50	30	CA	30	20	40
0B	50	F0	F0	4B	70	F0	D0	8B	70	60	30	CB	30	20	40
0C	F0	50	50	4C	70	F0	F0	8C	70	70	30	CC	40	20	40
0D	F0	50	F0	4D	70	D0	F0	8D	60	70	30	CD	40	20	30
0E	F0	F0	50	4E	70	B0	F0	8E	50	70	30	CE	40	20	30
0F	F0	F0	F0	4F	70	90	F0	8F	40	70	30	CF	40	20	20
10	00	00	00	50	B0	B0	F0	90	30	70	30	D0	40	20	20
11	10	10	10	51	C0	B0	F0	91	30	70	40	D1	40	20	20
12	20	20	20	52	D0	B0	F0	92	30	70	50	D2	40	30	20
13	20	20	20	53	E0	B0	F0	93	30	70	60	D3	40	30	20
14	30	30	30	54	F0	B0	F0	94	30	70	70	D4	40	40	20
15	40	40	40	55	F0	B0	E0	95	30	60	70	D5	30	40	20
16	50	50	50	56	F0	B0	D0	96	30	50	70	D6	30	40	20
17	60	60	60	57	F0	B0	C0	97	30	40	70	D7	20	40	20
18	70	70	70	58	F0	B0	B0	98	50	50	70	D8	20	40	20
19	80	80	80	59	F0	C0	B0	99	50	50	70	D9	20	40	20
1A	90	90	90	5A	F0	D0	B0	9A	60	50	70	DA	20	40	30
1B	A0	A0	A0	5B	F0	E0	B0	9B	60	50	70	DB	20	40	30

Table 4-4: Suggested LUT Values to Simulate VGA Default 256 Color Palette (Continued)

Index	R	G	B	Index	R	G	B	Index	R	G	B	Index	R	G	B
1C	B0	B0	B0	5C	F0	F0	B0	9C	70	50	70	DC	20	40	40
1D	C0	C0	C0	5D	E0	F0	B0	9D	70	50	60	DD	20	30	40
1E	E0	E0	E0	5E	D0	F0	B0	9E	70	50	60	DE	20	30	40
1F	F0	F0	F0	5F	C0	F0	B0	9F	70	50	50	DF	20	20	40
20	00	00	F0	60	B0	F0	B0	A0	70	50	50	E0	20	20	40
21	40	00	F0	61	B0	F0	C0	A1	70	50	50	E1	30	20	40
22	70	00	F0	62	B0	F0	D0	A2	70	60	50	E2	30	20	40
23	B0	00	F0	63	B0	F0	E0	A3	70	60	50	E3	30	20	40
24	F0	00	F0	64	B0	F0	F0	A4	70	70	50	E4	40	20	40
25	F0	00	B0	65	B0	E0	F0	A5	60	70	50	E5	40	20	30
26	F0	00	70	66	B0	D0	F0	A6	60	70	50	E6	40	20	30
27	F0	00	40	67	B0	C0	F0	A7	50	70	50	E7	40	20	30
28	F0	00	00	68	00	00	70	A8	50	70	50	E8	40	20	20
29	F0	40	00	69	10	00	70	A9	50	70	50	E9	40	30	20
2A	F0	70	00	6A	30	00	70	AA	50	70	60	EA	40	30	20
2B	F0	B0	00	6B	50	00	70	AB	50	70	60	EB	40	30	20
2C	F0	F0	00	6C	70	00	70	AC	50	70	70	EC	40	40	20
2D	B0	F0	00	6D	70	00	50	AD	50	60	70	ED	30	40	20
2E	70	F0	00	6E	70	00	30	AE	50	60	70	EE	30	40	20
2F	40	F0	00	6F	70	00	10	AF	50	50	70	EF	30	40	20
30	00	F0	00	70	70	00	00	B0	00	00	40	F0	20	40	20
31	00	F0	40	71	70	10	00	B1	10	00	40	F1	20	40	30
32	00	F0	70	72	70	30	00	B2	20	00	40	F2	20	40	30
33	00	F0	B0	73	70	50	00	B3	30	00	40	F3	20	40	30
34	00	F0	F0	74	70	70	00	B4	40	00	40	F4	20	40	40
35	00	B0	F0	75	50	70	00	B5	40	00	30	F5	20	30	40
36	00	70	F0	76	30	70	00	B6	40	00	20	F6	20	30	40
37	00	40	F0	77	10	70	00	B7	40	00	10	F7	20	30	40
38	70	70	F0	78	00	70	00	B8	40	00	00	F8	00	00	00
39	90	70	F0	79	00	70	10	B9	40	10	00	F9	00	00	00
3A	B0	70	F0	7A	00	70	30	BA	40	20	00	FA	00	00	00
3B	D0	70	F0	7B	00	70	50	BB	40	30	00	FB	00	00	00
3C	F0	70	F0	7C	00	70	70	BC	40	40	00	FC	00	00	00
3D	F0	70	D0	7D	00	50	70	BD	30	40	00	FD	00	00	00
3E	F0	70	B0	7E	00	30	70	BE	20	40	00	FE	00	00	00
3F	F0	70	90	7F	00	10	70	BF	10	40	00	FF	00	00	00

4.2.2 Gray Shade Modes

Gray shade modes are monochrome display modes. Monochrome display modes use the Look-Up Table in a very similar fashion to the color modes. This most significant difference is that the monochrome display modes use only the intensity of the green element of the Look-Up Table to form the gray level.

One side effect of using only green for intensity selection is that in gray shade modes there are only sixteen possible intensities. 8 bit-per-pixel is not supported for gray shade modes.

1 bpp gray shade

When the S1D13705 is configured for 1 bpp gray shade mode, the LUT is limited to selecting colors from the first two green entries. The two LUT entries can be set to any of sixteen possible intensities. Typically they would be set to 0h (black) and Fh (white).

Each byte in the display buffer contains eight adjacent pixels. If a bit has a value of "0" then the color in the green LUT 0 index is displayed. A bit value of "1" results in the color in green LUT 1 index being displayed.

The following table shows the recommended values 1 bpp gray shade display mode.

Table 4-5: Recommended LUT Values for 1 Bpp Gray Shade

Address	Red	Green	Blue
00	00	00	00
01	00	F0	00
02	00	00	00
...	00	00	00
FF	00	00	00

🟨	unused entries

2 bpp gray shade

When the S1D13705 is configured for 2 bpp gray shade, the displayed colors are selected from the first four green entries in the Look-Up Table. The remaining entries of the LUT are unused. Each of the four entries can be set to any of the sixteen possible colors.

Each byte in the display buffer contains four adjacent pixels. If a bit combination has a value of "00" then the intensity in the green LUT index 0 is displayed. A bit value of "01" results in the intensity represented by the green in LUT index 1 being displayed. Likewise the bit combination of "10" displays from the third LUT entry and "11" displays a from the fourth LUT entry.

The following table shows the example values for 2 bit-per-pixel display mode.

Table 4-6: Suggested Values for 2 Bpp Gray Shade

Index	Red	Green	Blue
0	00	00	00
1	00	50	00
2	00	A0	00
3	00	F0	00
4	00	00	00
...	00	00	00
FF	00	00	00

🟨	indicates unused entries

4 bpp gray shade

When the S1D13705 is configured for 4 bpp gray shade mode the displayed colors are selected from the green values of the first sixteen entries of the Look-Up Table. Each of the sixteen entries can be set to any of the sixteen possible intensity levels.

Each byte in the display buffer contains two adjacent pixels. If a nibble pattern is “0000” then the green intensity of LUT index 0 is displayed. A nibble value of “0001” results in the green intensity in LUT index 1 being displayed. The pattern continues to the nibble pattern of “1111” which results in the sixteenth intensity of Look-Up Table being displayed.

The following table shows the example values for 4 bit-per-pixel display mode.

Table 4-7: Suggested LUT Values for 4 Bpp Gray Shade

Index	Red	Green	Blue
00	00	00	00
01	00	10	00
02	00	20	00
03	00	30	00
04	00	40	00
05	00	50	00
06	00	60	00
07	00	70	00
08	00	80	00
09	00	90	00
0A	00	A0	00
0B	00	B0	00
0C	00	C0	00
0D	00	D0	00
0E	00	E0	00
0F	00	F0	00
10	00	00	00
...	00	00	00
FF	00	00	00

🟨	indicates unused entries

5 Advanced Techniques

This section contains programming suggestions for the following:

- virtual display
- panning and scrolling
- split screen display

5.1 Virtual Display

Virtual display refers to the situation where the image to be viewed is larger than the physical display. The difference can be in the horizontal, vertical or both dimensions. To view the image, the display is used as a window into the display buffer. At any given time only a portion of the image is visible. Panning and scrolling are used to view the full image.

The Memory Address Offset register determines the number of horizontal pixels in the virtual image. The offset register can be used to specify from 0 to 255 additional words for each scan line. At 1 bpp, 255 words span an additional 4,080 pixels. At 8 bpp, 255 words span an additional 510 pixels.

The maximum vertical size of the virtual image is the result of dividing 81920 bytes of display memory by the number of bytes on each line (i.e. at 1 bpp with a 320x240 panel set for a virtual width of 640x480 there is enough memory for 1024 lines).

Figure 5-1: “Viewport Inside a Virtual Display,” depicts a typical use of a virtual display. The display panel is 320x240 pixels, an image of 640x480 pixels can be viewed by navigating a 320x240 pixel viewport around the image using panning and scrolling.

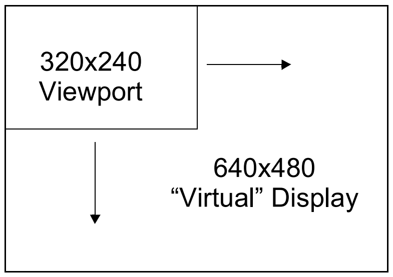


Figure 5-1: Viewport Inside a Virtual Display

5.1.1 Registers

REG[11h] Memory Address Offset Register							
Memory Address Offset Bit 7	Memory Address Offset Bit 6	Memory Address Offset Bit 5	Memory Address Offset Bit 4	Memory Address Offset Bit 3	Memory Address Offset Bit 2	Memory Address Offset Bit 1	Memory Address Offset Bit 0

Memory Address Offset Register

REG[11h] forms an 8-bit value called the Memory Address Offset. This offset is the number of additional words on each line of the display. If the offset is set to zero there is no virtual width.

Note

This value does not represent the number of words to be shown on the display. The display width is set in the Horizontal Display Width register.

5.1.2 Examples

Example 1: In this example we go through the calculations to display a 640x480 image on a 320x240 panel at 2 bpp.

Step 1: Calculate the number of pixels per word for this color depth.

At 2 bpp each byte is comprised of 4 pixels, therefore each word contains 8 pixels.

pixels_per_word = 16 / bpp = 16 / 2 = 8

Step 2: Calculate the Memory Address Offset register value

We require a total of 640 pixels. The horizontal display register will account for 320 pixels, this leaves 320 pixels for the Memory Address Offset register to account for.

offset = pixels / pixels_per_word = 320 / 8 = 40 = 28h

The Memory Address Offset register, REG[11h], will have to be set to 28h to satisfy the above condition.

Example 2: From the above, what is the maximum number of lines our image can contain?

Step 1: Calculate the number of bytes on each line.

bytes_per_line = pixels_per_line / pixels_per_byte = 640 / 4 = 160

Each line of the display requires 160 bytes.

Step 2: Calculate the number of lines the S1D13705 is capable of.

total_lines = memory / bytes_per_line = 81920 / 160 = 512

We can display a maximum of 512 lines. Our example image requires 480 lines so this example can be done.

5.2 Panning and Scrolling

Panning and scrolling describe the operation of moving a physical display viewport about a virtual image in order to view the entire image a portion at time. For example, after setting up the previous example (virtual display) and drawing an image into it we would only be able to view one quarter of the image. Panning and scrolling are used to reveal the rest of the image.

Panning describes the horizontal (side to side) motion of the viewport. When panning to the right the image in the viewport appears to slide to the left. When panning to the left the image to appears to slide to the right. Scrolling describes the vertical (up and down) motion of the viewport. Scrolling down causes the image to appear to slide up and scrolling up causes the image to appear to slide down.

Both panning and scrolling are performed by modifying the start address register. The start address registers in the S1D13705 are a word offset to the data to be displayed in the top left corner of a frame. Changing the start address by one means a change on the display of the number of pixels in one word. The number of pixels in word varies according to the color depth. At 1 bit-per-pixel a word contains sixteen pixels. At 2 bit-per-pixel there are eight pixels, at 4 bit-per-pixel there are four pixels and at 8 bit-per-pixel there is two pixels in each word. The number of pixels in each word represent the finest step we can pan to the left or right.

When portrait mode (see Hardware Rotation on page 35) is enabled the start address registers become offsets to bytes. In this mode the step rate for the start address registers if halved making for smoother panning.

5.2.1 Registers

REG[0Ch] Screen 1 Display Start Address 0 (LSB)							
Start Addr Bit 7	Start Addr Bit 6	Start Addr Bit 5	Start Addr Bit 4	Start Addr Bit 3	Start Addr Bit 2	Start Addr Bit 1	Start Addr Bit 0

REG[0Dh] Screen 1 Display Start Address 1 (MSB)							
Start Addr Bit 15	Start Addr Bit 14	Start Addr Bit 13	Start Addr Bit 12	Start Addr Bit 11	Start Addr Bit 10	Start Addr Bit 9	Start Addr Bit 8

REG[10h] Screen 1 Display Start Address 2 (MSB)							
n/a	n/a	n/a	n/a	n/a	n/a	n/a	Start Addr Bit 16

Screen 1 Start Address Registers

These three registers form the seventeen bit screen 1 start address. Screen 1 is displayed starting at the top left corner of the display.

In landscape mode these registers form the word offset to the first byte in display memory to be displayed in the upper left corner of the screen. Changing these registers by one will shift the display image 2 to 16 pixels, depending on the current color depth.

In portrait mode these registers form the offset to the display memory byte where screen 1 will start displaying. Changing these registers in portrait mode will result in a shift of 1 to 8 pixels depending on the color depth.

Refer to Table 5-1: “Number of Pixels Panned Using Start Address” to see the minimum number of pixels affected by a change of one to these registers

Table 5-1: Number of Pixels Panned Using Start Address

Color Depth (bpp)	Pixels per Word	Landscape Mode Number of Pixels Panned	Pixels Per Byte	Portrait Mode Number of Pixels Panned
1	16	16	8	8
2	8	8	4	4
4	4	4	2	2
8	2	2	1	1

5.2.2 Examples

For the following examples we base our calculations on a 4 bit-per-pixel image displayed on a 256w x 64h panel. We have set up a virtual size of 320w x 240h. Width is greater than height so we are in landscape display mode. Refer to Section 2, “Initialization” on page 6 and Section 5.1, “Virtual Display” on page 23 for assistance with these settings.

These examples are shown using a C-like syntax.

Example 3: Panning (Right and Left)

To pan to the right increase the start address value by one. To pan to the left decrease the start address value. Keep in mind that, with the exception of 8 bit-per-pixel portrait display mode, the display will jump by more than one pixel as a result of changing the start address registers.

Panning to the right.

```
StartWord = GetStartAddress();
StartWord ++;
SetStartAddress(StartWord);
```

Panning to the left.

```
StartWord = GetStartAddress();
StartWord --;
if (StartWord < 0)
      StartWord = 0;
SetStartAddress(StartWord);
```

The routine GetStartAddress() is one which will read the start address registers and return the start address as a long value. It would be written similar to:

```
long GetStartAddress()
{
      return ((REG[10] & 1) * 65536) + (REG[0D] * 256) + (REG[0C]);
}
```

The routine SetStartAddress() break up its long integer argument into three register values and store the values.

```
void SetStartAddress(long SA)
{
      REG[0C] =  SA        & 0xFF;
      REG[0D] = (SA >>  8) & 0xFF;
      Reg[10] = (SA >> 16) & 0xFF;
}
```

In this example code the notation REG[] refers to whatever mechanism is employed to read/write the registers.

Example 4: Scrolling (Up and Down)

To scroll down, increase the value in the Screen 1 Display Start Address Register by the number of words in one *virtual* scan line. To scroll up, decrease the value in the Screen 1 Display Start Address Register by the number of words in one *virtual* scan line. A virtual scan line includes both the number of bytes required by the physical display and any extra bytes that may be being used for creating a virtual width on the display.

The previous dimensions are still in effect for this example (i.e. 320w x 240h virtual size, 256h x 64w physical size at 4 bpp)

Step 1: Determine the number of words in one virtual scanline.

bytes_per_line = pixels_per_line / pixels_per_byte = 320 / 2 = 160

words_per_line = bytes_per_line / 2 = 160 /2 = 80

Step 2: Scroll up or down

To scroll up.

```
StartWord = GetStartAddress();
StartWord -= words_per_line;
if (StartWord < 0)
      StartWord = 0;
SetStartAddress(StartWord);
```

To scroll down.

```
StartWord = GetStartAddress();
StartWord += words_per_line;
SetStartAddress(StartWord);

      }
```

5.3 Split Screen

Occasionally the need arises to display two different but related images. Take, for example, a game where the main play area requires rapid updates and game status, displayed at the bottom of the screen, requires infrequent updates.

The Split Screen feature of the S1D13705 allows a programmer to setup a display in such a manor. When correctly configured the programmer has only to update the main area on a regular basis. Occasionally, as the need arises, the secondary area is updated.

The figure below illustrates how a 320x240 panel may be configured to have one image displaying from scan line 0 to scan line 199 and image 2 displaying from scan line 200 to scan line 239. Although this example picks specific values, the split between image 1 and image 2 may occur at any line of the display.

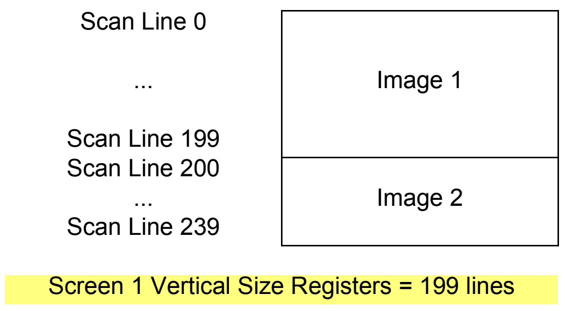


Figure 5-2: 320x240 Single Panel For Split Screen

In split screen operation “Image 1" is taken from the display memory location pointed to by the Screen 1 Start Address registers and is always located at the top of the screen. “Image 2" is taken from the display memory location pointed to by the Screen 2 Start Address registers. The line where “Image 1" end and “Image 2" begins is determined by the Screen 1 Vertical Size register.

5.3.1 Registers

Split screen operation is performed primarily by manipulating three register sets. Screen 1 Start Address and Screen 2 Start Address determine from where in display memory the first and second images will be taken from. The Vertical Size registers determine how many lines Screen 1 will use. The following is a description of the registers used to do split screen.

REG[12] Screen 1 Vertical Size (LSB)							
Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0

REG[13] Screen 1 Vertical Size (MSB)							
n/a	n/a	n/a	n/a	n/a	n/a	Bit 9	Bit 8

Screen 1 Vertical Size

These two registers form a ten bit value which determines the size of screen 1. When the vertical size is equal to or greater than the physical number of lines being displayed there is no visible effect on the display. When the vertical size value is less than the number of physical display lines, operation is like this:

1. From the beginning of a frame to the number of lines indicated by vertical size the display data will come from the memory area pointed to by the Screen 1 Display Start Address.

2. After *vertical size* lines have been displayed the system will begin displaying data from the memory area pointed to by Screen 2 Display Start Address.

On thing that must be pointed out here is that Screen 1 memory is **always** displayed at the top of the screen followed by screen 2 memory. This relationship holds true regardless of where in display memory Screen 1 Start Address and Screen 2 Start Address are pointing. For instance, Screen 2 Start Address may point to offset zero of display memory while Screen 1 Start Address points to a location several thousand bytes higher. Screen 1 will still be shown first on the display. While not particularly useful, it is even possible to set screen 1 and screen 2 to the same address.

REG[0Eh] Screen 2 Display Start Address 0 (LSB)							
Start Addr Bit 7	Start Addr Bit 6	Start Addr Bit 5	Start Addr Bit 4	Start Addr Bit 3	Start Addr Bit 2	Start Addr Bit 1	Start Addr Bit 0

REG[0Fh] Screen 2 Display Start Address 1 (MSB)							
Start Addr Bit 15	Start Addr Bit 14	Start Addr Bit 13	Start Addr Bit 12	Start Addr Bit 11	Start Addr Bit 10	Start Addr Bit 9	Start Addr Bit 8

Screen 2 Start Address Registers

These three registers form the seventeen bit Screen 2 Start Address. Screen 2 is always displayed immediately following the screen 1 data and will begin at the left-most pixel on a line. Keep in mind that if the Screen 1 Vertical Size is equal to or greater than the physical display then Screen 2 will not be shown.

In landscape mode these registers form the word offset to the first byte in display memory to be displayed. Changing these registers by one will shift the display image 2 to 16 pixels, depending on the current color depth.

The S1D13705 does not support split screen operation in portrait mode. Screen 2 will never be used if portrait mode is selected.

Refer to Table 5-1: “Number of Pixels Panned Using Start Address” to see the minimum number of pixels affected by a change of one to these registers

Screen 1 Start Address registers, REG[0C], REG[0D] and REG[10] are discussed in Section 5.2.1 on page 26

5.3.2 Examples

Example 5: Display 200 scanlines of image 1 and 40 scanlines of image 2. Image 2 is located first (offset 0) in the display buffer followed immediately by image 1. Assume a 320x240 display and a color depth of 4 bpp.

1. Calculate the Screen 1Vertical Size register values.

   vertical_size = 200 = C8h

   Write the Vertical Size LSB, REG[12h], with C8h and Vertical Size MSB, REG[13h], with a 00h.

2. Calculate the Screen 1 Start Word Address register values.

   Screen 2 is located first in display memory, therefore we must calculate the number of bytes taken up by the screen 2 data.

   bytes_per_line = pixels_per_line / pixels_per_byte = 320 / 2 = 160

   total bytes = bytes_per_line x lines = 160 x 40 = 6400.

   Screen 2 requires 6400 bytes (0 to 6399) therefore the start address offset for screen 1 must be 6400 bytes. (6400 bytes = 3200 words = C80h words)

   Set the Screen 1 Start Word Address MSB, REG[0Dh], to 0Ch and the Screen 1 Start Word Address LSB, REG[0Ch], to 80h.

3. Calculate the Screen 2 Start Word Address register values.

Screen 2 display data is coming from the very beginning of the display buffer. All there is to do here is ensure that both the LSB and MSB of the Screen 2 Start Word Address registers are set to zero.

6 LCD Power Sequencing and Power Save Modes

6.1 LCD Power Sequencing

Correct power sequencing is required to prevent long term damage to LCD panels and to avoid unsightly “lines” during power-up and power-down. Power Sequencing allows the LCD power supply to discharge prior to shutting down the LCD logic signals.

Proper LCD power sequencing dictates there must be a time delay between the LCD power being disabled and the LCD signals being shut down. During power-up the LCD signals must be active prior to or when power is applied to the LCD. The time intervals vary depending on the power supply design.

The S1D13705 performs automatic power sequencing in response to both software power save (REG[03h]) or in response to a hardware power save. One frame after a power save mode is set, the S1D13705 disables LCD power, and the LCD logic signals continue for one hundred and twenty seven frames allowing the LCD power supply to completely discharge. For most applications the internal power sequencing is the appropriate choice.

There may be situations where the internal time delay is insufficient to discharge the LCD power supply before the LCD signals are shut down, or the delay is too long and the designer wishes to shorten it. This section details the sequences to manually power-up and power-down the LCD interface.

6.2 Registers

REG[03h] Mode Register 2							
				LCDPWR Override	Hardware Power Save Enable	Software Power Save bit 1	Software Power Save bit 0

The LCD Power (LCDPWR) Override bit forces LCD power inactive one frame after being toggled. As long as this bit is “1” LCD power will be disabled.

The Hardware Power Save Enable bit must be set in order to activate hardware power save through GPIO0.

The Software Power Save bits set and reset the software power save mode. These bits are set to “11” for normal operation and set to “00” for power save mode.

LCD logic signals to the display panel are active for 128 frames after setting either hardware or software power save modes. Power sequencing override is performed by setting the LCDPWR Override bit some time before setting a power save mode for power off sequences. During power on sequences the power save mode is reset some time before the LCDPWR Override is reset resulting in the LCD logic signals being active before power is applied to the panel.

6.3 LCD Enable/Disable

The descriptions below cover manually powering the LCD panel up and down. Use the sequences described in this section if the power supply connected to the panel requires more than 127 frames to discharge on power-down, or if the panel requires starting the LCD logic well in advance of enabling LCD power. Currently there are no known circumstances where the LCD logic must be active well in advance of LCD power.

Note

If 127 frame period is to long, blank the display, then reprogram the Horizontal and Vertical sizes to produce a shorter frame period before using these methods.

Power On/Enable Sequence

The following is a sequence for manually powering-up an LCD panel if LCD power had to be applied later than LCD logic.

1. Set REG[03h] bit 3 (LCDPWR Override) to “1”. This ensures that LCD power will be held disabled.
2. Enable LCD logic. This is done by either setting the GPIO0 pin low to disable hardware power save mode and/or by setting REG[03h] bits 1-0 to “11” to disable software power save.
3. Count “x” Vertical Non-Display Periods (OPTIONAL).
   “x” corresponds the length of time LCD logic must be enabled before LCD power-up, converted to the equivalent vertical non-display periods. For example, at 72 HZ counting 36 non-display periods results in a one half second delay.
4. Set REG[03h] bit 3 to “0” to enable LCD Power.

Power Off/Disable Sequence

The following is a sequence for manually powering-down an LCD panel. These steps would be used if the power supply discharge requirements are larger than the default 127 frames.

1. Set REG[03h] bit 3 (LCDPWR Override) to “1” which will disable LCD Power.
2. Count “x” Vertical Non-Display Periods.
   “x” corresponds to the power supply discharge time converted to the equivalent vertical non-display periods. (see the previous example)
3. Disable the LCD logic by setting the software power save in REG[03h] or setting hardware power save via GPIO0. Keep in mind that after setting the power save mode there will be 127 frames before the LCD logic signals are disabled.

7 Hardware Rotation

7.1 Introduction To Hardware Rotation

Many of todays applications use the LCD panel in a portrait orientation (typically LCD panels are landscape oriented). In this case it becomes necessary to “rotate” the displayed image. This rotation can be done by software at the expense of performance or, as with the S1D13705, it can be done by hardware with no performance penalty.

This discussion of display rotation is intended to augment the excellent description of the hardware functionality found in the Hardware Functional Specification.

The S1D13705 supports two portrait modes: Default Portrait Mode and Alternate Portrait Mode.

7.2 Default Portrait Mode

Default portrait mode was designed to reduce power consumption for portrait mode use. The reduced power consumption comes with certain trade offs.

The most obvious difference between the two modes is that Default Portrait Mode requires the portrait width be a power of two, e.g. a 240-line panel, used in portrait mode, requires setting a virtual width of 256 pixels. Also default portrait mode is only capable of scrolling the display in two line increments.

The benefits to using default portrait mode lies in the ability to use a slower input clock and in reduced power consumption.

The following figure depicts the ways to envision memory layouts for the S1D13705 in default portrait mode. This example uses a 320x240 panel.

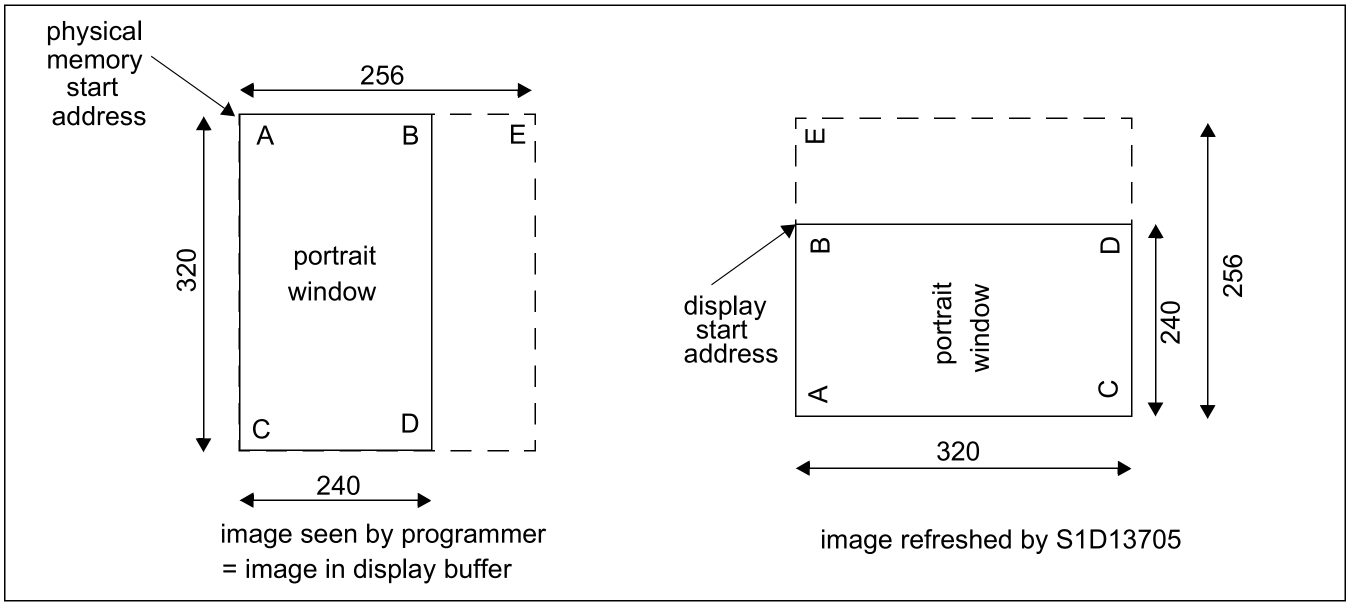

physical memory start address
256
A
B
E
320
portrait window
C
D
240
image seen by programmer = image in display buffer
E
B
D
display start address
portrait window
A
C
240
256
320
image refreshed by S1D13705

Figure 7-1: Relationship Between the Default Mode Screen Image and the Image Refreshed by S1D13705

From the programmers perspective the memory is laid out as shown on the left. The programmer accesses memory exactly as for a panel of with the dimensions of 240x320 setup to have a 256 pixel horizontal stride. The programmer sees memory addresses increasing from A->B and from B->C.

From a hardware perspective the S1D13705 always refreshes the LCD panel in the order B->D and down to do A->C.

7.3 Alternate Portrait Mode

Alternate portrait mode does not impose the power of two line width. To rotated the image on 240 line panel requires a portrait stride of 240 pixels. Alternate portrait mode is capable of scrolling by one line at a time in response to changes to the Start Address Registers. However, to achieve the same frame rate requires a 2 x faster input clock, therefore using more power.

The following figure depicts the ways to envision memory layouts for the S1D13705 in alternate portrait mode. This example also uses a 320x240 panel. Notice that in alternate portrait mode the stride may be as little as 240 pixels.

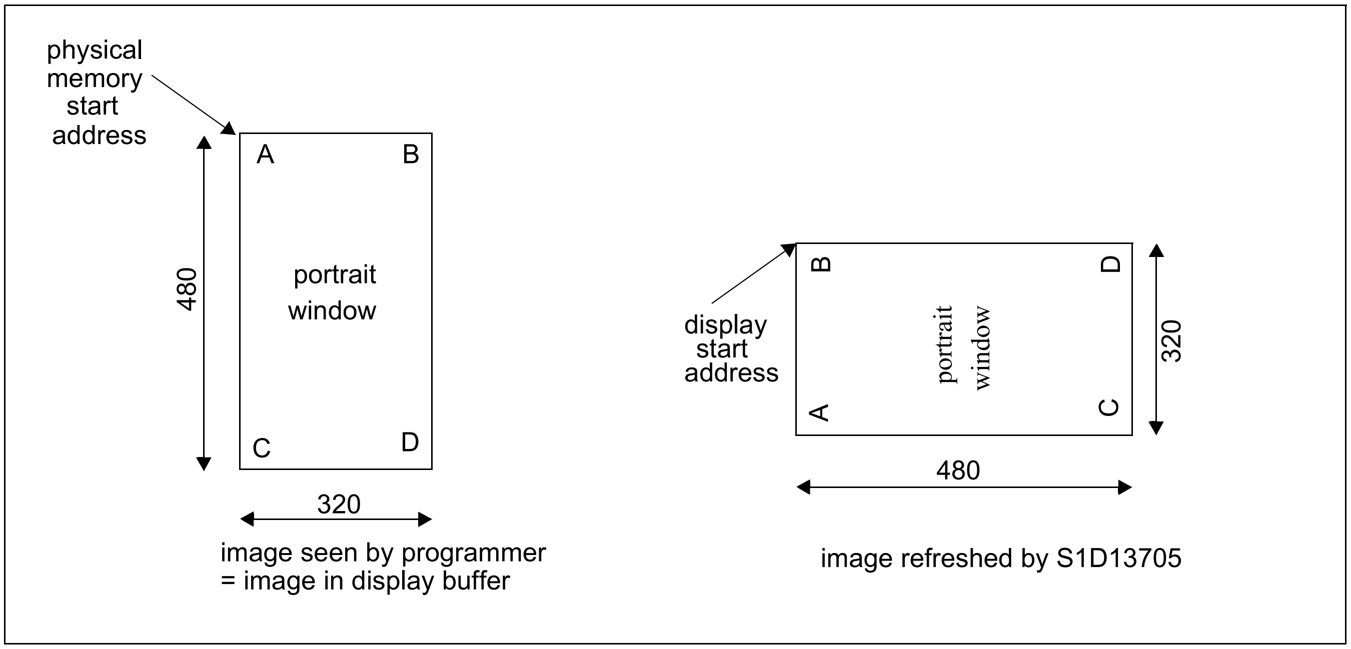

physical memory start address
A
B
480
portrait window
C
D
320
image seen by programmer = image in display buffer
display start address
B
D
portrait window
A
C
320
480
image refreshed by S1D13705

Figure 7-2: Relationship Between the Alternate Mode Screen Image and the Image Refreshed by S1D13705

From the programmers perspective the memory is laid out as shown on the left. The programmer accesses memory exactly as for a panel of with the dimensions of 240x320. The programmer sees memory addresses increasing from A->B and from B->C.

From a hardware perspective the S1D13705 always refreshes the LCD panel in the order B->D and down to do A->C

The greatest factor in selecting alternate portrait mode over default portrait mode would be for the ability to obtain an area of contiguous off screen memory. For example: A 640x480 panel in default portrait mode at two bit-per-pixel requires 81920 bytes (80 Kb). There is unused memory but it is not contiguous. The same situation using alternate portrait mode requires 76800 bytes leaving 5120 bytes of contiguous memory available to the application. In fact the change in memory usage may make the difference between being able to run certain panels in portrait mode or not being able to do so.

7.4 Registers

This section describes the registers used to set portrait mode operation.

REG[0Ch] Screen 1 Start Word Address LSB							
bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0

REG[0Dh] Screen 1 Start Word Address MSB							
bit 15	bit 14	bit 13	bit 12	bit 11	bit 10	bit 9	bit 8

REG[0Eh] Screen 1 Start Word Address MSB							
n/a	n/a	n/a	n/a	n/a	n/a	n/a	bit 16

The Screen 1 Start Address registers must be set correctly for portrait mode. In portrait mode the Start Address registers form a byte offset, as opposed to a word offset, into display memory.

The initial required offset is the portrait mode stride (in bytes) less one.

REG[1Ch] Line Byte Count Register							
bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0

The line byte count register informs the S1D13705 of the stride, in bytes, between two consecutive lines of display in portrait mode. The Line Byte Count register only affects portrait mode operation and are ignored when the S1D13705 is in landscape display mode.

REG[1Bh] Portrait Mode Register							
Portrait Mode Enable	Portrait Mode Select	n/a	n/a	n/a	Portrait Mode Memory Clock Select	Portrait Mode Pixel Clock Select Bit 1	Portrait Mode Pixel Clock Select Bit 0

The portrait mode register contains several items for portrait mode support.

The first is the Portrait Mode Enable bit. When this bit is “0” the S1D13705 is in landscape mode and the remainder of the settings in this register as well as the Line Byte Count in REG[1Ch] are ignored. Set this bit to “1” to enable portrait mode.

The portrait mode select bit selects between the “Default Mode” and the “Alternate Mode”. Setting this bit to “0” selects the default portrait mode while setting this bit to “1” enables the alternate portrait mode.

Portrait Mode Memory Clock Select is another power saving measure which can be enabled if the final MCLK value is less than or equal to 25 MHz. Memory Clock Select results in the S1D13705 temporarily increasing the memory clock circuitry on CPU access and resuming the slower speed when the access is complete. This results in better performance while using the least power.

In portrait display mode the CLKI (input clock) is routed to the portrait section of the S1D13705 as CLK. From the CLK signal the MCLK value can be determined from table 8-8 of the Hardware Functional Specification, document number X27A-A-001-xx. If MCLK is determined to be less than or equal to 25 MHz then Portrait Mode Memory Clock Select may be enabled.

7.5 Limitations

The only limitation to using portrait mode on the S1D13705 is that split screen operation is not supported.

A comparison of the two portrait modes is as follows:

Table 7-1: Default and Alternate Portrait Mode Comparison

Item	Default Portrait Mode	Alternate Portrait Mode
Memory Requirements	The width of the rotated image must be a power of 2. In most cases, a virtual image is required where the right-hand side of the virtual image is unused and memory is wasted. For example, a 320x480x4bpp image would normally require only 76,800 bytes - possible within the 80K byte address space, but the virtual image is 512x480x4bpp which needs 122,880 bytes - not possible.	Does not require a virtual image.
Clock Requirements	CLK need only be as fast as the required PCLK.	MCLK, and hence CLK, need to be 2x PCLK. For example, if the panel requires a 3MHz PCLK, then CLK must be 6MHz. Note that 25MHz is the maximum CLK, so PCLK cannot be higher than 12.5MHz in this mode.
Power Consumption	Lowest power consumption.	Higher than Default Mode.
Panning	Vertical panning in 2 line increments.	Vertical panning in 1 line increments.
Performance	Nominal performance. Note that performance can be increased by increasing CLK and setting MCLK = CLK (REG[1Bh] bit 2 = 1).	Higher performance than Default Mode. Note that performance can be increased by increasing CLK and setting MCLK = CLK (REG[1Bh] bit 2 = 1).

7.6 Examples

Example 6: Enable default portrait mode for a 320x240 panel at 4 bpp.

Before switching to portrait mode from landscape mode, display memory should be cleared to make the user perceived transition smoother. Images in display memory are not rotated automatically by hardware and a garbled image would be visible for a short period of time if video memory is not cleared.

If alternate portrait is used then the CLK signal is divided in half to get the PCLK signal. If the Input Clock Divide bit, in register[02] is set we can simply reset the divider. The result of this is a PCLK of exactly the same frequency as we used for landscape mode and we can use the current horizontal and vertical non-display periods. If the Input Clock Divide bit is not set then we must recalculate the frame rate based on the a PCLK value. In this example we will bypass recalculation of the horizontal and vertical non-display times (frame rate) by selecting the default portrait mode scheme.

1. Calculate and set the Screen 1 Start Word Address register.

   OffsetBytes = (Width x BitsPerPixel / 8) - 1 = (256 x 4 / 8) -1 = 127 = 007Fh

   ("Width" is the width of the portrait mode display - in this case the next power of two greater than 240 pixels or 256.)

   Set Screen1 Display Start Word Address LSB (REG [0Ch]) to 7Fh and Screen1 Display Start Word Address MSB (REG[0Dh]) to 00h.

2. Calculate the Line Byte Count

   The Line Byte Count also must be based on the power of two width.

   LineByteCount = Width x BitsPerPixel / 8 = 256 x 4 / 8 = 128 = 80h.

   Set the Line Byte Count (REG[1C]) to 80h.

3. Enable portrait mode.

   This example uses the default portrait mode scheme. If we do not change the Portrait Mode Pixel Clock Select bits then we will not have to recalculate the non-display timings to correct the frame rate.

   Write 80h to the Portrait Mode Register (REG[1Bh]).

The display is now configured for portrait mode use. Offset zero into display memory will corresponds to the upper left corner of the display. The only item to keep in mind is that the count from the first pixel of one line to the first pixel of the next line (referred to as the "stride") is 128 bytes.

Example 7: Enable alternate portrait mode for a 320x240 panel at 4 bpp.

Note

As we have to perform a frame rate calculation for this mode we need to know the following panel characteristics: 320x240 8-bit color to be run at 80 Hz with a 16 MHz input clock.

As in the previous example, before switching to portrait mode, display memory should be cleared. Images in display memory are not rotated automatically by hardware and the garbled image would be visible for a short period of time if video memory is not cleared.

1. Calculate and set the Screen 1 Start Word Address register.

   OffsetBytes = (Width x BitsPerPixel / 8) - 1 = (240 x 4 / 8) - 1 = 119 = 0077h

   Set Screen1 Display Start Word Address LSB (REG [0Ch]) to 77h and Screen1 Display Start Word Address MSB (REG[0Dh]) to 00h.

2. Calculate the Line Byte Count.

   LineByteCount = Width x BitsPerPixel / 8 = 240 x 4 / 8 = 120 = 78h.

   Set the Line Byte Count (REG[1C]) to 78h.

3. Enable portrait mode.

   This example uses the alternate portrait mode scheme. We will not change the MCLK Autoswitch or Pixel Clock Select settings.

   Write C0h to the Portrait Mode register (REG[1Bh])

4. Recalculate the frame rate dependents.

This example assumes the alternate portrait mode scheme. In this scheme, without touching the Pixel Clock Select bits the PCLK value will be equal to CLK/2.

These examples don't use the Pixel Clock Select bits. The ability to divide the PCLK value down further than the default values was added to the S1D13705 to support hardware portrait mode on very small panels.

The Pixel Clock value has changed so we must calculate horizontal and vertical non-display times to reach the desired frame rate. Rather than perform the frame rate calculations here I will refer the reader to the frame rate calculations in Frame Rate Calculation on page 7 and simply "arrive" at the following:

Horizontal Non-Display Period = 88h

Vertical Non-Display Period = 03h

Plugging the values into the frame rate calculations yields:

$$\mathrm{FrameRate} = \frac{\mathrm{PCLK}}{(\mathrm{HDP} + \mathrm{HNDP}) \times (\mathrm{VDP} + \mathrm{VNDP})}$$

$$\mathrm{FrameRate} = \frac{\frac{16,000,000}{2}}{(320 + 88) \times (240 + 3)} = 80.69$$

For this example the Horizontal Non-Display register [REG[08h]) needs to be set to 07h and the Vertical Non-Display register (REG[0Ah]) needs to be set to 03h.

The 16,000,000/2 in the formula above represents the input clock being divided by two when this alternate portrait mode is selected. With the values given for this example we must ensure the Input Clock Divide bit (REG[02h] b4) is reset (with the given values it was likely set as a result of the frame rate calculations for landscape display mode).

No other registers need to be altered.

The display is now configured for portrait mode use. Offset zero of display memory corresponds to the upper left corner of the display. Display memory is accessed exactly as it was for landscape mode.

As this is the alternate portrait mode the power of two stride issue encountered with the default portrait mode is no longer an issue. The stride is the same as the portrait mode width. In this case 120 bytes.

Example 8:* *Pan the above portrait mode image to the right by 4 pixels then scroll it up by 6 pixels.

To pan by four pixels the start address needs to be advanced.

1. Calculate the number of bytes to change start address by.

   Bytes = Pixels x BitsPerPixel / 8 = 4 x 4 / 8 = 2 bytes

2. Increment the start address registers by the just calculated value.

   In this case the value write to the start address register will be 81h (7Fh + 2 = 81h)

To scroll by 4 lines we have to change the start address by the offset of four lines of display.

1. Calculate the number of bytes to change start address by.

   BytesPerLine = LineByteCount = 128

   Bytes = Lines x BytesPerLine = 4 x 128 = 512 = 200h

2. Increment the start address registers by the just calculated value

   In this case 281h (81h + 200h) will be written to the Screen 1 Start Address register set.

   Set Screen1 Display Start Word Address LSB (REG[0Ch]) to 81h and Screen1 Display Start Word Address MSB (REG[0Dh]) to 02h.

8 Identifying the S1D13705

There are several similar products in the 135X and 137X LCD controller families. Products which can share significant portions of a generic code base. It may be important for a program to identify between products at run time.

Identification of the S1D13705 can be performed any time after the system has been powered up by reading REG[00h], the Revision Code register. The six most significant bits form the product identification code and the two least significant bits form the product revision.

From reset (power on) the steps to identifying the S1D13705 are as follows:

1. Read REG[00h]. Mask off the lower two bits, the revision code, to obtain the product code.
2. The product code for the S1D13705 is 024h.

9 Hardware Abstraction Layer (HAL)

9.1 Introduction

The HAL is a processor independent programming library provided by Epson. The HAL was developed to aid the implementation of internal test programs, and provides an easy, consistent method of programming the S1D13705 on different processor platforms. The HAL also allows for easier porting of programs between S1D1370X products. Integral to the HAL is an information structure (HAL_STRUCT) that contains configuration data on clocks, display modes, and default register values. This structure combined with the utility 13705CFG.EXE allows quick customization of a program for a new target display or environment.

Using the HAL keeps sample code simpler, although some programmers may find the HAL functions to be limited in their scope, and may wish to program the S1D13705 without using the HAL.

9.2 Contents of the HAL_STRUCT

The HAL_STRUCT below is contained in the file "hal.h" and is required to use the HAL library.

```
typedef struct tagHalStruct
{
   char  szIdString[16];
   WORD  wDetectEndian;
   WORD  wSize;
   BYTE  Regs [MAX_REG + 1];
   DWORD dwClkI;          /* Input Clock Frequency (in kHz) */
   DWORD dwDispMem;       /* Starting address of display buffer memory */
   WORD  wFrameRate;      /* Desired panel frame rate */
} HAL_STRUCT;
```

Within the Regs array ia a structure which defines all the registers described in the *S1D13705 Hardware Functional Specification*, document number X27A-A-001-xx. Using the 13705CFG.EXE utility you can adjust the content of the registers contained in HAL_STRUCT to allow for different LCD panel timing values and other default settings used by the HAL. In the simplest case, the program only calls a few basic HAL functions and the contents of the HAL_STRUCT are used to setup the S1D13705 for operation.

9.3 Using the HAL library

To utilize the HAL library, the programmer must include two ".h" files in their code. "Hal.h" contains the HAL library function prototypes and structure definitions, and "appcfg.h" contains the instance of the HAL_STRUCT that is defined in "Hal.h" and configured by 13705CFG.EXE. For a more thorough example of using the HAL see Section 10.1, "Sample code using the S1D13705 HAL API" on page 64.

Note

Many of the HAL library functions have pointers as parameters. The programmer should be aware that little validation of these pointers is performed, so it is up to the programmer to ensure that they adhere to the interface and use valid pointers. Programmers are recommended to use the highest warning levels of their compiler in order to verify the parameter types.

9.4 API for 13705HAL

This section is a description of the HAL library Application Programmers Interface (API). Updates and revisions to the HAL may include new functions not included in this documentation.

Table 9-1: HAL Functions

Function	Description
Initialization:	
seRegisterDevice	Registers the S1D13705 parameters with the HAL, calls seInitHal if necessary. seRegisterDevice MUST be the first HAL function called by an application.
seSetInit	Programs the S1D13705 for use with the default settings, calls seSetDisplayMode to do the work, clears display memory. Note: either seSetInit or seSetDisplayMode must be called AFTER calling seRegisterDevice
General HAL Support:	
seGetId	Interpret the revision code register to determine chip id
seGetHalVersion	Return version information on the HAL library
seGetLastUsableByte	Determine the offset of the last unreserved usable byte in the display buffer
seGetBytesPerScanline	Determine the number of bytes or memory consumed per scan line in current mode
seGetScreenSize	Determine the height and width of the display surface in pixels
seDelay	Use the frame rate timing to delay for required seconds (requires registers to be initialized)
seSetHighPerformance	Used in color modes less than 8-bpp to toggle the high performance bit on or off
Advanced HAL Functions:	
seSplitInit	Initialize split screen variables and setup start addresses
seSplitScreen	Set the size of either the top or bottom screen
seVirtInit	Initialize virtual screen mode setting x and y sizes
seVirtMove	pan/scroll the virtual screen surface(s)
Hardware Rotate:	
seSetHWRotate	Set the hardware rotation to either Portrait or Landscape
seSetPortraitMethod	Call before setting hardware portrait mode to set either Default or Alternate Portrait Mode

Table 9-1: HAL Functions (Continued)

Function	Description
Register / Memory Access:	
seSetReg	Write a Byte value to the specified S1D13705 register
seGetReg	Read a Byte value from the specified S1D13705 register
seWriteDisplayBytes	Write one or more bytes to the display buffer at the specified offset
seWriteDisplayWords	Write one or more words to the display buffer at the specified offset
seWriteDisplayDwords	Write one or more dwords to the display buffer at the specified offset
seReadDisplayByte	Read a byte from the display buffer from the specified offset
seReadDisplayWord	Read a word from the display buffer from the specified offset
seReadDisplayDword	Read a dword from the display buffer from the specified offset
Color Manipulation:	
seSetLut	Write to the Look-Up Table (LUT) entries starting at index 0
seGetLut	Read from the LUT starting at index 0
seSetLutEntry	Write one LUT entry (red, green, blue) at the specified index
seGetLutEntry	Read one LUT entry (red, green, blue) from the specified index
seSetBitsPerPixel	Set the color depth
seGetBitsPerPixel	Determine the current color depth
Drawing:	
seSetPixel	Draw a pixel at (x,y) in the specified color
seGetPixel	Read pixel's color at (x,y)
seDrawLine	Draw a line from (x1,y1) to (x2,y2) in specified color
seDrawRect	Draw a rectangle from (x1,y1) to (x2,y2) in specified color
Power Save:	
seSetPowerSaveMode	Control S1D13705 SW power save mode (enable/disable)

9.4.1 Initialization

The following section describes the HAL functions dealing with S1D13705 initialization. Typically a programmer has only to concern themselves with calls to seRegisterDevice() and seSetInit().

int seRegisterDevice(const LPHAL_STRUC lpHalInfo)

Description: This function registers the S1D13705 device parameters with the HAL library. The device parameters include address range, register values, desired frame rate, etc., and are stored in the HAL_STRUCT structure pointed to by lpHalInfo. Additionally this routine allocates system memory as address space for accessing registers and the display buffer.

Parameters: lpHalInfo - pointer to HAL_STRUCT information structure

Return Value: ERR_OK - operation completed with no problems
ERR_UNKNOWN_DEVICE - the HAL was unable to find an S1D13705.

Note
seRegisterDevice() MUST be called before any other HAL functions.
No S1D13705 registers are changed by calling seRegisterDevice().

seSetInit()

Description: Configures the S1D13705 for operation. This function sets all the S1D13705 control registers to their default values.

Initialization of the S1D13705 is a two step process to accommodate those programs (e.g. 13705PLAY.EXE) which do not initialize the S1D13705 on start-up.

Parameters: None

Return Value: ERR_OK - operation completed with no problems

Note
After this call the Look-Up Table will be set to a default state appropriate to the display type.

Unlike S1D1350x HAL versions, this function does not call seSetDisplayMode as this function does not exist in the 13705 HAL.

9.4.2 General HAL Support

Functions in this group do not fit into any specific category of support. They provide a miscellaneous range of support for working with the S1D13705

int seGetId(int * pId)

Description: Reads the S1D13705 revision code register to determine the chip product and revisions. The interpreted value is returned in pID.

Parameters: pId - pointer to an integer which will receive the controller ID.

S1D13705 values returned in pID are:
- ID_S1D13705_REV0
- ID_UNKNOWN

Other HAL libraries will return their respective controller IDs upon detection of their controller.

Return Value: ERR_OK - operation completed with no problems
ERR_UNKNOWN_DEVICE - the HAL was unable to identify the display controller. Returned when pID returns ID_UNKNOWN.

void seGetHalVersion(const char ** pVersion, const char ** pStatus, const char **pStatusRevision)

Description: Retrieves the HAL library version. The return pointers are all to ASCII strings. A typical return would be: *pVersion == "1.01" (HAL version 1.01),*pStatus == "B" (The 'B' is the beta designator), *pStatusRevision == "5". The programmer need only create pointers of const char type to pass as parameters (see Example below).

Parameters:

pVersion	- Pointer to string to return the version in.
	- must point to an allocated string of size VER_SIZE
pStatus	- Pointer to a string to return the release status in.
	- must point to an allocated string of size STATUS_SIZE
pStatusRevision	- Pointer to return the current revision of status.
	- must point to an allocated string of size STAT_REV_SIZE

Return Value: None

Example:

```
const char *pVersion, *pStatus, *pStatusRevision;
seGetHalVersion( &pVersion, &pStatus, &pStatusRevision);
```

int seSetBitsPerPixel(int BitsPerPixel)

Description: This routine sets the display color depth.

After performing validity checks to ensure the requested video mode can be set the appropriate registers are changed and the Look-Up Table is set its default values appropriate to the color depth.

This call is similar to a mode set call on a standard VGA.

Parameter: BitsPerPixel - desired color depth in bits per pixel.
- Valid arguments are: 1, 2, 4, and 8.

Return Value: ERR_OK - operation completed with no problems
ERR_FAILED- possible causes for this error include:

1) the desired frame rate may not be attainable with the specified input clock
2) the combination of width, height and color depth may require more memory than is available on the S1D13705.

int seGetBitsPerPixel(int * pBitsPerPixel)

Description: This function reads the S1D13705 registers to determine the current color depth and returns the result in *pBitsPerPixel*.

Parameters: pBitsPerPixel - pointer to an integer to receive current color depth.
- return values will be: 1, 2, 4, or 8.

Return Value: ERR_OK - operation completed with no problems

int seGetBytesPerScanline(int * pBytes)

Description: Determines the number of bytes per scan line of current display mode. It is assumed that the registers have already been correctly initialized before seGetBytesPer-Scanline() is called (i.e. after initializing the HAL, setting the Display mode and adjusting the bits per pixel or other values).

The number of bytes per scanline will include non-displayed bytes if the screen width is greater the display width, or in Default Portrait Mode.

Parameters: pBytes - pointer to an integer to receive the number of bytes per scan line

Return Value: ERR_OK - operation completed with no problems

int seGetScreenSize(int * Width, int * Height)

Description: Retrieves the width and height in pixels of the display surface. The width and height are derived by reading the horizontal and vertical size registers and calculating the dimensions. Virtual dimensions are not taken into account for this calculation.

When the display is in portrait mode the dimensions will be swapped. (i.e. a 640x480 display in portrait mode will return a width of 480 and height of 640).

Parameters:

Width	- pointer to an integer to receive the display width
Height	- pointer to an integer to receive the display height

Return value:

ERR_OK	- the operation completed successfully

int seDelay(int MilliSeconds)

Description: This function will delay for the length of time specified in “MilliSeconds” before returning to the caller.

This function was originally intended for non-PC platforms. Information about how to access the timers was not always available however we do know frame rate and can use that for timing calculations.

The S1D13705 registers must be initialized for this function to work correctly. On the PC platform this is simply a call to the C timing functions and is therefore independent of the register settings.

Parameters: MilliSeconds- time to delay in seconds

Return Value:

ERR_OK	- operation completed with no problems
ERR_FAILED	- returned on non-PC platforms when the S1D13705 registers have not bee initialized

int seGetLastUsableByte(long * plLastByte)

Description: This functions returns a pointer, as a long integer, to the last byte of usable display memory.

The returned value never changes for the S1D13705.

Parameters:

plLastByte	- pointer to a long integer to receive the offset to the last byte of display memory

Return Value:

ERR_OK	- operation completed with no problems

int seSetHighPerformance(BOOL OnOff)

Description: This function call enables or disable the high performance bit of the S1D13705.

When high performance is enabled then MClk equals PClk for all video display resolutions. In the high performance state CPU to video memory performance is improved at the cost of higher power consumption.

When high performance is disabled then MClk ranges from PClk/1 at 8 bit-per-pixel to PClk/8 at 1 bit-per-pixel. Without high performance CPU to video memory speeds are slower and the S1D13705 uses less power.

Parameters: OnOff - a boolean value (defined in HAL.H) to indicate whether to enable of disable high performance.

Return Value: ERR_OK - operation completed with no problems

9.4.3 Advanced HAL Functions

Advanced HAL functions include the functions to support split, virtual and rotated displays. While the concept for using these features is advanced the HAL makes actually using them easy.

int seSetPortraitMethod(int Style)

Description: This selects the portrait mode method to be used when seSetHWRotate() is called to put the S1D13705 into portrait mode.

Parameters: Style - call with style set to DEFAULT (-1) to select Default Portrait Mode
- call with style set to any other value to select Alternate Portrait Mode.

Return Value: ERR_OK - operation completed with no problems
ERR_FAILED - the operation failed.

int seSetHWRotate(int Rotate)

Description: This function sets the rotation scheme according to the value of 'Rotate'. When portrait mode is selected as the display rotation the scheme selected is the 'non-X2' scheme.

Parameters: Rotate - the direction to rotate the display
- Valid arguments for Rotate are: LANDSCAPE and PORTRAIT.

Return Value: ERR_OK - operation completed with no problems
ERR_FAILED - the operation failed to complete.
The most likely reason for failing to set a rotate mode is an inability to set the desired frame rate when setting portrait mode. Other factors which can cause a failure include having a 0 Hz frame rate or specifying a value other than LANDSCAPE or PORTRAIT for the rotation scheme.

int seSplitInit(WORD Scrn1Addr, WORD Scrn2Addr)

Description: This function prepares the system for split screen operation. In order for split screen to function the starting address in the display buffer for the upper portion(screen 1) and the lower portion (screen 2) must be specified. Screen 1 is always displayed above screen 2 on the display regardless of the location of their start addresses.

Parameters: Scrn1Addr	- offset, in bytes, to the start of screen 1
Scrn2Addr	- offset, in bytes, to the start of screen 2

Return Value: ERR_OK - operation completed with no problems

Note

It is assumed that the system has been initialized prior to calling seSplitInit().

int seSplitScreen(int Screen, int VisibleScanlines)

Description: Changes the relevant registers to adjust the split screen according to the number of visible lines requested. 'WhichScreen' determines which screen, 1 or 2, to base the changes on.

The smallest surface screen 1 can display is one line. This is due to the way the S1D13705 operates. Setting Screen 1 Vertical Size to zero results in one line of screen 1 being displayed. The remainder of the display will be screen 2 image.

Parameters: Screen	- must be set to 1 or 2 (or use the constants SCREEN1 or SCREEN2)
VisibleScanlines	- number of lines to display for the selected screen

Return Value: ERR_OK	- operation completed with no problems
ERR_HAL_BAD_ARG	- argument VisibleScanlines is negative or is greater than vertical panel size or WhichScreen is not SCREEN1 or SCREEN 2.

Note

Changing the number of lines for one screen will also change the number of lines for the other screen.

seSplitInit() must be called before calling seSplitScreen().

int seVirtInit(DWORD VirtX, DWORD * VirtY)

Description: This function prepares the system for virtual screen operation. The programmer passes the desired virtual width in pixels. When the routine returns *VirtY* will contain the maximum number of line that can be displayed at the requested virtual width.

Parameter:

VirtX	- horizontal size of virtual display in pixels. (Must be greater or equal to physical size of display)
VirtY	- pointer to an integer to receive the maximum number of displayable lines of 'VirtX' width.

Return Value: ERR_OK - operation completed with no problems
ERR_HAL_BAD_ARG - returned in three situations:
1) the virtual width (VirtX) is greater than the largest possible width (VirtX varies with color depth and ranges from 4096 pixels wider than the panel at 1 bit-per-pixel down to 512 pixels wider than the panel at 8 bit-per-pixel)
2) the virtual width is less than the physical width or
3) the maximum number of lines becomes less than the physical number of lines

Note
The system must have been initialized prior to calling seVirtInit()

int seVirtMove(int Screen, int x, int y)

Description: This routine pans and scrolls the display. In the case where split screen operation is being used, the Screen argument specifies which screen to move. The x and y parameters specify, in pixels, the starting location in the virtual image for the top left corner of the applicable display.

Parameter:

Screen	- must be set to 1 or 2, or use the constants SCREEN1 or SCREEN2, to identify which screen to base calculations on
x	- new starting X position in pixels
y	- new starting Y position in pixels

Return Value: ERR_OK - operation completed with no problems
ERR_HAL_BAD_ARG- there are several reasons for this return value:
1) WhichScreen is not SCREEN1 or SCREEN2.
2) the y argument is greater than the last available line less the screen height.

Note
seVirtInit() must be been called before calling seVirtMove().

9.4.4 Register / Memory Access

The Register/Memory Access functions provide access to the S1D13705 registers and display buffer through the HAL.

int seGetReg(int Index, BYTE * pValue)

Description: Reads the value in the register specified by index.

Parameters: Index - register index to read
pValue - pointer to a BYTE to receive the register value.

Return Value: ERR_OK - operation completed with no problems

int seSetReg(int Index, BYTE Value)

Description: Writes value specified in Value to the register specified by Index.

Parameters: Index - register index to set
Value - value to write to the register

Return Value: ERR_OK - operation completed with no problems

int seReadDisplayByte(DWORD Offset, BYTE *pByte)

Description: Reads a byte from the display buffer at the specified offset and returns the value in pByte.

Parameters: Offset - offset, in bytes from start of the display buffer, to read from
pByte - pointer to a BYTE to return the value in

Return Value: ERR_OK - operation completed with no problems
ERR_HAL_BAD_ARG - if the value for Addr is greater 80 kb

int seReadDisplayWord(DWORD Offset, WORD *pWord)

Description: Reads a word from the display buffer at the specified offset and returns the value in pWord.

Parameters: Offset - offset, in bytes from start of the display buffer, to read from
pWord - pointer to a WORD to return the value in

Return Value: ERR_OK - operation completed with no problems.
ERR_HAL_BAD_ARG - if the value for Addr is greater than 80 kb.

int seReadDisplayDword(DWORD Offset, DWORD *pDword)

Description: Reads a dword from the display buffer at the specified offset and returns the value in pDword.

Parameters:

Offset	- offset from start of the display buffer to read from
pDword	- pointer to a DWORD to return the value in

Return Value: ERR_OK - operation completed with no problems.
ERR_HAL_BAD_ARG - if the value for Addr is greater than 80 kb.

int seWriteDisplayBytes(DWORD Offset, BYTE Value, DWORD Count)

Description: This routine writes one or more bytes to the display buffer at the offset specified by Offset. If a count greater than one is specified all bytes will have the same value.

Parameters:

Offset	- offset from start of the display buffer to start writing at
Value	- BYTE value to write
Count	- number of bytes to write

Return Value: ERR_OK - operation completed with no problems
ERR_HAL_BAD_ARG - if the value for Addr or the value of Addr plus Count is greater than 80 kb.

Note
There are slight functionality differences between the S1D1370x and the S1D1350x HAL.

int seWriteDisplayWords(DWORD Offset, WORD Value, DWORD Count)

Description: Writes one or more WORDS to the display buffer at the offset specified by Addr. If a count greater than one is specified all WORDS will have the same value.

Parameters:

Offset	- offset from start of the display buffer
Value	- WORD value to write
Count	- number of words to write

Return Value: ERR_OK - operation completed with no problems
ERR_HAL_BAD_ARG - if the value for Addr or if Addr plus Count is greater than 80 kb.

Note
There are slight functionality differences between the S1D1370x and the S1D1350x HAL.

int seWriteDisplayDwords(DWORD Offset, DWORD Value, DWORD Count)

Description: Writes one or more DWORDS to the display buffer at the offset specified by Addr. If a count greater than one is specified all DWORDSs will have the same value.

Parameters:	Offset	- offset from start of the display buffer
	Value	- DWORD value to write
	Count	- number of dwords to write

Return Value: ERR_OK - operation completed with no problems
ERR_HAL_BAD_ARG - if the value for Addr or if Addr plus Count is greater than 80 kb.

Note
There are slight functionality differences between the S1D1370x and the S1D1350x HAL.

9.4.5 Power Save

This section covers the HAL functions dealing with the Power Save features of the S1D13705.

int seSetPowerSaveMode(int PwrSaveMode)

Description: This function sets on the S1D13705's software selectable power save modes.

Parameters: PwrSaveMode - integer value specifying the desired power save mode.

Acceptable values for PwrSaveMode are:
0 - (software power save mode) in this mode registers and memory are read/writable. LCD output is forced low.
3 - (normal operation) all outputs function normally.

Return Value: ERR_OK - operation completed with no problems

9.4.6 Drawing

The Drawing routines cover HAL functions that deal with displaying pixels, lines and shapes.

int seSetPixel(long x, long y, DWORD Color)

Description: Draws a pixel at coordinates (x,y) in the requested color. This routine can be used for any color depth.

Parameters:	x	- horizontal coordinate of the pixel (starting from 0)
	y	- vertical coordinate of the pixel (starting from 0)
	Color	- at 1, 2, 4, and 8 bpp Color is an index into the LUT. At 15 and 16 bpp Color defines the color directly (i.e. rrrrggggggbbbbb for 16 bpp)
Return Value:	ERR_OK	- operation completed with no problems.

int seGetPixel(long x, long y, DWORD *pColor)

Description: Reads the pixel color at coordinates (x,y). This routine can be used for any color depth.

Parameters:	x	- horizontal coordinate of the pixel (starting from 0)
	y	- vertical coordinate of the pixel (starting from 0)
	pColor	- at 1, 2, 4, and 8 bpp pColor points to an index into the LUT. At 15 and 16 bpp pColor points to the color directly (i.e. rrrrggggggbbbbb for 16 bpp)
Return Value:	ERR_OK	- operation completed with no problems.

int seDrawLine(int x1, int y1, int x2, int y2, DWORD Color)

Description: This routine draws a line on the display from the endpoints defined by x1,y1 to the endpoint x2,y2 in the requested 'Color'.

Currently seDrawLine() only draws horizontal and vertical lines.

Parameters:	(x1, y1)	- first endpoint of the line in pixels
	(x2, y2)	- second endpoint of the line in pixels (see note below)
	Color	- color to draw with. 'Color' is an index into the LUT.

Return Value: ERR_OK - operation completed with no problems

Note

Functionality differs from the 135x HAL.

int seDrawRect(long x1, long y1, long x2, long y2, DWORD Color, BOOL SolidFill)

Description: This routine draws and optionally fills a rectangular area of display buffer. The upper right corner is defined by x1,y1 and the lower right corner is defined by x2,y2. The color, defined by *Color*, applies both to the border and to the optional fill.

Parameters:

x1, y1	- top left corner of the rectangle (in pixels)
x2, y2	- bottom right corner of the rectangle (in pixels)
Color	- The color to draw the rectangle outline and fill with - Color is an index into the Look-Up Table.
SolidFill	- Flag whether to fill the rectangle or simply draw the border. - Set to 0 for no fill, set to non-0 to fill the inside of the rectangle

Return Value: ERR_OK - operation completed with no problems.

9.4.7 LUT Manipulation

These functions deal with altering the color values in the Look-Up Table.

int seSetLut(BYTE *pLut, int Count)

Description: This routine writes one or more LUT entries. The writes always start with Look-Up Table index 0 and continue for 'Count' entries.

A Look-Up Table entry consists of three bytes, one each for Red, Green, and Blue. The color information is stored in the four most significant bits of each byte.

Parameters:

pLut	- pointer to an array of BYTE lut[16][3] lut[x][0] == RED component lut[x][1] == GREEN component lut[x][2] == BLUE component
Count	- the number of LUT entries to write.

Return Value: ERR_OK - operation completed with no problems

int seGetLut(BYTE *pLUT, int Count)

Description: This routine reads one or more LUT entries and puts the result in the byte array pointed to by pLUT.

A Look-Up Table entry consists of three bytes, one each for Red, Green, and Blue. The color information is stored in the four most significant bits of each byte.

Parameters:

pLUT	- pointer to an array of BYTE lut[16][3] - pLUT must point to enough memory to hold 'Count′ x 3 bytes of data.
Count	- the number of LUT elements to read.

Return Value: ERR_OK - operation completed with no problems

int seSetLutEntry(int Index, BYTE *pEntry)

Description: This routine writes one LUT entry. Unlike seSetLut, the LUT entry indicated by 'Index' can be any value from 0 to 255.

A Look-Up Table entry consists of three bytes, one each for Red, Green, and Blue. The color information is stored in the four most significant bits of each byte.

Parameters:	Index	- index to LUT entry (0 to 255)
	pLUT	- pointer to an array of three bytes.

Return Value: ERR_OK - operation completed with no problems

int seGetLutEntry(int index, BYTE *pEntry)

Description: This routine reads one LUT entry from any index.

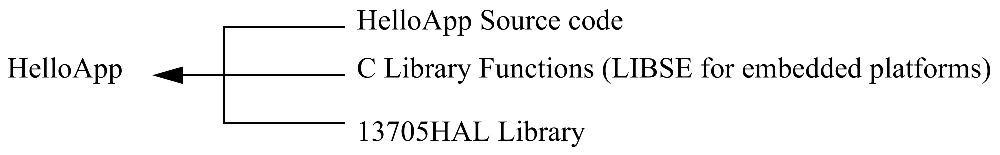
A Look-Up Table entry consists of three bytes, one each for Red, Green, and Blue. The color information is stored in the four most significant bits of each byte.

Parameters:	Index	- index to LUT entry (0 to 255)
	pEntry	- pointer to an array of three bytes

Return Value: ERR_OK - operation completed with no problems

9.5 Porting LIBSE to a new target platform

Building Epson Research and Development applications like a simple HelloApp for a new target platform requires 3 things, the HelloApp code, the 13705HAL library, and a some standard C functions (portable ones are encapsulated in our mini C library LIBSE).



Components needed to build 13705 HAL application

For example, when building HELLOAPP.EXE for the Intel 16-bit platform, you need the HELLOAPP source files, the 13705HAL library and its include files, and some Standard C library functions (which in this case would be supplied by the compiler as part of its run-time library). As this is a DOS .EXE application, you do not need to supply start-up code that sets up the chip selects or interrupts, etc... What if you wanted to build the application for an SH-3 target, one not running DOS?

Before you can build that application to load onto the target, you need to build a C library for the target that contains enough of the Standard C functions (like sprintf and strcpy) to let you build the application. Epson Research and Development supplies the LIBSE for this purpose, but your compiler may come with one included. You also need to build the 13705HAL library for the target. This library is the graphics chip dependent portion of the code. Finally, you need to build the final application, linked together with the libraries described earlier. The following examples assume that you have a copy of the complete source code for the S1D13705 utilities, including the nmake makefiles, as well as a copy of the GNU Compiler v2.7-96q3a for Hitachi SH3. These are available on the World Wide Web at vdc.epson.com.

9.5.1 Building the LIBSE library for SH3 target example

In the LIBSE files, there are three main types of files:

- C files that contain the library functions.
- assembler files that contain the target specific code.
- makefiles that describe the build process to construct the library.

The C files are generic to all platforms, although there are some customizations for targets in the form of #ifdef LCEVBSH3 code (the ifdef used for the example SH3 target Low Cost Eval Board SH3). The majority of this code remains constant whichever target you build for.

The assembler files contain some platform setup code (stacks, chip selects) and jumps into the main entry point of the C code that is contained in the C file entry.c. For our example, the assembler file is STARTSH3.S and it performs only some stack setup and a jump into the code at _mainEntry (entry.c).

In the embedded targets, printf (in file rprintf.c), putchar (putchar.c) and getch (kb.c) resolve to serial character input/output. For SH3, much of the detail of handling serial IO is hidden in the monitor of the evaluation board, but in general the primitives are fairly straight forward, providing the ability to get characters to/from the serial port.

For our target example, the nmake makefile is makesh3.mk. This makefile calls the Gnu compiler at a specific location (TOOLDIR), enumerates the list of files that go into the target and builds a .a library file as the output of the build process.

With nmake.exe in your path run:

nmake -fmakesh3.mk

9.5.2 Building the HAL library for the target example

Building the HAL for the target example is less complex because the code is written in C and requires little platform specific adjustment. The nmake makefile for our example is makesh3.mk.This makefile contains the rules for building sh3 objects, the files list for the library and the library creation rules. The Gnu compiler tools are pointed to by TOOLDIR.

With nmake in your path run:

nmake -fmakesh3.mk

10 Sample Code

Included in the sample code section are two examples of programing the S1D13705. The first sample uses the HAL to draw a red square, wait for user input then rotates to portrait mode and draws a blue square. The second sample code performs the same procedures but directly accesses the registers of the S1D13705. These code samples are for example purposes only.

10.1 Sample code using the S1D13705 HAL API

```
/*
**===========================================================================
** SAMPLE1.C - Sample code demonstrating a program using the S1D13705 HAL.
**---------------------------------------------------------------------------
** Created 1998, Vancouver Design Centre
** Copyright (c) 1998, 1999 Epson Research and Development, Inc.
** All Rights Reserved.
**---------------------------------------------------------------------------
**
** The HAL API code is configured for the following:
**
** 320x240 Single Color 4-bit STN
** 8 bpp - 70 Hz Frame Rate (6 MHz CLKi)
** High Performance enabled
**
**===========================================================================
*/
#include <conio.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "hal.h"                /* Structures, constants and prototypes. */
#include "appcfg.h"             /* HAL configuration information. */
/*-------------------------------------------------------------------------*/
void main(void)
{
        int   ChipId;
        /*
        ** Initialize the HAL.
        ** The call to seRegisterDevice() actually prepares the HAL library
        ** for use. The S1D13705 is not accessed, except to read the revision
        ** code register.
        */
        if (ERR_OK != seRegisterDevice(&HalInfo))
        {
              printf("\nERROR: Could not register S1D13705 device.");
              exit(1);
        }
        /*
        ** Get the product code to verify this is an S1D13705.
        */
        seGetId(&ChipId);
        if (ID_S1D13705_Rev1 != ChipId)
        {
              printf("\nERROR: Did not detect an S1D13705.");
              exit(1);
        }
        /*
        ** Initialize the S1D13705.
        ** This step programs the registers with values taken from
        ** the HalInfo struct in appcfg.h.
        */
        if (ERR_OK != seSetInit())
        {
              printf("\nERROR: Could not initialize device.");
              exit(1);
        }
        /*
        ** The default initialization cleared the display.
        ** Draw a 100x100 red (color 1) rectangle in the upper
        ** left corner (0,0) of the display.
        */
        seDrawRect(0, 0, 100, 100, 1, TRUE);
        /*
        ** Pause here.
        */
        getch();
        /*
        ** Clear the display. Do this by writing 81920 bytes
        */
        seWriteDisplayBytes(0, 0, EIGHTY_K);
        /*
        ** Setup portrait mode.
        */
        seSetHWRotate(PORTRAIT);
        /*
        ** Draw a solid blue 100x100 rectangle in center of the display.
        ** This starting co-ordinates, assuming a 320x240 display is
        ** (320-100)/2 , (240-100)/2 = 110,70.
        */
        seDrawRect(110, 70, 210, 170, 2, TRUE);
        /*
        ** Done!
        */
        exit(0);
}
```

10.2 Sample code without using the S1D13705 HAL API

This second sample demonstrates exactly the same sequence as the first however the HAL is not used, all manipulation is done by directly accessing the registers.

```
/*
**=============================================================================
** SAMPLE2.C - Sample code demonstrating a direct access of the S1D13705.
**-----------------------------------------------------------------------------
** Created 1998, Vancouver Design Centre
** Copyright (c) 1998, 1999 Epson Research and Development, Inc.
** All Rights Reserved.
**-----------------------------------------------------------------------------
**
** The sample code using direct S1D13705 access
** will configure for the following:
**
** 320x240 Single Color 4-bit STN
** 8 bpp color depth - 70 Hz Frame Rate (6 MHz CLKi)
**
** Notes:
** 1) This code is written to be compiled for use under 32-bit
**    Windows. In order to function the vxd file S1D13X0X.VXD must
**    be in the \WINDOWS\SYSTEM directory.
** 2) Register setup is done with discreet writes rather than being table
**    driven. This allows for clear commenting. It is more efficient to
**    loop through the array writing each element to a control register.
** 3) The array of register values as produced by 13705CFG.EXE is included
**    here. I write the registers directly rather than refer to the register
**    array in the sample code.
**
**=============================================================================
*/
#include <conio.h>
#include <windows.h>
#include <winioctl.h>
#include "ioctl.h"
/*
** Look-Up Table - 16 of 256 elements.
** For this sample only the first sixteen LUT elements are set.
*/
unsigned char LUT[16*3] =
{
        0x00, 0x00, 0x00,/* BLACK  */
        0x00, 0x00, 0xA0,/* BLUE   */
        0x00, 0xA0, 0x00,/* GREEN  */
        0x00, 0xA0, 0xA0,/* CYAN   */
        0xA0, 0x00, 0x00,/* RED    */
        0xA0, 0x00, 0xA0,/* PURPLE */
        0xA0, 0xA0, 0x00,/* YELLOW */
        0xA0, 0xA0, 0xA0,/* WHITE  */
        0x00, 0x00, 0x00,/* BLACK  */
        0x00, 0x00, 0xF0,/* LT BLUE   */
        0x00, 0xF0, 0x00,/* LT GREEN  */
        0x00, 0xF0, 0xF0,/* LT CYAN   */
        0xF0, 0x00, 0x00,/* LT RED    */
        0xF0, 0x00, 0xF0,/* LT PURPLE */
        0xF0, 0xF0, 0x00,/* LT YELLOW */
        0xF0, 0xF0, 0xF0/* LT WHITE  */
};
/*
** Register data.
** These values were generated using 13705CFG.EXE.
** The sample code uses these values but does not refer to this array.
** In a typical application these values would be written to the registers
** using a loop.
*/
unsigned char Reg[0x20] =
{
        0x00,  0x23,  0xC0,  0x03,  0x27,  0xEF,  0x00,  0x00,
        0x00,  0x00,  0x03,  0x00,  0x00,  0x00,  0x00,  0x00,
        0x00,  0x00,  0xFF,  0x03,  0x00,  0x00,  0x00,  0x00,
        0x00,  0x00,  0x00,  0x00,  0x00
};
#define MEM_SIZE  0x14000 /* 80 kb display buffer.          */
typedef unsigned short WORD;/* Some useful types */
typedef unsigned long  DWORD;
typedef unsigned char  BYTE;
typedef BYTE       *   PBYTE;
#define LOBYTE(w)           ((BYTE)(w))
#define HIBYTE(w)           ((BYTE)(((WORD)(w) >> 8) & 0xFF))
#define SET_REG(idx, val)   (*(pRegs + idx)) = (val)
/*-------------------------------------------------------------------------*/
void main(void)
{
        PBYTE p13705;
        PBYTE pRegs;
        PBYTE pMem;
        PBYTE pLUT;
        int x, y, tmp;
        int BitsPerPixel = 8;
        int Width      = 320;
        int Height     = 240;
        int OffsetBytes;
        int rc;
        /*
        ** Get a linear address we can use in our code to access the S1D13705.
        ** This is only needed to access the S1D13705 on the ISA eval board.
```

```
*/
DWORD dwLinearAddress;
rc = IntelGetLinAddressW32(0xF00000, &dwLinearAddress);
if (rc != 0)
{
      printf("Error getting linear address");
      return;
}
p13705 = (PBYTE)dwLinearAddress;
pRegs = p13705 + 0x1FFE0;
/*
** Check the revision code. Exit if we don't find an S1D13705.
*/
if (0x24 != *pRegs)
{
      printf("Didn't find an S1D13705");
      return;
}
/*
** Initialize the chip - after initialization the display will be
** setup for landscape use.
** Normally a loop would be used to write the register array near
** the top of this file to the registers.
** For purposes of documenting the sample code, each register write
** is performed individually.
*/
/*
** Register 01h: Mode Register 0 - Color, 8-bit format 2
*/
SET_REG(0x01, 0x20);
/*
** Register 02h: Mode Register 1 - 8BPP
*/
SET_REG(0x02, 0xC0);
/*
** Register 03h: Mode Register 2 - Normal power mode
*/
SET_REG(0x03, 0x03);
/*
** Register 04h: Horizontal Panel Size - 320 pixels - (320/8)-1 = 39 = 27h
*/
SET_REG(0x04, 0x27);
/*
** Register 05h: Vertical Panel Size LSB - 240 pixels
** Register 06h: Vertical Panel Size MSB - (240 - 1) = 239 = EFh
*/
SET_REG(0x05, 0xEF);
SET_REG(0x06, 0x00);
/*
```

```
** Register 07h - FPLINE Start Position - not used by STN
*/
SET_REG(0x07, 0x00);
/*
** Register 08h - Horizontal Non-Display Period = (Reg[08] + 4) * 8
**                                              = (0+4) * 8 = 32 pels
**              - HNDP and VNDP are calculated to achieve the
**                desired frame rate according to:
**
**                                       PCLK
**               Frame Rate = --------------------------
**                            (HDP + HNDP) * (VDP + VNDP)
*/
SET_REG(0x08, 0x00);
/*
** Register 09h - FPFRAME Start Position - not used by STN
*/
SET_REG(0x09, 0x00);
/*
** Register 0Ah - Vertical Non-Display Register = 3 lines
**              - Calculated in conjunction with register 08h (HNDP) to
**                achieve the desired frame rate.
*/
SET_REG(0x0A, 0x03);
/*
** Register 0Bh - MOD Rate - not used by this panel
*/
SET_REG(0x0B, 0x00);
/*
** Register 0Ch - Screen 1 Start Word Address LSB
** Register 0Dh - Screen 1 Start Word Address MSB
**              - Start address should be set to 0
*/
SET_REG(0x0C, 0x00);
SET_REG(0x0D, 0x00);
/*
** Register 0Eh - Screen 2 Start Word Address LSB
** Register 0Fh - Screen 2 Start Word Address MSB
**              - Set this start address to 0 too
*/
SET_REG(0x0E, 0x00);
SET_REG(0x0F, 0x00);
SET_REG(0x10, 0x00); /* Screen1/Screen2 Start Address High bits. */
/*
** Register 11h - Memory Address Offset
**              - Used for setting memory to a width greater than the
**                display size. Usually set to 0 during initialization
**                and programmed to desired value later.
*/
```

```
SET_REG(0x11, 0x00);
/*
** Register 12h - Screen 1 Vertical Size LSB
** Register 13h - Screen 1 Vertical Size MSB
**              - Set to maximum (i.e. 0x3FF). This register is used
**                for split screen operation. Normally it is set to
**                maximum value.
*/
SET_REG(0x12, 0xFF);
SET_REG(0x13, 0x03);
/*
** Look-Up Table registers
** The LUT is programmed at the end of the initialization sequence.
*/
/*
** Register 18h - GPIO Configuration - set to 0
**              - '0' configures the GPIO pins for input (power on default)
*/
SET_REG(0x18, 0x00);
/*
** Register 19h - GPIO Status - set to 0
**              - This step has no real purpose. It sets the GPIO
**                pins low should GPIO be set as outputs.
*/
SET_REG(0x19, 0x00);
/*
** Register 1Ah - Scratch Pad - set to 0
**              - Use this register to store whatever state data your
**                system may require.
*/
SET_REG(0x1A, 0x00);
/*
** Register 1Bh - Portrait Mode - set to 0 - disable portrait mode
*/
SET_REG(0x1B, 0x00);
/*
** Register 1Ch - Line Byte Count - set to 0 - used only by portrait mode.
*/
SET_REG(0x0C, 0x00);
/*
** Look-Up Table
** In this example we only set the first sixteen LUT entries.
** In typical use all 256 entries would be setup.
*/
/*
** Register 15h - Look-Up Table Address
**              - Set to 0 to start RGB sequencing at the first LUT entry.
*/
SET_REG(0x15, 0x00);
```

```
/*
** Register 17h - Look-Up Table Data
**              - Write 16 RGB triplets to the LUT.
*/
pLUT = LUT;
for (tmp = 0; tmp < 16; tmp++)
{
      SET_REG(0x17, *pLUT);// Set Red
      pLUT++;
      SET_REG(0x17, *pLUT);// Set Green
      pLUT++;
      SET_REG(0x17, *pLUT);// Set Blue
      pLUT++;
}
/*
** Clear all of video memory by writing 81920 bytes of 0.
*/
pMem = p13705;
for (tmp = 0; tmp < MEM_SIZE; tmp++)
{
      *pMem = 0;
      pMem++;
};
/*
** Draw a 100x100 red rectangle in the upper left corner (0,0)
** of the display.
*/
for (y = 0; y < 100; y++)
{
      /*
      ** Set the memory pointer at the start of each line.
      **   Pointer = MEM_OFFSET + (Y * Line_Width * BPP / 8) + (X * BPP / 8)
      */
      pMem = p13705 + (y * 320 * BitsPerPixel / 8) + 0;
      for (x = 0; x < 100; x++)
      {
            *pMem = 0x4;/* Draw a pixel with LUT color 4 */
            pMem++;
      }
}
/*
** Wait for the user to press a key before continuing.
*/
printf("Press any key to continue");
getch();
/*
** Set and use PORTRAIT mode.
*/
/*
```

```
    ** Clear the display, and all of video memory, by writing 81920 bytes
    ** of 0. This is done because an image in display memory is not rotated
    ** when the switch to portrait display mode occurs.
    */
    pMem = p13705;
    for (tmp = 0; tmp < MEM_SIZE; tmp++)
    {
          *pMem = 0;
          pMem++;
    };
    /*
    ** We will use the default portrait mode scheme so we have to adjust
    ** the ROTATED width to be a power of 2.
    ** (NOTE: current height will become the rotated width)
    */
    tmp = 1;
    while (Height > (1 << tmp))
    tmp++;
    Height = (1 << tmp);
    OffsetBytes = Height * BitsPerPixel / 8;
    /*
    ** Set:
    ** 1) Line Byte Count to size of the ROTATED width (i.e. current height)
    ** 2) Start Address to the offset of the width of the ROTATED display.
    **    (in portrait mode the start address registers point to bytes)
    */
    SET_REG(0x1C, (BYTE)OffsetBytes);
    OffsetBytes--;
    SET_REG(0x0C, LOBYTE(OffsetBytes));
    SET_REG(0x0D, HIBYTE(OffsetBytes));
    /*
    ** Set Portrait mode.
    ** Use the non-X2 (default) scheme so we don't have to re-calc the frame
    ** rate. MCLK will be <= 25 MHz so we can leave auto-switch enabled.
    */
    SET_REG(0x1B, 0x80);
    /*
    ** Draw a solid blue 100x100 rectangle centered on the display.
    ** Starting co-ordinates, assuming a 320x240 display are:
    **   (320-100)/2 , (240-100)/2 = 110,70.
    */
    for (y = 70; y < 180; y++)
    {
          /*
          ** Set the memory pointer at the start of each line.
          **    Pointer = MEM_OFFSET + (Y * Line_Width * BPP / 8) + (X * BPP / 8)
          ** NOTICE: as this is default portrait mode, the width is a power
          **         of two. In this case, we use a value of 256 pixels for
          **         our calculations instead of the panel dimension of 240.
```

```
             */
             x = 110;
             pMem = p13705 + (y * 256 * BitsPerPixel / 8) + (x * BitsPerPixel / 8);
             for (x = 110; x < 210; x++)
             {
                   *pMem = 0x01;          /* Draw a pixel in LUT color 1 */
                   pMem++;
             }
       }
}
/*
**=====================================================================
**
** IntelGetLinAddressW32(DWORD physaddr,DWORD *linaddr)
**
** return value:
**
**      0 : No error
**     -1 : Error
*/
int IntelGetLinAddressW32(DWORD physaddr, DWORD *linaddr)
{
       HANDLE hDriver;
       DWORD  cbReturned;
       int    rc, retVal;
       unsigned Arr[2];
       // First see if we are running under WinNT
       DWORD dwVersion = GetVersion();
       if (dwVersion < 0x80000000)
       {
             hDriver = CreateFile("\\\\.\\S1D13x0x", GENERIC_READ | GENERIC_WRITE,
                                  0, NULL, OPEN_EXISTING,FILE_ATTRIBUTE_NORMAL,
                                  NULL);
       }
       else    // Win95/98
       {
             // Dynamically load and prepare to call S1D13x0x.
             // The FILE_FLAG_DELETE_ON_CLOSE flag is used so that CloseHandle can
             // be used to dynamically unload the VxD.
             // The CREATE_NEW flag is not necessary
             hDriver = CreateFile("\\\\.\\S1D13x0x.VXD", 0,0,0,
                                  CREATE_NEW, FILE_FLAG_DELETE_ON_CLOSE, 0);
       }
       if (hDriver == INVALID_HANDLE_VALUE)
             return -1;
       /*
       ** From now on, the code is common for Win95 & WinNT
       */
       if (physaddr == 0)
```

```
            return -1;
        Arr[0] = physaddr;
        Arr[1] = 4 * 1024 * 1024;
        rc = DeviceIoControl(hDriver, IOCTL_SED_MAP_PHYSICAL_MEMORY,
                          &Arr[0], 2 * sizeof(ULONG), &retVal, sizeof(ULONG),
                          &cbReturned, NULL);
        if (rc)
            *linaddr = retVal;

        /*
        ** Close the handle.
        ** This will dynamically UNLOAD the Virtual Device for Win95.
        */
        CloseHandle(hDriver);
        if (rc)
            return 0;
        return -1;
}
```

10.3 Header Files

The header files included here are the required for the HAL sample to compile correctly.

```
/*
**===========================================================================
** HAL.H - Header file for use with programs written to use the S1D13705 HAL.
**---------------------------------------------------------------------------
** Created 1998, Vancouver Design Centre
** Copyright (c) 1998, 1999 Epson Research and Development, Inc.
** All Rights Reserved.
**===========================================================================
*/
#ifndef _HAL_H_
#define _HAL_H_
#include "hal_regs.h"
/*-------------------------------------------------------------------------*/
typedef unsigned char  BYTE;
typedef unsigned short WORD;
typedef unsigned long  DWORD;
typedef unsigned int   UINT;
typedef          int   BOOL;
#ifdef INTEL
typedef BYTE  far *LPBYTE;
typedef WORD  far *LPWORD;
typedef UINT  far *LPUINT;
typedef DWORD far *LPDWORD;
#else
typedef BYTE      *LPBYTE;
typedef WORD      *LPWORD;
typedef UINT      *LPUINT;
typedef DWORD     *LPDWORD;
#endif
#ifndef LOBYTE
#define LOBYTE(w)     ((BYTE)(w))
#endif
#ifndef HIBYTE
#define HIBYTE(w)     ((BYTE)(((UINT)(w) >> 8) & 0xFF))
#endif
#ifndef LOWORD
#define LOWORD(l)     ((WORD)(DWORD)(l))
#endif
#ifndef HIWORD
#define HIWORD(l)     ((WORD)((((DWORD)(l)) >> 16) & 0xFFFF))
#endif
#ifndef MAKEWORD
#define MAKEWORD(lo, hi) ((WORD)(((WORD)(lo)) | (((WORD)(hi)) << 8)) )
#endif
#ifndef MAKELONG
```

```
#define MAKELONG(lo, hi) ((long)(((WORD)(lo)) | (((DWORD)((WORD)(hi))) << 16)))
#endif
#ifndef TRUE
#define TRUE   1
#endif
#ifndef FALSE
#define FALSE  0
#endif
#define OFF 0
#define ON  1
#define SCREEN1 1
#define SCREEN22
/*
** Constants for HW rotate support
*/
#define DEFAULT0
#define LANDSCAPE 1
#define PORTRAIT2
#ifndef NULL
#ifdef __cplusplus
#define NULL    0
#else
#define NULL    ((void *)0)
#endif
#endif
/*-------------------------------------------------------------------------*/
/*
** SIZE_VERSION  is the size of the version string (eg. "1.00")
** SIZE_STATUS   is the size of the status string (eg. "b" for beta)
** SIZE_REVISION is the size of the status revision string (eg. "00")
*/
#define SIZE_VERSION5
#define SIZE_STATUS 2
#define SIZE_REVISION3
#ifdef ENABLE_DPF   /* Debug_printf() */
#define DPF(exp)  printf(#exp "\n")
#define DPF1(exp) printf(#exp " = %d\n", exp)
#define DPF2(exp1, exp2) printf(#exp1 "=%d  " #exp2 "=%d\n", exp1, exp2)
#define DPFL(exp) printf(#exp " = %x\n", exp)
#else
#define DPF(exp) ((void)0)
#define DPF1(exp) ((void)0)
#define DPFL(exp) ((void)0)
#endif
/*-------------------------------------------------------------------------*/
enum
{
ERR_OK = 0,                                       /* No error, call was successful.
*/
ERR_FAILED,                                      /* General purpose failure.
*/
ERR_UNKNOWN_DEVICE,       /* */
ERR_INVALID_PARAMETER,/* Function was called with invalid parameter. */
ERR_HAL_BAD_ARG,
ERR_TOOMANY_DEVS
};
/*******************************************
* Definitions for seGetId()
*******************************************/
#define PRODUCT_ID 0x24
enum
{
        ID_UNKNOWN,
        ID_S1D13705_Rev1
};
#define MAX_MEM_ADDR81920 - 1
#define EIGHTY_K81920
#define MAX_DEVICE    10
#define SE_RSVD           0
/*******************************************
* Definitions for Internal calculations.
*******************************************/
#define MIN_NON_DISP_X     32
#define MAX_NON_DISP_X     256
#define MIN_NON_DISP_Y     2
#define MAX_NON_DISP_Y     64
enum
{
        RED,
        GREEN,
        BLUE
};
/**************************************************************************/
typedef struct tagHalStruct
{
        char  szIdString[16];
        WORD  wDetectEndian;
        WORD  wSize;
        BYTE  Reg[MAX_REG + 1];
        DWORD dwClkI;     /* Input Clock Frequency (in kHz) */
        DWORD dwDispMem;/* */
        WORD  wFrameRate;/* */
} HAL_STRUCT;
typedef HAL_STRUCT * PHAL_STRUCT;
#ifdef INTEL_16BIT
typedef HAL_STRUCT far * LPHAL_STRUCT;
#else
typedef HAL_STRUCT     * LPHAL_STRUCT;
```

```
#endif
/*=========================================================================*/
/*                          FUNCTION  PROTO-TYPES                          */
/*=========================================================================*/
/*--------------------------- Initialization ----------------------------*/
int seRegisterDevice( const LPHAL_STRUCT lpHalInfo );
int seSetInit( void );
int seInitHal( void );
/*---------------------------- Miscellaneous ----------------------------*/
int  seGetId( int *pId );
void seGetHalVersion( const char **pVersion, const char **pStatus,
                      const char **pStatusRevision );
int seSetBitsPerPixel( int nBitsPerPixel );
int seGetBitsPerPixel( int *pBitsPerPixel );
int seGetBytesPerScanline( int *pBytes );
int seGetScreenSize( int *width, int *height );
void seDelay( int nMilliSeconds );
int seGetLastUsableByte( long *LastByte );
int seSetHighPerformance( BOOL OnOff );
/*------------------------------- Advanced ---------------------------------*/
int seSetHWRotate( int nMode );
int seSplitInit( WORD Scrn1Addr, WORD Scrn2Addr );
int seSplitScreen( int WhichScreen, int VisibleScanlines );
int seVirtInit( int xVirt, long *yVirt );
int seVirtMove( int nWhichScreen, int x, int y );
/*----------------------- Register/Memory Access -------------------------*/
int seGetReg( int index, BYTE *pValue );
int seSetReg( int index, BYTE value );
int seReadDisplayByte( DWORD offset, BYTE *pByte );
int seReadDisplayWord( DWORD offset, WORD *pWord );
int seReadDisplayDword( DWORD offset, DWORD *pDword );
int seWriteDisplayBytes( DWORD addr, BYTE val, DWORD count );
int seWriteDisplayWords( DWORD addr, WORD val, DWORD count );
int seWriteDisplayDwords( DWORD addr, DWORD val, DWORD count );
/*--------------------------------- Power Save ---------------------------*/
int seHWSuspend( int nDevID, BOOL val );
int seSetPowerSaveMode( int nDevID, int PowerSaveMode );
/*---------------------------------- Drawing ----------------------------*/
int seDrawLine( int x1, int y1, int x2, int y2, DWORD color );
int seDrawRect( int x1, int y1, int x2, int y2, DWORD color, BOOL Solidfill );
/*----------------------------- Color ----------------------------------*/
int seSetLut( BYTE *pLut );
int seGetLut( BYTE *pLut );
int seSetLutEntry( int index, BYTE *pEntry );
int seGetLutEntry( int index, BYTE *pEntry );
#endif      /* _HAL_H_ */
```

```
/*
**===========================================================================
** APPCFG.H - Application configuration information.
**---------------------------------------------------------------------------
** Created 1998 - Vancouver Design Centre
** Copyright (c) 1998, 1999 Epson Research and Development, Inc.
** All Rights Reserved.
**---------------------------------------------------------------------------
**
** The data in this file was generated using 13705CFG.EXE.
**
** The configureation parameters chosen were:
**   320x240 Single Color 4-bit STN
**   4 bpp - 100 Hz Frame Rate (12 MHz CLKi)
**   High Performance enabled
**
**===========================================================================
*/
/*************************************************************/
/*  13705 HAL HDR       (do not remove)                     */
/*  HAL_STRUCT Information generated by 13705CFG.EXE         */
/*  Copyright (c) 1998 Epson Research and Development, Inc. */
/*  All rights reserved.                                    */
/*                                                          */
/*  Include this file ONCE in your primary source file      */
/*************************************************************/
HAL_STRUCT HalInfo =
{
  "13705 HAL EXE",     /* ID string   */
  0x1234,             /* Detect Endian */
  sizeof(HAL_STRUCT), /* Size          */
  0x00,  0x20,  0xC0,  0x03,  0x27,  0xEF,  0x00,  0x00,
  0x00,  0x00,  0x03,  0x00,  0x00,  0x00,  0x00,  0x00,
  0x00,  0x00,  0xFF,  0x03,  0x00,  0x00,  0x00,  0x00,
  0x00,  0x00,  0x00,  0x00,  0x00,  0x00,  0x00,  0x00,
  6000,                  /* ClkI (kHz)      */
  0xF00000,              /* Display Address */
  70,                    /* Panel Frame Rate (Hz) */
};
```

```
/*
**=============================================================================
**  HAL_REGS.H
**-----------------------------------------------------------------------------
**  Created 1998, Epson Research & Development
**              Vancouver Design Center.
**  Copyright(c) Seiko Epson Corp. 1998.  All rights reserved.
**=============================================================================
*/
#ifndef __HAL_REGS_H__
#define __HAL_REGS_H__
/*
**     13705 register names
*/
#define REG_REVISION_CODE                       0x00
#define REG_MODE_REGISTER_0                     0x01
#define REG_MODE_REGISTER_1                     0x02
#define REG_MODE_REGISTER_2                     0x03
#define REG_HORZ_PANEL_SIZE                     0x04
#define REG_VERT_PANEL_SIZE_LSB                 0x05
#define REG_VERT_PANEL_SIZE_MSB                 0x06
#define REG_FPLINE_START_POS                    0x07
#define REG_HORZ_NONDISP_PERIOD                 0x08
#define REG_FPFRAME_START_POS                   0x09
#define REG_VERT_NONDISP_PERIOD                 0x0A
#define REG_MOD_RATE                            0x0B
#define REG_SCRN1_START_ADDR_LSB                0x0C
#define REG_SCRN1_START_ADDR_MSB                0x0D
#define REG_SCRN2_START_ADDR_LSB                0x0E
#define REG_SCRN2_START_ADDR_MSB                0x0F
#define REG_SCRN_START_ADDR_OVERFLOW            0x10
#define REG_MEMORY_ADDR_OFFSET                  0x11
#define REG_SCRN1_VERT_SIZE_LSB                 0x12
#define REG_SCRN1_VERT_SIZE_MSB                 0x13
#define REG_LUT_ADDR                            0x15
#define REG_LUT_BANK_SELECT                     0x16
#define REG_LUT_DATA                            0x17
#define REG_GPIO_CONFIG                         0x18
#define REG_GPIO_STATUS                         0x19
#define REG_SCRATCHPAD                          0x1A
#define REG_PORTRAIT_MODE                       0x1B
#define REG_LINE_BYTE_COUNT                     0x1C
#define REG_NOT_PRESENT_1                       0x1D
/*
** WARNING!!! MAX_REG must be the last available register!!!
*/
#define MAX_REG                                 0x1D
#endif       /*  __HAL_REGS_H__ */
```

```
/*-----------------------------------------------------------------------------
**
** Copyright (c) 1998, 1999 Epson Research and Development, Inc.
** All Rights Reserved.
**
** Module Name:
**
**    ioctl.h
**
**
** Abstract:
**
**   Include file for S1D13x0x PCI Board Driver.
**   Define the IOCTL codes we will use.  The IOCTL code contains a command
**   identifier, plus other information about the device, the type of access
**   with which the file must have been opened, and the type of buffering.
**
**-----------------------------------------------------------------------------
*/
#define SED_TYPE FILE_DEVICE_CONTROLLER
// The IOCTL function codes from 0x800 to 0xFFF are for customer use.
#define IOCTL_SED_QUERY_NUMBER_OF_PCI_BOARDS \
    CTL_CODE( SED_TYPE, 0x900, METHOD_BUFFERED, FILE_ANY_ACCESS)
#define IOCTL_SED_MAP_PCI_BOARD \
    CTL_CODE( SED_TYPE, 0x901, METHOD_BUFFERED, FILE_ANY_ACCESS)
#define IOCTL_SED_MAP_PHYSICAL_MEMORY \
    CTL_CODE( SED_TYPE, 0x902, METHOD_BUFFERED, FILE_ANY_ACCESS)
#define IOCTL_SED_UNMAP_LINEAR_MEMORY \
    CTL_CODE( SED_TYPE, 0x903, METHOD_BUFFERED, FILE_ANY_ACCESS)
```

11 Change Record

X27A-G-002-03	Revision 3.1 - Issued: March 26, 2018
	• updated Sales and Technical Support Section
	• updated some formatting

12 Sales and Technical Support

For more information on Epson Display Controllers, visit the Epson Global website.

https://global.epson.com/products_and_drivers/semicon/products/display_controllers/

For Sales and Technical Support, contact the Epson representative for your region.

https://global.epson.com/products_and_drivers/semicon/information/support.html