EPSON®
EXCEED YOUR VISION

S1D13505 Embedded RAMDAC LCD/CRT Controller

Programming Notes and Examples

Document Number: X23A-G-003-07

THIS PAGE LEFT BLANK

Table of Contents

List of Tables

THIS PAGE LEFT BLANK

List of Figures

THIS PAGE LEFT BLANK

1 Introduction

This guide describes how to program the S1D13505 Embedded RAMDAC LCD/CRT Controller. The guide presents the basic concepts of the LCD/CRT controller and provides methods to directly program the registers. It explains some of the advanced techniques used and the special features of the S1D13505.

The guide also introduces the Hardware Abstraction Layer (HAL), which is designed to simplify the programming of the S1D13505. Most S1D1350x and S1D1370x products support the HAL allowing OEMs to switch chips with relative ease.

This document is updated as appropriate. Please check the Epson Electronics America Website at http://www.eea.epson.com for the latest revision of this document before beginning any development.

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

2 Initialization

This section describes how to initialize the S1D13505. Sample code for performing initialization of the S1D13505 is provided in the file **init13505.c** which is available on the internet at http://www.eea.epson.com.

S1D13505 initialization can be broken into three steps. First, enable the S1D13505 controller (if necessary identify the specific controller). Next, set all the registers to their initial values. Finally, program the Look-Up Table (LUT) with color values. This section does not deal with programming the LUT, see Section 4 of this manual for LUT programming details.

Note

When using an ISA evaluation board in a PC (i.e. S5U13505B00C), there are two additional steps that must be carried out before initialization. First, confirm that 16-bit mode is enabled by writing to address F80000h. Then, if hardware suspend is enabled, disable suspend mode by writing to F00000h. For further information on ISA evaluation boards refer to the *S5U13505B00C Rev. 1.0 ISA Bus Evaluation Board User Manual*, document number X23A-G-004-xx.

The following table represents the sequence and values written to the S1D13505 registers to control a configuration with these specifications:

- 640x480 color dual passive format 1 LCD @ 75Hz.
- 8-bit data interface.
- 8 bit-per-pixel (bpp) - 256 colors.
- 31.5 MHz input clock.
- 50 ns EDO-DRAM, 2 CAS, 4 ms refresh, CAS before RAS.

Table 2-1: S1D13505 Initialization Sequence

Register	Value	Notes	See Also
[1B]	0000 0000	Enable the host interface	
[23]	1000 0000	Disable the FIFO	
[01]	0011 0000	Memory configuration - divide Clkl by 512 to get 4 ms for 256 refresh cycles - this is 2-CAS# EDO memory	
[22]	0100 1000	Performance Enhancement 0 - refer to the hardware specification for a complete description of these bits	S1D13505 Hardware Functional Specification, document number X23A-A-001-xx
[02]	0001 0110	Panel type - non-EL, 8-bit data, format 1, color, dual, passive	
[03]	0000 0000	Mod rate used by older monochrome panels - set to 0	
[04]	0100 1111	Horizontal display size = (Reg[04]+1)*8 = (79+1)*8 = 640 pixels	see note for REG[16h] and REG[17h]
[05]	0000 0011	Horizontal non-display size = (Reg[05]+1)*8 = (3+1)*8 = 32 pixels	

Table 2-1: S1D13505 Initialization Sequence (Continued)

Register	Value	Notes	See Also
[06]	0000 0000	FPLINE start position - only required for CRT or TFT/D-TFD	
[07]	0000 0000	FPLINE polarity set to active high	
[08]	1110 1111	Vertical display size = Reg[09][08] + 1 = 0000 0000 1110 1111 + 1 = 239+1 = 240 lines (total height/2 for dual panels)	
[09]	0000 0000		
[0A]	0011 1000	Vertical non-display size = Reg[0A] + 1 = 57 + 1 = 58 lines	
[0B]	0000 0000	FPFRAME start position - only required for CRT or TFT/D-TFD	
[0C]	0000 0000	FPFRAME polarity set to active high	
[0D]	0000 1100	Display mode - hardware portrait mode disabled, 8 bpp and LCD disabled, enable LCD in last step of this example.	
[0E]	1111 1111	Line compare (Regs[0Eh] and[0Fh] set to maximum allowable value. We can change this later if we want a split screen.	‘
[0F]	0000 0011		
[10]	0000 0000	Screen 1 Start Address (Regs [10h], [11h], and [12h]) set to 0. This will start the display in the first byte of the display buffer.	
[11]	0000 0000		
[12]	0000 0000		
[13]	0000 0000	Screen 2 Start Address (Regs [13h], [14h], and [15h]) to offset 0. Screen 2 Start Address in not used at this time.	
[14]	0000 0000		
[15]	0000 0000		
[16]	0100 0000	Memory Address Offset (Regs [17h] [16h]) - 640 pixels = 640 bytes = 320 words = 140h words **Note: When setting a horizontal resolution greater than 767 pixels, with a color depth of 15/16 bpp, the Memory Offset Registers (REG[16h], REG[17h]) must be set to a virtual horizontal pixel resolution of 1024.**	
[17]	0000 0001		
[18]	0000 0000	Set pixel panning for both screens to 0	
[19]	0000 0001	Clock Configuration - set PClk to MClk/2 - the specification says that for a dual color panel the maximum PClk is MClk/2	
[1A]	0000 0000	Enable LCD Power	
[1C]	0000 0000	MD Configuration Readback - we write a 0 here to keep the register configuration logic simpler	
[1D]	0000 0000		
[1E]	0000 0000	General I/O Pins - set to zero.	
[1F]	0000 0000		
[20]	0000 0000	General I/O Pins Control - set to zero.	
[21]	0000 0000		

Table 2-1: S1D13505 Initialization Sequence (Continued)

Register	Value	Notes	See Also
[24]	0000 0000	The remaining register control operation of the LUT and hardware cursor/ink layer. During the chip initialization none of these registers needs to be set. It is safe to write them to zero as this is the power-up value for the registers.	
[26]	0000 0000		
[27]	0000 0000		
[28]	0000 0000		
[29]	0000 0000		
[2A]	0000 0000		
[2B]	0000 0000		
[2C]	0000 0000		
[2D]	0000 0000		
[2E]	0000 0000		
[2F]	0000 0000		
[30]	0000 0000		
[31]	0000 0000		
[23]	0000 0000	Enable FIFO, mask in appropriate FIFO threshold bits	S1D13505 Hardware Functional Specification, document number X23A-A-001-xx
[0D]	0000 1101	Display mode - hardware portrait mode disabled, 8 bpp and LCD enabled	

2.1 Miscellaneous

This section of the notes contains recommendations which can be set at initialization time to improve display image quality.

At high color depths the display FIFO introduces two conditions which must be accounted for in software. Simultaneous display while using a dual passive panel introduces another possible register change.

Display FIFO Threshold

At 15/16 bit-per-pixel the display FIFO threshold (bits 0-4 of register [23h]) must be programmed to a value other than '0'. Product testing has shown that at these color depths a better quality image results when the display FIFO threshold is set to a value of 1Bh.

Memory Address Offset

When an 800x600 display mode is selected at 15 or 16 bpp, memory page breaks can disrupt the display buffer fetches. This disruption produces a visible flicker on the display. To avoid this set the Memory Address Offset (Reg [16h] and Reg [17h]) to 200h. This sets a 1024 pixel line which aligns the memory page breaks and reduces any flicker.

Half Frame Buffer Disable

The half frame buffer stores the display data for dual drive LCD panels. During LCD only or simultaneous display using a single LCD panel, no special adjustments are required.

However, for simultaneous display using a dual drive LCD panel, the half frame buffer must be disabled (REG[1Bh] bit 0 = 1). This results in reduced contrast on the LCD panel because the duty cycle of the LCD is halved. To compensate for this change, the pattern used by the Frame Rate Modulator (FRM) may need to be adjusted. Programming the Alternate FRM Register (REG[31h]) with the recommended value of FFh may produce more visually appealing output.

For further information on the half frame buffer and the Alternate FRM Register see the *S1D13505 Hardware Functional Specification*, document number X23A-A-001-xx.

3 Memory Models

The S1D13505 is capable of several color depths. The memory model for each color depth is packed pixel. Packed pixel data changes with each color depth from one byte containing eight consecutive pixels up to two bytes being required for one pixel.

3.1 Display Buffer Location

The S1D13505 supports either a 512k byte or 2M byte display buffer. The display buffer is memory mapped and can be accessed directly by software. The memory location allocated to the S1D13505 display buffer varies with each individual hardware platform, and is determined by the OEM.

For further information on the display buffer, see the *S1D13505 Hardware Functional Specification*, document number X23A-A-001-xx.

3.1.1 Memory Organization for One Bit-Per-Pixel (2 Colors/Gray Shades)

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

In this memory format each byte of display buffer contains eight adjacent pixels. Setting or resetting any pixel will require reading the entire byte, masking out the appropriate bits and, if necessary, setting the bits to '1'.

One bit pixels provide two gray shade/color possibilities. For monochrome panels the two gray shades are generated by indexing into the first two elements of the green component of the Look-Up Table (LUT). For color panels the two colors are derived by indexing into positions 0 and 1 of the Look-Up Table.

3.1.2 Memory Organization for Two Bit-Per-Pixel (4 Colors/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

In this memory format each byte of display buffer contains four adjacent pixels. Setting or resetting any pixel will require reading the entire byte, masking out the appropriate bits and, if necessary, setting the bits to '1'.

Two bit pixels are capable of displaying four gray shade/color combinations. For monochrome panels the four gray shades are generated by indexing into the first four elements of the green component of the Look-Up Table. For color panels the four colors are derived by indexing into positions 0 through 3 of the Look-Up Table.

3.1.3 Memory Organization for Four Bit-Per-Pixel (16 Colors/Gray Shades)

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

In this memory format each byte of display buffer contains two adjacent pixels. Setting or resetting any pixel will require reading the entire byte, masking out the upper or lower nibble (4 bits) and setting the appropriate bits to '1'.

Four bit pixels provide 16 gray shade/color possibilities. For monochrome panels the gray shades are generated by indexing into the first 16 elements of the green component of the Look-Up Table. For color panels the 16 colors are derived by indexing into the first 16 positions of the Look-Up Table.

3.1.4 Memory Organization for Eight Bit-Per-Pixel (256 Colors/16 Gray Shades)

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

Figure 3-4: Pixel Storage for 8 Bpp (256 Colors/16 Gray Shades) in One Byte of Display Buffer

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

Each byte indexes into one of the 256 positions of the Look-Up Table. The S1D13505 LUT supports four bits per primary color, therefore this translates into 4096 possible colors when color mode is selected. To display the fullest dynamic range of colors will require careful selection of the colors in the LUT indices and in the image to be displayed.

When monochrome mode is selected, the green component of the LUT is used to determine the gray shade intensity. The green indices, with only four bits, can resolve 16 gray shades. In this situation one might as well use four bit-per-pixel mode and conserve display buffer.

3.1.5 Memory Organization for Fifteen Bit-Per-Pixel (32768 Colors/16 Gray Shades)

Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8
Reserved	Red Bit 4	Red Bit 3	Red Bit 2	Red Bit 1	Red Bit 0	Green Bit 4	Green Bit 3
Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Green Bit 2	Green Bit 1	Green Bit 0	Blue Bit 4	Blue Bit 3	Blue Bit 2	Blue Bit 1	Blue Bit 0

Figure 3-5: Pixel Storage for 15 Bpp (32768 Colors/16 Gray Shades) in Two Bytes of Display Buffer

In 15 bit-per-pixel mode the S1D13505 is capable of displaying 32768 colors. The 32768 color pixel is divided into four parts: one reserved bit, five bits for red, five bits for green, and five bits for blue. In this mode the Look-Up Table is bypassed and output goes directly into the Frame Rate Modulator.

The full color range is only available on TFT/D-TFD or CRT displays. Passive LCD displays are limited to using the four most significant bits from each of the red, green and blue portions of each color. The result is 4096 ($2^4 * 2^4 * 2^4$) possible colors.

Should monochrome mode be chosen at this color depth, the output reverts to sending the four most significant bits of the green LUT component to the modulator for a total of 16 possible gray shades. In this situation one might as well use four bit-per-pixel mode and conserve display buffer.

3.1.6 Memory Organization for Sixteen Bit-Per-Pixel (65536 Colors/16 Gray Shades)

Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8
Red Bit 4	Red Bit 3	Red Bit 2	Red Bit 1	Red Bit 0	Green Bit 5	Green Bit 4	Green Bit 3
Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Green Bit 2	Green Bit 1	Green Bit 0	Blue Bit 4	Blue Bit 3	Blue Bit 2	Blue Bit 1	Blue Bit 0

Figure 3-6: Pixel Storage for 16 Bpp (65536 Colors/16 Gray Shades) in Two Bytes of Display Buffer

In 16 bit-per-pixel mode the S1D13505 is capable of generating 65536 colors. The 65536 color pixel is divided into three parts: five bits for red, six bits for green, and five bits for blue. In this mode the Look-Up Table is bypassed and output goes directly into the Frame Rate Modulator.

The full color range is only available on TFT/D-TFD or CRT displays. Passive LCD displays are limited to using the four most significant bits from each of the red, green and blue portions of each color. The result is 4096 ($2^4 * 2^4 * 2^4$) possible colors.

When monochrome mode is selected, the green component of the LUT is used to determine the gray shade intensity. The green indices, with only four bits, can resolve 16 gray shades. In this situation one might as well use four bit-per-pixel mode and conserve display buffer.

4 Look-Up Table (LUT)

This section is supplemental to the description of the Look-Up Table architecture found in the S1D13505 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 S1D13505 Hardware Functional Specification, document number X23A-A-001-xx for more detail.

The S1D13505 Look-Up Table is used for both the CRT and panel interface and consists of 256 indexed red/green/blue entries. Each entry is 4 bits wide. Two registers, at offsets 24h and 26h, control access to the LUT. Color depth affects how many indices will be used for image display.

In color modes, pixel values are used as indices to an RGB value stored in the Look-Up Table. In monochrome modes only the green component of the LUT is used. The value in the display buffer indexes into the LUT and the amount of green at that index controls the intensity. Monochrome mode look-ups are done for the panel interface only. The CRT interface always receives the RGB values from the Look-Up Table.

4.1 Look-Up Table Registers

REG[24h] 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. Writing to this register will select the red bank. After three successive reads or writes to the data register this register will be incremented by one.

REG[26h] 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 or read. With each successive read or write the internal bank select is incremented. Three reads from this register will result in reading the red, then the green, and finally the blue values associated with the index set in the LUT address register.

After the third read the LUT address register is incremented and the internal index points to the red bank again.

4.2 Look-Up Table Organization

- The Look-Up Table treats the value of a pixel as an index into an array of colors or gray shades. For example, a pixel value of zero would point to the first LUT entry; a pixel value of 7 would point to the eighth LUT entry.
- The value inside each LUT entry represents the intensity of the given color or gray shade. This intensity can range in value between 0 and 0Fh.
- The S1D13505 Look-Up Table is linear; increasing the LUT entry number results in a lighter color or gray shade. For example, a LUT entry of 0Fh into the red LUT entry will result in a bright red output while a LUT entry of 5 would result in a dull red.

Table 4-1: Look-Up Table Configurations

Display Mode	4-Bit Wide Look-Up Table			Effective Gray Shade/Colors on an Passive Panel
	RED	**GREEN**	**BLUE**	
1 bpp gray	🟨	2	🟨	2 gray shades
2 bpp gray	🟨	4	🟨	4 gray shades
4 bpp gray	🟨	16	🟨	16 gray shades
8 bpp gray	🟨	16	🟨	16 gray shades
15 bpp gray	🟨	🟨	🟨	16 gray shades
16 bpp gray	🟨	🟨	🟨	16 gray shades
1 bpp color	2	2	2	2 colors
2 bpp color	4	4	4	4 colors
4 bpp color	16	16	16	16 colors
8 bpp color	256	256	256	256 colors
15 bpp color	🟨	🟨	🟨	4096 colors*
16 bpp color	🟨	🟨	🟨	4096 colors*

* On an active matrix panel the effective colors are determined by the interface width. (i.e. 9-bit=512, 12-bit=4096, 18-bit=64K colors) Passive panels are limited to 12-bits through the Frame Rate Modulator.

🟨 = Indicates the Look-Up Table is not used for that display mode

Color Modes

In color display modes, depending on the color depth, 2 through 256 index entries are used. The selection of which entries are used is automatic.

1 bpp color

When the S1D13505 is configured for 1 bpp color mode, the LUT is limited to 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 8 bits, each pertaining to adjacent pixels. A bit value of '0' results in the LUT 0 index value being displayed. A bit value of '1' results in the LUT 1 index value 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 4-2: 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

🟨 = Indicates unused entries in the LUT

2 bpp color

When the S1D13505 is configured for 2 bpp color mode only the first 4 entries of the LUT are used. These four entries can be set to any desired values.

Each byte in the display buffer contains 4 adjacent pixels. Each pair of bits in the byte are used as an index into the LUT. The following table shows example values for 2 bpp color mode.

Table 4-3: 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 in the LUT

4 bpp color

When the S1D13505 is configured for 4 bpp color mode the first 16 entries in the LUT are used.

Each byte in the display buffer contains two adjacent pixels. The upper and lower nibbles of the byte are used as indices into the LUT.

The following table shows LUT values that will simulate those of a VGA operating in 16 color mode.

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

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

🟨 = Indicates unused entries in the LUT

8 bpp color

When the S1D13505 is configured for 8 bpp color mode all 256 entries in the LUT are used. Each byte in display buffer corresponds to one pixel and is used as an index value into the LUT.

The S1D13505 LUT has four bits (16 intensities) of intensity control per primary color while a standard VGA RAMDAC has six bits (64 intensities). This four to one difference has to be considered when attempting to match colors between a VGA RAMDAC and the S1D13505 LUT. (i.e. VGA levels 0 - 3 map to LUT level 0, VGA levels 4 - 7 map to LUT level 1...). Additionally, the significant bits of the color tables are located at different offsets within their respective bytes. After calculating the equivalent intensity value the result must be shifted into the correct bit positions.

The following table shows LUT values that will approximate the VGA default color palette.

Table 4-5: 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
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

Table 4-5: 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
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

15 bpp color

The Look-Up Table is bypassed at this color depth, hence programming the LUT is not necessary.

16 bpp color

The Look-Up Table is bypassed at this color depth, hence programming the LUT is not necessary.

Gray Shade Modes

This discussion of gray shade/monochrome modes only applies to the panel interface. Monochrome mode is selected when register [01] bit 2 = 0. In this mode the output value to the panel is derived solely from the green component of the LUT. The CRT image will continue to be formed from all three (RGB) Look-Up Table components.

Note
In order to match the colors on a CRT with the colors on a monochrome panel it is important to ensure that the red and blue components of the Look-Up Table be set to the same intensity as the green component.

1 bpp gray shade

In 1 bpp gray shade mode only the first two entries of the green LUT are used. All other LUT entries are unused.

Table 4-6: Recommended LUT Values for 1 Bpp Gray Shade

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

🟦 = Required to match CRT to panel
🟨 = Unused entries

2 bpp gray shade

In 2 bpp gray shade mode the first four green elements are used to provide values to the panel. The remaining indices are unused.

Table 4-7: Suggested Values for 2 Bpp Gray Shade

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

🟦 = Required to match CRT to panel
🟨 = Unused entries

4 bpp gray shade

The 4 bpp gray shade mode uses the first 16 LUT elements. The remaining indices of the LUT are unused.

Table 4-8: Suggested LUT Values for 4 Bpp Gray Shade

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

🟦	Required to match CRT to panel
🟨	Unused entries

8 bpp gray shade

When 8 bpp gray shade mode is selected the gray shade intensity is determined by the green LUT value. The green portion of the LUT has 16 possible intensities. There is no color advantage to selecting 8 bpp mode over 4 bpp mode; however, hardware rotate can be only used in 8 and 16 bpp modes.

15 bpp gray shade

The Look-Up Table is bypassed at this color depth, hence programming the LUT is not necessary.

As with 8 bpp there are limitations to the colors which can be displayed. In this mode the four most significant bits of green are used to set the absolute intensity of the image. Four bits of green resolves to 16 colors. Now however, each pixel requires two bytes.

16 bpp gray

The Look-Up Table is bypassed at this color depth, hence programming the LUT is not necessary.

As with 8 bpp there are limitations to the colors which can be displayed. In this mode the four most significant bits of green are used to set the absolute intensity of the image. Four bits of green resolves to 16 colors. Now however, each pixel requires two bytes.

5 Advanced Techniques

This section presents information on 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. This can be in the horizontal, the vertical or both dimensions. To view the image, the display is used as a window (or viewport) 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 registers are used to determine the number of horizontal pixels in the virtual image. The offset registers can be set for a maximum of 2^{11} or 2048 words. In 1 bpp display modes these 2048 words cover 16,384 pixels. At 16 bpp 2048 words cover 1024 pixels.

The maximum vertical size of the virtual image is the result of a number of variables. In its simplest, the number of lines is the total display buffer divided by the number of bytes per horizontal line. The number of bytes per line is the number of words in the offset register multiplied by two. At maximum horizontal size, the greatest number of lines that can be displayed is 1024. Reducing the horizontal size makes memory available to increase the virtual vertical size.

In addition to the calculated limit the virtual vertical size is limited by the size and location of the half frame buffer and the ink/cursor if present.

Seldom are the maximum sizes used. Figure 5-1: “Viewport Inside a Virtual Display,” depicts a more 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.

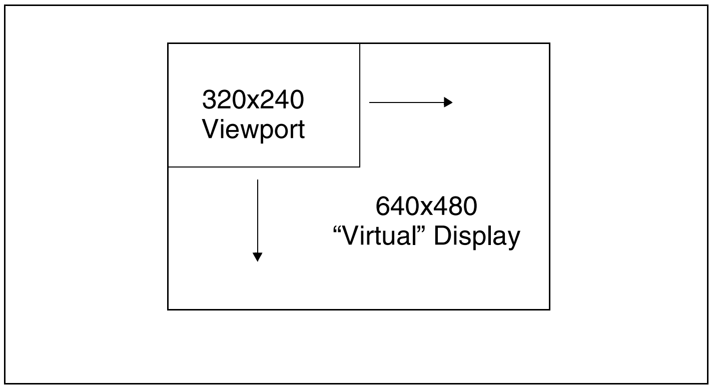

Figure 5-1: Viewport Inside a Virtual Display

5.1.1 Registers

REG[16h] Memory Address Offset Register 0							
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

REG[17h] Memory Address Offset Register 1							
n/a	n/a	n/a	n/a	n/a	Memory Address Offset Bit 10	Memory Address Offset Bit 9	Memory Address Offset Bit 8

Figure 5-2: Memory Address Offset Registers

Registers [16h] and [17h] form an 11-bit value called the memory address offset. This offset is the number of words from the beginning of one line of the display to the beginning of the next line of the display.

Note that this value does not necessarily represent the number of words to be shown on the display. The display width is set in the Horizontal Display Width register. If the offset is set to the same as the display width then there is no virtual width.

To maintain a constant virtual width as color depth changes, the memory address offset must also change. At 1 bpp each word contains 16 pixels, at 16 bpp each word contains one pixel. The formula to determine the value for these registers is:

offset = pixels_per_line / pixels_per_word

5.1.2 Examples

Example 1: Determine the offset value required for 800 pixels at a color depth of 8 bpp.

At 8 bpp each byte contains one pixel, therefore each word contains two pixels.

pixels_per_word = 16 / bpp = 16 / 8 = 2

Using the above formula.

offset = pixels_per_line / pixels_per_word = 800 / 2 = 400 = 190h words

Register [17h] would be set to 01h and register [16h] would be set to 90h.

Example 2: Program the Memory Address Offset Registers to support a 16 color (4 bpp) 640x480 virtual display on a 320x240 LCD panel.

To create a virtual display the offset registers must be programmed to the horizontal size of the larger "virtual" image. After determining the amount of memory used by each line, do a calculation to see if there is enough memory to support the desired number of lines.

1. Initialize the S1D13505 registers for a 320x240 panel. (See Introduction on page 11).
2. Determine the offset register value.

   pixels_per_word = 16 / bpp = 16 / 4 = 4

   offset = pixels_per_line / pixels_per_word = 640 / 4 = 160 words = 0A0h words

   Register [17h] will be written with 00h and register [16h] will be written with A0h.
3. Check that we have enough memory for the required virtual height.

   Each line uses 160 words and we need 480 lines for a total of (160*480) 76,800 words. This display could be done on a system with the minimum supported memory size of 512 K bytes. It is safe to continue with these values.

5.2 Panning and Scrolling

The terms panning and scrolling refer to the actions used to move the viewport about a virtual display. Although the image is stored entirely in the display buffer, only a portion is actually visible at any given time.

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 refers to the word offset in the display buffer where the image will start being displayed from. At color depths less than 15 bpp a second register, the pixel pan register, is required for smooth pixel level panning.

Internally, the S1D13505 latches different signals at different times. Due to this internal sequence, there is an order in which the start address and pixel pan registers should be accessed during scrolling operations to provide the smoothest scrolling. Setting the registers in the wrong sequence or at the wrong time will result in a “tearing” or jitter effect on the display.

The start address is latched at the beginning of each frame, therefore the start address can be set any time during the display period. The pixel pan register values are latched at the beginning of each display line and must be set during the vertical non-display period. The correct sequence for programing these registers is:

1. Wait until just after a vertical non-display period (read register [0Ah] and watch bit 7 for the non-display status).
2. Update the start address registers.
3. Wait until the next vertical non-display period.
4. Update the pixel paning register.

5.2.1 Registers

REG[10h] Screen 1 Display Start Address 0							
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[11h] Screen 1 Display Start Address 1							
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[12h] Screen 1 Display Start Address 2							
n/a	n/a	n/a	n/a	Start Addr Bit 19	Start Addr Bit 18	Start Addr Bit 17	Start Addr Bit 16

Figure 5-3: Screen 1 Start Address Registers

These three registers form the address of the word in the display buffer where screen 1 will start displaying from. Changing these registers by one will cause a change of 0 to 16 pixels depending on the current color depth. Refer to the following table 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	Number of Pixels Panned
1	16	16
2	8	8
4	4	4
8	2	2
15	1	1
16	1	1

REG[18h] Pixel Panning Register							
Screen 2 Pixel Pan Bit 3	Screen 2 Pixel Pan Bit 2	Screen 2 Pixel Pan Bit 1	Screen 2 Pixel Pan Bit 0	Screen 1 Pixel Pan Bit 3	Screen 1 Pixel Pan Bit 2	Screen 1 Pixel Pan Bit 1	Screen 1 Pixel Pan Bit 0

Figure 5-4: Pixel Panning Register

The pixel panning register offers finer control over pixel pans than is available with the Start Address Registers. Using this register it is possible to pan the displayed image one pixel at a time. Depending on the current color depth certain bits of the pixel pan register are not used. The following table shows this.

Table 5-2: Active Pixel Pan Bits

Color Depth (bpp)	Pixel Pan bits used
1	bits [3:0]
2	bits [2:0]
4	bits [1:0]
8	bit 0
15/16	---

5.2.2 Examples

For the examples in this section assume that the display system has been set up to view a 640x480 pixel image in a 320x240 viewport. Refer to Section 2, “Initialization” on page 12 and Section 5.1, “Virtual Display” on page 29 for assistance with these settings.

Example 3: Panning - Right and Left

To pan to the right, increment the pixel pan value. If the pixel pan value is equal to the current color depth then set the pixel pan value to zero and increment the start address value. To pan to the left decrement the pixel pan value. If the pixel pan value is less than zero set it to the color depth (bpp) less one and decrement the start address.

Note
Scrolling operations are easier to follow if a value, call it pan_value, is used to track both the pixel pan and start address. The least significant bits of pan_value will represent the pixel pan value and the more significant bits are the start address value.

The following pans to the right by one pixel in 4 bpp display mode.

1. This is a pan to the right. Increment pan_value.

   pan_value = pan_value + 1

2. Mask off the values from pan_value for the pixel panning and start address register portions. In this case, 4 bpp, the lower two bits are the pixel panning value and the upper bits are the start address.

   pixel_pan = pan_value AND 3

   start_address = pan_value SHR 3
   (the fist two bits of the shift account for the pixel_pan the last bit of the shift converts the start_address value from bytes to words)

3. Write the pixel panning and start address values to their respective registers using the procedure outlined in the registers section.

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.

Example 5: Scroll down one line for a 16 color 640x480 virtual image using a 320x240 single panel LCD.

1. To scroll down we need to know how many words each line takes up. At 16 colors (4 bpp) each byte contains two pixels so each word contains 4 pixels.

   offset_words = pixels_per_line / pixels_per_word = 640 / 4 = 160 = A0h

   We now know how much to add to the start address to scroll down one line.

2. Increment the start address by the number of words per virtual line.

   start_address = start_address + words

3. Separate the start address value into three bytes. Write the LSB to register [10h] and the MSB to register [12h].

5.3 Split Screen

Occasionally the need arises to display two distinct images on the display. For example, we may write a game where the main play area will rapidly update and we want a status display at the bottom of the screen.

The Split Screen feature of the S1D13505 allows a programmer to setup a display for such an application. The figure below illustrates setting a 320x240 panel to have Image 1 displaying from scan line 0 to scan line 99 and image 2 displaying from scan line 100 to scan line 239. Although this example picks specific values, image 1 and image 2 can be shown as varying portions of the screen

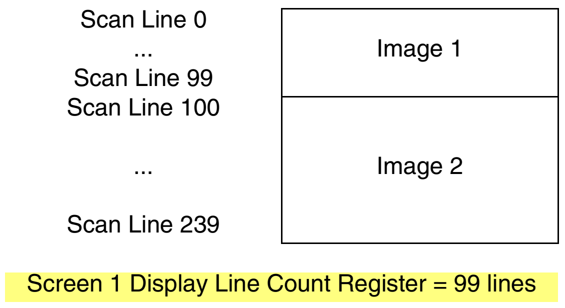

Figure 5-5: 320x240 Single Panel For Split Screen

5.3.1 Registers

The other registers required for split screen operations, [10h] through [12h] (Screen 1 Display Start Address) and [18h] (Pixel Panning Register), are described in Section 5.2.1 on page 32.

REG[0E] Screen 1 Line Compare Register 0							
Line Compare Bit 7	Line Compare Bit 6	Line Compare Bit 5	Line Compare Bit 4	Line Compare Bit 3	Line Compare Bit 2	Line Compare Bit 1	Line Compare Bit 0

REG[0F] Screen 1 Line Compare Register 1							
n/a	n/a	n/a	n/a	n/a	n/a	Line Compare Bit 9	Line Compare Bit 8

Figure 5-6: Screen 1 Line Compare

These two registers form a value known as the line compare. When the line compare value is equal to or greater than the physical number of lines being displayed there is no visible effect on the display. When the line compare value is less than the number of physically displayed lines, display operation works like this:

1. From the end of vertical non-display to the number of lines indicated by line compare the display data will be from the memory pointed to by the Screen 1 Display Start Address.

2. After *line compare* lines have been displayed the display will begin showing data from Screen 2 Display Start Address memory.

REG[13h] Screen 2 Display Start Address Register 0							
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[14h] Screen 2 Display Start Address Register 1							
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[15h] Screen 2 Display Start Address Register 2							
n/a	n/a	n/a	n/a	Start Addr Bit 19	Start Addr Bit 18	Start Addr Bit 17	Start Addr Bit 16

Figure 5-7: Screen 2 Display Start Address

These three registers form the twenty bit offset to the first word in the display buffer that will be shown in the screen 2 portion of the display.

Screen 1 memory is **always** displayed first at the top of the screen followed by screen 2 memory. The start address for the screen 2 image may be lower in memory than that of screen 1 (i.e. screen 2 could be coming from offset 0 in the display buffer while screen 1 was coming from an offset located several thousand bytes into the display buffer). While not particularly useful, it is possible to set screen 1 and screen 2 to the same address.

5.3.2 Examples

Example 6: Display 380 scanlines of image 1 and 100 scanlines of image 2. Image 2 is located immediately after image 1 in the display buffer. Assume a 640x480 display and a color depth of 1 bpp.

1. The value for the line compare is not dependent on any other setting so we can set it immediately (380 = 17Ch).

   Write the line compare registers [0Fh] with 01h and register [0Eh] with 7Ch.

2. Screen 1 is coming from offset 0 in the display buffer. Although not necessary, ensure that the screen 1 start address is set to zero.

   Write 00h to registers [10h], [11h] and [12h].

3. Calculate the size of the screen 1 image (so we know where the screen 2 image is located). This calculation must be performed on the virtual size (offset register) of the display. Since a virtual size was not specified assume the virtual size to be the same as the physical size.

   offset = pixels_per_line / pixels_per_word = 640 / 16 = 40 words per line

   screen1_size = offset * lines = 40 * 480 = 19,200 words = 4B00h words

4. Set the screen 2 start address to the value we just calculated.

   Write the screen 2 start address registers [15h], [14h] and [13h] with the values 00h, 4Bh and 00h respectively.

6 LCD Power Sequencing and Power Save Modes

The S1D13505 design includes a pin (LCDPWR) which may be used to control an external LCD bias power supply. If the hardware design makes use of LCDPWR, automatic LCD power sequencing and power save modes are available to the programmer. If LCDPWR is not used to control an external LCD bias power supply, this section is not applicable.

6.1 LCD Power Sequencing

The S1D13505 is designed with internal circuitry which automates LCD power sequencing (the process of powering-on and powering-off the LCD panel). LCD power sequencing allows the LCD bias voltage to discharge prior to shutting down the LCD signals. Power sequencing prevents long term damage to the panel and avoids unsightly "lines" at power-on/power-off.

Proper LCD power sequencing for power-off requires a time delay from the time the LCD power is disabled to the time the LCD signals are shut down. Power-on requires the LCD signals to be active prior to applying power to the LCD. This time interval varies depending on the LCD bias power supply design. For example, the LCD bias power supply on the S5U13505 Evaluation board requires approximately 0.5 seconds to fully discharge. Your power supply design may vary.

For most applications internal power sequencing is the appropriate choice. However, there may be situations where the internal time delay is insufficient to discharge the LCD bias power supply before the LCD signals are shut down. For the sequence used to manually power-off the LCD panel, see Section 6.1.2, "LCD Power Disable" on page 39.

6.1.1 Registers

REG[0Dh] Display Mode Register							
SwivelView Enable	Simultaneous Display Option Select Bit 1	Simultaneous Display Option Select Bit 0	Bit-Per-Pixel Select Bit 2	Bit-Per-Pixel Select Bit 1	Bit-Per-Pixel Select Bit 0	CRT Enable	LCD Enable

The LCD Enable bit triggers all automatic power sequencing.

Setting the LCD Enable bit to 1 causes the S1D13505 to enable the LCD display. The following sequence of events occurs:

1. Confirms the LCD power is disabled.
2. Enables the LCD signals.
3. Counts 128 frames.
4. Enables the LCD power.

Setting the LCD Enable bit to 0 causes the S1D13505 to disable the LCD display. The following sequence of events occurs:

1. Disables the LCD power.
2. Counts 128 frames to wait for the LCD bias power supply to discharge.
3. Disables the LCD signals.

REG[1Ah] Power Save Configuration Register							
Power Save Status (RO)	n/a	n/a	n/a	LCD Power Disable	Suspend Refresh Select Bit 1	Suspend Refresh Select Bit 0	Software Suspend Mode Enable

The LCD Power Disable bit is used to manually power-off the LCD bias power supply. Setting the LCD Power Disable bit to 1 begins discharging the LCD bias power supply. Setting the LCD Power Disable bit to 0 causes the LCD bias power supply to power-on.

If your situation requires using the LCD Power Disable bit, see Section 6.1.2, “LCD Power Disable” on page 39 for the correct procedure. The LCD Enable bit (REG[0Dh] bit 0) should be set to 1 to allow the S1D13505 to power-on the LCD using the automatic LCD Power Sequencing.

6.1.2 LCD Power Disable

If the LCD bias power supply timing requirements are different than those timings built into the S1D13505 power disable sequence, it may be necessary to manually power-off an LCD panel. One of two situations may be true:

- Delay is too short.
- Delay is too long.

Different procedures should be used for each situation. Choose the appropriate procedure based on your requirements from the following:

Delay Too Short

To lengthen the 128 frame delay on LCDPWR.

1. Set REG[1Ah] bit 3 to 1 - 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.
3. Set REG[0Dh] bit 0 to 0 - disable the LCD outputs.

Delay Too Long

To shorten 128 frame delay on LCDPWR.

1. Set REG[23h] bit 7 to 1 - Blanks screen by disabling the FIFO.
2. Set REG[04h] to 3 (changes display width to 32 pixels)
   Set REG[08h] to 0 (changes display height to 1 line)
   - This changes the display resolution to minimum (32x1).
3. Set REG[1Ah] bit 0 to 0 - Enables power save mode.
4. Wait delay time (based on new frame rate, see *S1D13505 Hardware Functional Specification*, document number X23A-A-001-xx)
   - at this time any clocks can be disabled.
   .
   .
   .
5. Enable any clocks that were disabled in step 4.
6. Set REG[1Ah] bit 0 to 0 - Disables power save mode.
7. Set REG[04h] to original setting
   Set REG[08h] to original setting
   - Re-initializes the original resolution.
8. Set REG[023h] bit 7 to 0 - Un-blanks screen by enabling the FIFO.

6.2 Software Power Save

The S1D13505 supports a software initiated suspend power save mode. This mode is controllable using the Software Suspend Mode Enable bit in REG[1Ah]. The type of memory refresh used during suspend can also be controlled by software.

While software suspend is enabled the following conditions apply.

- display(s) are inactive
- registers are accessible
- memory is not-accessible
- LUT is accessible

6.2.1 Registers

REG[1Ah] Power Save Configuration Register							
Power Save Status (RO)	n/a	n/a	n/a	LCD Power Disable	Suspend Refresh Select Bit 1	Suspend Refresh Select Bit 0	Software Suspend Mode Enable

The Software Suspend Mode Enable bit initiates Software suspend when set to 1. Setting the bit back to 0 returns the controller back to normal mode.

REG[1Ah] Power Save Configuration Register							
Power Save Status (RO)	n/a	n/a	n/a	LCD Power Disable	Suspend Refresh Select Bit 1	Suspend Refresh Select Bit 0	Software Suspend Mode Enable

The Suspend Refresh Select Bits specify the type of DRAM refresh used during suspend mode. The type of DRAM refresh is as follows:

Table 6-1: Suspend Refresh Selection

Suspend Refresh Select Bits [1:0]	DRAM Refresh Type
00	CAS-before-RAS (CBR) refresh
01	Self-Refresh
1X	No Refresh

Note
The Suspend Refresh Select bits should never be changed while in suspend mode.

REG[1Ah] Power Save Configuration Register							
Power Save Status (RO)	n/a	n/a	n/a	LCD Power Disable	Suspend Refresh Select Bit 1	Suspend Refresh Select Bit 0	Software Suspend Mode Enable

The Power Save Status bit is a read-only status bit which indicates the power-save state of the S1D13505. When this bit returns a 1, the panel is powered-off and the memory is in a suspend memory refresh mode. When this bit returns a 0, the S1D13505 is either powered-on, in transition of powering-on, or in transition of powering-off.

6.3 Hardware Power Save

The S1D13505 supports a hardware suspend power save mode. This mode is not programmable by software. It is controlled directly by the S1D13505 SUSPEND# pin.

While hardware suspend is enabled the following conditions apply.

- display(s) are inactive
- registers are not-accessible
- memory is not-accessible
- LUT is not-accessible

7 Hardware Cursor/Ink Layer

7.1 Introduction

The S1D13505 provides hardware support for a cursor or an ink layer. These features are mutually exclusive and therefore only one or the other may be active at any given time.

A hardware cursor improves video throughput in graphical operating systems by off-loading much of the work typically assigned to software. Take the actions which must be performed when the user moves the mouse. On a system without hardware support, the operating system must restore the area under the current cursor position then save the area under the new location and finally draw the cursor shape. Contrast that with the hardware assisted system where the operating system must simply update the cursor X and cursor Y position registers.

An ink layer is used to support stylus or pen input. Without an ink layer the operating system would have to save an area (possibly all) of the display buffer where pen input was to occur. After the system recognized the user entered characters, the display would have to be restored and the characters redrawn in a system font. With an ink layer the stylus path is drawn in the ink layer, where it overlays the displayed image. After character recognition takes place the display is updated with the new characters and the ink layer is simply cleared. There is no need to save and restore display data thus providing faster throughput.

The S1D13505 hardware cursor/ink layer supports a 2 bpp (four color) overlay image. Two of the available colors are transparent and invert. The remaining two colors are user definable.

7.2 Registers

There are a total of eleven registers dedicated to the operation of the hardware cursor/ink layer. Many of the registers need only be set once. Others, such as the positional registers, will be updated frequently.

REG[27h] Ink/Cursor Control Register							
Ink/Cursor Mode bit 1	Ink/Cursor Mode bit 0	n/a	n/a	Cursor High Threshold bit 3	Cursor High Threshold bit 2	Cursor High Threshold bit 1	Cursor High Threshold bit 0

The Ink/Cursor mode bits determine if the hardware will function as a hardware cursor or as an ink layer. See Table 7-1: for an explanation of these bits.

Table 7-1: Ink/Cursor Mode

Register [27h]		**Operating**
bit 7	**bit 6**	**Mode**
0	0	Inactive
0	1	Cursor
1	0	Ink
1	1	Reserved

When cursor mode is selected the cursor image is always 64x64 pixels. Selecting an ink layer will result in a large enough area to completely cover the display.

The cursor threshold bits are used to control the Ink/Cursor FIFO depth to sustain uninterrupted display fetches.

REG[28h] Cursor X Position Register 0							
Cursor X Position bit 7	Cursor X Position bit 6	Cursor X Position bit 5	Cursor X Position bit 4	Cursor X Position bit 3	Cursor X Position bit 2	Cursor X Position bit 1	Cursor X Position bit 0

REG[29h] Cursor X Position Register 1							
Reserved	n/a	n/a	n/a	n/a	n/a	Cursor X Position bit 9	Cursor X Position bit 8

Registers [28h] and [29h] control the horizontal position of the hardware cursor. The value in this register specifies the location of the left edge of the cursor. When ink mode is selected these registers should be set to zero.

Cursor X Position bits 9-0 determine the horizontal location of the cursor. With 10 bits of resolution the horizontal cursor range is 1024 pixels.

REG[2Ah] Cursor Y Position Register 0							
Cursor Y Position bit 7	Cursor Y Position bit 6	Cursor Y Position bit 5	Cursor Y Position bit 4	Cursor Y Position bit 3	Cursor Y Position bit 2	Cursor Y Position bit 1	Cursor Y Position bit 0

REG[2Bh] Cursor Y Position Register 0							
Reserved	n/a	n/a	n/a	n/a	n/a	Cursor Y Position bit 9	Cursor Y Position bit 8

Registers [2Ah] and [2Bh] control the vertical position of the hardware cursor. The value in this register specifies the location of the left edge of the cursor. When ink mode is selected these registers should be set to zero.

Cursor Y Position bits 9-0 determine the location of the cursor. With ten bits of resolution the vertical cursor range is 1024 pixels.

REG[2Ch] Ink/Cursor Color 0 Register 0							
Cursor Color 0 bit 7	Cursor Color 0 bit 6	Cursor Color 0 bit 5	Cursor Color 0 bit 4	Cursor Color 0 bit 3	Cursor Color 0 bit 2	Cursor Color 0 bit 1	Cursor Color 0 bit 0

REG[2Dh] Ink/Cursor Color 0 Register 1							
Cursor Color 0 bit 15	Cursor Color 0 bit 14	Cursor Color 0 bit 13	Cursor Color 0 bit 12	Cursor Color 0 bit 11	Cursor Color 0 bit 10	Cursor Color 0 bit 9	Cursor Color 0 bit 8

REG[2Eh] Ink/Cursor Color 1 Register 0							
Cursor Color 1 bit 7	Cursor Color 1 bit 6	Cursor Color 1 bit 5	Cursor Color 1 bit 4	Cursor Color 1 bit 3	Cursor Color 1 bit 2	Cursor Color 1 bit 1	Cursor Color 1 bit 0

REG[2Fh] Ink/Cursor Color 1 Register 1							
Cursor Color 1 bit 15	Cursor Color 1 bit 14	Cursor Color 1 bit 13	Cursor Color 1 bit 12	Cursor Color 1 bit 11	Cursor Color 1 bit 10	Cursor Color 1 bit 9	Cursor Color 1 bit 8

Acting in pairs, Registers [2Ch], [2Dh] and registers [2Eh], [2Fh] are used to form the 16 bpp (5-6-5) RGB values for the two user defined colors.

REG[30h] Ink/Cursor Start Address Select Register							
Ink/Cursor Start Address bit 7	Ink/Cursor Start Address bit 6	Ink/Cursor Start Address bit 5	Ink/Cursor Start Address bit 4	Ink/Cursor Start Address bit 3	Ink/Cursor Start Address bit 2	Ink/Cursor Start Address bit 1	Ink/Cursor Start Address bit 0

Register [30h] determines the location in the display buffer where the cursor/ink layer will be located. Table 7-2: can be used to determine this location.

Note

Bit 7 is write only, when reading back the register this bit reads a '0'.

Table 7-2: Cursor/Ink Start Address Encoding

Ink/Cursor Start Address Bits [7:0]	Start Address (Bytes)
0	Display Buffer Size - 1024
1 - FFh	Display Buffer Size - (n * 8192)

7.3 Limitations

There are limitations for using the hardware cursor/ink layer which should be noted.

7.3.1 Updating Hardware Cursor Addresses

All hardware cursor addresses must be set during VNDP (vertical non-display period). Check the VNDP status bit (REG[0Ah] bit 7) to determine if you are in VNDP, then update the cursor address register.

7.3.2 Reg[29h] And Reg[2Bh]

Bit seven of registers [29h] and [2Bh] are write only, and must always be set to zero as setting these bits to one, will cause undefined cursor behavior.

7.3.3 Reg [30h]

Bit 7 of register [30h] is write only, therefore programs cannot determine the current cursor/ink layer start address by reading register [30h]. It is suggested that values written to this register be stored elsewhere and used when the current state of this register is required.

7.3.4 No Top/Left Clipping on Hardware Cursor

The S1D13505 does not clip the hardware cursor on the top or left edges of the display. For cursor shapes where the hot spot is not the upper left corner of the image (the hourglass for instance), the cursor image will have to be modified to clip the cursor shape.

7.4 Examples

See Section 12, “Sample Code” for hardware cursor programming examples.

8 SwivelView

8.1 Introduction To SwivelView

LCD panels are typically designed with row and column drivers mounted such that the panel's horizontal size is larger than the vertical size. These panels are typically referred to as “Landscape” panels. A minority of panels have the row and column drivers mounted such that the vertical size is larger than the horizontal size. These panels are typically referred to as “Portrait” panels. The SwivelView feature is designed to allow landscape panels to operate in a portrait orientation without the Operating System driver or software knowing the panel is not in its natural orientation. Vice-versa, this 90° rotation also allows a portrait panel to operate as a landscape panel.

The S1D13505 SwivelView option allows only 90° rotation. The display image is rotated 90° in a clockwise direction allowing the panel to be mounted 90° counter-clockwise from its normal orientation. SwivelView also provides 180° and 270° rotation on some S1D13x0x products, however, the S1D13505 does not support 180° or 270° rotation.

8.2 S1D13505 SwivelView

The S1D13505 provides hardware support for SwivelView in 8, 15 and 16 bpp modes.

Enabling SwivelView carries several conditions:

- The (virtual) display offset must be set to 1024 pixels.
- The display start address is calculated differently with SwivelView enabled.
- Calculations that would result in panning in landscape mode, result in scrolling when SwivelView is enabled and vice-versa.

8.3 Registers

This section will detail each of the registers used to setup SwivelView operations on the S1D13505. The functionality of most of these registers has been covered in previous sections but is included here to make this section complete.

The first step toward setting up SwivelView operation is to set the SwivelView Enable bit to 1 (bit 7 of register [0Dh]).

REG[0Dh] Display Mode Register							
SwivelView Enable	Simultaneous Display Option Select Bit 1	Simultaneous Display Option Select Bit 0	Bit-Per-Pixel Select Bit 2	Bit-Per-Pixel Select Bit 1	Bit-Per-Pixel Select Bit 0	CRT Enable	LCD Enable

Step two involves setting the screen 1 start address registers. Set to 1024 - width for 16 bpp modes and to (1024 - width) / 2 for 8 bpp modes.

REG[10h] Screen 1 Display Start Address Register 0							
Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0

REG[11h] Screen 1 Display Start Address Register 1							
Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8

REG[12h] Screen 1 Display Start Address Register 2							
n/a	n/a	n/a	n/a	Bit 19	Bit 18	Bit 17	Bit 16

Finally set the memory address offset registers to 1024 pixels. In 16 bpp mode load registers [17h:16h] with 1024 and in 8 bpp mode load the registers with 512.

REG[16h] Memory Address Offset Register 0							
Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0

REG[17h] Memory Address Offset Register 1							
n/a	n/a	n/a	n/a	n/a	Bit 10	Bit 9	Bit 8

8.4 Limitations

The following limitations apply to SwivelView:

- Only 8/15/16 bpp modes are supported - 1/2/4 bpp modes are not supported.
- Hardware Cursor and Ink Layer images are not rotated - software rotation must be used. SwivelView must be turned off when the programmer is accessing the Hardware Cursor or the Ink Layer.
- Split screen images appear side-by-side, i.e. when SwivelView is enabled the screen is split vertically.
- Pixel panning works vertically.

Note

Drawing into the Hardware Cursor/Ink Layer with SwivelView enabled does not work without some form of address manipulation. The easiest way to ensure correct cursor/ink images is to disable SwivelView, draw in the cursor/ink memory, then re-enable SwivelView. While writing the cursor/ink memory each pixel must be transformed to its rotated position.

8.5 Examples

Example 7: Enable SwivelView for a 640x480 display at a color depth of 8 bpp.

Before enabling SwivelView, the display buffer should be cleared to make the transition smoother. Currently displayed images cannot simply be rotated by hardware.

1. Set the line offset to 1024 pixels. The Line Offset register is the offset in words.

   Write 200h to registers [17h]:[16h]. That is write 02h to register [17h] and 00h to register [16h].

2. Set the Display 1 Start Address. The Display Start Address registers form a pointer to a word, therefore the value to set the start.

   Write C0h (192 or (1024 - 480)/2) to registers [10h], [11h] and [12h]. That is write Ch) to register [10h], 00h to register [11h] and 00h to register [12h].

3. Enable SwivelView by setting bit 7 of register [0Dh].

4. The display is now configured for SwivelView. Offset zero into display memory will correspond to the upper left corner of the display. The only difference seen by the programmer will be in acknowledging that the display offset is now 1024 pixels regardless of the physical dimensions of the display surface.

Example 8: Pan the above SwivelView image to the right by 3 pixels then scroll it up by 4 pixels.

1. With SwivelView enabled, the x and y control is rotated as well. Simply swap the x and y co-ordinates and calculate as if the display were not rotated.

2. Calculate the new start address and pixel pan values.

   BytesPerScanline = 1024

   PixelPan = newX & 01h;
   StartAddr = (newY * BytesPerScanline / 2) + (newX & FFFEh) >> 1;

3. Write the start address during the display enabled portion of the frame.

   a) loop waiting for vertical non-display (b7 of register [0Ah] high).

```
do register = ReadRegister(0Ah)
    while (80h != (register & 80h));
```

b) Loop waiting for the end of vertical non-display.

```
do register = ReadRegister(0Ah)
  while (80h == (register & 80h));
```

c) Write the new start address.

```
SetRegister(REG_SCRN1_DISP_START_ADDR0, (BYTE) (dwAddr & FFh));
SetRegister(REG_SCRN1_DISP_START_ADDR1, (BYTE)((dwAddr >> 8) & 
FFh));
SetRegister(REG_SCRN1_DISP_START_ADDR2, (BYTE)((dwAddr >> 16) & 
0Fh));
do register = ReadRegister(0Ah)
  while (80h == (register & 80h));
```

4. Write the pixel pan value during the vertical non-display portion of the frame.

a) Coming from the above code wait for beginning of the non-display period.

```
do register = ReadRegister(0Ah)
  while (80h != (register & 80h));
```

b) Write the new pixel panning value.

```
register  = ReadRegister(18h);
register &= F0h;
register |= (PixelPan & 0Fh);
WriteRegister(18h, register);
```

9 CRT Considerations

9.1 Introduction

The S1D13505 is capable of driving either an LCD panel, or a CRT display, or both simultaneously.

As display devices, panels tend to be lax in their horizontal and vertical timing requirements. CRT displays often cannot vary by more than a very small percentage in their timing requirements before the image is degraded.

Central to the following sections are VESA timings. Rather than fill this section of the guide with pages full of register values it is recommended that the program 13505CFG.EXE be used to generate a header file with the appropriate values. For more information on VESA timings contact the Video Electronics Standards association on the world-wide web at www.vesa.org.

9.1.1 CRT Only

All CRT output should meet VESA timing specifications. The VESA specification details all the parameters of the display and non-display times as well as the input clock required to meet the times. Given a proper VESA input clock the configuration program 13505CFG.EXE will generate correct VESA timings for 640x480 and for 800x600 modes.

9.1.2 Simultaneous Display

As mentioned in the previous section, CRT timings should always comply to the VESA specification. This requirement implies that during simultaneous operation the timing must still be VESA compliant. For most panels, being run at CRT frequencies is not a problem. One side effect of running with these usually slower timings will be a flicker on the panel.

One limitation of simultaneous display is that should a dual panel be the second display device the half frame buffer must be disabled for correct operation.

10 Identifying the S1D13505

The S1D13505 can only be identified once the host interface has been enabled. The steps to identify the S1D13505 are:

1. If using an ISA evaluation board in a PC follow steps a. and b.
    a. If a reset has occurred, confirm that 16-bit mode is enabled by writing to address F8 0000h.
    b. If hardware suspend is enabled then disable the suspend by writing to address F0 0000h.
2. Enable the host interface by writing 00h to REG[1Bh].
3. Read REG[00h].
4. The production version of the S1D13505 will return a value of 0Ch.

11 Hardware Abstraction Layer (HAL)

11.1 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 S1D13505 on different processor platforms. The HAL also allows for easier porting of programs between S1D1350X 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 13505CFG.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 S1D13505 without using the HAL.

11.2 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;
   WORD  wDefaultMode;
   BYTE  Regs[MAX_DISP_MODE][MAX_REG + 1];
   DWORD dwClkI;            /* Input Clock Frequency (in kHz) */
   DWORD dwBusClk;          /* Bus Clock Frequency (in kHz) */
   DWORD dwRegAddr;         /* Starting address of registers */
   DWORD dwDispMem;         /* Starting address of display buffer memory */
   WORD  wPanelFrameRate;   /* Desired panel frame rate */
   WORD  wCrtFrameRate;     /* Desired CRT rate */
   WORD  wMemSpeed;         /* Memory speed in ns */
   WORD  wTrc;              /* Ras to Cas Delay in ns */
   WORD  wTrp;              /* Ras Precharge time in ns */
   WORD  wTrac;             /* Ras Access Charge time in ns */
   WORD  wHostBusWidth;     /* Host CPU bus width in bits */
} HAL_STRUCT;
```

Within the Regs array in the structure are all the registers defined in the *S1D13505 Hardware Functional Specification*, document number X23A-A-001-xx. Using the 13505CFG.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 S1D13505 for operation (see Section 11.6.3, "Building a complete application for the target example" on page 80).

11.3 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 13505CFG.EXE. Additionally, "hal_regs.h" can be included if the programmer intends to change the S1D13505 registers directly using the seGetReg() or seSetReg() functions. For a more thorough example of using the HAL see Section 12.1.1, "Sample code using the S1D13505 HAL API" on page 84.

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.

11.4 API for 13505HAL

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 the following documentation.

Table 11-1: HAL Functions

Function	Description
Initialization:	
seRegisterDevice	Registers the S1D13505 parameters with the HAL, calls seInitHal if necessary. seRegisterDevice MUST be the first HAL function called by an application.
seInitHal	Initialize the variables used by the HAL library (called by seRegisterDevice)
seSetInit	Programs the S1D13505 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
seSetDisplayMode	Programs the S1D13505 for use with the passed display mode and flags.
General HAL Support:	
seGetId	Interpret the revision code register to determine chip id
seGetHalVersion	Return some Version information on the HAL library
seGetLibseVersion	Return version information on the LIBSE libraries (for non-x86 platforms)
seGetMemSize	Determines the amount of installed video memory

Table 11-1: HAL Functions (Continued)

Function	Description
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
seSelectBusWidth	Select the bus width on the ISA evaluation card
seGetHostBusWidth	Determine the bus width set in the HAL_STRUCT
seDisplayEnable	Turn the display(s) on/off
seDisplayFifo	Turn the FIFO on/off
seDelay	Use the frame rate timing to delay for required seconds (requires registers to be initialized)
seGetLinearDispAddr	Get a pointer to the logical start address of the display buffer
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)
Register / Memory Access:	
seSetReg	Write a Byte value to the specified S1D13505 register
seSetWordReg	Write a Word value to the specified S1D13505 register
seSetDwordReg	Write a Dword value to the specified S1D13505 register
seGetReg	Read a Byte value from the specified S1D13505 register
seGetWordReg	Read a Word value from the specified S1D13505 register
seGetDwordReg	Read a Dword value from the specified S1D13505 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
seDrawEllipse	Draw an ellipse centered at (xc,yc) of radius (xr,yr) in specified color
seDrawCircle	Draw a circle centered at (x,y) of radius r in specified color
Hardware Cursor:	
seInitCursor	Initialize hardware cursor registers and variables for use; enable cursor

Table 11-1: HAL Functions (Continued)

Function	Description
seCursorOn	Enable the cursor
seCursorOff	Disable the cursor
seGetCursorStartAddr	Determine the offset of the first byte of cursor memory in the display buffer (landscape mode)
seMoveCursor	Move the cursor to the (x.y) position specified
seSetCursorColor	Sets the specified cursor color entry (0-1) to color
seSetCursorPixel	Draw one pixel into the cursor memory at (x,y) from top left corner of cursor
seDrawCursorLine	Draw a line into the cursor memory from (x1,y1) to (x2,y2) in specified color
seDrawCursorRect	Draw a rectangle into the cursor memory from (x1,y1) to (x2,y2) in specified color
seDrawCursorEllipse	Draw an ellipse into the cursor memory centered at (xc,yc) of radius (xr,yr) in specified color
seDrawCursorCircle	Draw a circle into the cursor memory centered at (x,y) of radius r in specified color
Ink Layer:	
seInitInk	Initialize the Ink layer variables and registers; enable ink layer
seInkOn	Enables the Ink layer
seInkOff	Disables the Ink layer
seGetInkStartAddr	Determine the offset of the first byte of Ink layer memory in the display buffer (landscape mode)
seSetInkColor	Sets the specified Ink layer color entry (0-1) to color
seSetInkPixel	Draw one pixel into the Ink layer memory at (x,y) from top left corner of cursor
seDrawInkLine	Draw a line into the Ink layer memory from (x1,y1) to (x2,y2) in specified color
seDrawInkRect	Draw a rectangle into the Ink layer memory from (x1,y1) to (x2,y2) in specified color
seDrawInkEllipse	Draw an ellipse into the Ink layer memory centered at (xc,yc) of radius (xr,yr) in specified color
seDrawInkCircle	Draw a circle into the Ink layer memory centered at (x,y) of radius r in specified color
Power Save:	
seSWSuspend	Control S1D13505 SW suspend mode (enable/disable)
seHWSuspend	Control S1D13505 HW suspend mode (enable/disable)

11.5 Initialization

The following section describes the HAL functions dealing with initialization of the S1D13505. Typically a programmer will only use the calls seRegisterDevice() and seSetInit().

int seRegisterDevice(const LPHAL_STRUC lpHalInfo, int * pDevice)

Description: This function registers the S1D13505 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 as defined in appcfg.h (HalInfo)
pDevice	- pointer to the integer to receive the device ID

Return Value: ERR_OK - operation completed with no problems

Example: seRegisterDevice(&HalInfo, &DeviceId);

Note

No S1D13505 registers are changed by calling seRegisterDevice().
seRegisterDevice() MUST be called before any other HAL functions.

int seInitHal(void)

Description: This function initializes the variables used by the HAL library. This function or seRegisterDevice() must be called once when an application starts.

Normally programmers do not have to concern themselves with seInitHal(). On PC platforms, seRegisterDevice() automatically calls seInitHal(). Consecutive calls to seRegisterDevice() will not call seInitHal() again. On non-PC platforms the start-up code, supplied by Epson, will call seInitHal().However, if support code for a new operating platform is written the programmer must ensure that seInitHAL is called prior to calling other HAL functions.

Parameters: None

Return Value: ERR_OK - operation completed with no problems

seSetInit(int DevID)

Description: This routine sets the S1D13505 registers for operation using the default settings. Initialization of the S1D13505 is a two step process consisting of initializing the HAL (seInitHal) and initializing the S1D13505 registers (seSetInit). Unlike the HAL the registers do not necessarily require initialization at program startup and may be initialized as needed (e.g. 13505PLAY.EXE).

Parameters: DevID - registered device ID (acquired in seRegisterDevice)

Return Value: ERR_OK - operation completed with no problems
ERR_FAILED- unable to complete operation. Occurs as a result an invalid register in the HAL_STRUCT.

Note

This function calls seSetDisplayMode() and uses the configuration designated to be the default by 13505CFG.EXE (wDefaultMode in HAL_STRUCT). The programmer could call
seSetDisplayMode() directly allowing the selection of any DisplayMode configuration along with the options of clearing memory and blanking the display (DISP_FIFO_OFF).

Note

It is strongly recommended that the programmer call either seSetInit() or seSetDisplay-Mode()
after seRegisterDevice() before calling any other HAL functions. If not, the programmer must manually disable hardware suspend and enable the host interface before accessing the registers

int seSetDisplayMode(int DevID, int DisplayMode, int flags)

Description: This routine sets the S1D13505 registers according to the values contained in the HAL_STRUCT register section.

Setting all the registers means that timing, display surface dimensions, and all other aspects of chip operation are set with this call, including loading default values into the color Look-Up Tables (LUTs).

Parameters:

DevID	- a valid registered device ID
DisplayMode	- the HAL_STRUCT register set to use: DISP_MODE_LCD, DISP_MODE_CRT, or DISP_MODE_SIMULTANEOUS
flags	- Can be set to one or more flags. Each flag added by using the logical OR command. Do not add mutually exclusive flags. Flags can be set to 0 to use defaults. DONT_CLEAR_MEM (default) - do not clear memory CLEAR_MEM - clear display buffer memory DISP_FIFO_OFF - turn off display FIFO (blank screen except for cursor or ink layer) DISP_FIFO_ON (default) - turn on display FIFO

Return Value:

ERR_OK	- no problems encountered
ERR_FAILED	- unable to complete operation. Occurs as a result of an invalid register in the HAL_STRUCT.

See Also: seDisplayFifo() - for enabling/disabling the FIFO.

Example: seSetDisplayMode(DevID, DISP_MODE_LCD, CLEAR_MEM | DISP_FIFO_OFF);
The above example will initialize for the LCD, and then clear display buffer memory and blank the screen. The advantage to this approach is that afterwards the application can write to the display without showing the image until memory is completely updated; the application would then call seDisplayFIFO(DevID, ON).

Note
See note from seSetInit().

11.5.1 General HAL Support

General HAL support covers the miscellaneous functions. There is usually no more than one or two functions devoted to any particular aspect of S1D13505 operation.

int seGetId(int DevID, int * pId)

Description: Reads the S1D13505 revision code register to determine the chip product and revisions. The interpreted value is returned in pID.

Parameters:

DevID	- registered device ID
pId	- pointer to the int to receive the controller ID.

For the S1D13505 the return values are currently:
ID_S1D13505_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 - returned when pID returns ID_UNKNOWN. (The HAL was unable to identify the display controller).

Note

seGetId() will disable hardware suspend on x86 platforms, and will enable the host interface (register [1Bh]) on all platforms.

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 of HAL version code
pStatus - pointer to string of HAL status code (NULL is release)
pStatusRevision - pointer to string of HAL statusRevision

Return Value: None

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

Note

This document was written for HAL version “1.04”, so any later versions should be a superset of the functions described here.

void seGetLibseVersion(int ** Version)

Description: Retrieves the LIBSE library version for non-x86 platforms. The return pointer in parameter Version is valid if the function return value is ERR_OK.

Parameters: Version - pointer to an int to store LIBSE version code

Return Value: ERR_OK - no problems encountered, version code is valid
ERR_FAILED - unable to complete operation. Probably on x86 platform where LIBSE is not used.

int seGetMemSize(int DevID, DWORD * pSize)

Description: This routine returns the amount of installed video memory. The memory size is determined by reading the status of MD6 and MD7. *pSize will be set to either 80000h (512 KB) or 200000h (2 MB).

Parameters: DevID - registered device ID
pSize - pointer to a DWORD to receive the size

Return Value: ERR_OK - the operation completed successfully

Note

Memory size is only checked when calling seRegisterDevice(), seSetDisplayMode() or seSetInit(). Afterwards, the memory size is stored and made available through seGetMemSize().

int seGetLastUsableByte(int DevID, DWORD * pLastByte)

Description: Calculates the offset of the last byte in the display buffer which can be used by applications. Locations following LastByte are reserved for system use. Items such as the half frame buffer, hardware cursor and ink layer will be located in memory from GetLastUsableByte() + 1 to the end of memory.

It is assumed that the registers will have been initialized before calling seGetLastUsableByte(). Factors such as the half frame buffer and hardware cursor / ink layer being enabled dynamically alter the amount of display buffer available to an application. Call seGetLastUsableByte() any time the true end of usable memory is required.

Parameters: DevID - registered device ID
pLastByte - pointer to a DWORD to receive the offset to the last usable byte of display buffer

Return Value: ERR_OK - operation completed with no problems

int seGetBytesPerScanline(int DevID, UINT * pBytes)

Description: Determines the number of bytes per scan line of the current display mode. It is assumed that the registers have already been correctly initialized before seGetBytesPerScanline() is called.

The number of bytes per scanline calculation includes the value in the offset register. For rotated modes the return value will be either 1024 (8 bpp) or 2048 (15/16 bpp) to reflect the 1024 x 1024 virtual area of the rotated memory.

Parameters: DevID - registered device ID
pBytes - pointer to an integer which indicates the number of bytes per scan line

Return Value: ERR_OK - operation completed with no problems
ERR_FAILED- returned when this function is called for rotated display modes other than 8, 15 or 16 bpp.

int seGetScreenSize(int DevID, UINT * Width, UINT * Height)

Description: Gets 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.

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

Parameters:	DevID	- registered device ID
	Width	- unsigned integer to receive the display width
	Height	- unsigned integer to receive the display height

Return value: ERR_OK - the operation completed successfully

int seSelectBusWidth(int DevID, int Width)

Description: Call this function to select the interface bus width on the ISA evaluation card. Selectable widths are 8 bit and 16 bit.

Parameters:	DevID	- registered device ID
	Width	- desired bus width. Must be 8 or 16.

Return Value: ERR_OK - the operation completed successfully
ERR_FAILED- the function was called on a non-ISA platform or width was not set to 8 or 16.

Note
This call applies to the S1D13505 ISA evaluation cards only.

int seGetHostBusWidth(int DevID, int * Width)

Description: This function retrieves the default (as set by 13505CFG.EXE) value for the host bus interface width and returns it in Width.

Parameters:	DevID	- registered device ID
	Width	- integer to hold the returned value of the host bus width

Return Value: ERR_OK - the function completed successfully

int seDisplayEnable(int DevID, BYTE State)

Description: This routine turns the display on or off by enabling or disabling the ENABLE bit of the display device (PANEL, CRT, or SIMULTANEOUS). The Display Mode setting (LCD, CRT or SIMULTANEOUS) determines which device(s) will be affected, the default mode is stored in the HAL_STRUCT.

Parameters:	DevID	- registered device ID
	State	- set to ON or OFF to respectively enable or disable the display

Return Value: ERR_OK - the function completed successfully

int seDisplayFifo(int DevID, BYTE State)

Description: This routine turns the display on or off by enabling or disabling the display FIFO (the hardware cursor and ink layer are not affected).

To quickly blank the display, use seDisplayFifo() instead of seDisplayEnable(). Enabling and disabling the display FIFO is much faster, allowing full CPU bandwidth to the display buffer.

Parameters:	DevID	- registered device ID
	State	- set to ON or OFF respectively to enable or disable the display FIFO

Return Value:	ERR_OK	- the function completed successfully

Note
Disabling the display FIFO will force all display data outputs to zero but horizontal and vertical sync pulses and panel power supply are still active. As stated earlier, the hardware cursor and ink layer are not affected by disabling the FIFO.

int seDelay(int DevID, DWORD Seconds)

Description: This function will delay for the number of seconds given in ***Seconds*** before returning to the caller.

This function was originally intended for non-PC platforms. Because information on how to access the timers was not always immediately available, we use the frame rate for timing calculations. The S1D13505 registers must be initialized for this function to work correctly.

The PC platform version of seDelay() calls the C timing functions and is therefore independent of the register settings.

Parameters:	DevID	- registered device ID
	Seconds	- time to delay in seconds

Return Value: ERR_OK - operation completed with no problems
ERR_FAILED- returned only on non-PC platforms when the S1D13505 registers have not been initialized.

int seGetLinearDispAddr(int device, DWORD * pDispLogicalAddr)

Description: Determines the logical address of the start of the display buffer. This address may be used in programs for direct control over the display buffer.

Parameter: device - registered device ID
pDispLogicalAddr - logical address is returned in this variable.

Return Value: ERR_OK - operation completed with no problems.

11.5.2 Advanced HAL Functions

Advanced HAL functions include the functions to support split and virtual screen operations and are the same features that were described in the section on advanced programming techniques.

int seSplitInit(int DevID, DWORD Scrn1Addr, DWORD 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 respective starting addresses.

Parameters:

DevID	- registered device ID
Scrn1Addr	- offset in display buffer, in bytes, to the start of screen 1
Scrn2Addr	- offset in display buffer, 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 properly initialized prior to calling seSplitInit().

int seSplitScreen(int DevID, int WhichScreen, long VisibleScanlines)

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

The smallest screen 1 can be set to is one line. This is due to the way the register values are used internally on the S1D13505. Setting the line compare register to zero results in one line of screen 1 being displayed followed by screen 2.

Parameters:

DevID	- registered device ID
WhichScreen	- must be set to 1 or 2, or use the constants SCREEN1 or SCREEN2, to identify which screen to base calculations on
VisibleScanlines	- number of lines to show 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
seSplitInit() must be called before calling seSplitScreen()
Changing the number of lines for one screen will also change the number of lines in the other screen (e.g. increasing screen 1 lines by 5 will reduce screen 2 lines by 5).

int seVirtInit(int DevID, DWORD VirtX, DWORD * VirtY)

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

Parameter:

DevID	- registered device ID
VirtX	- horizontal size of virtual display in pixels. (Must be greater or equal to physical size of display)
VirtY	- a return placeholder for the maximum number of lines available given VirtX

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 attainable width The maximum allowable xVirt is 7FFh * (16 / bpp))
2) the virtual width is less than the physical width, or
3) the maximum number of lines is less than the physical number of lines

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

int seVirtMove(int DevID, int WhichScreen, DWORD x, DWORD y)

Description: This routine pans and scrolls the display. In the case where split screen operation is being used the WhichScreen 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:

DevID	- registered device ID
WhichScreen	- 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.

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

11.5.3 Register / Memory Access

The Register/Memory Access functions provide access to the S1D13505 registers and display buffer through the HAL.

int seSetReg(int DevID, int Index, BYTE Value)

Description: Writes Value to the register specified by Index.

Parameters:

DevID	- registered device ID
Index	- register index to set
Value	- value to write to the register

Return Value: ERR_OK - operation completed with no problems

int seSetWordReg(int DevID, int Index, WORD Value)

Description: Writes WORD sized Value to the register specified by Index.

Parameters:	DevID	- registered device ID
	Index	- register index to set
	Value	- value to write to the register

Return Value: ERR_OK - operation completed with no problems

int seSetDwordReg(int DevID, int Index, DWORD Value)

Description: Writes DWORD sized Value to the register specified by Index.

Parameters:	DevID	- registered device ID
	Index	- register index to set
	Value	- value to write to the register

Return Value: ERR_OK - operation completed with no problems

int seGetReg(int DevID, int Index, BYTE * pValue)

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

Parameters:	DevID	- registered device ID
	Index	- register index to read
	pValue	- return value of the register

Return Value: ERR_OK - operation completed with no problems

int seGetWordReg(int DevID, int Index, WORD * pValue)

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

Parameters:	DevID	- registered device ID
	Index	- register index to read
	pValue	- return value of the register

Return Value: ERR_OK - operation completed with no problems

int seGetDwordReg(int DevID, int Index, DWORD * pValue)

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

Parameters:	DevID	- registered device ID
	Index	- register index to read
	pValue	- return value of the register

Return Value: ERR_OK - operation completed with no problems

int seWriteDisplayBytes(int DevID, 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:	DevID	- registered device ID
	Offset	- offset from start of the display buffer
	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 Offset is greater than the amount of installed memory.

Note
If offset + count > memory size, this function will limit the writes to the end of memory.

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

Description: Writes one or more words to the display buffer.

Parameters:	DevID	- registered device ID
	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 Offset is greater than the amount of installed memory.

Note
If offset + (count*2) > memory size, this function will limit the writes to the end of memory.

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

Description: Writes one or more dwords to the display buffer.

Parameters:	DevID	- registered device ID
	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 Offset is greater than the amount of installed memory.

Note
If offset + (count*4) > memory size, this function will limit the writes to the end of memory.

int seReadDisplayByte(int DevID, DWORD Offset, BYTE *pByte)

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

Parameters:	DevID	- registered device ID
	Offset	- offset, in bytes, from start of the display buffer
	pByte	- return value of the display buffer location.

Return Value: ERR_OK - operation completed with no problems
ERR_HAL_BAD_ARG - if the value for Offset is greater than the amount of installed memory.

int seReadDisplayWord(int DevID, DWORD Offset, WORD *pWord)

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

Parameters:	DevID	- registered device ID
	Offset	- offset, in bytes, from start of the display buffer
	pWord	- return value of the display buffer location

Return Value: ERR_OK - operation completed with no problems.
ERR_HAL_BAD_ARG - if the value for Offset is greater than the amount of installed memory.

int seReadDisplayDword(int DevID, DWORD Offset, DWORD *pDword)

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

Parameters:	DevID	- registered device ID
	Offset	- offset from start of the display buffer
	pDword	- return value of the display buffer location

Return Value: ERR_OK - operation completed with no problems.
ERR_HAL_BAD_ARG - if the value for Offset is greater than the amount of installed memory.

11.5.4 Color Manipulation

The functions in the Color Manipulation section deal with altering the color values in the Look-Up Table directly through the accessor functions and indirectly through the color depth setting functions.

int seSetLut(int DevID, BYTE *pLut, int Count)

Description: This routine can write 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:	DevID	- registered device ID
	pLut	- pointer to an array of BYTE lut[16][3]
		lut[x][0] == RED component

lut[x][1] == GREEN component
l ut[x][2] == BLUE component
Count - the number of LUT entries to write.

Return Value: ERR_OK - operation completed with no problems

int seGetLut(int DevID, 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: DevID - registered device ID
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 DevID, 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: DevID - registered device ID
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

int seGetLutEntry(int DevID, int index, BYTE *pEntry)

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

Parameters: DevID - registered device ID
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

int seSetBitsPerPixel(int DevID, UINT BitsPerPixel)

Description: This routine sets the system color depth. Valid arguments for *BitsPerPixel* is are: 1, 2, 4, 8, 15, and 16.

After performing validity checks for the requested color depth the appropriate registers are changed and the Look-Up Table is set its default value.

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

Parameter:	DevID	- registered device ID
	BitsPerPixel	- desired color depth in bits per pixel

Return Value:	ERR_OK	- operation completed with no problems
	ERR_FAILED	- possible causes for this error message include: 1) attempted to set other than 8 or 15/16 bpp in portrait mode (portrait mode only supports 8 and 15/16 bpp) 2) factors such as input clock and memory speed will affect the ability to set some color depths. If the requested color depth cannot be set this call will fail

int seGetBitsPerPixel(int DevID, UINT * pBitsPerPixel)

Description: This function reads the S1D13505 registers to determine the current color depth and returns the result in *pBitsPerPixel.*

Determines the color depth of current display mode.

Parameters:	DevID	- registered device ID
	pBitsPerPixel	- return value is the current color depth (1/2/4/8/15/16 bpp)

Return Value:	ERR_OK	- operation completed with no problems

11.5.5 Drawing

The Drawing section covers HAL functions that deal with displaying pixels, lines and shapes.

int seSetPixel(int DevID, 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:	DevID	- Registered device ID
	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. rrrrrggggggbbbbb for 16 bpp)

Return Value:	ERR_OK	- operation completed with no problems.

int seGetPixel(int DevID, 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:	DevID	- Registered device ID
	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. rrrrrggggggbbbbb for 16 bpp)
Return Value:	ERR_OK	- operation completed with no problems.

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

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

seDrawLine() supports horizontal, vertical, and diagonal lines.

Parameters:	DevID	- registered device ID.
	(x1, y1)	- top left corner of line
	(x2, y2)	- bottom right corner of line (see note below)
	Color	- color of line
		- For 1, 2, 4, and 8 bpp, 'Color' refers to the pixel value which points to the respective LUT/DAC entry.
		- For 15 and 16 bpp, 'Color' refers to the pixel value which stores the red, green, and blue intensities within a WORD.
Return Value:	ERR_OK	- operation completed with no problems
	ERR_INVALID_REG_DEVICE	- device argument is not valid.

int seDrawRect(int DevID, 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 of the rectangle is defined by (x1,y1) and the lower right corner is defined by (x2,y2). The color, defined by *Color*, applies to the border and to the optional fill.

Parameters:	DevID	- registered device ID
	(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 solid fill
		- At 1, 2, 4, and 8 bpp Color is an index into the Look-Up Table.
		- At 15/16 bpp Color defines the color directly (i.e. rrrrrggggggbbbbb for 16 bpp)
	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.

int seDrawEllipse(int DevID, long xc, long yc, long xr, long yr, DWORD Color, BOOL SolidFill)

Description:	This routine draws an ellipse with the center located at (xc,yc). The xr and yr parameters specify the x any y radii, in pixels, respectively. The ellipse will be drawn in the color specified in 'Color'.	
Parameters:	DevID	- registered device ID
	(xc, yc)	- The center location of the ellipse (in pixels)
	xr	- horizontal radius of the ellipse (in pixels)
	yr	- vertical radius of the ellipse (in pixels)
	Color	- The color to draw the ellipse - At 1, 2, 4, and 8 bpp Color is an index into the Look-Up Table. - At 15/16 bpp Color defines the color directly (i.e. rrrrggggggbbbbb for 16 bpp)
	SolidFill	- unused
Return Value:	ERR_OK	- operation completed with no problems.

Note
The 'SolidFill' argument is currently unused and is included for future considerations.

int seDrawCircle(int DevID, long xc, long yc, long Radius, DWORD Color, BOOL SolidFill)

Description:	This routine draws an circle with the center located at (xc,yc) and a radius of Radius. The circle will be drawn in the color specified in *Color*.	
Parameters:	DevID	- registered device ID
	xc, yc	- The center of the circle (in pixels)
	Radius	- the circles radius (in pixels)
	Color	- The color to draw the ellipse - At 1, 2, 4, and 8 bpp Color is an index into the Look-Up Table. - At 15/16 bpp Color defines the color directly (i.e. rrrrggggggbbbbb for 16 bpp)
	SolidFill	- unused
Return Value:	ERR_OK	- operation completed with no problems.

Note
The *SolidFill* argument is currently unused and is included for future considerations.

11.5.6 Hardware Cursor

The routines in this section support hardware cursor functionality. Several of the calls look similar to normal drawing calls (i.e. seDrawCursorLine()); however, these calls remove the programmer from having to know the particulars of the cursor memory location, layout and whether portrait mode is enabled. Note that hardware cursor and ink layers utilize some of the same registers and are mutually exclusive.

int seInitCursor(int DevID)

Description: Prepares the hardware cursor for use. This consists of determining a location in display buffer for the cursor, setting cursor memory to the transparent color and enabling the cursor.

When this call returns the cursor is enabled, the cursor image is transparent and ready to be drawn.

Parameters:	DevID	- a registered device ID
Return Value:	ERR_OK	- operation completed with no problems

int seCursorOn(int DevID)

Description: This function enables the cursor after it has been disabled through a call to seCursorOff(). After enabling the cursor will have the same shape and position as it did prior to being disabled. The exception to the size and position occurs if the ink layer was used while the cursor was disabled.

Parameters:	DevID	- a registered device ID
Return Value:	ERR_OK	- operation completed with no problems

int seCursorOff(int DevID)

Description: This routine disables the cursor. While disabled the cursor is invisible.

Parameters:	DevID	- a registered device ID
Return Value:	ERR_OK	- operation completed with no problems

int seGetCursorStartAddr(int DevID, DWORD * Offset)

Description: This function retrieves the offset to the first byte of hardware cursor memory.

Parameters:	DevID	- a registered device ID
	Offset	- a DWORD to hold the return value.
Return Value:	ERR_OK	- the operation completed with no problems.

int seMoveCursor(int DevID, long x, long y)

Description: Moves the upper left corner of the hardware cursor to the pixel position (x,y).

Parameters:	DevID	- a registered device ID
	(x, y)	- the (x,y) position (in pixels) to move the cursor to
Return Value:	ERR_OK	- operation completed with no problems

int seSetCursorColor(int DevID, int Index, DWORD Color)

Description: Sets the color of the specified ink/cursor index to 'Color'. The user definable hardware cursor colors are 16-bit 5-6-5 RGB colors.

The hardware cursor image is always 2 bpp or four colors. Two of the colors are defined to be transparent and inverse. This leaves two colors which are user definable.

Parameters:	DevID	- a registered device ID
	Index	- the cursor index to set. Valid values are 0 and 1
	Color	- a DWORD value which hold the requested color

Return Value:	ERR_OK	- operation completed with no problems
	ERR_FAILED	- returned if Index if other than 0 or 1

int seSetCursorPixel(int DevID, long x, long y, DWORD Color)

Description: Draws a single pixel into the hardware cursor. The pixel will be of color 'Color' located at (x,y) pixels relative to the top left of the hardware cursor.

The value of 'Color' must be 0 to 3. Values 0 and 1 refer to the two user definable colors. If 'Color' is 2 then the pixel will be transparent and if the value is 3 the pixel will be an inversion of the underlying screen color.

Parameters:	DevID	- a registered device ID
	(x, y)	- draw coordinates, in pixels, relative to the top left corner of the cursor
	Color	- a value of 0 to 3 to draw the pixel with

Return Value:	ERR_OK	- operation completed with no problems

int seDrawCursorLine(int DevID, long x1, long y1, long x2, long y2, DWORD Color)

Description: Draws a line between the two endpoints, (x1,y1) and (x2,y2), in the hardware cursor display buffer using color 'Color'.

The value of 'Color' must be 0 to 3. Values 0 and 1 refer to the two user definable colors. If 'Color' is 2 then the pixel will be transparent and if the value is 3 the pixel will be an inversion of the underlying screen color.

Parameters:	DevID	- a registered device ID
	(x1,y1)	- first line endpoint (in pixels)
	(x2,y2)	- second line endpoint (in pixels)
	Color	- a value of 0 to 3 to draw the pixel with

Return Value:	ERR_OK	- operation completed with no problems

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

Description: This routine will draw a rectangle in hardware cursor memory. The upper left corner of the rectangle is defined by the point (x1,y1) and the lower right is the point (x2,y2). Both points are relative to the upper left corner of the cursor.

The value of 'Color' must be 0 to 3. Values 0 and 1 refer to the two user definable colors. If 'Color' is 2 then the pixel will be transparent and if the value is 3 the pixel result will be an inversion of the underlying screen color.

If 'SolidFill' is specified the interior of the rectangle will be filled with 'Color', otherwise the rectangle is only outlined in 'Color'.

Parameters:	DevID	- a registered device ID
	(x1,y1)	- upper left corner of the rectangle (in pixels)
	(x2,y2)	- lower right corner of the rectangle (in pixels)
	Color	- a 0 to 3 value to draw the rectangle with
	SolidFill	- flag for filling the rectangle interior - if equal to 0 then outline the rectangle; if not equal to 0 then fill the rectangle with Color

Return Value: ERR_OK - operation completed with no problems

int seDrawCursorEllipse(int DevID, long xc, long yc, long xr, long yr, DWORD Color, BOOL SolidFill)

Description: This routine draws an ellipse within the hardware cursor display buffer. The ellipse will be centered on the point (xc,yc) and will have a horizontal radius of xr and a vertical radius of yr.

The value of 'Color' must be 0 to 3. Values 0 and 1 refer to the two user definable colors. If 'Color' is 2 then the pixel will be transparent and if the value is 3 the pixel will be an inversion of the underlying screen color.

Currently seDrawCursorEllipse() does not support solid fill of the ellipse.

Parameters:	DevID	- a registered device ID
	(xc, yc)	- center of the ellipse (in pixels)
	xr	- horizontal radius (in pixels)
	yr	- vertical radius (in pixels)
	Color	- 0 to 3 value to draw the pixels with
	SolidFill	- flag to solid fill the ellipse (not currently used)

Return Value: ERR_OK - operation completed with no problems

int seDrawCursorCircle(int DevID, long x, long y, long Radius, DWORD Color, BOOL SolidFill)

Description: This routine draws a circle in hardware cursor display buffer. The center of the circle will be at (x,y) and the circle will have a radius of 'Radius' pixels.

The value of 'Color' must be 0 to 3. Values 0 and 1 refer to the two user definable colors. If 'Color' is 2 then the pixel will be transparent and if the value is 3 the pixel will be an inversion of the underlying screen color.

Currently seDrawCursorCircle() does not support the solid fill option.

Parameters:	DevID	- a registered device ID
	(x,y)	- center of the circle (in pixels)
	Radius	- radius of the circle (in pixels)
	Color	- 0 to 3 value to draw the circle with
	SolidFill	- flag to solid fill the circle (currently not used)

Return Value: ERR_OK - operation completed with no problems

11.5.7 Ink Layer

The functions in this section support the hardware ink layer. Overall these functions are nearly identical to the hardware cursor routines. In fact the same S1D13505 hardware is used for both features which means that only the cursor or the ink layer can be active at any given time.The difference between the hardware cursor and the ink layer is that in cursor mode the image is a maximum of 64x64 pixels and can be moved around the display while in ink layer mode the image is as large as the physical size of the display and is in a fixed position. Both the Ink layer and Hardware cursor have the same number of colors and handle these colors identically.

int seInitInk(int DevID)

Description: This routine prepares the ink layer for use. This consists of determining the start address for the ink layer, setting the ink layer to the transparent color and enabling the ink layer.

When this function returns the ink layer is enabled, transparent and ready to be drawn on.

Parameters: DevID - a registered device ID

Return Value: ERR_OK - operation completed with no problems
ERR_FAILED- if the ink layer cannot be enabled due to timing constraints this value will be returned.

int seInkOn(int DevID)

Description: Enables the ink layer after a call to seInkOff(). If the hardware cursor has not been used between the time seInkOff() was called and this call then the contents of the ink layer should be exactly as it was prior to the call to seInkOff().

Parameters: DevID - a registered device ID

Return Value: ERR_OK - operation completed with no problems

int seInkOff(int DevID)

Description: Disables the ink layer. When disabled the ink layer is not visible.

Parameters: DevID - a registered device ID

Return Value: ERR_OK - operation completed with no problems

int seGetInkStartAddr(int DevID, DWORD * Offset)

Description: This function retrieves the offset to the first byte of hardware ink layer memory.

Parameters: DevID - a registered device ID
Offset - a DWORD to hold the return value.

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

int seSetInkColor(int DevID, int Index, DWORD Color)

Description:	Sets the color of the specified ink/cursor index to 'Color'. The user definable hardware cursor colors are sixteen bit 5-6-5 RGB colors. The hardware ink layer image is always 2 bpp or four colors. Two of the colors are defined to be transparent and inverse. This leaves two colors which are user definable.	
Parameters:	DevID	- a registered device ID
	Index	- the index, 0 or 1, to write the color to
	Color	- a sixteen bit RRRRRGGGGGGBBBBB color to write to 'Index'
Return Value:	ERR_OK	- operation completed with no problems
	ERR_FAILED	- an index other than 0 or 1 was specified.

int seSetInkPixel(int DevID, long x, long y, DWORD Color)

Description:	Sets one pixel located at (x,y) to the value 'Color'. The point (x,y) is relative to the upper left corner of the display. The value of 'Color' must be 0 to 3. Values 0 and 1 refer to the two user definable colors. If 'Color' is 2 then the pixel will be transparent and if the value is 3 the pixel will be an inversion of the underlying screen color.	
Parameters:	DevID	- a registered device ID
	(x,y)	- coordinates of the pixel to draw
	Color	- a 0 to 3 value to draw the pixel with
Return Value:	ERR_OK	- operation completed with no problems

int seDrawInkLine(int DevID, long x1, long y1, long x2, long y2, DWORD Color)

Description:	This routine draws a line in 'Color' between the endpoints (x1,y1) and (x2,y2). The value of 'Color' must be 0 to 3. Values 0 and 1 refer to the two user definable colors. If 'Color' is 2 then the pixel will be transparent and if the value is 3 the pixel will be an inversion of the underlying screen color.	
Parameters:	DevID	- a registered device ID
	(x1,y1)	- first endpoint of the line (in pixels)
	(x2,y2)	- second endpoint of the line (in pixels)
	Color	-a value from 0 to 3 to draw the line with
Return Value:	ERR_OK	- operation completed with no problems

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

Description: Draws a rectangle of color 'Color' and optionally fills it. The upper left corner of the rectangle is the point (x1,y1) and the lower right corner of the rectangle is the point (x2,y2).

The value of 'Color' must be 0 to 3. Values 0 and 1 refer to the two user definable colors. If 'Color' is 2 then the pixel will be transparent and if the value is 3 the pixel will be an inversion of the underlying screen color.

Parameters:	DevID	- a registered device ID
	(x1,y1)	- upper left corner of the rectangle (in pixels)
	(x2.y2)	- lower right corner of the rectangle (in pixels)
	Color	- a two bit value (0 to 3) to draw the rectangle with
	SolidFill	- a flag to indicate that the interior should be filled

Return Value:	ERR_OK	- operation completed with no problems

int seDrawInkEllipse(int DevID, long xc, long yc, long xr, long yr, DWORD Color, BOOL SolidFill)

Description: This routine draws an ellipse with the center located at xc,yc. The xr and yr parameters specify the x and y radii, in pixels, respectively. The ellipse will be drawn in the color specified by *'Color'*.

The value of 'Color' must be 0 to 3. Values 0 and 1 refer to the two user definable colors. If 'Color' is 2 then the pixel will be transparent and if the value is 3 the pixel will be an inversion of the underlying screen color.

This solid fill option is not yet available for this function.

Parameters:	DevID	- a registered device ID
	xc,yc	- center point for the ellipse (in pixels)
	xr	- horizontal radius of the ellipse (in pixels)
	yr	- vertical radius of the ellipse (in pixels)
	Color	- a two bit value (0 to 3) to draw the rectangle with
	SolidFill	- flag to enable filling the interior of the ellipse (currently not used)

Return Value:	ERR_OK	- operation completed with no problems

int seDrawInkCircle(int DevID, long x, long y, long Radius, DWORD Color, BOOL SolidFill)

Description: This routine draws a circle in the ink layer display buffer. The center of the circle will be at x,y and the circle will have a radius of 'Radius' pixels.

The value of 'Color' must be 0 to 3. Values 0 and 1 refer to the two user definable colors. If 'Color' is 2 then the pixel will be transparent and if the value is 3 the pixel will be an inversion of the underlying screen color.

Currently seDrawCursorCircle() does not support the solid fill option.

Parameters:	DevID	- a registered device ID
	x,y	- center of the circle (in pixels)
	Radius	- circle radius (in pixels)
	Color	- a two bit (0 to 3) value to draw the circle with
	SolidFill	- flag to fill the interior of the circle (currently not used)

Return Value:	ERR_OK	- operation completed with no problems

11.5.8 Power Save

This section covers the HAL functions dealing with the Power Save features of the S1D13505.

int seSWSuspend(int DevID, BOOL Suspend)

Description: Causes the S1D13505 to enter software suspend mode.

When software suspend mode is engaged the display is disabled and display buffer is inaccessible. In this mode the registers and the LUT are accessible.

Parameters:	DevID	- a registered device ID
	Suspend	- boolean flag to indicate which state to engage. - enter suspend mode when non-zero and return to normal power when equal to zero.
Return Value:	ERR_OK	- operation completed with no problems

int seHWSuspend(int DevID, BOOL Suspend)

Description: Causes the S1D13505 to enter/leave hardware suspend mode. This option in only supported on S1D13505B0B ISA evaluation boards.

When hardware suspend mode is engaged the display is disabled and display buffer is inaccessible and the registers and LUT are inaccessible.

Parameters:	DevID	- a registered device ID
	Suspend	- boolean flag to indicate which state to engage. - enter suspend mode when non-zero and return to normal power when equal to zero.
Return Value:	ERR_OK	- operation completed with no problems

11.6 Porting LIBSE to a new target platform

Building Epson applications like a simple HelloApp for a new target platform requires 3 things, the HelloApp code, the 13505HAL library, and a some standard C functions (portable ones are encapsulated in our mini C library LIBSE).

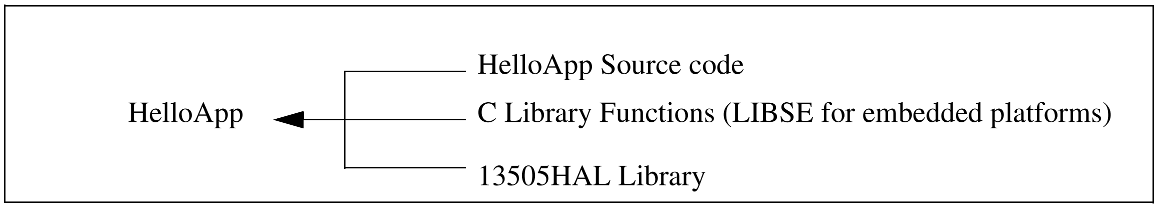

Figure 11-1: Components needed to build 13505 HAL application

For example, when building HELLOAPP.EXE for the x86 16-bit platform, you need the HELLOAPP source files, the 13505HAL 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 supplies the LIBSE for this purpose, but your compiler may come with one included. You also need to build the 13505HAL 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 S1D13505 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 Epson Electronics America Website at http://www.eea.epson.com.

11.6.1 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
```

11.6.2 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

11.6.3 Building a complete application for the target example

The following source code is available on the Epson Electronics America Website at http://www.eea.epson.com.

```
#include <stdio.h>
#include "Hal.h"
#include "Appcfg.h"
#include "Hal_regs.h"
int main(void);
#define RED16BPP   0xf800
#define GREEN16BPP 0x07e0
#define BLUE16BPP  0x001f
int main(void)
{
               int DevId;
               UINT height, width, Bpp;
               const char *p1, *p2, *p3;
               DWORD color_red, color_blue;
               BYTE RedBlueLut[3][3] = {
                                 {0, 0, 0},                        /* Black */
                                 {0xF0, 0, 0},                     /* Red */
                                 {0, 0, 0xF0}                      /* Blue */
               };
               BOOL verbose = TRUE;
               long x1, x2, y1, y2;
               /*
               ** Call this to get hal.c linked into the image, and HalInfoArray
               ** which is defined in hal.c and used by other HAL pieces.
               */
               seGetHalVersion( &p1, &p2, &p3 );
               printf("1355 Hal version %s\n", p1);
               /*
               ** Register the device with the HAL
               ** NOTE: HalInfo is an instance of HAL_STRUCT and is defined
               ** in Appcfg.h
               */
               if (seRegisterDevice(&HalInfo, &DevId) != ERR_OK)
```

```
                {
                                printf("\r\nERROR: Unable to register device with 
HAL\r\n");
                                return -1;
                }
                /*
                ** Init the SED1355 with the defaults stored in the HAL_STRUCT
                */
                if (seSetInit(DevId) != ERR_OK)
                {
                                printf("\r\nERROR: Unable to initialize the 
SED1355\r\n");
                                return -1;
                }
                /*
                ** Determine the screen size
                */
                if (seGetScreenSize(DevId, &width, &height) != ERR_OK)
                {
                                printf("\r\nERROR: Unable to get screen size\r\n");
                                return -1;
                }
                /*
                ** Determine the Bpp mode, and set colors appropriately
                ** Note: if less than 15Bpp set the color Lookup Table (LUT)
                ** local color variables contain either index into LUT or RGB value
                */
                seGetBitsPerPixel(DevId, &Bpp);
                if (verbose)
                                printf("Bpp is %d\n", Bpp);
                switch(Bpp)
                {
                                case 1: /* Can't really do red and blue here */
                                                seSetLut(DevId, (BYTE *)&RedBlue-
Lut[0][0], 3);
                                                color_red =  1;
                                                color_blue = 1;
                                                break;
                                /* Set the LUT to values appropriate to Black, Red, 
and Blue */
                                case 2:
                                case 4:
                                case 8:
                                                seSetLut(DevId, (BYTE *)&RedBlue-
Lut[0][0], 3);
                                                color_red  = 1;
                                                color_blue = 2;
                                                break;
                                default: /* 15 or 16 bpp */
                                                color_red  = RED16BPP;
```

```
                                                   color_blue = BLUE16BPP;
                                                   break;
               }
               /*
               ** Draw a Blue line from top left hand corner to bottom right hand
corner
               */
               if (seDrawLine(DevId, 0,0, width-1, height-1, color_blue) != ERR_OK)
               {
                                 printf("\r\nERROR: Unable to draw line\r\n");
                                 return -1;
               }
               /*
               ** Delay for 2 seconds and then draw a filled rectangle
               */
               seDelay(DevId, (DWORD)2);

               /*
               ** Centre the rectangle at 1/4 x,y and 3/4 x,y
               */
               x1 = width/4;
               x2 = width/2 + x1;
               y1 = height/4;
               y2 = height/2 + y1;
               seDrawRect(DevId, x1, y1, x2, y2, color_red, TRUE);

               /*
               ** Draw a box around the screen
               */
               if ((seDrawLine(DevId, 0, 0, width-1, 0, color_blue) != ERR_OK)
                  |(seDrawLine(DevId, 0, height-1, width-1, height-1, color_blue) !=
ERR_OK)
                  |(seDrawLine(DevId, 0, 0, 0, height-1, color_blue) != ERR_OK)
                  |(seDrawLine(DevId, width-1, 0, width-1, height-1, color_blue) !=
ERR_OK))
               {
                                 printf("\r\nERROR: Unable to draw box\r\n");
                                 return -1;
               }
               /*
               ** Load a cursor with a blue outlined green rectangle
               */
               seInitCursor(DevId);
               seCursorOff(DevId);
               seSetCursorColor(DevId, 0, GREEN16BPP);
               seSetCursorColor(DevId, 1, BLUE16BPP);
               seDrawCursorRect(DevId, 0, 0, 63, 63, 1, FALSE);
               seDrawCursorRect(DevId, 1, 1, 62, 62, 0, TRUE);
               seCursorOn(DevId);
```

```
                /*
                ** Delay for 2 seconds
                */
                seDelay(DevId, (DWORD)2);
                /*
                **                 Move the cursor
                */
                seMoveCursor(DevId, width-1-63, 0);

                return 0;
}
```

12 Sample Code

12.1 Introduction

There are two included examples of programming the S1D13505 color graphics controller. First is a demonstration using the HAL library and the second without. These code samples are for example purposes only. Lastly, are three header files that may make some of the structures used clearer.

12.1.1 Sample code using the S1D13505 HAL API

```
*/
// Sample code using 1355HAL API
*/

*/
**---------------------------------------------------------------------------
**
** Created 1998, Epson Research & Development
** Vancouver Design Centre
** Copyright (c) Epson Research and Development, Inc. 1998. All rights reserved.
**
** The HAL API code is configured for the following:
**
** 25.175 MHz ClkI
** 640x480 8 bit dual color STN panel @60Hz
** 50 ns EDO, 32 ms (self) refresh time
** Initial color depth - 8 bpp
**
**---------------------------------------------------------------------------
*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "hal.h"                                                     /* Structures,
constants and prototypes. */
#include "appcfg.h"                                                  /* HAL configu-
ration information. */

/*---------------------------------------------------------------------------*/
void main(void)
{
               int ChipId;
               int Device;

               /*
               ** Initialize the HAL.
```

```
        ** This step sets up the HAL for use but does not access the 1355.
        */
        switch (seRegisterDevice(&HalInfo, &Device))
        {
                  case ERR_OK:
                                    break;
                  case HAL_DEVICE_ERR:
                                    printf("\nERROR: Too many devices 
registered.");
                                    exit(1);
                  default:
                                    printf("\nERROR: Could not regis-
ter SED1355 device.");
                                    exit(1);
        }

        /*
        ** Identify that this is indeed an SED1355.
        */
        seGetId( Device, &ChipId);
        if (ID_SED1355F0A != ChipId)
        {
                  printf("\nERROR: Did not detect SED1355.");
                  exit(1);
        }

        /*
        ** Initialize the SED1355.
        ** This step will actually program the registers with values taken 
from
        ** the default register table in appcfg.h.
        */
        if (ERR_OK != seSetInit(Device))
        {
                  printf("\nERROR: Could not initialize device.");
                  exit(1);
        }

        /*
        ** The default initialization clears the display.
        ** Draw a 100x100 red rectangle in the upper left corner (0,0)
        ** of the display.
        */
        seDrawRect(Device, 0, 0, 100, 100, 1, TRUE);

        /*
        ** Init the HW cursor. The HAL performs several calculations to
        ** determine the best location to place the cursor image and
        ** will use that location from here on.
```

```
                ** The background must be set to transparent.
                */
                seInitCursor(Device);
                seDrawCursorRect(Device, 0, 0, 63, 63, 2, TRUE);



                /*
                ** Set the first user definable color to black and
                ** the second user definable color to white.
                */
                seSetCursorColor(Device, 0, 0);
                seSetCursorColor(Device, 1, 0xFFFFFFFF);

                /*
                ** Draw a hollow rectangle around the cursor and move
                ** the cursor to 101,101.
                */
                seDrawCursorRect(Device, 0, 0, 63, 63, 1, FALSE);
                seMoveCursor(Device, 101, 101);
                exit(0);
}
```

12.1.2 Sample code without using the S1D13505 HAL API

```
/*
**============================================================================
**  INIT1355.C - sample code demonstrating the initialization of the SED1355.
**               Beta release 2.0  98-10-29
**
**  The code in this example will perform initialization to the following
**  specification:
**
**  - 640 x 480 dual 16-bit color passive panel.
**  - 75 Hz frame rate.
**  - 8 BPP (256 colors).
**  - 33 MHz input clock.
**  - 2 MB of 60 ns EDO memory.
**
**                    *** This is sample code only! ***
**  This means:
**  1) Generic C is used. I assume that pointers can access the
**     relevant memory addresses (this is not always the case).
**     i.e. using the 1355B0B card on an x86 16 bit platform will require
**          changes to use a DOS extender to access memory and registers.
**  2) Register setup is done with discrete writes rather than being
**     table driven. This allows for clearer commenting. A real program
**     would probably store the register settings in an array and loop
**     through the array writing each element to a control register.
**  3) The pointer assignment for the register offset does not work on
**     x86 16 bit platforms.
**
**-----------------------------------------------------------------------------
**  Copyright (c) 1998 Epson Research and Development, Inc.
**  All Rights Reserved.
**=============================================================================
*/
/*
** Note that only the upper four bits of the LUT are actually used.
*/
unsigned char LUT8[256*3] =
{
/* Primary and secondary colors */
0x00, 0x00, 0x00,  0x00, 0x00, 0xA0,  0x00, 0xA0, 0x00,  0x00, 0xA0, 0xA0,
0xA0, 0x00, 0x00,  0xA0, 0x00, 0xA0,  0xA0, 0xA0, 0x00,  0xA0, 0xA0, 0xA0,
0x50, 0x50, 0x50,  0x00, 0x00, 0xF0,  0x00, 0xF0, 0x00,  0x00, 0xF0, 0xF0,
0xF0, 0x00, 0x00,  0xF0, 0x00, 0xF0,  0xF0, 0xF0, 0x00,  0xF0, 0xF0, 0xF0,
/* Gray shades */
0x00, 0x00, 0x00,  0x10, 0x10, 0x10,  0x20, 0x20, 0x20,  0x30, 0x30, 0x30,
0x40, 0x40, 0x40,  0x50, 0x50, 0x50,  0x60, 0x60, 0x60,  0x70, 0x70, 0x70,
0x80, 0x80, 0x80,  0x90, 0x90, 0x90,  0xA0, 0xA0, 0xA0,  0xB0, 0xB0, 0xB0,
0xC0, 0xC0, 0xC0,  0xD0, 0xD0, 0xD0,  0xE0, 0xE0, 0xE0,  0xF0, 0xF0, 0xF0,
/* Black to red */
0x00, 0x00, 0x00,  0x10, 0x00, 0x00,  0x20, 0x00, 0x00,  0x30, 0x00, 0x00,
0x40, 0x00, 0x00,  0x50, 0x00, 0x00,  0x60, 0x00, 0x00,  0x70, 0x00, 0x00,
0x80, 0x00, 0x00,  0x90, 0x00, 0x00,  0xA0, 0x00, 0x00,  0xB0, 0x00, 0x00,
0xC0, 0x00, 0x00,  0xD0, 0x00, 0x00,  0xE0, 0x00, 0x00,  0xF0, 0x00, 0x00,
/* Black to green */
0x00, 0x00, 0x00,  0x00, 0x10, 0x00,  0x00, 0x20, 0x00,  0x00, 0x30, 0x00,
0x00, 0x40, 0x00,  0x00, 0x50, 0x00,  0x00, 0x60, 0x00,  0x00, 0x70, 0x00,
0x00, 0x80, 0x00,  0x00, 0x90, 0x00,  0x00, 0xA0, 0x00,  0x00, 0xB0, 0x00,
0x00, 0xC0, 0x00,  0x00, 0xD0, 0x00,  0x00, 0xE0, 0x00,  0x00, 0xF0, 0x00,
/* Black to blue */
0x00, 0x00, 0x00,  0x00, 0x00, 0x10,  0x00, 0x00, 0x20,  0x00, 0x00, 0x30,
0x00, 0x00, 0x40,  0x00, 0x00, 0x50,  0x00, 0x00, 0x60,  0x00, 0x00, 0x70,
0x00, 0x00, 0x80,  0x00, 0x00, 0x90,  0x00, 0x00, 0xA0,  0x00, 0x00, 0xB0,
0x00, 0x00, 0xC0,  0x00, 0x00, 0xD0,  0x00, 0x00, 0xE0,  0x00, 0x00, 0xF0,
/* Blue to cyan (blue and green) */
0x00, 0x00, 0xF0,  0x00, 0x10, 0xF0,  0x00, 0x20, 0xF0,  0x00, 0x30, 0xF0,
0x00, 0x40, 0xF0,  0x00, 0x50, 0xF0,  0x00, 0x60, 0xF0,  0x00, 0x70, 0xF0,
0x00, 0x80, 0xF0,  0x00, 0x90, 0xF0,  0x00, 0xA0, 0xF0,  0x00, 0xB0, 0xF0,
0x00, 0xC0, 0xF0,  0x00, 0xD0, 0xF0,  0x00, 0xE0, 0xF0,  0x00, 0xF0, 0xF0,
/* Cyan (blue and green) to green */
0x00, 0xF0, 0xF0,  0x00, 0xF0, 0xE0,  0x00, 0xF0, 0xD0,  0x00, 0xF0, 0xC0,
0x00, 0xF0, 0xB0,  0x00, 0xF0, 0xA0,  0x00, 0xF0, 0x90,  0x00, 0xF0, 0x80,
0x00, 0xF0, 0x70,  0x00, 0xF0, 0x60,  0x00, 0xF0, 0x50,  0x00, 0xF0, 0x40,
0x00, 0xF0, 0x30,  0x00, 0xF0, 0x20,  0x00, 0xF0, 0x10,  0x00, 0xF0, 0x00,
/* Green to yellow (red and green) */
```

```
0x00, 0xF0, 0x00,  0x10, 0xF0, 0x00,  0x20, 0xF0, 0x00,  0x30, 0xF0, 0x00,
0x40, 0xF0, 0x00,  0x50, 0xF0, 0x00,  0x60, 0xF0, 0x00,  0x70, 0xF0, 0x00,
0x80, 0xF0, 0x00,  0x90, 0xF0, 0x00,  0xA0, 0xF0, 0x00,  0xB0, 0xF0, 0x00,
0xC0, 0xF0, 0x00,  0xD0, 0xF0, 0x00,  0xE0, 0xF0, 0x00,  0xF0, 0xF0, 0x00,
/* Yellow (red and green) to red */
0xF0, 0xF0, 0x00,  0xF0, 0xE0, 0x00,  0xF0, 0xD0, 0x00,  0xF0, 0xC0, 0x00,
0xF0, 0xB0, 0x00,  0xF0, 0xA0, 0x00,  0xF0, 0x90, 0x00,  0xF0, 0x80, 0x00,
0xF0, 0x70, 0x00,  0xF0, 0x60, 0x00,  0xF0, 0x50, 0x00,  0xF0, 0x40, 0x00,
0xF0, 0x30, 0x00,  0xF0, 0x20, 0x00,  0xF0, 0x10, 0x00,  0xF0, 0x00, 0x00,
/* Red to magenta (blue and red) */
0xF0, 0x00, 0x00,  0xF0, 0x00, 0x10,  0xF0, 0x00, 0x20,  0xF0, 0x00, 0x30,
0xF0, 0x00, 0x40,  0xF0, 0x00, 0x50,  0xF0, 0x00, 0x60,  0xF0, 0x00, 0x70,
0xF0, 0x00, 0x80,  0xF0, 0x00, 0x90,  0xF0, 0x00, 0xA0,  0xF0, 0x00, 0xB0,
0xF0, 0x00, 0xC0,  0xF0, 0x00, 0xD0,  0xF0, 0x00, 0xE0,  0xF0, 0x00, 0xF0,
/* Magenta (blue and red) to blue */
0xF0, 0x00, 0xF0,  0xE0, 0x00, 0xF0,  0xD0, 0x00, 0xF0,  0xC0, 0x00, 0xF0,
0xB0, 0x00, 0xF0,  0xA0, 0x00, 0xF0,  0x90, 0x00, 0xF0,  0x80, 0x00, 0xF0,
0x70, 0x00, 0xF0,  0x60, 0x00, 0xF0,  0x50, 0x00, 0xF0,  0x40, 0x00, 0xF0,
0x30, 0x00, 0xF0,  0x20, 0x00, 0xF0,  0x10, 0x00, 0xF0,  0x00, 0x00, 0xF0,
/* Black to magenta (blue and red) */
0x00, 0x00, 0x00,  0x10, 0x00, 0x10,  0x20, 0x00, 0x20,  0x30, 0x00, 0x30,
0x40, 0x00, 0x40,  0x50, 0x00, 0x50,  0x60, 0x00, 0x60,  0x70, 0x00, 0x70,
0x80, 0x00, 0x80,  0x90, 0x00, 0x90,  0xA0, 0x00, 0xA0,  0xB0, 0x00, 0xB0,
0xC0, 0x00, 0xC0,  0xD0, 0x00, 0xD0,  0xE0, 0x00, 0xE0,  0xF0, 0x00, 0xF0,
/* Black to cyan (blue and green) */
0x00, 0x00, 0x00,  0x00, 0x10, 0x10,  0x00, 0x20, 0x20,  0x00, 0x30, 0x30,
0x00, 0x40, 0x40,  0x00, 0x50, 0x50,  0x00, 0x60, 0x60,  0x00, 0x70, 0x70,
0x00, 0x80, 0x80,  0x00, 0x90, 0x90,  0x00, 0xA0, 0xA0,  0x00, 0xB0, 0xB0,
0x00, 0xC0, 0xC0,  0x00, 0xD0, 0xD0,  0x00, 0xE0, 0xE0,  0x00, 0xF0, 0xF0,
/* Red to white */
0xF0, 0x00, 0x00,  0xF0, 0x10, 0x10,  0xF0, 0x20, 0x20,  0xF0, 0x30, 0x30,
0xF0, 0x40, 0x40,  0xF0, 0x50, 0x50,  0xF0, 0x60, 0x60,  0xF0, 0x70, 0x70,
0xF0, 0x80, 0x80,  0xF0, 0x90, 0x90,  0xF0, 0xA0, 0xA0,  0xF0, 0xB0, 0xB0,
0xF0, 0xC0, 0xC0,  0xF0, 0xD0, 0xD0,  0xF0, 0xE0, 0xE0,  0xF0, 0xF0, 0xF0,
/* Green to white */
0x00, 0xF0, 0x00,  0x10, 0xF0, 0x10,  0x20, 0xF0, 0x20,  0x30, 0xF0, 0x30,
0x40, 0xF0, 0x40,  0x50, 0xF0, 0x50,  0x60, 0xF0, 0x60,  0x70, 0xF0, 0x70,
0x80, 0xF0, 0x80,  0x90, 0xF0, 0x90,  0xA0, 0xF0, 0xA0,  0xB0, 0xF0, 0xB0,
0xC0, 0xF0, 0xC0,  0xD0, 0xF0, 0xD0,  0xE0, 0xF0, 0xE0,  0xF0, 0xF0, 0xF0,
/* Blue to white */
0x00, 0x00, 0xF0,  0x10, 0x10, 0xF0,  0x20, 0x20, 0xF0,  0x30, 0x30, 0xF0,
0x40, 0x40, 0xF0,  0x50, 0x50, 0xF0,  0x60, 0x60, 0xF0,  0x70, 0x70, 0xF0,
0x80, 0x80, 0xF0,  0x90, 0x90, 0xF0,  0xA0, 0xA0, 0xF0,  0xB0, 0xB0, 0xF0,
0xC0, 0xC0, 0xF0,  0xD0, 0xD0, 0xF0,  0xE0, 0xE0, 0xF0,  0xF0, 0xF0, 0xF0
};

/*
** REGISTER_OFFSET points to the starting address of the SED1355 registers
*/
```

```
#define REGISTER_OFFSET    ((unsigned char *) 0x14000000)
/*
** DISP_MEM_OFFSET points to the starting address of the display buffer memory
*/
#define DISP_MEM_OFFSET  ((unsigned char *) 0x4000000)
/*
** DISP_MEMORY_SIZE is the size of display buffer memory
*/
#define DISP_MEMORY_SIZE   0x200000
/*
** Calculate the value to put in Ink/Cursor Start Address Select Register
**   Offset = (DISP_MEM_SIZE - (X * 8192)
** We want the offset to be just past the end of display memory so:
**   (640 * 480) = DISP_MEMORY_SIZE - (X * 8192)
**
**   CURSOR_START = (DISP_MEMORY_SIZE - (640 * 480)) / 8192
*/
#define CURSOR_START   218
void main(void)
{
  unsigned char * pRegs = REGISTER_OFFSET;
  unsigned char * pMem;
  unsigned char * pLUT;
  unsigned char * pTmp;
  unsigned char * pCursor;
  long lpCnt;
  int idx;
  int rgb;
  long x, y;
  /*
  ** Initialize the chip.
  */
  /*
  ** Step 1: Enable the host interface.
  **
  ** Register 1B: Miscellaneous Disable - host interface enabled, half frame
  **              buffer enabled.
  */
  *(pRegs + 0x1B) = 0x00;             /* 0000 0000 */
  /*
  ** Step 2: Disable the FIFO
  */
  *(pRegs + 0x23) = 0x80;             /* 1000 0000 */
  /*
  ** Step 3: Set Memory Configuration
  **
  ** Register 1: Memory Configuration - 4 ms refresh, EDO
  */
  *(pRegs + 0x01) = 0x30;             /* 0011 0000 */
```

```
/*
** Step 4: Set Performance Enhancement 0 register
*/
*(pRegs + 0x22) = 0x24;              /* 0010 0100 */
/*
** Step 5: Set the rest of the registers in order.
*/
/*
** Register 2: Panel Type - 16-bit, format 1, color, dual, passive.
*/
*(pRegs + 0x02) = 0x26;              /* 0010 0110 */
/*
** Register 3: Mod Rate
*/
*(pRegs + 0x03) = 0x00;              /* 0000 0000 */
/*
** Register 4: Horizontal Display Width (HDP) - 640 pixels
**             (640 / 8) - 1 = 79t = 4Fh
*/
*(pRegs + 0x04) = 0x4f;              /* 0100 1111 */
/*
** Register 5: Horizontal Non-Display Period (HNDP)
**                                        PCLK
**             Frame Rate = ----------------------------
**                           (HDP + HNDP) * (VDP + VNDP)
**
**                                     16,500,000
**                        = ----------------------------
**                           (640 + HNDP) * (480 + VNDP)
**
** HNDP and VNDP must be calculated such that the desired frame rate
** is achieved.
*/
*(pRegs + 0x05) = 0x1F;              /* 0001 1111 */
/*
** Register 6: HRTC/FPLINE Start Position - applicable to CRT/TFT only.
*/
*(pRegs + 0x06) = 0x00;              /* 0000 0000 */
/*
** Register 7: HRTC/FPLINE Pulse Width - applicable to CRT/TFT only.
*/
*(pRegs + 0x07) = 0x00;              /* 0000 0000 */
/*
** Registers 8-9: Vertical Display Height (VDP) - 480 lines.
**                480/2 - 1 = 239t = 0xEF
*/
*(pRegs + 0x08) = 0xEF;              /* 1110 1111 */
*(pRegs + 0x09) = 0x00;              /* 0000 0000 */
/*
```

```
** Register A: Vertical Non-Display Period (VNDP)
**             This register must be programed with register 5 (HNDP)
**             to arrive at the frame rate closest to the desired
**             frame rate.
*/
*(pRegs + 0x0A) = 0x01;             /* 0000 0001 */
/*
** Register B: VRTC/FPFRAME Start Position - applicable to CRT/TFT only.
*/
*(pRegs + 0x0B) = 0x00;             /* 0000 0000 */
/*
** Register C: VRTC/FPFRAME Pulse Width - applicable to CRT/TFT only.
*/
*(pRegs + 0x0C) = 0x00;             /* 0000 0000 */
/*
** Register D: Display Mode - 8 BPP, LCD disabled.
*/
*(pRegs + 0x0D) = 0x0C;             /* 0000 1100 */
/*
** Registers E-F: Screen 1 Line Compare - unless setting up for
**                split screen operation use 0x3FF.
*/
*(pRegs + 0x0E) = 0xFF;             /* 1111 1111 */
*(pRegs + 0x0F) = 0x03;             /* 0000 0011 */
/*
** Registers 10-12: Screen 1 Display Start Address - start at the
**                  first byte in display memory.
*/
*(pRegs + 0x10) = 0x00;             /* 0000 0000 */
*(pRegs + 0x11) = 0x00;             /* 0000 0000 */
*(pRegs + 0x12) = 0x00;             /* 0000 0000 */
/*
** Register 13-15: Screen 2 Display Start Address - not applicable
**                 unless setting up for split screen operation.
*/
*(pRegs + 0x13) = 0x00;             /* 0000 0000 */
*(pRegs + 0x14) = 0x00;             /* 0000 0000 */
*(pRegs + 0x15) = 0x00;             /* 0000 0000 */
/*
** Register 16-17: Memory Address Offset - this address represents the
**                 starting WORD. At 8BPP our 640 pixel width is 320
**                 WORDS
*/
*(pRegs + 0x16) = 0x40;             /* 0100 0000 */
*(pRegs + 0x17) = 0x01;             /* 0000 0001 */
/*
** Register 18: Pixel Panning
*/
*(pRegs + 0x18) = 0x00;             /* 0000 0000 */
```

```
/*
** Register 19: Clock Configuration - In this case we must divide
**              PCLK by 2 to arrive at the best frequency to set
**              our desired panel frame rate.
*/
*(pRegs + 0x19) = 0x01;              /* 0000 0001 */
/*
** Register 1A: Power Save Configuration - enable LCD power, CBR refresh,
**              not suspended.
*/
*(pRegs + 0x1A) = 0x00;              /* 0000 0000 */
/*
** Register 1C-1D: MD Configuration Readback - these registers are
**                 read only, but it's OK to write a 0 to keep
**                 the register configuration logic simpler.
*/
*(pRegs + 0x1C) = 0x00;              /* 0000 0000 */
*(pRegs + 0x1D) = 0x00;              /* 0000 0000 */
/*
** Register 1E-1F: General I/O Pins Configuration
*/
*(pRegs + 0x1E) = 0x00;              /* 0000 0000 */
*(pRegs + 0x1F) = 0x00;              /* 0000 0000 */
/*
** Register 20-21: General I/O Pins Control
*/
*(pRegs + 0x20) = 0x00;              /* 0000 0000 */
*(pRegs + 0x21) = 0x00;              /* 0000 0000 */
/*
** Registers 24-26: LUT control.
**                  For this example do a typical 8 BPP LUT setup.
**
** Setup the pointer to the LUT data and reset the LUT index register.
** Then, loop writing each of the RGB LUT data elements.
*/
pLUT = LUT8;
*(pRegs + 0x24) = 0;
for (idx = 0; idx < 256; idx++)
{
  for (rgb = 0; rgb < 3; rgb++)
  {
    *(pRegs + 0x26) = *pLUT;
    pLUT++;
  }
}
/*
** Register 27: Ink/Cursor Control - disable ink/cursor
*/
*(pRegs + 0x27) = 0x00;              /* 0000 0000 */
```

```
/*
** Registers 28-29: Cursor X Position
*/
*(pRegs + 0x28) = 0x00;              /* 0000 0000 */
*(pRegs + 0x29) = 0x00;              /* 0000 0000 */
/*
** Registers 2A-2B: Cursor Y Position
*/
*(pRegs + 0x2A) = 0x00;              /* 0000 0000 */
*(pRegs + 0x2B) = 0x00;              /* 0000 0000 */
/*
** Registers 2C-2D: Ink/Cursor Color 0 - blue
*/
*(pRegs + 0x2C) = 0x1F;              /* 0001 1111 */
*(pRegs + 0x2D) = 0x00;              /* 0000 0000 */
/*
** Registers 2E-2F: Ink/Cursor Color 1 - green
*/
*(pRegs + 0x2E) = 0xE0;              /* 1110 0000 */
*(pRegs + 0x2F) = 0x07;              /* 0000 0111 */
/*
** Register 30: Ink/Cursor Start Address Select
*/
*(pRegs + 0x30) = 0x00;              /* 0000 0000 */
/*
** Register 31: Alternate FRM Register
*/
*(pRegs + 0x31) = 0x00;
/*
** Register 23: Performance Enhancement - display FIFO enabled, optimum
**              performance. The FIFO threshold is set to 0x00; for
**              15/16 bpp modes, set the FIFO threshold
**              to a higher value, such as 0x1B.
*/
*(pRegs + 0x23) = 0x00;              /* 0000 0000 */
/*
** Register D: Display Mode - 8 BPP, LCD enable.
*/
*(pRegs + 0x0D) = 0x0D;              /* 0000 1101 */
/*
** Clear memory by filling 2 MB with 0
*/
pMem = DISP_MEM_OFFSET;
for (lpCnt = 0; lpCnt < DISP_MEMORY_SIZE; lpCnt++)
{
  *pMem = 0;
  pMem++;
}
/*
```

```
** Draw a 100x100 red rectangle in the upper left corner (0, 0)
** of the display.
*/
pMem = DISP_MEM_OFFSET;
for (y = 0; y < 100; y++)
{
  pTmp = pMem + y * 640L;
  for (x = 0; x < 100; x++)
  {
    *pTmp = 0x0c;
    pTmp++;
  }
}
/*
** Init the HW cursor. In this example the cursor memory will be located
** immediately after display memory. Why here? Because it's an easy
** location to calculate and will not interfere with the half frame buffer.
** Additionally, the HW cursor can be turned into an ink layer quite
** easily from this location.
*/
*(pRegs + 0x30) = CURSOR_START;
pTmp = pCursor = pMem + (DISP_MEMORY_SIZE - (CURSOR_START * 8192L));
/*
** Set the contents of the cursor memory such that the cursor
** is transparent. To do so, write a 10101010b pattern in each byte.
** The cursor is 2 bpp so a 64x64 cursor requires
** 64/4 * 64 = 1024 bytes of memory.
*/
for (lpCnt = 0; lpCnt < 1024; lpCnt++)
  {
  *pTmp = 0xAA;
  pTmp++;
  }
/*
** Set the first user definable cursor color to black and
** the second user definable cursor color to white.
*/
*(pRegs + 0x2C) = 0;
*(pRegs + 0x2D) = 0;
*(pRegs + 0x2E) = 0xFF;
*(pRegs + 0x2F) = 0xFF;
/*
** Draw a hollow rectangle around the cursor.
*/
pTmp = pCursor;
for (lpCnt = 0; lpCnt < 16; lpCnt++)
{
  *pTmp = 0x55;
  pTmp++;
  }
  for (lpCnt = 0; lpCnt < 14; lpCnt++)
  {
    *pTmp = 0x6A;
    pTmp += 15;
    *pTmp = 0xA9;
    pTmp++;
  }
  for (lpCnt = 0; lpCnt < 16; lpCnt++)
  {
    *pTmp = 0x55;
    pTmp++;
  }
  /*
  ** Move the cursor to 100, 100.
  */
  /*
  ** First we wait for the next vertical non-display
  ** period before updating the position registers.
  */
  while (*(pRegs + 0x0A) & 0x80);      /* wait while in VNDP */
  while (!(*(pRegs + 0x0A) & 0x80));  /* wait while in VDP */
  /*
  ** Now update the position registers.
  */
  *(pRegs + 0x28) = 100;   /* Set Cursor X = 100 */
  *(pRegs + 0x29) = 0x00;
  *(pRegs + 0x2A) = 100;   /* Set Cursor Y = 100 */
  *(pRegs + 0x2B) = 0x00;
  /*
  ** Enable the hardware cursor.
  */
  *(pRegs + 0x27) = 0x40;
}
}
```

12.1.3 Header Files

The following header files are included as they help to explain some of the structures used when programming the S1D13505.

The following header file defines the structure used to store the configuration information contained in all utilities using the S1D13505 HAL API.

```
/*****************************************************************************/
/*  1355 HAL INF       (do not remove)                                      */
/*  HAL_STRUCT Information generated by 1355CFG.EXE                         */
/*  Copyright (c) 1998 Epson Research and Development Inc. All rights reserved. */
/*                                                                           */
/*  Include this file ONCE in your primary source file                        */
/******************************************************************************/

HAL_STRUCT HalInfo =
{
  "1355 HAL EXE",      /* ID string   */
  0x1234,              /* Detect Endian */
  sizeof(HAL_STRUCT),  /* Size           */
  0,                   /* Default Mode */

  {
    {                /* LCD */
      0x00,  0x50,  0x16,  0x00,  0x4F,  0x03,  0x00,  0x00,
      0xEF,  0x00,  0x34,  0x00,  0x00,  0x0D,  0xFF,  0x03,
      0x00,  0x00,  0x00,  0x00,  0x00,  0x00,  0x40,  0x01,
      0x00,  0x01,  0x02,  0x00,  0x00,  0x00,  0x00,  0x00,
      0x00,  0x00,  0x48,  0x00,  0x00,  0x00,  0x00,  0x00,
      0x00,  0x00,  0x00,  0x00,  0x00,  0x00,  0x00,  0x00,
      0x00,  0x00
    },

    {                /* CRT */
      0x00,  0x50,  0x16,  0x00,  0x4F,  0x13,  0x01,  0x0B,
      0xDF,  0x01,  0x2B,  0x09,  0x01,  0x0E,  0xFF,  0x03,
      0x00,  0x00,  0x00,  0x00,  0x00,  0x00,  0x40,  0x01,
      0x00,  0x00,  0x02,  0x01,  0x00,  0x00,  0x00,  0x00,
      0x00,  0x00,  0x48,  0x00,  0x00,  0x00,  0x00,  0x00,
      0x00,  0x00,  0x00,  0x00,  0x00,  0x00,  0x00,  0x00,
      0x00,  0x00
    },

    {                /* SIMUL */
      0xFF,  0x50,  0x16,  0x00,  0x4F,  0x13,  0x01,  0x0B,
      0xDF,  0x01,  0x2B,  0x09,  0x01,  0x0F,  0xFF,  0x03,
      0x00,  0x00,  0x00,  0x00,  0x00,  0x00,  0x40,  0x01,
      0x00,  0x01,  0x02,  0x01,  0x00,  0x00,  0x00,  0x00,
      0x00,  0x00,  0x48,  0x00,  0x00,  0x00,  0x00,  0x00,
      0x00,  0x00,  0x00,  0x00,  0x00,  0x00,  0x00,  0x00,
      0x00,  0x00
    },

  },

  25175,           /* ClkI (kHz)       */
  8000,          /* BusClk (kHz)       */
  0xE00000,          /* Register Address */
  0xC00000,          /* Display Address */
  60,              /* Panel Frame Rate (Hz) */
  60,              /* CRT Frame Rate (Hz) */
  50,              /* Memory speed in ns */
  84,              /* Ras to Cas Delay in ns */
  30,              /* Ras Access Charge time in ns */
  50,              /* RAS Access Charge time in ns */
  16              /* Host CPU bus width in bits */
};
```

The following header file defines the S1D13505 HAL registers.

```
/*============================================================================
**  HAL_REGS.H
**  Created 1998, Epson Research & Development
**                Vancouver Design Center.
**  Copyright(c) Epson Research and Development Inc. 1997, 1998.  All rights
                                 reserved.
============================================================================*/

#ifndef __HAL_REGS_H__
#define __HAL_REGS_H__

/*
** 1355 register names
*/

#define REG_REVISION_CODE              0x00
#define REG_MEMORY_CONFIG              0x01
#define REG_PANEL_TYPE                 0x02
#define REG_MOD_RATE                   0x03
#define REG_HORZ_DISP_WIDTH            0x04
#define REG_HORZ_NONDISP_PERIOD        0x05
#define REG_HRTC_START_POSITION        0x06
#define REG_HRTC_PULSE_WIDTH           0x07
#define REG_VERT_DISP_HEIGHT0          0x08
#define REG_VERT_DISP_HEIGHT1          0x09
#define REG_VERT_NONDISP_PERIOD        0x0A
#define REG_VRTC_START_POSITION        0x0B
#define REG_VRTC_PULSE_WIDTH           0x0C
#define REG_DISPLAY_MODE               0x0D
#define REG_SCRN1_LINE_COMPARE0        0x0E
#define REG_SCRN1_LINE_COMPARE1        0x0F
#define REG_SCRN1_DISP_START_ADDR0     0x10
#define REG_SCRN1_DISP_START_ADDR1     0x11
#define REG_SCRN1_DISP_START_ADDR2     0x12
#define REG_SCRN2_DISP_START_ADDR0     0x13
#define REG_SCRN2_DISP_START_ADDR1     0x14
#define REG_SCRN2_DISP_START_ADDR2     0x15
#define REG_MEM_ADDR_OFFSET0           0x16
#define REG_MEM_ADDR_OFFSET1           0x17
#define REG_PIXEL_PANNING              0x18
```

```
#define REG_CLOCK_CONFIG                0x19
#define REG_POWER_SAVE_CONFIG           0x1A
#define REG_MISC                        0x1B
#define REG_MD_CONFIG_READBACK0         0x1C
#define REG_MD_CONFIG_READBACK1         0x1D
#define REG_GPIO_CONFIG0                0x1E
#define REG_GPIO_CONFIG1                0x1F
#define REG_GPIO_CONTROL0               0x20
#define REG_GPIO_CONTROL1               0x21
#define REG_PERF_ENHANCEMENT0           0x22
#define REG_PERF_ENHANCEMENT1           0x23
#define REG_LUT_ADDR                    0x24
#define REG_RESERVED_1                  0x25
#define REG_LUT_DATA                    0x26
#define REG_INK_CURSOR_CONTROL          0x27
#define REG_CURSOR_X_POSITION0          0x28
#define REG_CURSOR_X_POSITION1          0x29
#define REG_CURSOR_Y_POSITION0          0x2A
#define REG_CURSOR_Y_POSITION1          0x2B
#define REG_INK_CURSOR_COLOR0_0         0x2C
#define REG_INK_CURSOR_COLOR0_1         0x2D
#define REG_INK_CURSOR_COLOR1_0         0x2E
#define REG_INK_CURSOR_COLOR1_1         0x2F
#define REG_INK_CURSOR_START_ADDR       0x30
#define REG_ALTERNATE_FRM               0x31



/*
** WARNING!!! MAX_REG must be the last available register!!!
*/
#define MAX_REG                         0x31

#endif      /*  __HAL_REGS_H__ */
```

The following header file defines the structures used in the S1D13505 HAL API.

```
**============================================================================
** HAL.H
**----------------------------------------------------------------------------
**  Created 1998, Epson Research & Development
**                Vancouver Design Center.
**  Copyright(c) Epson Research and Development Inc. 1997, 1998.  All rights
                                  reserved.
**============================================================================
*/

#ifndef _HAL_H_
#define _HAL_H_
```

```
#pragma warning(disable:4001)   // Disable the 'single line comment' warning.


#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 DWORD far *LPDWORD;
#else
 typedef BYTE      *LPBYTE;
 typedef WORD      *LPWORD;
 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

#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_VERSION    5
#define SIZE_STATUS        2
#define SIZE_REVISION      3


#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,

   ERR_INVALID_STD_DEVICE
};

/*******************************************
 * Definitions for seGetId()
 *******************************************/
enum
{
   ID_UNKNOWN,
   ID_SED1355,
   ID_SED1355F0A
};

#define MAX_DEVICE      10

/*
** SE_RESERVED is for reserved device
*/
#define  SE_RESERVED    0

/*
** DetectEndian is used to determine whether the most significant
** and least significant bytes are reversed by the given compiler.
*/
#define ENDIAN       0x1234
#define REV_ENDIAN   0x3412


/*******************************************
 * 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


/*******************************************
```

```
 * Definitions for seSetFont
 ******************************************/

enum
{
   HAL_STDOUT,
   HAL_STDIN,
   HAL_DEVICE_ERR
};


#define FONT_NORMAL          0x00
#define FONT_DOUBLE_WIDTH    0x01
#define FONT_DOUBLE_HEIGHT   0x02

enum
{
   RED,
   GREEN,
   BLUE
};


/******************************************
 * Definitions for seSplitScreen()
 ******************************************/

enum
{
SCREEN1 = 1,
SCREEN2
};

/******************************************
 * Definitions for sePowerSaveMode()
 ******************************************/

#define PWR_CBR_REFRESH   0x00
#define PWR_SELF_REFRESH  0x01
#define PWR_NO_REFRESH    0x02

/*************************************************************************/

enum
{
DISP_MODE_LCD = 0,
DISP_MODE_CRT,
DISP_MODE_SIMULTANEOUS,
MAX_DISP_MODE
};


typedef struct tagHalStruct
{
   char  szIdString[16];
   WORD  wDetectEndian;
   WORD  wSize;
   WORD  wDefaultMode;

   BYTE Regs[MAX_DISP_MODE][MAX_REG + 1];

   DWORD dwClkI;            /* Input Clock Frequency (in kHz) */
   DWORD dwBusClk;          /* Bus Clock Frequency (in kHz) */
   DWORD dwRegAddr;         /* Starting address of registers */
   DWORD dwDispMem;         /* Starting address of display buffer memory */
   WORD  wPanelFrameRate;   /* Desired panel frame rate */

   WORD  wCrtFrameRate;     /* Desired CRT rate */
   WORD  wMemSpeed;         /* Memory speed in ns */
   WORD  wTrc;              /* Ras to Cas Delay in ns */
   WORD  wTrp;              /* Ras Precharge time in ns */
   WORD  wTrac;             /* Ras Access Charge time in ns */
   WORD  wHostBusWidth;     /* Host CPU bus width in bits */

} HAL_STRUCT;

typedef HAL_STRUCT * PHAL_STRUCT;

#ifdef INTEL
typedef HAL_STRUCT far * LPHAL_STRUCT;
#else
typedef HAL_STRUCT     * LPHAL_STRUCT;
#endif


/*=========================================================================*/
/*                          FUNCTION  PROTO-TYPES                          */
/*=========================================================================*/

/*--------------------------- HAL Support --------------------------------*/

int seInitHal( void );
int seGetDetectedBusWidth(int *bits);
int seRegisterDevice( const LPHAL_STRUCT lpHalInfo, int *Device );
int seGetMemSize( int seReserved1, DWORD *val );

#define CLEAR_MEM        TRUE
```

```
#define DONT_CLEAR_MEM   FALSE
int seSetDisplayMode(int device, int DisplayMode, int ClearMem);

int seSetInit(int device);

int  seGetId( int seReserved1, int *pId );
void seGetHalVersion( const char **pVersion, const char **pStatus, const char **pSta-
tusRevision );

/*---------------------------- Chip Access --------------------------------*/

int seGetReg( int seReserved1, int index, BYTE *pValue );
int seSetReg( int seReserved1, int index, BYTE value );

/*------------------------------- Misc ------------------------------------*/

int seSetBitsPerPixel( int seReserved1, UINT nBitsPerPixel );
int seGetBitsPerPixel( int seReserved1, UINT *pBitsPerPixel );

int seGetBytesPerScanline( int seReserved1, UINT *pBytes );
int seGetScreenSize( int seReserved1, UINT *width, UINT *height );
int seHWSuspend(int seReserved1, BOOL val);
int seSelectBusWidth(int seReserved1, int width);

int seDelay( int seReserved1, DWORD Seconds );

int seGetLastUsableByte( int seReserved1, DWORD *LastByte );
int seDisplayEnable(int seReserved1, BYTE NewState);

int seSplitInit( int seReserved1, DWORD wScrn1Addr, DWORD wScrn2Addr );
int seSplitScreen( int nReserved1, int WhichScreen, long VisibleScanlines );
int seVirtInit( int seReserved1, DWORD xVirt, DWORD *yVirt );
int seVirtMove( int seReserved1, int nWhichScreen, DWORD x, DWORD y );

/*-------------------------- Power Save -----------------------------------*/

int seSetPowerSaveMode( int seReserved1, int PowerSaveMode );

/*------------------------- Memory Access ---------------------------------*/

int seReadDisplayByte( int seReserved1, DWORD offset, BYTE *pByte );
int seReadDisplayWord( int seReserved1, DWORD offset, WORD *pWord );
int seReadDisplayDword( int seReserved1, DWORD offset, DWORD *pDword );

int seWriteDisplayBytes( int seReserved1, DWORD addr, BYTE val, DWORD count );
int seWriteDisplayWords( int seReserved1, DWORD addr, WORD val, DWORD count );
int seWriteDisplayDwords( int seReserved1, DWORD addr, DWORD val, DWORD count );

/*------------------------------- Drawing ---------------------------------*/
```

```
int seGetInkStartAddr(int seReserved1, DWORD *addr);

int seGetPixel( int seReserved1, long x, long y, DWORD *pVal );

int seSetPixel( int seReserved1, long x, long y, DWORD color );
int seDrawLine( int seReserved1, long x1, long y1, long x2, long y2, DWORD color );
int seDrawRect( int seReserved1, long x1, long y1, long x2, long y2, DWORD color,
BOOL SolidFill );
int seDrawEllipse(int seReserved1, long xc, long yc, long xr, long yr, DWORD color,
BOOL SolidFill);
int seDrawCircle( int seReserved1, long xCenter, long yCenter, long radius, DWORD
color, BOOL SolidFill );

/*------------------------------ Hardware Cursor -----------------------------*/

int seInitCursor(int seReserved1);
int seCursorOff(int seReserved1);
int seGetCursorStartAddr(int seReserved1, DWORD *addr);
int seMoveCursor(int seReserved1, long x, long y);
int seSetCursorColor(int seReserved1, int index, DWORD color);
int seSetCursorPixel( int seReserved1, long x, long y, DWORD color );
int seDrawCursorLine( int seReserved1, long x1, long y1, long x2, long y2, DWORD
color );
int seDrawCursorRect( int seReserved1, long x1, long y1, long x2, long y2, DWORD
color, BOOL SolidFill );
int seDrawCursorEllipse(int seReserved1, long xc, long yc, long xr, long yr, DWORD
color, BOOL SolidFill);
int seDrawCursorCircle( int seReserved1, long xCenter, long yCenter, long radius,
DWORD color, BOOL SolidFill );

/*------------------------------ Hardware Ink Layer --------------------------*/

int seInitInk(int seReserved1);
int seInkOff(int seReserved1);
int seGetInkStartAddr(int seReserved1, DWORD *addr);
int seSetInkColor(int seReserved1, int index, DWORD color);
int seSetInkPixel( int seReserved1, long x, long y, DWORD color );
int seDrawInkLine( int seReserved1, long x1, long y1, long x2, long y2, DWORD color
);
int seDrawInkRect( int seReserved1, long x1, long y1, long x2, long y2, DWORD color,
BOOL SolidFill );
int seDrawInkEllipse(int seReserved1, long xc, long yc, long xr, long yr, DWORD
color, BOOL SolidFill);
int seDrawInkCircle( int seReserved1, long xCenter, long yCenter, long radius, DWORD
color, BOOL SolidFill );

/*------------------------------ Color -------------------------------------*/

int seSetLut( int seReserved1, BYTE *pLut, int count );
int seGetLut( int seReserved1, BYTE *pLut, int count );
int seSetLutEntry( int seReserved1, int index, BYTE *pEntry );
int seGetLutEntry( int seReserved1, int index, BYTE *pEntry );

/*-------------------------- C Like Support ------------------------------*/

int seDrawText( int seReserved1, char *fmt, ... );
int sePutChar( int seReserved1, int ch );
int seGetChar( void );

/*-------------------------- XLIB Support --------------------------------*/

int seGetLinearDispAddr(int seReserved1, DWORD *pDispLogicalAddr);
int InitLinear(int seReserved1);

#endif      /* _HAL_H_ */
```

Appendix A Supported Panel Values

A.1 Supported Panel Values

The following tables show related register data for different panels. All the examples are based on 8 bpp and 2M bytes of 50 ns EDO-DRAM.

Note

The following settings may not reflect the ideal settings for your system configuration. Power, speed, and cost requirements may dictate different starting parameters for your system (e.g. 320x240@78Hz using 12MHz clock).

Table 12-1: Passive Single Panel @ 320x240 with 40MHz Pixel Clock

Register	**Mono 4-Bit** 320X240@60Hz	**Mono 4-Bit EL** 320X240@60Hz	**Color 8-Bit** 320X240@60Hz	**Color 8-Bit Format 2** 320X240@60Hz	Notes
REG[02h]	0000 0000	1000 0000	0001 0100	0001 1100	set panel type
REG[03h]	0000 0000	0000 0000	0000 0000	0000 0000	set MOD rate
REG[04h]	0010 0111	0010 0111	0010 0111	0010 0111	set horizontal display width
REG[05h]	0001 0111	0001 0111	0001 0111	0001 0111	set horizontal non-display period
REG[08h]	1110 1111	1110 1111	1110 1111	1110 1111	set vertical display height bits 7-0
REG[09h]	0000 0000	0000 0000	0000 0000	0000 0000	set vertical display height bits 9-8
REG[0Ah]	0011 1110	0011 1110	0011 1110	0011 1110	set vertical non-display period
REG[0Dh]	0000 1101	0000 1101	0000 1101	0000 1101	set 8 bpp and LCD enable
REG[19h]	0000 0011	0000 0011	0000 0011	0000 0011	set MCLK and PCLK divide
REG[1Bh]	0000 0001	0000 0001	0000 0001	0000 0001	disable half frame buffer
REG[24h]	0000 0000	0000 0000	0000 0000	0000 0000	set Look-Up Table address to 0
REG[26h]	load LUT	load LUT	load LUT	load LUT	load Look-Up Table

Table 12-2: Passive Single Panel @ 640x480 with 40MHz Pixel Clock

Register	**Mono 8-Bit** 640X480@60Hz	**Color 8-Bit** 640X480@60Hz	Color 16-Bit 640X480@60Hz	Notes
REG[02h]	0001 0000	0001 0100	0010 0100	set panel type
REG[03h]	0000 0000	0000 0000	0000 0000	set MOD rate
REG[04h]	0100 1111	0100 1111	0100 1111	set horizontal display width
REG[05h]	0000 0011	0000 0011	0000 0011	set horizontal non-display period
REG[08h]	1101 1111	1101 1111	1101 1111	set vertical display height bits 7-0
REG[09h]	0000 0001	0000 0001	0000 0001	set vertical display height bits 9-8
REG[0Ah]	0000 0010	0000 0010	0000 0010	set vertical non-display period
REG[0Dh]	0000 1101	0000 1101	0000 1101	set 8 bpp and LCD enable
REG[19h]	0000 0001	0000 0001	0000 0001	set MCLK and PCLK divide
REG[1Bh]	0000 0001	0000 0001	0000 0001	disable half frame buffer
REG[24h]	0000 0000	0000 0000	0000 0000	set Look-Up Table address to 0
REG[26h]	load LUT	load LUT	load LUT	load Look-Up Table

Table 12-3: Passive Dual Panel @ 640x480 with 40MHz Pixel Clock

Register	**Mono 4-Bit EL** 640X480@60Hz	**Mono 8-Bit** 640X480@60Hz	Color 8-Bit 640X480@60Hz	Color 16-Bit 640X480@60Hz	Notes
REG[02h]	1000 0010	0001 0010	0001 0110	0010 0110	set panel type
REG[03h]	0000 0000	0000 0000	0000 0000	0000 0000	set MOD rate
REG[04h]	0100 1111	0100 1111	0100 1111	0100 1111	set horizontal display width
REG[05h]	0000 0101	0000 0101	0000 0101	0000 0101	set horizontal non-display period
REG[08h]	1110 1111	1110 1111	1110 1111	1110 1111	set vertical display height bits 7-0
REG[09h]	0000 0000	0000 0000	0000 0000	0000 0000	set vertical display height bits 9-8
REG[0Ah]	0011 1110	0011 1110	0011 1110	0011 1110	set vertical non-display period
REG[0Dh]	0000 1101	0000 1101	0000 1101	0000 1101	set 8 bpp and LCD enable
REG[19h]	0000 0010	0000 0010	0000 0010	0000 0010	set MCLK and PCLK divide
REG[1Bh]	0000 0000	0000 0000	0000 0000	0000 0000	enable half frame buffer
REG[24h]	0000 0000	0000 0000	0000 0000	0000 0000	set Look-Up Table address to 0
REG[26h]	load LUT	load LUT	load LUT	load LUT	load Look-Up Table

Table 12-4: TFT Single Panel @ 640x480 with 25.175 MHz Pixel Clock

Register	Color 16-Bit 640X480@60Hz	Notes
REG[02h]	0010 0101	set panel type
REG[03h]	0000 0000	set MOD rate
REG[04h]	0100 1111	set horizontal display width
REG[05h]	0001 0011	set horizontal non-display period
REG[06h]	0000 0001	set HSYNC start position
REG[07h]	0000 1011	set HSYNC polarity and pulse width
REG[08h]	1101 1111	set vertical display height bits 7-0
REG[09h]	0000 0001	set vertical display height bits 9-8
REG[0Ah]	0010 1011	set vertical non-display period
REG[0Bh]	0000 1001	set VSYNC start position
REG[0Ch]	0000 0001	set VSYNC polarity and pulse width
REG[0Dh]	0000 1101	set 8 bpp and LCD enable
REG[19h]	0000 0000	set MCLK and PCLK divide
REG[1Bh]	0000 0001	disable half frame buffer
REG[24h]	0000 0000	set Look-Up Table address to 0
REG[26h]	load LUT	load Look-Up Table