EPSON
EXCEED YOUR VISION

S1D13A05 LCD/USB Companion Chip

Programming Notes and Examples

Document Number: X40A-G-003-06.1

SEIKO EPSON CORPORATION

Rev. 6.1

NOTICE

No part of this material may be reproduced or duplicated in any form or by any means without the written permission of Seiko Epson. Seiko Epson reserves the right to make changes to this material without notice. Seiko Epson does not assume any liability of any kind arising out of any inaccuracies contained in this material or due to its application or use in any product or circuit and, further, there is no representation that this material is applicable to products requiring high level reliability, such as, medical products. Moreover, no license to any intellectual property rights is granted by implication or otherwise, and there is no representation or warranty that anything made in accordance with this material will be free from any patent or copyright infringement of a third party. When exporting the products or technology described in this material, you should comply with the applicable export control laws and regulations and follow the procedures required by such laws and regulations. You are requested not to use, to resell, to export and/or to otherwise dispose of the products (and any technical information furnished, if any) for the development and/or manufacture of weapon of mass destruction or for other military purposes.

Table of Contents

THIS PAGE LEFT BLANK

1 Introduction

This guide discusses programming issues and provides examples for the main features of the S1D13A05, such as SwivelView, Picture-in-Picture Plus, and the BitBLT engine. The example source code referenced in this guide is available on the web at vdc.epson.com.

This guide also introduces the Hardware Abstraction Layer (HAL), which is designed to simplify the programming of the S1D13A05. Most S1D13xxx products have HAL support, thus allowing OEMs to do multiple designs with a common code base.

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

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

2 Identifying the S1D13A05

The S1D13A05 can be identified by reading the value contained in the Product Information Register (REG[00h]). To identify the S1D13A05 follow the steps below.

1. Read REG[00h].
2. The production version of the S1D13A05 returns a value of 2Dxx402Dh (where xx depends on the configuration of the CNF[6:0] pins). This value can be broken down into the following.
    1. The product code for the S1D13A05 is 0Bh (001011 binary) and can be found in bits 7-2. The product code is repeated in bits 31-26.
    2. The revision code is 1h (01 binary) and can be found in bits 1-0. The revision code is repeated in bits 25-24.
    3. The display buffer size is encoded as 40h (00101000 binary) and is contained in bits 15-8.

3 Initialization

This section describes how to initialize the S1D13A05. Sample code for performing initialization of the S1D13A05 is provided in the file **init13A05.c** which is available on the internet at vdc.epson.com.

S1D13A05 initialization can be broken into the following steps.

1. Set all registers to initial values. The values are obtained by using a **s1d13A0x.h** file exported from the 13A05CFG configuration utility. For more information on 13A05CFG, see the *13A05CFG User Manual*, document number X40A-B-001-xx.
2. Program the Look-Up Table (LUT) with color values. For details on programming the LUT, see Section 5, “Look-Up Table (LUT)” on page 13.
3. Clear the display buffer.

Refer to the HAL (Hardware Abstraction Layer) sample code available on the internet at vdc.epson.com for S5U13A05B00C specific initialization (i.e. programming the Cypress clock chip).

4 Memory Models

The S1D13A05 contains a display buffer of 256K bytes and supports color depths of 1, 2, 4, 8, and 16 bit-per-pixel. For each color depth, the data format is packed pixel.

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

4.1 Display Buffer Location

The S1D13A05 display buffer is 256K bytes of embedded SRAM. The display buffer is memory mapped and is accessible directly by software. The memory block location assigned to the S1D13A05 display buffer varies with each individual hardware platform.

For further information on the display buffer, see the *S1D13A05 Hardware Functional Specification*, document number X40A-A-001-xx.

4.2 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 4-1: Pixel Storage for 1 Bpp in One Byte of Display Buffer

At a color depth of 1 bpp, each byte of display buffer contains eight adjacent pixels. Setting or resetting any pixel requires reading the entire byte, masking out the unchanged bits and setting the appropriate bits to 1.

One bit pixels provide 2 gray shades/color possibilities. For monochrome panels the 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 2 colors are derived by indexing into the first 2 positions of the LUT.

4.3 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 bits 1-0		Pixel 1 bits 1-0		Pixel 2 bits 1-0		Pixel 3 bits 1-0	

Figure 4-2: Pixel Storage for 2 Bpp in One Byte of Display Buffer

At a color depth of 2 bpp, each byte of display buffer contains four adjacent pixels. Setting or resetting any pixel requires reading the entire byte, masking out the unchanged bits and setting the appropriate bits to 1.

Two bit pixels provide 4 gray shades/color possibilities. For monochrome panels the gray shades are generated by indexing into the first 4 elements of the green component of the Look-Up Table (LUT). For color panels the 4 colors are derived by indexing into the first 4 positions of the LUT.

4.4 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 bits 3-0				Pixel 1 bits 3-0			

Figure 4-3: Pixel Storage for 4 Bpp in One Byte of Display Buffer

At a color depth of 4 bpp, each byte of display buffer contains two adjacent pixels. Setting or resetting any pixel requires reading the entire byte, masking out the upper or lower nibble (4 bits) and setting the appropriate bits to 1.

Four bit pixels provide 16 gray shades/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 (LUT). For color panels the 16 colors are derived by indexing into the first 16 positions of the LUT.

4.5 Memory Organization for 8 Bpp (256 Colors/64 Gray Shades)

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

Figure 4-4: Pixel Storage for 8 Bpp in One Byte of Display Buffer

At a color depth of 8 bpp, each byte of display buffer represents one pixel on the display. At this color depth the read-modify-write cycles are eliminated making the update of each pixel faster.

Each byte indexes into one of the 256 positions of the LUT. The S1D13A05 LUT supports six bits per primary color. This translates into 256K possible colors when color mode is selected. Therefore the display has 256 colors available out of a possible 256K colors.

When a monochrome panel is selected, the green component of the LUT is used to determine the intensity. The green indices, with six bits, can resolve 64 gray shades. Display memory values > 64 are truncated. Thus a display memory value of 65 (1000 0001) displays the same intensity as a display memory value of 1.

4.6 Memory Organization for 16 Bpp (65536 Colors/64 Gray Shades)

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

Figure 4-5: Pixel Storage for 16 Bpp in Two Bytes of Display Buffer

At a color depth of 16 bpp the S1D13A05 is capable of displaying 64K (65536) colors. The 64K color pixel is divided into three parts: five bits for red, six bits for green, and five bits for blue. In this mode the LUT is bypassed and output goes directly into the Frame Rate Modulator.

Should monochrome mode be chosen at this color depth, the output sends the six bits of the green component to the modulator for a total of 64 possible gray shades.

5 Look-Up Table (LUT)

This section discusses programming the S1D13A05 Look-Up Table (LUT). Included is a summary of the LUT registers, recommendations for color/gray shade LUT values, and additional programming considerations. For a discussion of the LUT architecture, refer to the *S1D13A05 Hardware Functional Specification*, document number X40A-A-001-xx.

The S1D13A05 is designed with a LUT consisting of 256 red/green/blue entries. Each LUT entry is six bits wide. The color depth (bpp) determines how many indices are used. For example, 1 bpp uses the first 2 indices, 2 bpp uses the first 4 indices, 4 bpp uses the first 16 indices and 8 bpp uses all 256 indices. 16 bpp bypasses the LUT.

In color modes, the pixel values stored in the display buffer index directly to an RGB value stored in the LUT. In monochrome modes, the pixel value indexes into the green component of the LUT and the amount of green at that index controls the intensity.

5.1 Registers

5.1.1 Look-Up Table Write Register

Look-Up Table Write Register															
REG[18h]			Default = 00000000h												Write Only
LUT Write Address								LUT Red Write Data						n/a	
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
LUT Green Write Data						n/a		LUT Blue Write Data						n/a	
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

This register receives the data to be written to the red (bits 23-18), green (bits 15-10), and blue (bits 7-2) components of the Look-Up Table (LUT). Also contained in this register is the LUT Write Address (bits 31-24) which forms a pointer to the location in the LUT where the RGB components will be written.

This register should be accessed using a 32-bit write cycle to ensure proper operation. If the Look-Up Table Write Register is accessed with 8 or 16-bit write, it is important to understand that the RGB data is latched into the LUT only after the completion of the write to the LUT Write Address bits. On little endian systems, this means a write to bits 31-24. On big endian systems, this means a write to bits 7-2.

This is a write-only register and returns 00000000h if read.

Note

For further information on the S1D13A05 LUT architecture, see the *S1D13A05 Hardware Functional Specification*, document number X40A-A-001-xx.

5.1.2 Look-Up Table Read Register

Look-Up Table Read Register
REG[1Ch] Default = 00000000h Write Only (bits 31-24)/Read Only

LUT Read Address (write only)								LUT Red Read Data						n/a	
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
LUT Green Read Data						n/a		LUT Blue Read Data						n/a	
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

This register contains the data returned from the red (bits 23-18), green (bits 15-10), and blue (bits 7-2) components of the Look-Up Table. This register also contains the LUT Read Address (bits 31-24) which forms a pointer to the offset in the LUT where the RGB components are read from.

Reading the LUT is a two step process. First the desired index must be set by writing the LUT Read Address bits with the desired index. Second, the LUT values are retrieved by reading the Look-Up Table Read Register.

Bits 31-24 are write only and will return 00h when read.

Note
For further information on the S1D13A05 LUT architecture, see the *S1D13A05 Hardware Functional Specification*, document number X40A-A-001-xx.

5.2 Look-Up Table Organization

- The Look-Up Table treats the value of a pixel as an index into an array. For example, a pixel value of zero would point to the first LUT entry, whereas a pixel value of seven would point to the eighth LUT entry.
- The value contained in each LUT entry represents the intensity of the given color or gray shade. This intensity can range in value from 0 to 3Fh.
- Increasing the LUT data value results in a brighter color or gray shade. For example, a LUT entry of FCh in the red bank results in bright red output while a LUT entry of 1Ch results in dull red.

Table 5-1: Look-Up Table Configurations

Color Depth	Look-Up Table Indices Used			Effective Gray Shades/Colors
	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		64		64 gray shades
16 bpp gray				64 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

Table 5-1: Look-Up Table Configurations

Color Depth	Look-Up Table Indices Used			Effective Gray Shades/Colors
	RED	GREEN	BLUE	
8 bpp color	256	256	256	256 colors
16 bpp color	🟨	🟨	🟨	65536 colors

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

5.2.1 Gray Shade Modes

Gray shade (monochrome) modes are selected by the Color/Mono Panel Select bit (REG[0Ch] bit 6). When this bit is set to 0, the value output to the panel is derived solely from the green component of the LUT.

For each gray shade depth a table of sample LUT values is provided. These LUT values are a standardized set of intensities used by the Epson S1D13A05 utility programs.

Note

These LUT values show eight bits of significance. The S1D13A05 LUT uses only the six MSB. The 2 LSB are ignored.

1 bpp gray shade

The 1 bpp gray shade mode uses the green component of the first 2 LUT entries. The remaining indices of the LUT are unused.

Table 5-2: Suggested LUT Values for 1 Bpp Gray Shade

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

🟨 Unused entries

2 bpp gray shade

The 2 bpp gray shade mode uses the green component of the first 4 LUT entries. The remaining indices of the LUT are unused.

Table 5-3: Suggested LUT Values for 4 Bpp Gray Shade

Index	Red	Green	Blue
00	00	00	00
01	00	55	00
02	00	AA	00
03	00	FF	00
04	00	00	00
...	00	00	00
FF	00	00	00

🟨 Unused entries

4 bpp gray shade

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

Table 5-4: Suggested LUT Values for 4 Bpp Gray Shade

Index	Red	Green	Blue
00	00	00	00
01	00	10	00
02	00	20	00
03	00	34	00
04	00	44	00
05	00	54	00
06	00	68	00
07	00	78	00
08	00	88	00
09	00	9C	00
0A	00	AC	00
0B	00	BC	00
0C	00	CC	00
0D	00	DC	00
0E	00	EC	00
0F	00	FC	00
10	00	00	00
...	00	00	00
FF	00	00	00

🟨 Unused entries

8 bpp gray shade

When configured for 8 bpp gray shade mode, the green component of all 256 LUT entries may be used. The green component of the LUT has six bits of resolution resulting in 64 gray shades. The remaining 192 green color indices can be programmed, but only to one of the existing 64 intensities.

Table 5-5: Suggested LUT Values for 8 Bpp Gray Shade

Index	Red	Green	Blue	Index	Red	Green	Blue
00	00	00	00	20	00	80	00
01	00	04	00	21	00	84	00
02	00	08	00	22	00	88	00
03	00	0C	00	23	00	8C	00
04	00	10	00	24	00	90	00
05	00	14	00	25	00	94	00
06	00	18	00	26	00	98	00
07	00	1C	00	27	00	9C	00
08	00	20	00	28	00	A0	00
09	00	24	00	29	00	A4	00
0A	00	28	00	2A	00	A8	00
0B	00	2C	00	2B	00	AC	00
0C	00	30	00	2C	00	B0	00
0D	00	34	00	2D	00	B4	00
0E	00	38	00	2E	00	B8	00
0F	00	3C	00	2F	00	BC	00
10	00	40	00	30	00	C0	00
11	00	44	00	31	00	C4	00
12	00	48	00	32	00	C8	00
13	00	4C	00	33	00	CC	00
14	00	50	00	34	00	D0	00
15	00	54	00	35	00	D4	00
16	00	58	00	36	00	D8	00
17	00	5C	00	37	00	DC	00
18	00	60	00	38	00	E0	00
19	00	64	00	39	00	E4	00
1A	00	68	00	3A	00	E8	00
1B	00	6C	00	3B	00	EC	00
1C	00	70	00	3C	00	F0	00
1D	00	74	00	3D	00	F4	00
1E	00	78	00	3E	00	F8	00
1F	00	7C	00	3F	00	FC	00
				40	00	00	00
				...	00	00	00
				FF	00	00	00

🟨 Unused entries

16 bpp gray shade

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

In this mode, the six bits of green in each pixel are used to set the absolute intensity of the image. This results in 64 gray shades.

5.2.2 Color Modes

In color display modes, the number of LUT entries used is determined by the color depth. For each supported color depth a table of sample LUT values is provided. These LUT values are a standardized set of colors used by the Epson S1D13A05 utility programs.

Note

These LUT values show eight bits of significance. The S1D13A05 LUT uses only the six MSB. The 2 LSB are ignored.

1 bpp color

When the S1D13A05 is configured for 1 bpp color mode the first 2 entries in the LUT are used. The remaining indices of the LUT are unused.

Table 5-6: Suggested LUT Values for 1 bpp Color

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

🟨 = Indicates unused entries in the LUT

2 bpp color

When the S1D13A05 is configured for 2 bpp color mode the first 4 entries in the LUT are used. The remaining indices of the LUT are unused.

Table 5-7: Suggested LUT Values for 2 bpp Color

Index	Red	Green	Blue
00	00	00	00
01	00	00	FF
02	FF	00	00
03	FF	FF	FF
04	00	00	00
...	00	00	00
FF	00	00	00

🟨 = Indicates unused entries in the LUT

4 bpp color

When the S1D13A05 is configured for 4 bpp color mode the first 16 entries in the LUT are used. The remaining indices of the LUT are unused.

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

Table 5-8: Suggested LUT Values for 4 bpp Color

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

🟨 = Indicates unused entries in the LUT

8 bpp color

When the S1D13A05 is configured for 8 bpp color mode all 256 entries in the LUT are used.

The S1D13A05 LUT has six bits (64 intensities) of intensity control per primary color which is the same as a standard VGA RAMDAC.

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

Table 5-9: Suggested LUT Values 8 bpp Color

Index	R	G	B	Index	R	G	B	Index	R	G	B	Index	R	G	B
00	00	00	00	40	00	00	00	80	FF	FF	00	C0	00	00	00
01	00	00	AA	41	00	00	11	81	FF	EF	00	C1	00	11	11
02	00	AA	00	42	00	00	22	82	FF	DE	00	C2	00	22	22
03	00	AA	AA	43	00	00	33	83	FF	CD	00	C3	00	33	33
04	AA	00	00	44	00	00	44	84	FF	BC	00	C4	00	44	44
05	AA	00	AA	45	00	00	55	85	FF	AB	00	C5	00	55	55
06	AA	AA	00	46	00	00	66	86	FF	9A	00	C6	00	66	66
07	AA	AA	AA	47	00	00	77	87	FF	89	00	C7	00	77	77
08	55	55	55	48	00	00	89	88	FF	77	00	C8	00	89	89
09	00	00	FF	49	00	00	9A	89	FF	66	00	C9	00	9A	9A
0A	00	FF	00	4A	00	00	AB	8A	FF	55	00	CA	00	AB	AB
0B	00	FF	FF	4B	00	00	BC	8B	FF	44	00	CB	00	BC	BC
0C	FF	00	00	4C	00	00	CD	8C	FF	33	00	CC	00	CD	CD
0D	FF	00	FF	4D	00	00	DE	8D	FF	22	00	CD	00	DE	DE
0E	FF	FF	00	4E	00	00	EF	8E	FF	11	00	CE	00	EF	EF
0F	FF	FF	FF	4F	00	00	FF	8F	FF	00	00	CF	00	FF	FF
10	00	00	00	50	00	00	FF	90	FF	00	00	D0	FF	00	00
11	11	11	11	51	00	11	FF	91	FF	00	11	D1	FF	11	11
12	22	22	22	52	00	22	FF	92	FF	00	22	D2	FF	22	22
13	33	33	33	53	00	33	FF	93	FF	00	33	D3	FF	33	33
14	44	44	44	54	00	44	FF	94	FF	00	44	D4	FF	44	44
15	55	55	55	55	00	55	FF	95	FF	00	55	D5	FF	55	55
16	66	66	66	56	00	66	FF	96	FF	00	66	D6	FF	66	66
17	77	77	77	57	00	77	FF	97	FF	00	77	D7	FF	77	77
18	89	89	89	58	00	89	FF	98	FF	00	89	D8	FF	89	89
19	9A	9A	9A	59	00	9A	FF	99	FF	00	9A	D9	FF	9A	9A
1A	AB	AB	AB	5A	00	AB	FF	9A	FF	00	AB	DA	FF	AB	AB
1B	BC	BC	BC	5B	00	BC	FF	9B	FF	00	BC	DB	FF	BC	BC
1C	CD	CD	CD	5C	00	CD	FF	9C	FF	00	CD	DC	FF	CD	CD
1D	DE	DE	DE	5D	00	DE	FF	9D	FF	00	DE	DD	FF	DE	DE
1E	EF	EF	EF	5E	00	EF	FF	9E	FF	00	EF	DE	FF	EF	EF
1F	FF	FF	FF	5F	00	FF	FF	9F	FF	00	FF	DF	FF	FF	FF
20	00	00	00	60	00	FF	FF	A0	FF	00	FF	E0	00	FF	00
21	11	00	00	61	00	FF	EF	A1	EF	00	FF	E1	11	FF	11
22	22	00	00	62	00	FF	DE	A2	DE	00	FF	E2	22	FF	22
23	33	00	00	63	00	FF	CD	A3	CD	00	FF	E3	33	FF	33

Table 5-9: Suggested LUT Values 8 bpp Color (Continued)

Index	R	G	B	Index	R	G	B	Index	R	G	B	Index	R	G	B
24	44	00	00	64	00	FF	BC	A4	BC	00	FF	E4	44	FF	44
25	55	00	00	65	00	FF	AB	A5	AB	00	FF	E5	55	FF	55
26	66	00	00	66	00	FF	9A	A6	9A	00	FF	E6	66	FF	66
27	77	00	00	67	00	FF	89	A7	89	00	FF	E7	77	FF	77
28	89	00	00	68	00	FF	77	A8	77	00	FF	E8	89	FF	89
29	9A	00	00	69	00	FF	66	A9	66	00	FF	E9	9A	FF	9A
2A	AB	00	00	6A	00	FF	55	AA	55	00	FF	EA	AB	FF	AB
2B	BC	00	00	6B	00	FF	44	AB	44	00	FF	EB	BC	FF	BC
2C	CD	00	00	6C	00	FF	33	AC	33	00	FF	EC	CD	FF	CD
2D	DE	00	00	6D	00	FF	22	AD	22	00	FF	ED	DE	FF	DE
2E	EF	00	00	6E	00	FF	11	AE	11	00	FF	EE	EF	FF	EF
2F	FF	00	00	6F	00	FF	00	AF	00	00	FF	EF	FF	FF	FF
30	00	00	00	70	00	FF	00	B0	00	00	00	F0	00	00	FF
31	00	11	00	71	11	FF	00	B1	11	00	11	F1	11	11	FF
32	00	22	00	72	22	FF	00	B2	22	00	22	F2	22	22	FF
33	00	33	00	73	33	FF	00	B3	33	00	33	F3	33	33	FF
34	00	44	00	74	44	FF	00	B4	44	00	44	F4	44	44	FF
35	00	55	00	75	55	FF	00	B5	55	00	55	F5	55	55	FF
36	00	66	00	76	66	FF	00	B6	66	00	66	F6	66	66	FF
37	00	77	00	77	77	FF	00	B7	77	00	77	F7	77	77	FF
38	00	89	00	78	89	FF	00	B8	89	00	89	F8	89	89	FF
39	00	9A	00	79	9A	FF	00	B9	9A	00	9A	F9	9A	9A	FF
3A	00	AB	00	7A	AB	FF	00	BA	AB	00	AB	FA	AB	AB	FF
3B	00	BC	00	7B	BC	FF	00	BB	BC	00	BC	FB	BC	BC	FF
3C	00	CD	00	7C	CD	FF	00	BC	CD	00	CD	FC	CD	CD	FF
3D	00	DE	00	7D	DE	FF	00	BD	DE	00	DE	FD	DE	DE	FF
3E	00	EF	00	7E	EF	FF	00	BE	EF	00	EF	FE	EF	EF	FF
3F	00	FF	00	7F	FF	FF	00	BF	FF	00	FF	FF	FF	FF	FF

16 bpp color

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

6 Power Save Mode

The S1D13A05 is designed for very low-power applications. During normal operation, the internal clocks are dynamically disabled when not required. The S1D13A05 design also includes a Power Save Mode to further save power. When Power Save Mode is initiated, LCD power sequencing is required to ensure the LCD bias power supply is disabled properly. For further information on LCD power sequencing, see Section 6.3, “LCD Power Sequencing” on page 25.

For Power Save Mode AC Timing, see the *S1D13A05 Hardware Functional Specification*, document number X40A-A-001-xx.

6.1 Overview

The S1D13A05 includes a software initiated Power Save Mode. Enabling/disabling Power Save Mode is controlled using the Power Save Mode Enable bit (REG[14h] bit 4).

While Power Save Mode is enabled the following conditions apply.

- Most registers are accessible.
  - USB registers are not accessible
- Memory writes are possible[1]
- Memory reads are not possible
- LCD display is inactive.
- LCD interface outputs are forced low.

Note

[1] Memory writes are possible during power save mode because the S1D13A05 dynamically enables the memory controller for display buffer writes.

6.2 Registers

6.2.1 Power Save Mode Enable

Power Save Configuration Register															
REG[14h]			Default = 00000010h												Read/Write
n/a															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
n/a								VNDP Status (RO)	Memory Power Save Status (RO)	n/a	Power Save Mode Enable	n/a			Reserved
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

The Power Save Mode Enable bit initiates Power Save Mode when set to 1. Setting the bit to 0 disables Power Save Mode and returns the S1D13A05 to normal mode. At reset this bit is set to 1.

Note

Enabling/disabling Power Save Mode requires proper LCD Power Sequencing. See Section 6.3, “LCD Power Sequencing” on page 25.

6.2.2 Memory Controller Power Save Status

Power Save Configuration Register															
REG[14h]			Default = 00000010h												Read/Write
n/a															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
n/a								VNDP Status (RO)	Memory Power Save Status (RO)	n/a	Power Save Mode Enable	n/a			Reserved
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

The Memory Controller Power Save Status bit is a read-only status bit which indicates the power save state of the S1D13A05 SRAM interface. When this bit returns a 1, the SRAM interface is powered down and the memory clock source may be disabled. When this bit returns a 0, the SRAM interface is active. This bit returns a 0 after a chip reset.

Note

Memory writes are possible during power save mode because the S1D13A05 dynamically enables the memory controller for display buffer writes.

6.3 LCD Power Sequencing

The S1D13A05 requires 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, preventing long term damage to the panel and avoiding unsightly “lines” at power-on/power-off.

Proper LCD power sequencing for power-off requires a 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 depends on the LCD bias power supply design. For example, the LCD bias power supply on the S1D13A05 Evaluation Board requires 0.5 seconds to fully discharge. Other power supply designs may vary.

This section assumes the LCD bias power is controlled through GPIO0. The S1D13A05 GPIO pins are multi-use pins and may not be available in all system designs. For further information on the availability of GPIO pins, see the *S1D13A05 Hardware Functional Specification*, document number X40A-A-001-xx.

6.4 Enabling Power Save Mode

Enable Power Save Mode using the following steps.

1. Turn off the LCD bias power.

Note
The S1D13A05 evaluation board uses GPIO0 to control the LCD bias power supplies. Your system design may vary.

2. Wait for the LCD bias power supply to discharge. The discharge time is based on the discharge rate of the power supply.
3. Enable Power Save Mode - set REG[14h] bit 4 to 1.

The S1D13A05 is now in Power Save Mode. To further increase power savings PCLK and MCLK can be switched off (see steps 4 and 5).

4. At this time, the LCD pixel clock source may be disabled.
5. After the Memory Controller Power Save Status bit (REG[14h] bit 6) returns a 1, the Memory Clock source may be shut down.

6.5 Disabling Power Save Mode

Bring the S1D13A05 out of Power Save Mode using the following steps.

1. If the Memory Clock source is shut down, it must be started.
2. If the pixel clock is disabled, it must be started.
3. Disable Power Save Mode - set REG[14h] bit 4 to 0.
4. Wait for the LCD bias power supply to charge. The charge time is based on the time required for the LCD power supply to reach operating voltage.
5. Enable the LCD bias power.

Note
The S1D13A05 evaluation board uses GPIO0 to control the LCD bias power supplies. Your system design may vary.

7 SwivelView™

Computer displays usually show an image in only one orientation, which is typically wider than it is high. For example, a display size of 320x240 is 320 pixels wide and 240 lines high.

SwivelView uses hardware to rotate the displayed image counter-clockwise in ninety degree increments. Rotating the image on a 320x240 display by 90° or 270° yields a display that is now 240 pixels wide and 320 lines high.

The S1D13A05 provides hardware support for SwivelView in the following configurations:

- SwivelView 0° (landscape) with or without pixel doubling
- SwivelView 90° without pixel doubling
- SwivelView 180° with or without pixel doubling
- SwivelView 270° without pixel doubling

The SwivelView feature only affects the main display window and the PIP^+ window.

7.1 SwivelView 90°

When design constraints require physical rotation of a display by 90°, enable SwivelView 90° mode. The following example shows a LCD panel, which is physically rotated clockwise by 90°. The top left corner of the physical panel is marked for reference.

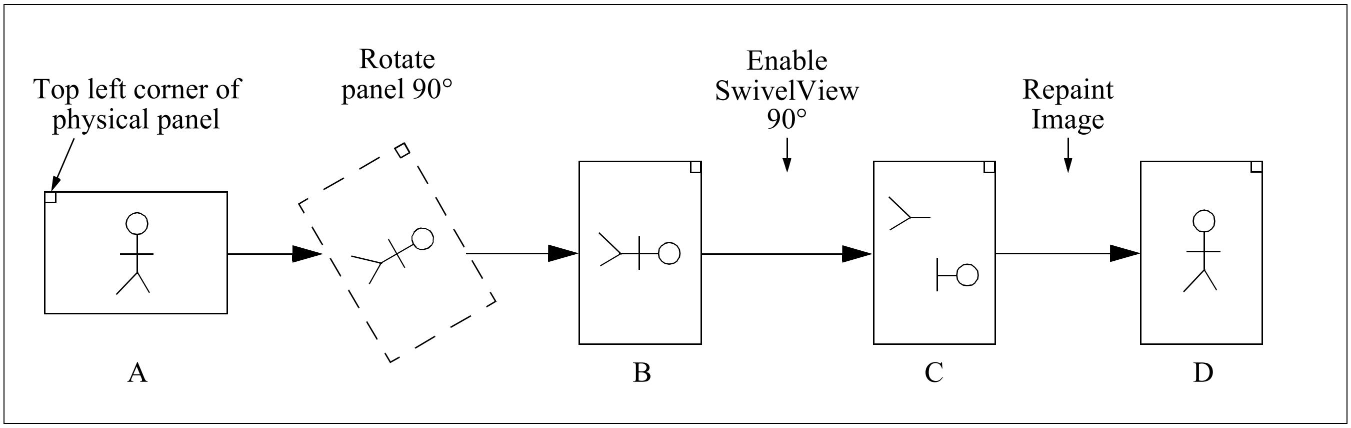


Figure 7-1: SwivelView 90° Rotation

The above illustration shows a series of transitions:

- A to B shows the LCD panel being physically rotated. Note that the physical display is rotated clockwise and the image is rotated counter-clockwise.
- B to C shows the effect of enabling SwivelView 90°. The broken image in C indicates that after the registers are programmed the image in display memory is invalid and must be repainted. The following register values must be updated to select SwivelView 90°:
  - SwivelView Mode Select (REG[010h] bits 17 and 16)
  - Main Window Display Start Address (REG[040h])
  - Main Window Line Address Offset (REG[044h])
  - PIP$^+$ Line Address Offset (REG[054h])
  - PIP$^+$ X End Position (REG[058h] bits 25-16)
  - PIP$^+$ X Start Position (REG[058h] bits 9-0)
  - PIP$^+$ Y End Position (REG[05Ch] bits 25-16)
  - PIP$^+$ Y Start Position (REG[05Ch] bits 9-0)
- C to D shows the effect of repainting the display in SwivelView 90°. The image must be drawn based on the new display dimensions.

7.2 SwivelView 180°

When design constraints require physical rotation of a display by 180°, enable SwivelView 180° mode. The following example shows a LCD panel which is physically rotated by 180°. The top left corner of the physical panel is marked for reference.

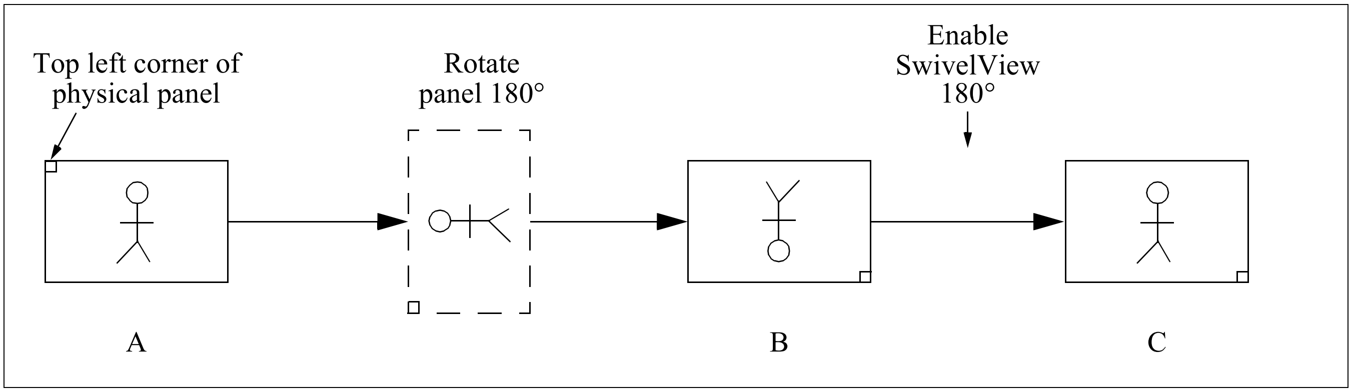


Figure 7-2: SwivelView 90° Rotation

The above illustration shows a series of transitions:

- A to B shows the LCD panel being physically rotated.
- B to C shows the effect of enabling SwivelView 180°. The following register values must be updated to support SwivelView 180°:
  - SwivelView Mode Select (REG[010h] bits 17 and 16)
  - Main Window Display Start Address (REG[040h])
  - Main Window Line Address Offset (REG[044h])
  - PIP$^+$ X End Position (REG[058h] bits 25-16)
  - PIP$^+$ X Start Position (REG[058h] bits 9-0)
  - PIP$^+$ Y End Position (REG[05Ch] bits 25-16)
  - PIP$^+$ Y Start Position (REG[05Ch] bits 9-0)
- Notice that the image does not have to be repainted when SwivelView 180° is enabled.

7.3 SwivelView 270°

When design constraints require physical rotation of a display by 270°, enable SwivelView 270° mode. The following example shows a LCD panel which is physically rotated clockwise by 270°. The top left corner of the physical panel is marked for reference.

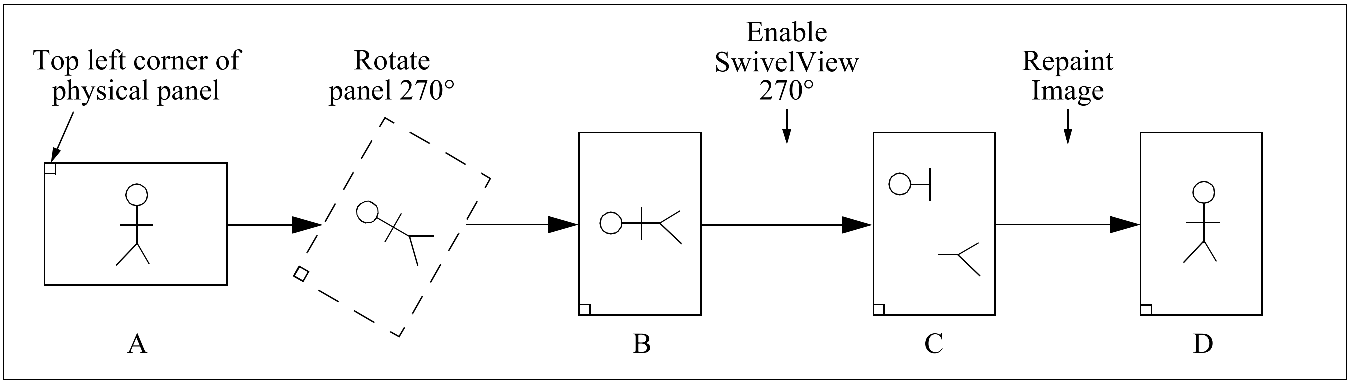


Figure 7-3: SwivelView 270° Rotation

The above illustration shows a series of transitions:

- A to B shows the LCD panel being physically rotated.
- B to C shows the effect of enabling SwivelView 270°. The broken image in Box C indicates that after registers are programmed the image in display memory is invalid and must be repainted. The following register values must be updated to support SwivelView 270°:
  - SwivelView Mode Select (REG[010h] bits 17 and 16)
  - Main Window Display Start Address (REG[040h])
  - Main Window Line Address Offset (REG[044h])
  - PIP^+ X End Position (REG[058h] bits 25-16)
  - PIP^+ X Start Position (REG[058h] bits 9-0)
  - PIP^+ Y End Position (REG[05Ch] bits 25-16)
  - PIP^+ Y Start Position (REG[05Ch] bits 9-0)
- C to D shows the effect of repainting the display in SwivelView 270°. The image must be drawn based on the new display dimensions.

7.4 SwivelView Registers

These registers control the SwivelView feature.

Display Settings Register
REG[10h] Default = 00000000h Read/Write

n/a						Pixel Doubling Vertical	Pixel Doubling Horiz.	Display Blank	Dithering Disable	n/a	SW Video Invert	PIP$^+$ Window Enable	n/a	SwivelView Mode Select	
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
n/a											Bits-per-pixel Select (actual value: 1, 2, 4, 8 or 16 bpp)				
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

SwivelView Mode Select

The SwivelView modes are selected using the SwivelView Mode Select Bits[1:0] (bits 17-16). The combinations of these bits provide the following rotations.

Table 7-1: SwivelView Mode Select Bits

SwivelView Mode Select Bit 1	SwivelView Mode Select Bit 0	SwivelView Orientation
0	0	0° (normal)
0	1	90°
1	0	180°
1	1	270°

Main Window Display Start Address Register
REG[40h] Default = 00000000h Read/Write

n/a															bit 16
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Main Window Display Start Address bits 15-0															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Main Window Display Start Address

The Main Window Display Start Address Register represents a DWORD address which points to the start of the main window image in the display buffer. An address of 0 is the start of the display buffer. The Main Window Display Start Address is programmed to the address of display memory which is at the upper left corner of the rotated image.

For the following SwivelView mode descriptions, calculations refer to the U*pper Left Coordinate Address* as the address of display memory for the coordinates at the upper left corner of the display whether or not the image is rotated. Normally this coordinate is zero and therefore the address is zero. However, this address is dependent on the number of bits-per-pixel and is programmed into the Main Window Display Start Address register in dwords.

In SwivelView 0°, program the Main Window Display Start Address
= Upper Left Coordinate Address ÷ 4

In SwivelView 90°, program the Main Window Display Start Address
= ((Upper Left Coordinate Address + (unrotated panel height × bpp ÷ 8)
+ ((4 - (unrotated panel height × bpp ÷ 8)) & 03h)) ÷ 4) - 1

In SwivelView 180°, calculate the value of the Main Window Stride and then program the Main Window Display Start Address
= ((Upper Left Coordinate Address + (Main Window Stride × (unrotated panel height − 1)) + (unrotated panel width × bpp ÷ 8) + ((4 - (unrotated panel width × bpp ÷ 8)) & 03h)) ÷ 4) - 1

In SwivelView 270°, calculate the value of the Main Window Stride and then program the Main Window Display Start Address
= (Upper Left Coordinate Address + ((unrotated panel width - 1) × Main Window Stride)) ÷ 4

Note

Truncate all fractional values before writing to the address registers.

SwivelView 0° and 180° require the unrotated panel width to be a multiple of (32 ÷ bits-per-pixel). SwivelView 90° and 270° require the unrotated panel height to be a multiple of (32 ÷ bits-per-pixel). If this is not possible, refer to Section 7.6, “Limitations” on page 42.

Main Window Line Address Offset Register
REG[44h] Default = 00000000h Read/Write

n/a															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
n/a						Main Window Line Address Offset bits 9-0									
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Main Window Line Address Offset

The Main Window Line Address Offset Register indicates the number of dwords per line in the main window image.

For SwivelView 0° and 180°, the image width must be at least the rotated panel width. For SwivelView 90° and 270°, the image width must be at least the rotated panel height. In addition, the image width must be a multiple of (32 ÷ bpp). If the image width is not such a multiple, a slightly larger width must be chosen (see Section 7.6, “Limitations”).

Unrotated panel width and *Unrotated panel height* refer to the physical panel dimensions in pixels. *Stride* is the number of bytes required for one line of the image; the offset register represents the stride in DWORD steps.

Main Window Stride = unrotated panel width × bpp ÷ 8

Note

In SwivelView 0° or 180° the image width must be equal to or greater than the unrotated panel width and in SwivelView 90° or 270° the image width must be equal to or greater than the unrotated panel height.

number of dwords per line = unrotated panel width ÷ (32 ÷ bpp)

7.4.1 SwivelView 0° (Landscape)

This section describes how to program the registers to establish SwivelView 0° in a series of steps. Calculations in each step must be truncated to integers. For the following SwivelView mode descriptions, calculations refer to the *Upper Left Coordinate Address* as the address of display memory for coordinates at the upper left corner of the display.

1. Determine the Stride.

   Stride
   = unrotated panel width × bpp ÷ 8

2. Determine the Upper Left Coordinate Address.

   Upper Left Coordinate Address
   =(upper coordinate × Stride) + (left coordinate ÷ bpp)

3. Determine and program the Main Window Display Start Address Register.

   Main Window Display Start Address Register
   = Upper Left Coordinate Address ÷ 4

4. Determine and program the Main Window Line Offset Register.

   Main Window Line Offset Register
   = display width in pixels ÷ (32 ÷ bpp)

7.4.2 SwivelView 90°

This section describes how to program the registers to establish SwivelView 90° in a series of steps. Calculations in each step must be truncated to integers. For the following SwivelView mode descriptions, calculations refer to the *Upper Left Coordinate Address* as the address of display memory for the coordinates at the upper left corner of the rotated display.

1. Determine the Stride.

   Stride
   = unrotated panel width × bpp ÷ 8

2. Determine the Upper Left Coordinate Address.

   Upper Left Coordinate Address
   = (upper-coordinate × Stride) + (left-coordinate ÷ bpp)

3. Determine and program the Main Window Display Start Address Register.

   Main Window Display Start Address Register
   = ((Upper Left Coordinate Address + unrotated panel height × bpp ÷ 8))
   + ((4 - (unrotated panel height × bpp ÷ 8))) & 03h)) ÷ 4) - 1

4. Determine and program the Main Window Line Address Offset Register.

   Main Window Line Address Offset Register
   = unrotated panel height ÷ (32 ÷ bpp)

7.4.3 SwivelView 180°

This section describes how to program the registers to establish SwivelView 180° in a series of steps. Calculations in each step must be truncated to integers. For the following SwivelView mode descriptions, calculations refer to the *Upper Left Coordinate Address* as the address of display memory for the coordinates at the upper left corner of the rotated display.

1. Determine the Stride.

   Stride
   = rotated panel width × bpp ÷ 8

2. Determine the Upper Left Coordinate Address.

   Upper Left Coordinate Address
   =(upper coordinate × Stride) + (left coordinate ÷ bpp)

3. Determine and program the Main Window Display Start Address Register.

   Main Window Display Start Address Register
   = ((Upper Left Coordinate Address + (Stride × (unrotated panel height - 1))
   + (unrotated panel width × bpp ÷ 8)) + ((4 - unrotated panel width × bpp ÷ 8))
   & 03h)) ÷ 4) - 1

4. Determine and program the Main Window Line Address Offset Register.

   Main Window Line Address Offset Register
   = unrotated panel height ÷ (32 ÷ bpp)

7.4.4 SwivelView 270°

This section describes how to program the registers to establish SwivelView 270° in a series of steps. Calculations in each step must be truncated to integers. For the following SwivelView mode descriptions, calculations refer to the *Upper Left Coordinate Address* as the address of display memory for the coordinates at the upper left corner of the rotated display.

1. Determine the Stride.

   Stride
   = unrotated panel height × bpp ÷ 8

2. Determine the Upper Left Coordinate Address.

   Upper Left Coordinate Address
   = (upper coordinate × Stride) + (left coordinate ÷ bpp)

3. Determine and program the Main Window Display Start Address Register.

   Main Window Display Start Address Register
   = ((Upper Left Coordinate Address + ((unrotated panel width - 1) × Stride)) ÷ 4

4. Determine and program the Main Window Line Address Offset Register.

   Main Window Line Address Offset Register
   = unrotated panel height ÷ (32 ÷ bpp)

7.5 Examples

Example 1: ***In SwivelView 0° (normal) mode, program the main window registers for a 320x240 panel at a color depth of 4 bpp.***

1. Determine the Stride.

   Stride
   = unrotated panel width × bpp ÷ 8

2. Determine the Upper Left Coordinate Address.

   Upper Left Coordinate Address
   = (0 × Stride) + (0 ÷ bpp)
   = 0

3. Determine and program the Main Window Display Start Address. The main window is typically placed at the start of display memory which is at display address 0.

   Main Window Display Start Address Register
   = Upper Left Coordinate Address ÷ 4
   = 0

   Program the Main Window Display Start Address register. REG[40h] is set to 00000000h.

4. Determine the Main Window Line Address Offset.

   number of dwords per line
   = unrotated panel width ÷ (32 ÷ bpp)
   = 320 ÷ (32 ÷ 4)
   = 40
   = 28h

   Program the Main Window Line Address Offset register. REG[44h] is set to 00000028h.

Example 2: In SwivelView 90° mode, program the main window registers for a 320x240 panel at a color depth of 4 bpp.

1. Determine the Stride.

   Stride
   = unrotated panel width × bpp ÷ 8

2. Determine the Upper Left Coordinate Address.

   Upper Left Coordinate Address
   = (0 × Stride) + (0 ÷ bpp)
   = 0

3. Determine and program the Main Window Display Start Address. The main window is typically placed at the start of display memory, which is at display address 0.

   Main Window Display Start Address Register
   = ((Upper Left Coordinate Address + (unrotated panel height × bpp ÷ 8)
   + ((4 - (unrotated panel height × bpp ÷ 8)) & 03h)) ÷ 4) - 1
   = ((0 + (240 × 4 ÷ 8) + ((4 - (240 × 4 ÷ 8)) & 03h)) ÷ 4) - 1
   = 29
   = 1Dh

   Program the Main Window Display Start Address register. REG[40h] is set to 0000001Dh.

4. Determine and program the Main Window Line Address Offset.

   number of dwords per line
   = unrotated panel height ÷ (32 ÷ bpp)
   = 240 ÷ (32 ÷ 4)
   = 30
   = 1Eh

   Program the Main Window Line Address Offset register. REG[44h] is set to 0000001Eh.

Example 3: In SwivelView 180° mode, program the main window registers for a 320x240 panel at a color depth of 4 bpp.

1. Determine the Stride.

   Stride
   = unrotated panel width × bpp ÷ 8
   = 320 × 4 ÷ 8
   = 160
   = A0h

2. Determine the Upper Left Coordinate Address.

   Upper Left Coordinate Address
   = (0 × Stride) + (0 ÷ bpp)
   = 0

3. Determine and program the Main Window Display Start Address. The main window is typically placed at the start of display memory which is at display address 0.

   Main Window Display Start Address Register
   = ((Upper Left Coordinate Address + (Stride × (unrotated panel height – 1)) + (unrotated panel width × bpp ÷ 8) + ((4 - (unrotated panel width × bpp ÷ 8)) & 03h)) ÷ 4) - 1
   = ((0+(160 × (240 – 1)) + (320 × 4 ÷ 8) + ((4 - (320 × 4 ÷ 8))& 03h)) ÷ 4) - 1
   = 9599
   = 257Fh

   Program the Main Window Display Start Address register. REG[40h] is set to 0000257Fh.

4. Determine and program the Main Window Line Address Offset.

   number of dwords per line
   = unrotated panel width ÷ (32 ÷ bpp)
   = 320 ÷ (32 ÷ 4)
   = 40
   = 28h

   Program the Main Window Line Address Offset register. REG[44h] is set to 00000028h.

***Example 4:* In SwivelView 270° mode, program the main window registers for a 320x240 panel at a color depth of 4 bpp.**

1. Determine the Stride.

   Stride
   = unrotated panel height × bpp ÷ 8
   = 240 × 4 ÷ 8
   = 120
   = 78h

2. Determine the Upper Left Coordinate Address.

   Upper Left Coordinate Address
   = (0 × Stride) + (0 ÷ bpp)
   = 0

3. Determine and program Main Window Display Start Address. The main window is typically placed at the start of display memory, which is at display address 0.

   Main Window Display Start Address Register
   = (Upper Left Coordinate Address + ((unrotated panel width - 1) × Stride)) ÷ 4
   = (0 + ((320 - 1) × 120)) ÷ 4
   = 9570
   = 2562h

   Program the Main Window Display Start Address register. REG[40h] is set to 00002562h.

4. Determine and program the Main Window Line Address Offset.

   number of dwords per line
   = unrotated panel height ÷ (32 ÷ bpp)
   = 240 ÷ (32 ÷ 4)
   = 30
   = 1Eh

   Program the Main Window Line Address Offset register. REG[44h] is set to 0000001Eh.

7.6 Limitations

7.6.1 SwivelView 0° and 180°

In SwivelView 0° and 180°, the Main Window Line Address Offset Register (REG[44h]) requires the *panel width* to be a multiple of (32 ÷ bits-per-pixel). If this is not the case, then the Main Window Line Address Offset Register must be programmed to a longer line which meets this requirement. This longer line creates a virtual image where the width is *Main Window Line Address Offset Register × (32 ÷ bits-per-pixel)*.

In SwivelView 0°, this virtual image should be drawn in display memory as left justified, and in SwivelView 180°, this virtual image should be drawn in display memory as right justified.

A left-justified image is one drawn in display memory such that each of the image’s lines only use the left most portion of the line width defined by the line address offset register (i.e. starting at horizontal position 0).

A right-justified image is one drawn in display memory such that each of the image’s lines only use the right most portion of the line width defined by the line address offset register (i.e. starting at a non-zero horizontal position which is the virtual width - image width).

7.6.2 SwivelView 90° and 270°

In SwivelView 90° and 270°, the Main Window Line Address Offset Register (REG[44h]) requires the *panel height* to be a multiple of (32 ÷ bits-per-pixel). If this is not the case, then the Main Window Line Address Offset Register must be programmed to a longer line which meets this requirement. This longer line creates a virtual image whose width is *Main Window Line Address Offset Register × (32 ÷ bits-per-pixel)*.

In SwivelView 270°, this virtual image should be drawn in display memory as left justified, and in SwivelView 90°, this virtual image should be drawn in display memory as right justified.

A left-justified image is one drawn in display memory such that each of the image’s lines only use the left most portion of the line width defined by the line address offset register (i.e. starting at horizontal position 0).

A right-justified image is one drawn in display memory such that each of the image’s lines only use the right most portion of the line width defined by the line address offset register (i.e. starting at a non-zero horizontal position which is the virtual width - image width).

8 Picture-In-Picture Plus (PIP$^+$)

Picture-in-Picture Plus (PIP$^+$) enables a secondary window (or PIP$^+$ window) within the main display window. The PIP$^+$ window may be positioned anywhere within the virtual display and is controlled through the PIP$^+$ Window control registers (REG[50h] through REG[5Ch]). The PIP$^+$ window retains the same color depth and SwivelView orientation as the main window.

A PIP$^+$ window can be used to display temporary items such as a dialog box or to “float” the display item so that the system doesn’t have to exclude the area during screen repaints.

The following diagram shows an example of a PIP$^+$ window within a main window.

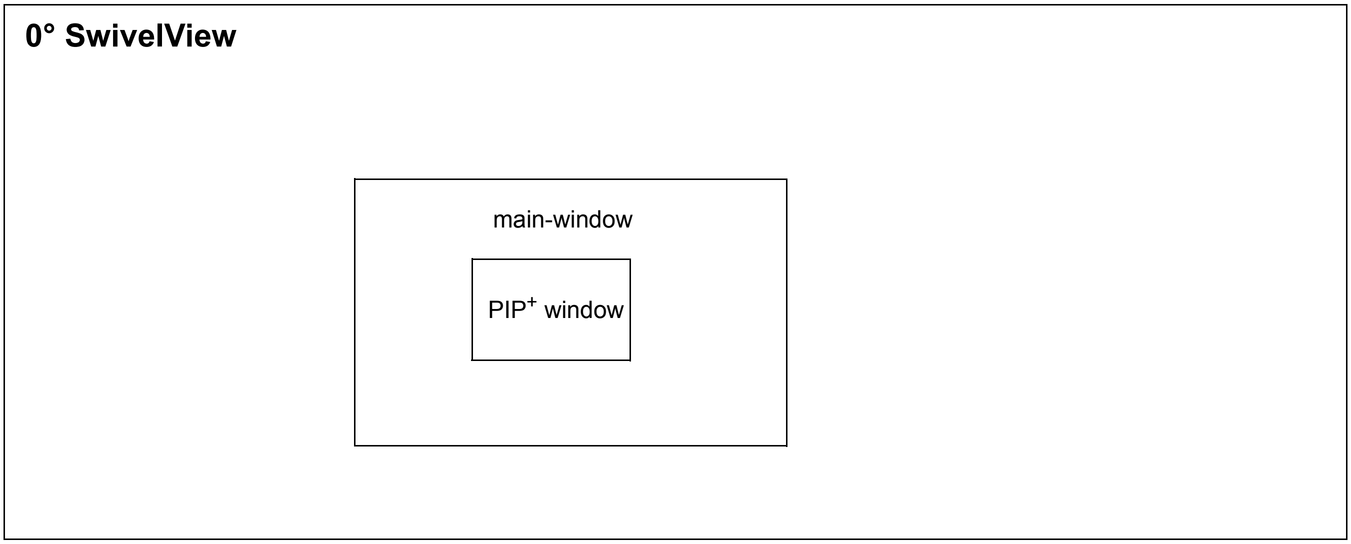


Figure 8-1: Picture-in-Picture Plus with SwivelView Disabled

8.1 Registers

The following registers control the Picture-In-Picture Plus feature.

Display Settings Register
REG[10h] Default = 00000000h Read/Write

n/a						Pixel Doubling Vertical	Pixel Doubling Horiz.	Display Blank	Dithering Disable	n/a	SW Video Invert	PIP$^+$ Window Enable	n/a	SwivelView Mode Select	
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
n/a											Bits-per-pixel Select (actual value: 1, 2, 4, 8 or 16 bpp)				
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

PIP$^+$ Window Enable

The PIP$^+$ Window Enable bit enables a PIP$^+$ window within the main window. The location of the PIP$^+$ window within the landscape window is determined by the PIP$^+$ X Position register (REG[58h]) and PIP$^+$ Y Position register (REG[5Ch]). The PIP$^+$ window has its own Display Start Address register (REG[50h]) and Line Address Offset register (REG[54h]). The PIP$^+$ window shares the same color depth and SwivelView^{TM} orientation as the main window.

PIP$^+$ Display Start Address Register
REG[50h] Default = 00000000h Read/Write

n/a															bit 16
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
PIP$^+$ Display Start Address bits 15-0															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

PIP$^+$ Display Start Address

The PIP$^+$ Display Start Address register is a DWORD which represents an address that points to the start of the PIP$^+$ window image in the display buffer. An address of 0 is the start of the display buffer. For the following PIP$^+$ descriptions, the *desired byte address* is the starting display address for the PIP$^+$ window image.

In SwivelView 0°, program the start address
= desired byte address ÷ 4

In SwivelView 90°, program the start address
= ((desired byte address + (PIP$^+$ width × bpp ÷ 8)
+ ((4 - (PIP$^+$ width × bpp ÷ 8)) & 03h)) ÷ 4) - 1

In SwivelView 180°, program the start address
= ((desired byte address + (PIP$^+$ Stride × (PIP$^+$ height − 1))
+ (PIP$^+$ width × bpp ÷ 8) + ((4 - (PIP$^+$ width × bpp ÷ 8)) & 03h)) ÷ 4) - 1

In SwivelView 270°, program the start address
= (desired byte address + ((PIP$^+$ height - 1) × PIP$^+$ Stride)) ÷ 4

Note

Truncate all fractional values before writing to the address registers.

SwivelView 0° and 180° require the PIP$^+$ width to be a multiple of (32 ÷ bits-per-pixel). SwivelView 90° and 270° require the PIP$^+$ height to be a multiple of (32 ÷ bits-per-pixel). If this is not possible, refer to Section 8.3, “Limitations” .

PIP$^+$ Line Address Offset Register
REG[54h] Default = 00000000h Read/Write

n/a															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
n/a						PIP$^+$ Line Address Offset bits 9-0									
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

PIP$^+$ Line Address Offset

The PIP$^+$ Line Address Offset register indicates the number of dwords per line in the PIP$^+$ window image.

The image width must be a multiple of (32 ÷ bpp). If the image width is not such a multiple, a slightly larger width must be chosen (see Section 8.3, “Limitations”).

PIP$^+$ width and *PIP$^+$ height* refer to the PIP$^+$ dimensions as seen in SwivelView 0° (landscape mode). *Stride* is the number of bytes required for one line of the image; the offset register represents the stride in DWORD steps.

PIP$^+$ Stride = image width × bpp ÷ 8

For SwivelView 0° and 180°, program the line address offset
= PIP$^+$ Width
= ((REG[58h] bits 25:16) - (REG[58h] bits 9:0) + 1) × (32 ÷ bpp)

For SwivelView 90° and 270°, program the line address offset
= PIP$^+$ Width
= ((REG[5Ch] bits 25:16) - (REG[5Ch] bits 9:0) + 1) × (32 ÷ bpp)

PIP$^+$ X Positions Register
REG[58h] Default = 00000000h Read/Write

n/a						PIP$^+$ X End Position bits 9-0									
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
n/a						PIP$^+$ X Start Position bits 9-0									
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

PIP$^+$ X End Position

The PIP$^+$ X End Position bits determine the horizontal end of the PIP$^+$ window in 0° and 180° SwivelView orientations. These bits determine the vertical end position in 90° and 270° SwivelView. For further information on defining the value of the X End Position, see Section 8.2, “Picture-In-Picture-Plus Examples” on page 51.

This register increments differently based on the SwivelView orientation. For 0° and 180° SwivelView the X End Position is incremented by *X* pixels where *X* is relative to the current color depth. For 90° and 270° SwivelView the X End Position is incremented in 1 line increments.

Table 8-1: 32-bit Address Increments for PIP$^+$ X Position in SwivelView 0° and 180°

Bits-Per-Pixel (Color Depth)	Pixel Increment (X)
1 bpp	32
2 bpp	16
4 bpp	8
8 bpp	4
16 bpp	2

In SwivelView 0°, these bits set the horizontal coordinates (x) of the PIP$^+$ window’s right edge. Increasing x moves the right edge towards the right in steps of (32 ÷ bits-per-pixel) (see Table 8-1:). The horizontal coordinates start at pixel 0.

Program the PIP$^+$ Window X End Position so that
PIP$^+$ Window X End Position = x ÷ (32 ÷ bits-per-pixel)

Note
Truncate the fractional part of the above equation.

In SwivelView 90°, these bits set the vertical coordinates (y) of the PIP$^+$ window’s bottom edge. Increasing y moves the bottom edge downward in 1 line steps. The vertical coordinates start at line 0.

Program the PIP$^+$ Window X End Position so that
PIP$^+$ Window X End Position = y

In SwivelView 180°, these bits set the horizontal coordinates (x) of the PIP$^+$ window's left edge. Increasing x moves the left edge towards the right in steps of (32 ÷ bits-per-pixel) (see Table 8-1:). The horizontal coordinates start at pixel 0.

Program the PIP$^+$ Window X End Position so that
PIP$^+$ Window X End Position = (panel width - x - 1) ÷ (32 ÷ bits-per-pixel)

Note
Truncate the fractional part of the above equation.

In SwivelView 270°, these bits set the vertical coordinates (y) of the PIP$^+$ window's top edge. Increasing y moves the top edge downwards in 1 line steps. The vertical coordinates start at line 0.

Program the PIP$^+$ Window X End Position so that
PIP$^+$ Window X End Position = panel width - y - 1

PIP$^+$ X Start Position

The PIP$^+$ X Start Position bits determine the horizontal position of the start of the PIP$^+$ window in 0° and 180° SwivelView orientations. These bits determine the vertical start position in 90° and 270° SwivelView. For further information on defining the value of the X Start Position, see Section 8.2, "Picture-In-Picture-Plus Examples" on page 51.

The register increments differently based on the SwivelView orientation. For 0° and 180° SwivelView the X Start Position is incremented by *X* pixels where *X* is relative to the current color depth. For 90° and 270° SwivelView the X Start Position is incremented in 1 line increments.

Table 8-2: 32-bit Address Increments for Color Depth

Bits-per-pixel (Color Depth)	Pixel Increment (X)
1 bpp	32
2 bpp	16
4 bpp	8
8 bpp	4
16 bpp	2

In SwivelView 0°, these bits set the horizontal coordinates (x) of the PIP$^+$ windows's left edge. Increasing x moves the left edge towards the right in steps of (32 ÷ bits-per-pixel) (see Table 8-2:). The horizontal coordinates start at pixel 0.

Program the PIP$^+$ Window X Start Position so that
PIP$^+$ Window X Start Position = x ÷ (32 ÷ bits-per-pixel)

Note
Truncate the fractional part of the above equation.

In SwivelView 90°, these bits set the vertical coordinates (y) of the PIP$^+$ window's top edge. Increasing y moves the top edge downward in 1 line steps. The vertical coordinates start at line 0.

Program the PIP$^+$ Window X Start Position so that
PIP$^+$ Window X Start Position = y

In SwivelView 180°, these bits set the horizontal coordinates (x) of the PIP$^+$ window's right edge. Increasing x moves the right edge towards the right in steps of (32 ÷ bits-per-pixel) (see Table 8-2:). The horizontal coordinates start at pixel 0.

Program the PIP$^+$ Window X Start Position so that
PIP$^+$ Window X Start Position = (panel width - x - 1) ÷ (32 ÷ bits-per-pixel)

Note

Truncate the fractional part of the above equation.

In SwivelView 270°, these bits set the vertical coordinates (y) of the PIP$^+$ window's bottom edge. Increasing y moves the bottom edge downwards in 1 line steps. The vertical coordinates start at line 0.

Program the PIP$^+$ Window X Start Position so that
PIP$^+$ Window X Start Position = panel width - y - 1

PIP$^+$ Y Positions Register
REG[5Ch] Default = 00000000h Read/Write

n/a						PIP$^+$ Y End Position bits 9-0									
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
n/a						PIP$^+$ Y Start Position bits 9-0									
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

PIP$^+$ Y End Position

The PIP$^+$ Y End Position bits determine the vertical end position of the PIP$^+$ window in 0° and 180° SwivelView orientations. These bits determine the horizontal end position in 90° and 270° SwivelView. For further information on defining the value of the Y End Position, see Section 8.2, "Picture-In-Picture-Plus Examples" on page 51.

The register increments differently based on the SwivelView orientation. For 0° and 180° SwivelView the Y End Position is incremented in 1 line increments. For 90° and 270° SwivelView the Y End Position is incremented by *Y* pixels where *Y* is relative to the current color depth.

Table 8-3: 32-bit Address Increments for Color Depth

Bits-Per-Pixel (Color Depth)	Pixel Increment (Y)
1 bpp	32

Table 8-3: 32-bit Address Increments for Color Depth

Bits-Per-Pixel (Color Depth)	Pixel Increment (Y)
2 bpp	16
4 bpp	8
8 bpp	4
16 bpp	2

In SwivelView 0°, these bits set the vertical coordinates (y) of the PIP$^+$ windows's bottom edge. Increasing y moves the bottom edge downwards in 1 line steps. The vertical coordinates start at line 0.

Program the PIP$^+$ Window Y End Position so that
PIP$^+$ Window Y End Position = y

In SwivelView 90°, these bits set the horizontal coordinates (x) of the PIP$^+$ window's left edge. Increasing x moves the left edge towards the right in steps of (32 ÷ bits-per-pixel) (see Table 8-3:). The horizontal coordinates start at pixel 0.

Program the PIP$^+$ Window Y End Position so that
PIP$^+$ Window Y End Position = (panel height - x - 1) ÷ (32 ÷ bits-per-pixel)

Note
Truncate the fractional part of the above equation.

In SwivelView 180°, these bits set the vertical coordinates (y) of the PIP$^+$ window's top edge. Increasing y moves the top edge downwards in 1 line steps. The vertical coordinates start at line 0.

Program the PIP$^+$ Window Y End Position so that
PIP$^+$ Window Y End Position = panel height - y - 1

In SwivelView 270°, these bits set the horizontal coordinates (x) of the PIP$^+$ window's right edge. Increasing x moves the right edge towards the right in steps of (32 ÷ bits-per-pixel) (see Table 8-3:). The horizontal coordinates start at pixel 0.

Program the PIP$^+$ Window Y End Position so that
PIP$^+$ Window Y End Position = x ÷ (32 ÷ bits-per-pixel)

Note
Truncate the fractional part of the above equation.

PIP$^+$ Y Start Position

The PIP$^+$ Y Start Position bits determine the vertical start position of the PIP$^+$ window in 0° and 180° SwivelView orientations. These bits determine the horizontal start position in 90° and 270° SwivelView. For further information on defining the value of the Y Start Position, see Section 8.2, “Picture-In-Picture-Plus Examples” on page 51.

The register increments differently based on the SwivelView orientation. For 0° and 180° SwivelView the Y Start Position is incremented in 1 line increments. For 90° and 270° SwivelView the Y Start Position is incremented by *Y* pixels where *Y* is relative to the current color depth.

Table 8-4: 32-bit Address Increments for Color Depth

Bits-Per-Pixel (Color Depth)	Pixel Increment (Y)
1 bpp	32
2 bpp	16
4 bpp	8
8 bpp	4
16 bpp	2

In SwivelView 0°, these bits set the vertical coordinates (y) of the PIP$^+$ windows's top edge. Increasing y moves the top edge downwards in 1 line steps. The vertical coordinates start at line 0.

Program the PIP$^+$ Window Y Start Position so that
PIP$^+$ Window Y Start Position = y

In SwivelView 90°, these bits set the horizontal coordinates (x) of the PIP$^+$ window's right edge. Increasing x moves the right edge towards the right in steps of (32 ÷ bits-per-pixel) (see Table 8-4:). The horizontal coordinates start at pixel 0.

Program the PIP$^+$ Window Y Start Position so that
PIP$^+$ Window Y Start Position = (panel height - x - 1) ÷ (32 ÷ bits-per-pixel)

Note
Truncate the fractional part of the above equation.

In SwivelView 180°, these bits set the vertical coordinates (y) of the PIP$^+$ window's bottom edge. Increasing y moves the bottom edge downwards in 1 line steps. The vertical coordinates start at line 0.

Program the PIP$^+$ Window Y Start Position so that
PIP$^+$ Window Y Start Position = panel height - y - 1

In SwivelView 270°, these bits set the horizontal coordinates (x) of the PIP$^+$ window's left edge. Increasing x moves the left edge towards the right in steps of (32 ÷ bits-per-pixel) (see Table 8-4:). The horizontal coordinates start at pixel 0.

Program the PIP$^+$ Window Y Start Position so that
PIP$^+$ Window Y Start Position = x ÷ (32 ÷ bits-per-pixel)

Note
Truncate the fractional part of the above equation.

8.2 Picture-In-Picture-Plus Examples

8.2.1 SwivelView 0° (Landscape Mode)

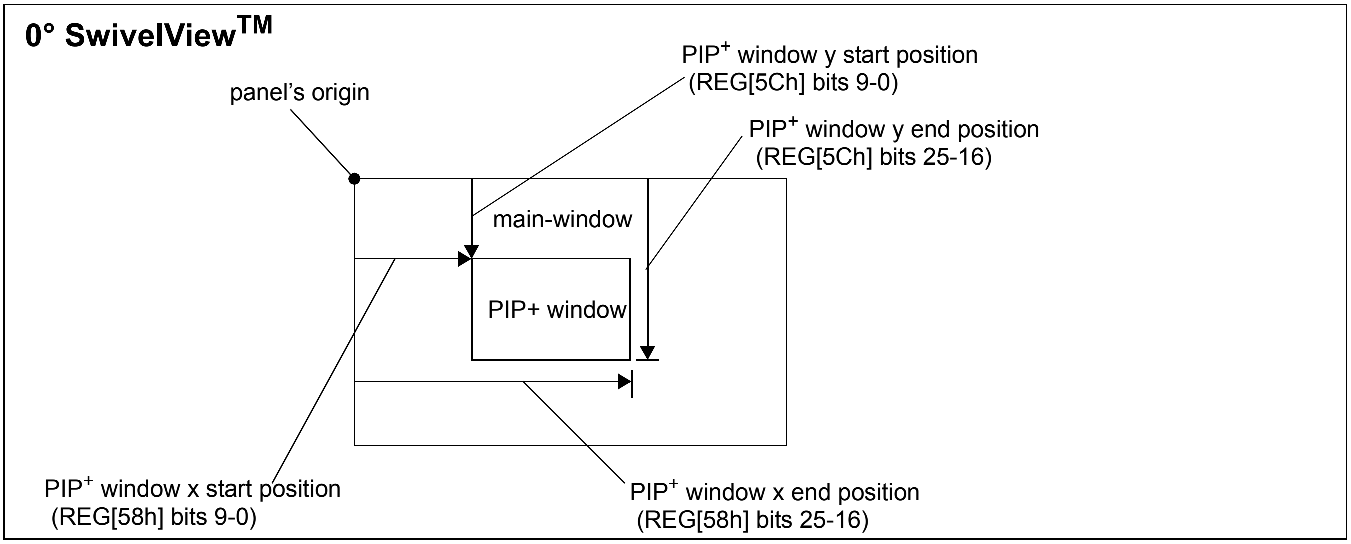

0° SwivelView^{TM}
panel's origin
PIP$^+$ window y start position (REG[5Ch] bits 9-0)
PIP$^+$ window y end position (REG[5Ch] bits 25-16)
main-window
PIP+ window
PIP$^+$ window x start position (REG[58h] bits 9-0)
PIP$^+$ window x end position (REG[58h] bits 25-16)

Figure 8-2: Picture-in-Picture Plus with SwivelView disabled

SwivelView 0° (or landscape) is a mode in which both the main and PIP$^+$ window are non-rotated. The images for each window are typically placed consecutively, with the main window image starting at address 0 and followed by the PIP$^+$ window image. In addition, both images must start at addresses which are dword-aligned (the last two bits of the starting address must be 0).

Note

It is possible to use the same image for both the main window and PIP$^+$ window. To do so, set the PIP$^+$ Line Address Offset register (REG[54h]) to the same value as the Main Window Line Address Offset register (REG[44h].

Example 5: Program the PIP$^+$ window registers for a 320x240 panel at 4 bpp, with the PIP$^+$ window positioned at (80, 60) with a width of 160 and a height of 120.

1. Determine the value for the PIP$^+$ Window X Positions and PIP$^+$ Window Y Positions registers. Let the top left corner of the PIP$^+$ window be (x1, y1), and let the bottom right corner be (x2, y2), where x2 = x1 + width - 1 and y2 = y1 + height - 1. The PIP$^+$ Window X Positions register sets the horizontal coordinates of the PIP$^+$ window's top left and bottom right corners. The PIP$^+$ Window Y Positions register sets the vertical coordinates of the PIP$^+$ window's top left and bottom right corners.

   The required values are calculated as follows:

   X Start Position
   = x1 ÷ (32 ÷ bpp)
   = 80 ÷ (32 ÷ 4)
   = 10
   = 0Ah

   Y Start Position
   = y1
   = 60
   = 3Ch

   X End Position
   = x2 ÷ (32 ÷ bpp)
   = (80 + 160 - 1) ÷ (32 ÷ 4)
   = 29.875
   = 1Dh (truncated fractional part)

   Y End Position
   = y2
   = 60 + 120 - 1
   = 179
   = B3h

2. Program the PIP$^+$ Window X Positions register with the X Start Position in bits 9-0 and the X End Position in bits 25-16. REG[58h] is set to 001D000Ah.
   Program the PIP$^+$ Window Y Positions register with the Y Start Position in bits 9-0 and the Y End Position in bits 25-16. REG[5Ch] is set to 00B3003Ch.

   Due to truncation, the dimensions of the PIP$^+$ window may have changed. Recalculate the PIP$^+$ window width and height below:
   PIP$^+$ Width
   = ((REG[58h] bits 25:16) - (REG[58h] bits 9:0) + 1) × (32 ÷ bpp)
   = (1Dh - 0Ah + 1) × (32 ÷ 4)
   = 160 pixels

   PIP Height
   = (REG[5Ch] bits 25:16) - (REG[5Ch] bits 9:0) + 1
   = B3h - 3Ch + 1
   = 120 lines

3. Determine the PIP$^+$ display start address.
   The main window image must take up 320 x 240 pixels × bpp ÷ 8 = 9600h bytes. If the main window starts at address 0h, the PIP$^+$ window can start at 9600h.

   PIP$^+$ display start address
   = desired byte address ÷ 4
   = 9600h ÷ 4
   = 2580h.

   Program the PIP$^+$ Display Start Address register. REG[50h] is set to 00002580h.

4. Determine the PIP$^+$ line address offset.

   number of dwords per line
   = image width ÷ (32 ÷ bpp)
   = 160 ÷ (32 ÷ 4)
   = 20
   = 14h

   Program the PIP$^+$ Line Address Offset register. REG[54h] is set to 00000014h.

5. Enable the PIP$^+$ window.

   Program the PIP$^+$ Window Enable bit. REG[10h] bit 19 is set to 1.

8.2.2 SwivelView 90°

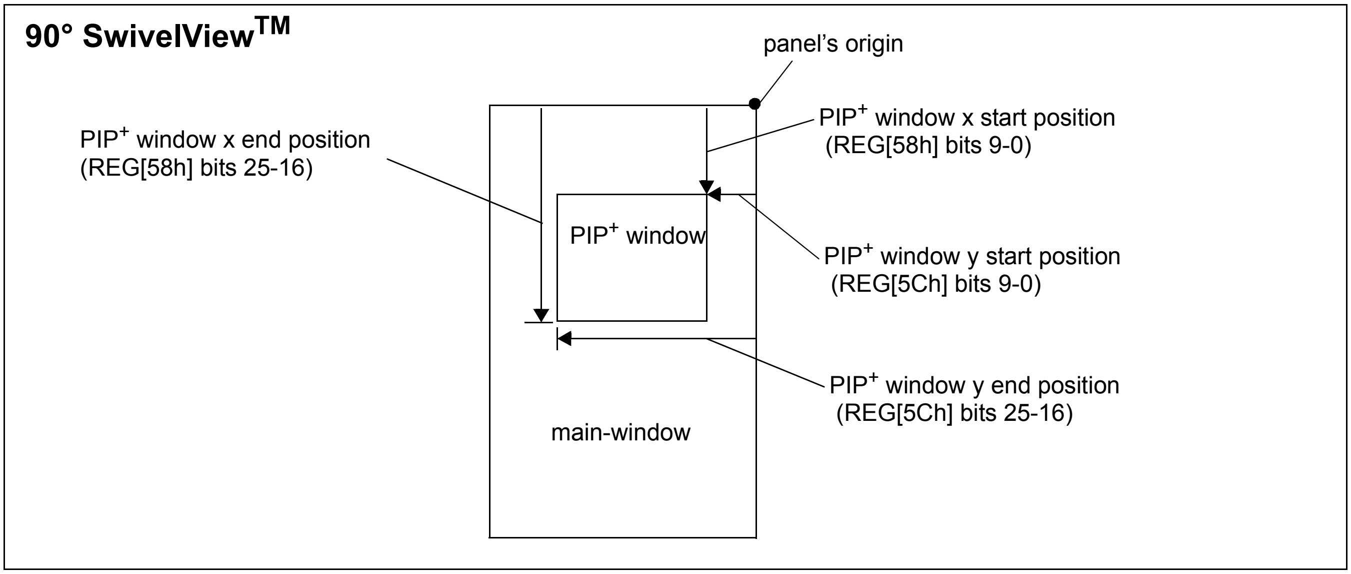

90° SwivelView™
panel's origin
PIP+ window x end position (REG[58h] bits 25-16)
PIP+ window x start position (REG[58h] bits 9-0)
PIP+ window
PIP+ window y start position (REG[5Ch] bits 9-0)
PIP+ window y end position (REG[5Ch] bits 25-16)
main-window

Figure 8-3: Picture-in-Picture Plus with SwivelView 90° enabled

SwivelView 90° is a mode in which both the main and PIP$^+$ windows are rotated 90° counter-clockwise when shown on the panel. The images for each window are typically placed consecutively, with the main window image starting at address 0 and followed by the PIP$^+$ window image. In addition, both images must start at addresses which are dword-aligned (the last two bits of the starting address must be 0).

Note

It is possible to use the same image for both the main window and PIP$^+$ window. To do so, set the PIP$^+$ Line Address Offset register (REG[54h]) to the same value as the Main Window Line Address Offset register (REG[44h]).

Example 6: In SwivelView 90°, program the PIP$^+$ window registers for a 320x240 panel at 4 bpp, with the PIP$^+$ window positioned at SwivelView 90° coordinates (60, 80) with a width of 120 and a height of 160.

1. Determine the value for the PIP$^+$ Window X Positions and PIP$^+$ Window Y Positions registers. Let the top left corner of the PIP$^+$ window be (x1, y1), and let the bottom right corner be (x2, y2), where x2 = x1 + width - 1 and y2 = y1 + height - 1. The PIP$^+$ Window X Positions register sets the vertical coordinates of the PIP$^+$ window's top right and bottom left corners. The PIP$^+$ Window Y Positions register sets the horizontal coordinates of the PIP$^+$ window's top right and bottom left corners.

   The required values are calculated as follows:

   X Start Position
   = y1
   = 80
   = 50h

   Y Start Position
   = (panel height - x2 - 1) ÷ (32 ÷ bpp)
   = (240 - (60 + 120 - 1) - 1) ÷ (32 ÷ 4)
   = 7.5
   = 07h (truncated fractional part)

   X End Position
   = y2
   = 80 + 160 - 1
   = 239
   = EFh

   Y End Position
   = (panel height - x1 - 1) ÷ (32 ÷ bpp)
   = (240 - 60 - 1) ÷ (32 ÷ 4)
   = 22.375
   = 16h (truncated fractional part)

2. Program the PIP$^+$ Window X Positions register with the X Start Position in bits 9-0 and the X End Position in bits 25-16. REG[58h] is set to 00EF0050h.
   Program the PIP$^+$ Window Y Positions register with the Y Start Position in bits 9-0 and the Y End Position in bits 25-16. REG[5Ch] is set to 00160007h.

   Due to truncation, the dimensions of the PIP$^+$ window may have changed. Recalculate the PIP$^+$ window width and height below:

   PIP$^+$ Width
   = ((REG[5Ch] bits 25:16) - (REG[5Ch] bits 9:0) + 1) × (32 ÷ bpp)
   = (16h - 07h + 1) × (32 ÷ 4)
   = 128 pixels **(note that this is different from the desired width)**

   PIP Height
   = (REG[58h] bits 25:16) - (REG[58h] bits 9:0) + 1
   = EFh - 50h + 1
   = 160 lines

3. Determine the PIP$^+$ display start address.
   The main window image must take up 320 x 240 pixels × bpp ÷ 8 = 9600h bytes. If the main window starts at address 0h, then the PIP$^+$ window can start at 9600h.

   PIP$^+$ display start address
   = ((desired byte address + (PIP$^+$ width × bpp ÷ 8)
   + ((4 - (PIP$^+$ width × bpp ÷ 8)) & 03h)) ÷ 4) - 1
   = ((9600h + (128 × 4 ÷ 8) + ((4 - (128 × 4 ÷ 8)) & 03h)) ÷ 4) - 1
   = 9615
   = 258Fh

   Program the PIP$^+$ Display Start Address register. REG[50h] is set to 0000258Fh.

4. Determine the PIP$^+$ line address offset.

   number of dwords per line
   = image width ÷ (32 ÷ bpp)
   = 128 ÷ (32 ÷ 4)
   = 16
   = 10h

   Program the PIP$^+$ Line Address Offset register. REG[54h] is set to 00000010h.

5. Enable the PIP$^+$ window.

   Program the PIP$^+$ Window Enable bit. REG[10h] bit 19 is set to 1.

8.2.3 SwivelView 180°

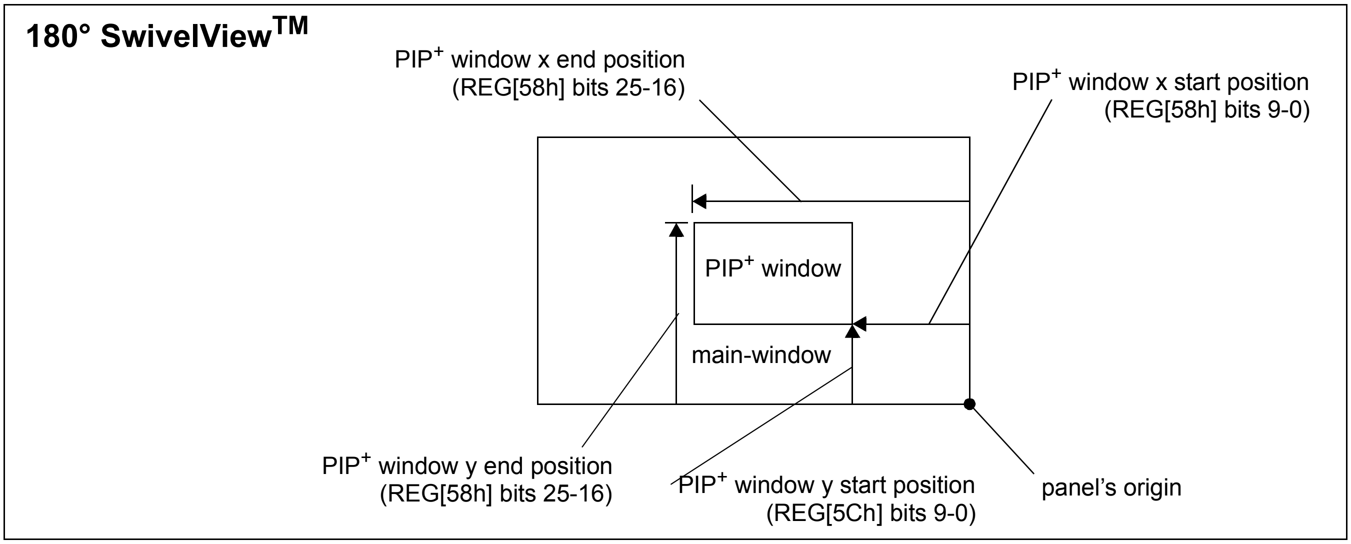

180° SwivelView™
PIP+ window x end position (REG[58h] bits 25-16)
PIP+ window x start position (REG[58h] bits 9-0)
PIP+ window
main-window
PIP+ window y end position (REG[58h] bits 25-16)
PIP+ window y start position (REG[5Ch] bits 9-0)
panel's origin

Figure 8-4: Picture-in-Picture Plus with SwivelView 180° enabled

SwivelView 180° is a mode in which both the main and PIP$^+$ windows are rotated 180° counter-clockwise when shown on the panel. The images for each window are typically placed consecutively, with the main window image starting at address 0 and followed by the PIP$^+$ window image. In addition, both images must start at addresses which are dword-aligned (the last two bits of the starting address must be 0).

Note

It is possible to use the same image for both the main window and PIP$^+$ window. To do so, set the PIP$^+$ Line Address Offset register (REG[54h]) to the same value as the Main Window Line Address Offset register (REG[44h]).

***Example 7:** In SwivelView 180°, program the PIP$^+$ window registers for a 320x240 panel at 4 bpp, with the PIP$^+$ window positioned at SwivelView 180° co-ordinates (80, 60) with a width of 160 and a height of 120.*

1. Determine the value for the PIP$^+$ Window X Positions and PIP$^+$ Window Y Positions registers. Let the top left corner of the PIP$^+$ window be (x1, y1), and let the bottom right corner be (x2, y2), where x2 = x1 + width - 1 and y2 = y1 + height - 1. The PIP$^+$ Window X Positions register sets the horizontal coordinates of the PIP$^+$ window's bottom right and top left corner. The PIP$^+$ Window Y Positions register sets the vertical coordinates of the PIP$^+$ window's bottom right and top left corner.

   The required values are calculated as follows:

   X Start Position
   = (panel width - x2 - 1) ÷ (32 ÷ bpp)
   = (320 - (80 + 160 - 1) - 1) ÷ (32 ÷ 4)
   = 10
   = 0Ah
   Y Start Position
   = panel height - y2 - 1
   = 240 - (60 + 120 - 1) - 1
   = 60
   = 3Ch
   X End Position
   = (panel width - x1 - 1) ÷ (32 ÷ bpp)
   = (320 - 80 - 1) ÷ (32 ÷ 4)
   = 29.875
   = 1Dh (truncated fractional part)
   Y End Position
   = panel height - y1 - 1
   = 240 - 60 - 1
   = 179
   = B3h

   Program the PIP$^+$ Window X Positions register with the X Start Position in bits 9-0 and the X End Position in bits 25-16. REG[58h] is set to 001D000Ah.
   Program the PIP$^+$ Window Y Positions register with the Y Start Position in bits 9-0 and the Y End Position in bits 25-16. REG[5Ch] is set to 00B3003Ch.

   Due to truncation, the dimensions of the PIP$^+$ window may have changed. Recalculate the PIP$^+$ window width and height below:

   PIP$^+$ Width
   = ((REG[58h] bits 25:16) - (REG[58h] bits 9:0) + 1) × (32 ÷ bpp)
   = (1Dh - 0Ah + 1) × (32 ÷ 4)
   = 160 pixels

   PIP Height

= (REG[5Ch] bits 25:16) - (REG[5Ch] bits 9:0) + 1
= B3h - 3Ch + 1
= 120 lines

2. Determine the PIP$^+$ display start address.
   The main window image must take up 320 x 240 pixels × bpp ÷ 8 = 9600h bytes. If the main window starts at address 0h, then the PIP$^+$ window can start at 9600h.

   PIP$^+$ Stride
   = image width × bpp ÷ 8
   = 160 × 4 ÷ 8
   = 80
   = 50h

   PIP$^+$ display start address
   = ((desired byte address + (PIP$^+$ Stride × (PIP$^+$ height - 1))
   + (PIP$^+$ width × bpp ÷ 8) + ((4 - (PIP width × bpp ÷ 8)) & 03h)) ÷ 4) - 1
   = ((9600h + (80 × (120 - 1)) + (160 × 4 ÷ 8) + ((4 - (160 × 4 ÷ 8))&03h)) ÷ 4) - 1
   = 11999
   = 2EDFh

   Program the PIP$^+$ Display Start Address register. REG[50h] is set to 00002EDFh.

3. Determine the PIP$^+$ line address offset.

   number of dwords per line
   = image width ÷ (32 ÷ bpp)
   = 160 ÷ (32 ÷ 4)
   = 20
   = 14h

   Program the PIP$^+$ Line Address Offset register. REG[54h] is set to 00000014h.

4. Enable the PIP$^+$ window.

   Program the PIP$^+$ Window Enable bit. REG[10h] bit 19 is set to 1.

8.2.4 SwivelView 270°

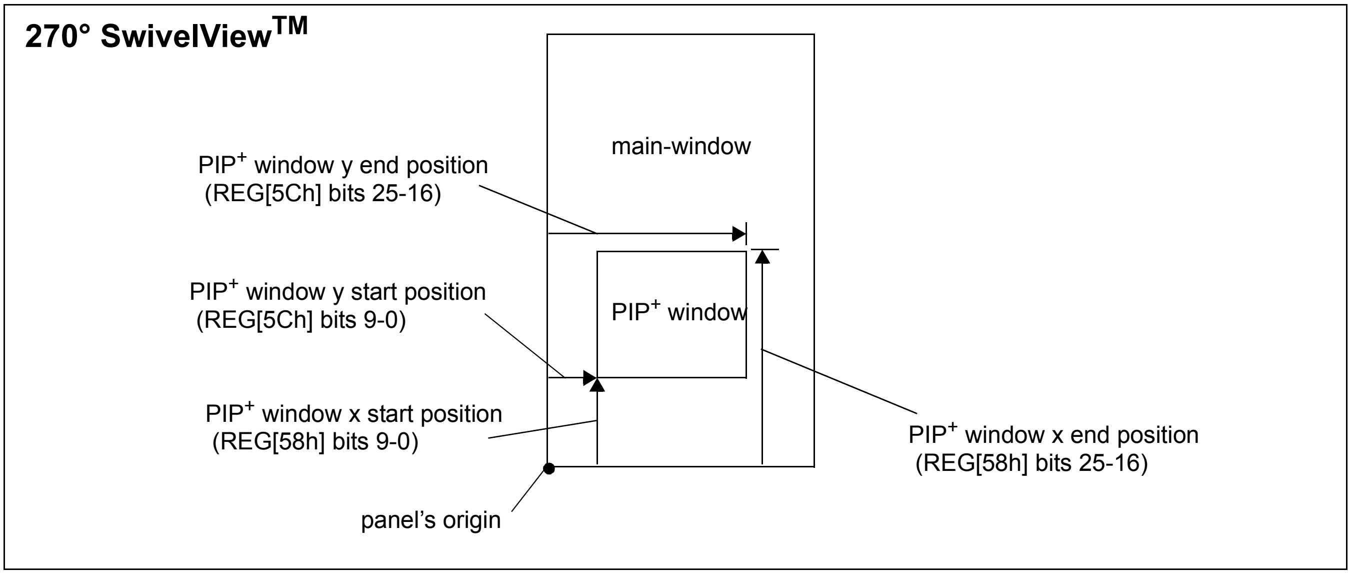

270° SwivelView™
main-window
PIP+ window
PIP+ window y end position (REG[5Ch] bits 25-16)
PIP+ window y start position (REG[5Ch] bits 9-0)
PIP+ window x start position (REG[58h] bits 9-0)
PIP+ window x end position (REG[58h] bits 25-16)
panel's origin

Figure 8-5: Picture-in-Picture Plus with SwivelView 270° enabled

SwivelView 270° is a mode in which both the main and PIP$^+$ windows are rotated 270° counter-clockwise when shown on the panel. The images for each window are typically placed consecutively, with the main window image starting at address 0 and followed by the PIP$^+$ window image. In addition, both images must start at addresses which are dword-aligned (the last two bits of the starting address must be 0).

Note

It is possible to use the same image for both the main window and PIP$^+$ window. To do so, set the PIP$^+$ Line Address Offset register (REG[54h]) to the same value as the Main Window Line Address Offset register (REG[44h]).

***Example 8:** In SwivelView 270°, program the PIP$^+$ window registers for a 320x240 panel at 4 bpp, with the PIP$^+$ window positioned at SwivelView 270° co-ordinates (60, 80) with a width of 120 and a height of 160.*

1. Determine the value for the PIP$^+$ Window X Positions and PIP$^+$ Window Y Positions registers. Let the top left corner of the PIP$^+$ window be (x1, y1), and let the bottom right corner be (x2, y2), where x2 = x1 + width - 1 and y2 = y1 + height - 1. The PIP$^+$ Window X Positions register sets the vertical coordinates of the PIP$^+$ window's top right and bottom left corner. The PIP$^+$ Window Y Positions register sets the horizontal coordinates of the PIP$^+$ window's top right and bottom left corner.

   The required values are calculated as follows:

   X Start Position
   = panel width - y2 - 1
   = 320 - (80 + 160 - 1) - 1
   = 80
   = 50h

   Y Start Position
   = x1 ÷ (32 ÷ bpp)
   = 60 ÷ (32 ÷ 4)
   = 7.5
   = 07h (truncated fractional part)

   X End Position
   = panel width - y1 - 1
   = 320 - 80 - 1
   = 239
   = EFh

   Y End Position
   = x2 ÷ (32 ÷ bpp)
   = (60 + 120 - 1) ÷ (32 ÷ 4)
   = 22.375
   = 16h (truncated fractional part)

2. Program the PIP$^+$ Window X Positions register with the X Start Position in bits 9-0 and the X End Position in bits 25-16. REG[58h] is set to 00EF0050h.
   Program the PIP$^+$ Window Y Positions register with the Y Start Position in bits 9-0 and the Y End Position in bits 25-16. REG[5Ch] is set to 00160007h.

   Due to truncation, the dimensions of the PIP$^+$ window may have changed. Recalculate the PIP$^+$ window width and height below:

   PIP$^+$ Width
   = ((REG[5Ch] bits 25:16) - (REG[5Ch] bits 9:0) + 1) × (32 ÷ bpp)
   = (16h - 07h + 1) × (32 ÷ 4)
   = 128 pixels **(note that this is different from the desired width)**

   PIP Height
   = (REG[58h] bits 25:16) - (REG[58h] bits 9:0) + 1
   = EFh - 50h + 1
   = 160 lines

3. Determine the PIP$^+$ display start address.
   The main window image must take up 320 x 240 pixels × bpp ÷ 8 = 9600h bytes. If the main window starts at address 0h, then the PIP$^+$ window can start at 9600h.

   PIP$^+$ Stride
   = image width × bpp ÷ 8
   = 128 × 4 ÷ 8
   = 64
   = 40h

   PIP$^+$ display start address
   = (desired byte address + ((PIP$^+$ height - 1) × PIP$^+$ Stride)) ÷ 4
   = (9600h + ((160 - 1) × 64)) ÷ 4
   = 12144
   = 2F70h

   Program the PIP$^+$ Display Start Address register. REG[50h] is set to 00002F70h.

4. Determine the PIP$^+$ line address offset.

   number of dwords per line
   = image width ÷ (32 ÷ bpp)
   = 128 ÷ (32 ÷ 4)
   = 16
   = 10h

   Program the PIP$^+$ Line Address Offset register. REG[54h] is set to 00000010h.

5. Enable the PIP$^+$ window.

   Program the PIP$^+$ Window Enable bit. REG[10h] bit 19 is set to 1.

8.3 Limitations

8.3.1 SwivelView 0° and 180°

The PIP$^+$ Line Address Offset register (REG[54h]) requires the PIP$^+$ window image *width* to be a multiple of (32 ÷ bits-per-pixel). If this formula is not satisfied, then the PIP$^+$ Line Address Offset register must be programmed to the next larger value that satisfies the formula.

8.3.2 SwivelView 90° and 270°

The PIP$^+$ Line Address Offset register (REG[54h]) requires the PIP$^+$ window image *width* to be a multiple of (32 ÷ bits-per-pixel). If this formula is not satisfied, then the PIP$^+$ Line Address Offset register must be programmed to the next larger value that satisfies the formula.

9 2D BitBLT Engine

BitBLT is an acronym for Bit Block Transfer. The 2D BitBLT Engine in the S1D13A05 is designed to increase the speed of the most common GUI operations by off-loading work from the CPU, reducing traffic on the system bus and freeing the CPU sooner for other tasks.

All BitBLTs require a destination - a place to write the data. Most BitBLTs have a source of data for the BitBLT and many also incorporate a pattern. The pattern, source, and destination operands are combined using logical AND, OR, XOR and NOT operations. The combining process is called a Raster Operation (ROP) and results in the final pixel data to be written to the destination address.

The S1D13A05 2D BitBLT engine supports a total of sixteen ROPs and works at 8 bpp and 16 bpp color depths. This section describes the BitBLT registers and provides some sample BitBLT operations.

9.1 Registers

The S1D13A05 BitBLT registers are located 8000h bytes from the start of S1D13A05 address space. The registers are labelled, according to their byte offset, as REG[8000h] through REG[8024h]. The following is a description of all BitBLT registers.

BitBLT Control Register
REG[8000h] Default = 00000000h Read/Write

n/a													Color Format Select	Dest Linear Select	Source Linear Select
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
n/a															BitBLT Enable (WO)
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Color Format Select

The Color Format Select bit indicates to the BitBLT engine what color depth to assume for the BitBLT operation. The BitBLT engine uses this information to set the step size for internal counters.

When this bit = 0, 8 bpp is selected and when this bit = 1, 16 bpp is selected.

Destination Linear Select

The Destination Linear Select bit determines how the BitBLT destination address pointer is updated when the BitBLT reaches the end of a row.

When the end of a row is reached and rectangular is selected the destination address is updated to point to the beginning of the next row of a rectangular area. The offset to the start of the next row is contained in the BitBLT Memory Address Offset register (REG[8014h]).

When the end of a row is reached and destination linear is selected the destination address is updated to the next available memory offset. The result is data which is jammed together with one row immediately following the next in display memory. This is useful when it is desired to compactly save a rectangular area into off screen memory.

When this bit = 0, the BitBLT destination is stored as a rectangular region of memory.
When this bit = 1, the BitBLT destination is stored as a contiguous linear block of memory.

Source Linear Select

The Source Linear Select bit determines how the source address pointer is updated when the BitBLT reaches the end of a row.

When the end of a row is reached and rectangular is selected the source address is updated to point to the beginning of the next row of a rectangular area. The offset to the start of the next row is contained in the BitBLT Memory Address Offset register (REG[8014h]).

When the end of a row is reached and source linear is selected the source address is updated to the next available memory offset. The result is data, which was jammed together with one row immediately following the next in display memory, can now be expanded back to a rectangular area.

When this bit = 0, the BitBLT source is stored as a rectangular region of memory.
When this bit = 1, the BitBLT source is stored as a contiguous linear block of memory.

BitBLT Enable

This bit is write only.
Setting this bit to 1 begins the 2D BitBLT operation. **This bit must not be set to 0 while a BitBLT operation is in progress.**

Note

To determine the status of a BitBLT operation use the BitBLT Busy Status bit (REG[8004h] bit 0).

BitBLT Status Register
REG[8004h] Default = 00000000h Read Only

n/a			Number of Used FIFO Entries					n/a			Number of Free FIFO Entries (0 means full)				
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
n/a									FIFO Not Empty	FIFO Half Full	FIFO Full Status	n/a			BitBLT Busy Status
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Number of Used FIFO Entries

This is a read-only status.
This field indicates the minimum number of FIFO entries currently in use. If these bits return a 0, the FIFO is empty.

Number of Free FIFO Entries

This is a read-only status bit
This field indicates the number of empty FIFO entries available. If these bits return a 0, the FIFO is full.

FIFO Not-Empty

This is a read-only status bit.
When this bit = 0, the BitBLT FIFO is empty. When this bit = 1, the BitBLT FiFO has at least one data. To reduce system latency, software can monitor this bit prior to a BitBLT read burst operation.

The following table shows the number of words available in the BitBLT FIFO under different status conditions.

Table 9-1: BitBLT FIFO Words Available

BitBLT FIFO Full Status (REG[8004h] Bit 4)	BitBLT FIFO Half Full Status (REG[8004h] Bit 5)	BitBLT FIFO Not Empty Status (REG[8004h] Bit 6)	Number of Words available in BitBLT FIFO
0	0	0	0
0	0	1	1 to 6
0	1	1	7 to 14
1	1	1	15 to 16

BitBLT FIFO Half Full Status

This is a read-only status bit.
When this bit = 1, the BitBLT FIFO is half full or greater than half full. When this bit = 0, the BitBLT FIFO is less than half full.

BitBLT FIFO Full Status

This is a read-only status bit.
When this bit = 1, the BitBLT FIFO is full. When this bit = 0, the BitBLT FIFO is not full.

BitBLT Busy Status

This bit is a read-only status bit.
When this bit = 1, the BitBLT operation is in progress. When this bit = 0, the BitBLT operation is complete.

Note

During a BitBLT Read operation, the BitBLT engine does not attempt to keep the FIFO full. If the FIFO becomes full, the BitBLT operation stops temporarily as data is read out of the FIFO. The BitBLT will restart only when less than 14 values remain in the FIFO.

BitBLT Command Register
REG[8008h] Default = 00000000h Read/Write

n/a												BitBLT ROP Code bits 3-0			
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
n/a												BitBLT Operation bits 3-0			
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

BitBLT ROP Code

The BitBLT ROP Code specifies the Raster Operation to be used for Write and Move BitBLTs. In addition, for Color Expansion, the BitBLT ROP Code bits 2-0 specify the start bit position for Color Expansion BitBLTs.

Table 9-2 : BitBLT ROP Code/Color Expansion Function Selection

BitBLT ROP Code Bits [3:0]	Boolean Function for Write BitBLT and Move BitBLT	Boolean Function for Pattern Fill	Start Bit Position for Color Expansion
0000	0 (Blackness)	0 (Blackness)	bit 0
0001	~S . ~D or ~(S + D)	~P . ~D or ~(P + D)	bit 1
0010	~S . D	~P . D	bit 2
0011	~S	~P	bit 3
0100	S . ~D	P . ~D	bit 4
0101	~D	~D	bit 5
0110	S ^ D	P ^ D	bit 6
0111	~S + ~D or ~(S . D)	~P + ~D or ~(P . D)	bit 7
1000	S . D	P . D	bit 0
1001	~(S ^ D)	~(P ^ D)	bit 1
1010	D	D	bit 2
1011	~S + D	~P + D	bit 3
1100	S	P	bit 4
1101	S + ~D	P + ~D	bit 5
1110	S + D	P + D	bit 6
1111	1 (Whiteness)	1 (Whiteness)	bit 7

Note

S = Source, D = Destination, P = Pattern.
~ = NOT, . = Logical AND, + = Logical OR, ^ = Logical XOR

BitBLT Operation

The BitBLT Operation selects which BitBLT operation performed. The following table lists the available BitBLT operations.

Table 9-3 : BitBLT Operation Selection

BitBLT Operation Bits [3:0]	BitBLT Operation
0000	Write BitBLT with ROP This operation refers to BitBLTs where data is to be transferred from system memory to display memory
0001	Read BitBLT This operation refers to BitBLTs where data is to be transferred from display memory to system memory
0010	Move BitBLT in positive direction with ROP This operation is used to transfer data from display memory to display memory
0011	Move BitBLT in negative direction with ROP This operation is used to transfer data from display memory to display memory
0100	Transparent Write BitBLT Like the Write BitBLT this operation is used when transferring data from system memory to display memory, the difference is that destination pixels will be left “as is” when source pixels of a specified color are encountered.
0101	Transparent Move BitBLT in positive direction As with the Move BitBLTs this operation is used to transfer data from display memory to display memory. The difference is that destination pixels will be left “as is” when source pixels of a specified color are encountered.
0110	Pattern Fill with ROP Fills the specified area of display memory with a repeating pattern stored in display memory.
0111	Pattern Fill with transparency As with the Pattern Fill, this BitBLT fills a specified area of display memory with a repeating pattern, destination pixels will be left “as is” when source pixels of a specified color are encountered.
1000	Color Expansion This BitBLT expands the bits of the source data into full pixels at the destination. If a source bit is 0 the destination pixel will be background color and if the source bit is 1 the destination pixel will be of foreground color. The source data for Color Expansion BitBLTs is always system memory.
1001	Color Expansion with transparency Like the Color Expansion BitBLT, this operations expands each bit of the source data to occupy a full destination pixel. The difference, is that destination pixels corresponding to source bits of 0 will be left “as is”. The data source is system memory
1010	Move BitBLT with Color Expansion This BitBLT works the same as the Color Expansion BitBLT however the source of the BitBLT is display memory.
1011	Move BitBLT with Color Expansion and transparency This BitBLT works the same as the Color Expansion with Transparency BitBLT however the source of the BitBLT is display memory.
1100	Solid Fill BitBLT Use this BitBLT to fill a given area with one solid color.
Other combinations	Reserved

BitBLT Source Start Address Register
REG[800Ch] Default = 00000000h Read/Write

n/a											BitBLT Source Start Address bits 20-16				
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
BitBLT Source Start Address bits 15-0															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

BitBLT Source Start Address

This register has multiple meanings depending on the BitBLT operation being performed. It can be either:

- the start address in display memory of the source data for BitBLTs where the source is display memory (i.e. Move BitBLTs).
- in pattern fill operations, the BitBLT Source Start Address determines where in the pattern to begin the BitBLT operation and is defined by the following equation:
  Value programmed to the Source Start Address Register =
  Pattern Base Address + Pattern Line Offset + Pixel Offset.
- the data alignment for 16 bpp BitBLTs where the source of BitBLT data is the CPU (i.e. Write BitBLTs).

The following table shows how Source Start Address Register is defined for 8 and 16 bpp color depths.

Table 9-4 : BitBLT Source Start Address Selection

Color Format	Pattern Base Address[20:0]	Pattern Line Offset[2:0]	Pixel Offset[3:0]
8 bpp	BitBLT Source Start Address[20:6]	BitBLT Source Start Address[5:3]	BitBLT Source Start Address[2:0]
16 bpp	BitBLT Source Start Address[20:7]	BitBLT Source Start Address[6:4]	BitBLT Source Start Address[3:0]

BitBLT Destination Start Address Register
REG[8010h] Default = 00000000h Read/Write

n/a											BitBLT Destination Start Address bits 20-16				
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
BitBLT Destination Start Address bits 15-0															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

BitBLT Destination Start Address

This register specifies the initial destination address for BitBLT operations. For rectangular destinations this address represents the upper left corner of the BitBLT rectangle. If the operation is a Move BitBLT in a Negative Direction, these bits define the address of the lower right corner of the rectangle.

BitBLT Memory Address Offset Register
REG[8014h] Default = 00000000h Read/Write

n/a															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
n/a					BitBLT Memory Address Offset bits 10-0										
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

BitBLT Memory Address Offset

This register specifies the 11-bit address offset from the starting word of line *n* to the starting word of line *n + 1*. The offset value is only used for address calculation when the BitBLT is configured as rectangular.

BitBLT Width Register
REG[8018h] Default = 00000000h Read/Write

n/a															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
n/a						BitBLT Width bits 9-0									
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

BitBLT Width

This register specifies the width of a BitBLT in pixels - 1.

BitBLT width (in pixels) = REG[8018h] + 1

BitBLT Height Register
REG[801Ch] Default = 00000000h Read/Write

n/a															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
n/a						BitBLT Height bits 9-0									
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

BitBLT Height

This register specifies the height of the BitBLT in lines - 1.

BitBLT height (in lines) = REG[801Ch] + 1

BitBLT Background Color Register
REG[8020h] Default = 00000000h Read/Write

n/a															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
BitBLT Background Color bits 15-0															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

BitBLT Background Color

This register specifies either:

- the BitBLT background color for Color Expansion or

- the key color for Transparent BitBLT. For 8 bpp BitBLTs, bits 7-0 are used to specify the key color and for 16 bpp BitBLTs, bits 15-0 are used.

BitBLT Foreground Color Register															
REG[8024h]			Default = 00000000h												Read/Write
n/a															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
BitBLT Foreground Color bits 15-0															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

BitBLT Foreground Color

This register specifies the foreground color for Color Expansion or Solid Fill BitBLTs. For 8 bpp BitBLTs, bits 7-0 are used to specify the color and for 16 bpp BitBLTs, bits 15-0 are used.

2D Accelerator (BitBLT) Data Memory Mapped Region Register															
AB16-AB0 = 10000h-1FFFEh, even addresses															Read/Write
BitBLT Data bits 31-16															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
BitBLT Data bits 15-0															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

BitBLT Data bits

This register is used by the local CPU to send data to the BitBLT engine for Write and Color Expansion BitBLTs and is used to read data from the BitBLT engine for Read BitBLTs. The register should be treated as any other register it is however loosely decoded from 10000h to 1FFFEh.

Note

The BitBLT data registers are 32 bits wide but are accessed on WORD boundaries using 16 bit accesses. Byte access to the BitBLT data registers is not allowed.

Note

Accesses to this register, other than for purposes of a BitBLT operation may cause the 13A05 to stop responding and the system to hang.

9.2 BitBLT Descriptions

The S1D13A05 supports 13 fundamental BitBLT operations:

- Write BitBLT with ROP
- Read BitBLT
- Move BitBLT in positive direction with ROP
- Move BitBLT in negative direction with ROP
- Transparent Write BitBLT
- Transparent Move BitBLT in positive direction
- Pattern Fill with ROP
- Pattern Fill with Transparency
- Color Expansion
- Color Expansion with Transparency
- Move BitBLT with Color Expansion
- Move BitBLT with Color Expansion and Transparency
- Solid Fill

Most of the 13 operations are self completing. This means once the BitBLT operation begins it completes without further assistance from the local CPU. No data transfers are required to or from the local CPU. Five BitBLT operations (Write BitBLT with ROP, Transparent Write BitBLT, Color Expansion, Color Expansion with Transparency, Read BitBLT) require data to be written to/read from the BitBLT engine. This data must be transferred one word (16-bits) at a time. This does not imply only 16-bit CPU instructions are acceptable. If a system is able to separate one DWORD write into two WORD writes and the CPU writes the low word before the high word, then 32-bit CPU instructions are acceptable. Otherwise, 16-bit CPU instructions are required.

The data is not directly written to/read from the display buffer. It is written to/read from the BitBLT FIFO through the 64K byte BitBLT aperture specified at the address of REG[10000h]. The 16 word FIFO can be written to only when not full and can be read from only when not empty. Failing to monitor the FIFO status can result in a BitBLT FIFO overflow or underflow.

While the FIFO is being written to by the CPU, it is also being emptied by the S1D13A05. If the S1D13A05 empties the FIFO faster than the CPU can fill it, it may not be possible to cause an overflow/underflow. In these cases, performance can be improved by not monitoring the FIFO status. However, this is very much platform dependent and must be determined for each system.

9.2.1 Write BitBLT with ROP

Write BitBLTs increase the speed of transferring data from system memory to the display buffer. The Write BitBLT with ROP accepts data from the CPU and *writes* it into display memory. This BitBLT is typically used to copy a bitmap image from system memory to the display buffer.

Write BitBLTs support 16 ROPs, the most frequently used being ROP 0Ch (Copy Source to Destination). Write BitBLTs support both rectangular and linear destinations. Using a linear destination it is possible to move an image to off screen memory in a compact format for later restoration using a Move BitBLT.

During a Write BitBLT operation the BitBLT engine expects to receive a particular number of WORDs and it is the responsibility of the CPU to provide the required amount of data.

When performing BitBLT at 16 bpp color depth the number of WORDS to be sent is the same as the number of pixels to be transferred as each pixel is one WORD wide. The number of WORD writes the BitBLT engine expects is calculated using the following formula.

$$\begin{aligned} \text{WORDS} &= \text{Pixels} \\ &= \text{BitBLTWidth} \times \text{BitBLTHeight} \end{aligned}$$

When the color depth is 8 bpp the formula must take into consideration that the BitBLT engine accepts only WORD accesses and each pixel is one BYTE. This may lead to a different number of WORD transfers than there are pixels to transfer.

The number of WORD accesses is dependant on the position of the first pixel within the first WORD of each row. Is the pixel stored in the low byte or the high byte of the WORD? This aspect of the BitBLT is called phase and is determined as follows:

Source phase is 0 when the first pixel is in the low byte and the second pixel is in the high byte of the WORD. When the source phase is 0, bit 0 of the Source Start Address Register is 0. The Source Phase is 1 if the first pixel of each row is contained in the high byte of the WORD, the contents of the low byte are ignored. When the source phase is 1, bit 0 of the Source Start Address Register is set.

Depending on the Source Phase and the BitBLT Width, the last WORD may contain only one pixel. In this case it is always in the low byte. The number of WORD writes the BitBLT engine expects for 8 bpp color depths is shown in the following formula.

$$\text{WORDS} = ((\text{BitBLTWidth} + 1 + \text{SourcePhase}) \div 2) \times \text{BitBLTHeight}$$

The BitBLT engine requires this number of WORDS to be sent from the local CPU before it will end the Write BitBLT operation.

Note

The BitBLT engine counts WORD writes made to the BitBLT register space. This does not imply only 16-bit CPU instructions are acceptable. If a system is able to separate one DWORD write into two WORD writes **and** the CPU writes the low word before the high word, then 32-bit CPU instructions are acceptable. Otherwise, 16-bit CPU instructions are required.

Example 9: Write a 100 x 20 rectangle at the screen coordinates x = 25, y = 38 using a 320x240 display at a color depth of 8 bpp.

1. Calculate the destination address (upper left corner of the screen BitBLT rectangle) using the following formula.

   $$\begin{aligned} \text{DestinationAddress} &= (y \times \text{ScreenStride}) + (x \times \text{BytesPerPixel}) \\ &= (38 \times 320) + (25 \times 1) \\ &= 12185 \\ &= \text{2F99h} \end{aligned}$$

   where:
   BytesPerPixel = 1 for 8 bpp
   ScreenStride = DisplayWidthInPixels × BytesPerPixel = 320 for 8 bpp

   Program the BitBLT Destination Start Address Register. REG[8010h] is set to 2F99h.

2. Program the BitBLT Width Register to 100 - 1. REG[8018h] is set to 63h (99 decimal).

3. Program the BitBLT Height Register to 20 - 1. REG[801Ch] is set to 13h (19 decimal).

4. Program the Source Phase in the BitBLT Source Start Address Register. In this example the data is WORD aligned, so the source phase is 0. REG[800Ch] is set to 00h.

5. Program the BitBLT Operation Register to select the Write BitBLT with ROP. REG[8008h] bits 3-0 are set to 0h.

6. Program the BitBLT ROP Code Register to select Destination = Source. REG[8008h] bits 19-16 are set to 0Ch.

7. Program the BitBLT Color Format Select bit for 8 bpp operations. REG[8000h] bit 18 is set to 0.

8. Program the BitBLT Memory Offset Register to the ScreenStride in WORDS:

   $$\begin{aligned} \text{BLTMemoryOffset} &= \text{DisplayWidthInPixels} \div \text{BytesPerPixel} \\ &= 320 \div 2 \\ &= \text{A0h} \end{aligned}$$

   REG[8014h] is set to A0h.

9. Calculate the number of WORDS the BitBLT engine expects to receive.

   $$\begin{aligned}\text{WORDS} &= ((\text{BLTWidth} + 1 + \text{SourcePhase}) \div 2) \times \text{BLTHeight}\\ &= (100 + 1) \div 2 \times 20\\ &= 1000\\ &= 3E8h\end{aligned}$$

10. Program the BitBLT Destination/Source Linear Select bits for a rectangular BitBLT (BitBLT Destination Linear Select = 0, BitBLT Source Linear Select = 0).

    Start the BitBLT operation. REG[8004h] bit 0 returns a 1.

11. Prior to writing any data to the BitBLT FIFO, confirm the BitBLT FIFO is not full (REG[8004h] bit 4 returns a 0).
    If the BitBLT FIFO Not Empty Status (REG[8004h] bit 6) returns a 0, the FIFO is empty. Write up to 16 WORDS to the BitBLT data register area.
    If the BitBLT FIFO Not Empty Status (REG[8004h] bit 6) returns a 1 and the BitBLT FIFO Half Full Status (REG[8004h] bit 5) returns a 0 then you can write up to 8 WORDS.
    If the BitBLT FIFO Full Status returns a 1, do not write to the BitBLT FIFO until it returns a 0.

    The following table summarizes how many words can be written to the BitBLT FIFO.

Table 9-5: Possible BitBLT FIFO Writes

BitBLT Status Register (REG[8004h])			Word Writes Available
FIFO Not Empty Status	FIFO Half Full Status	FIFO Full Status	
0	0	0	16
1	0	0	8
1	1	0	up to 8
1	1	1	0 (do not write)

Note
The sequence of register initialization is irrelevant as long as all required registers are programmed before the BitBLT is started.

9.2.2 Color Expansion BitBLT

Similar to the Write BitBLT, the Color Expansion BitBLT requires the CPU to feed data to the BitBLT data register It differs in that bits set to one in the source data becomes a complete pixel of foreground color. Source bits set to zero are converted to a pixel of background color. The intended use of this BitBLT operation is to increase the speed of writing text to display memory. As with the Write BitBLT, all data sent to the BitBLT engine must be WORD (16-bit) writes.

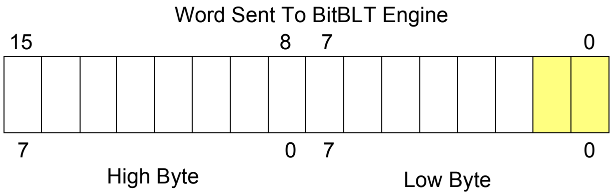
The BitBLT engine expands first the low byte, then the high byte starting at bit 7 of each byte. The start byte of the first WORD to be expanded and the start bit position within this byte must be specified. The start byte position is selected by setting source address bit 0 to 0 to start expanding the low byte or 1 to start expanding the high byte.

Partially “masked” color expansion BitBLTs can be used when drawing a portion of a pattern (i.e. a portion of a character) on the screen. The following examples illustrate how one WORD is expanded using the Color Expansion BitBLT.

1. To expand bits 0-1 of the word:

   Source Address = 0
   Start Bit Position = 1
   BitBLT Width = 2

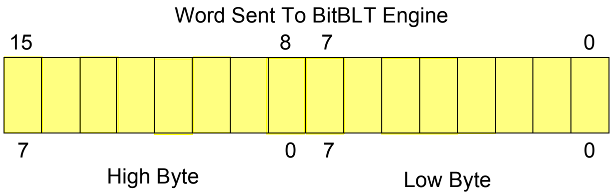
   The following bits are expanded.



2. To expand bits 0-15 of the word (entire word)

   Source Address = 0
   Start Bit Position = 7 (bit seven of the low byte)
   BitBLT Width = 16

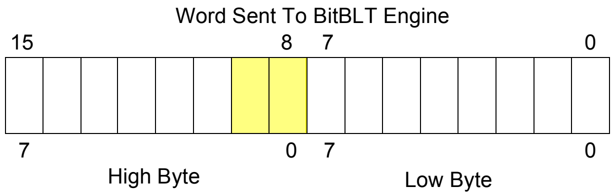
   The following bits are expanded.



3. To expand bits 8-9 of the word

   Source Address = 1
   Start Bit Position = 1
   BitBLT Width = 2

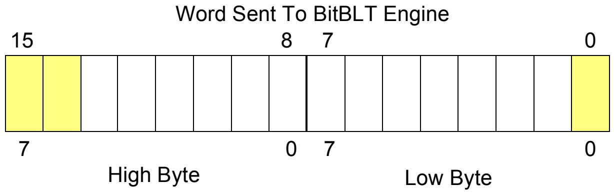
   The following bits are expanded.



4. To expand bits 0,15-14 of the word

   Source Address = 0
   Start Bit Position = 0
   BitBLT Width = 3

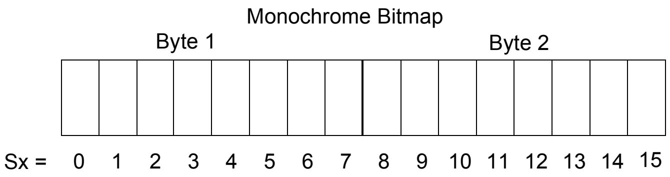
   The following bits are expanded.



All subsequent WORDS in one BitBLT line are then serially expanded starting at bit 7 of the low byte until the end of the BitBLT line. All unused bits in the last WORD are discarded. It is extremely important that the exact number of WORDS is provided to the BitBLT engine. The number of WORDS is calculated from the following formula. This formula is valid for all color depths (8/16 bpp).

$$\text{WORDS} = ((\text{Sx MOD } 16 + \text{BitBLTWidth} + 15) \div 16) \times \text{BitBLTHeight}$$

where:
Sx is the X coordinate of the starting pixel in a word aligned monochrome bitmap.



Example 10: Color expand a rectangle of 12 x 18 starting at the coordinates Sx = 125, Sy = 17 using a 320x240 display at a color depth of 8 bpp.

This example assumes a monochrome, WORD aligned bitmap of dimensions 300 x 600 with the origin at an address A. The color expanded rectangle will be displayed at the screen coordinates X = 20, Y = 30. The foreground color corresponds to the LUT entry at index 134, the background color to index 124.

1. First we need to calculate the address of the WORD within the monochrome bitmap containing the pixel x = 125,y = 17.

   SourceAddress = BitmapOrigin + (y × SourceStride) + (x ÷ 8)
   = A + (Sy × SourceStride) + (Sx ÷ 8)
   = A + (17 × 38) + (125 ÷ 8)
   = A + 646 + 15
   = A + 661

   where:
   SourceStride = (BitmapWidth + 15) ÷ 16
   = (300 + 15) ÷ 16
   = 19 WORDS per line
   = 38 BYTES per line

2. Calculate the destination address (upper left corner of the screen BitBLT rectangle) using the following formula.

   DestinationAddress = (Y × ScreenStride) + (X × BytesPerPixel)
   = (30 × 320) + (20 × 1)
   = 9620
   = 2594h

   where:
   BytesPerPixel = 1 for 8 bpp
   ScreenStride = DisplayWidthInPixels × BytesPerPixel = 320 for 8 bpp

   Program the BitBLT Destination Start Address Register. REG[8010h] is set to 2594h.

3. Program the BitBLT Width Register to 12 - 1. REG[8018h] is set to 0Bh (11 decimal).

4. Program the BitBLT Height Register to 18 - 1. REG[801Ch] is set to 11h (17 decimal).

5. Program the Source Phase in the BitBLT Source Start Address Register. In this example the source address equals A + 661 (odd), so REG[800Ch] is set to 1.

   Since only bit 0 flags the source phase, more efficient code would simply write the low byte of the SourceAddress into REG[800Ch] directly -- not needing to test for an odd/even address. Note that in 16 bpp color depths the Source address is guaranteed to be even.

6. Program the BitBLT Operation Register to select the Color Expand BitBLT. REG[8008h] bits 3-0 are set to 8h.

7. Program the Color Expansion Register. The formula for this example is as follows.

   Color Expansion = 7 - (Sx MOD 8)
   = 7 - (125 MOD 8)
   = 7 - 5
   = 2

   REG[8008h] is set to 2h.

8. Program the Background Color Register to the background color. REG[8020h] is set to 7Ch (124 decimal).

9. Program the Foreground Color Register to the foreground color. REG[8024h] is set to 86h (134 decimal).

10. Program the BitBLT Color Format Register for 8 bpp operation. REG[8000h] bit 18 is set to 0.

11. Program the BitBLT Memory Offset Register to the ScreenStride in WORDS.

    BltMemoryOffset = ScreenStride ÷ 2
    = 320 ÷ 2
    = A0h

    REG[8014h] is set to A0h.

12. Calculate the number of WORDS the BitBLT engine expects to receive.

    First, the number of WORDS in one BitBLT line must be calculated as follows.

    WordsOneLine = ((125 MOD 16) + 12 + 15) ÷ 16
    = (13 + 12 + 15) ÷ 16
    = 40 ÷ 16
    = 2

    Therefore, the total WORDS the BitBLT engine expects to receive is calculated as follows.

    WORDS = WordsOneLine × 18
    = 2 × 18
    = 36

13. Program the BitBLT Destination/Source Linear Select bits for a rectangular BitBLT (BitBLT Destination Linear Select = 0, BitBLT Source Linear Select = 0).

    Start the BitBLT operation. REG[8004h] bit 0 returns a 1.

14. Prior to writing all WORDS to the BitBLT FIFO, confirm the BitBLT FIFO is not full (REG[8004h] bit 4 returns a 0). One WORD expands into 16 pixels which fills all 16 FIFO words in 16 bpp or 8 FIFO words in 8 bpp.

    The following table summarizes how many words can be written to the BitBLT FIFO.

Table 9-6: Possible BitBLT FIFO Writes

BitBLT Status Register (REG[8004h])			8 bpp Word Writes Available	16 bpp Word Writes Available
FIFO Not Empty Status	FIFO Half Full Status	FIFO Full Status		
0	0	0	2	1
1	0	0	1	0 (do not write)
1	1	0	0 (do not write)	
1	1	1		

Note
The sequence of register setup is irrelevant as long as all required registers are programmed before the BitBLT is started.

9.2.3 Color Expansion BitBLT With Transparency

This BitBLT operation is virtually identical to the Color Expand BitBLT, the difference is in how background bits are handled. Bits in the source bitmap which are set to zero result in the destination pixel remaining untouched. Bits set to one are expanded to the foreground color.

Use this BitBLT operation to overlay text onto any background while leaving the background intact.

Refer to the Color Expansion BitBLT for sample calculations and keep the following points in mind:

- Program the BitBLT operation bits, REG[8008h] bits 3-0, to 09h instead of 08h.
- Setting a background color, REG[8020h], is not required.

9.2.4 Solid Fill BitBLT

The Solid Fill BitBLT fills a rectangular area of the display buffer with a solid color. This operation is used to paint large screen areas or to set areas of the display buffer to a given value.

This BitBLT operation is self completing. After setting the width, height, destination start position and (foreground) color the BitBLT engine is started. When the region of display memory is filled with the given color the BitBLT engine will automatically stop.

Example 11: Fill a red 9 x 301 rectangle at the screen coordinates x = 100, y = 10 using a 320x240 display at a color depth of 16 bpp.

1. Calculate the destination address (upper left corner of the destination rectangle) using the following formula.

   DestinationAddress = (y × ScreenStride) + (x × BytesPerPixel)
   = (10 × (320 × 2)) + (100 × 2)
   = 6600
   = 19C8h

   where:
   BytesPerPixel = 2 for 16 bpp
   ScreenStride = DisplayWidthInPixels × BytesPerPixel = 640 for 16 bpp.

   Program the BitBLT Destination Start Address Register. REG[8010h] is set to 19C8h.

2. Program the BitBLT Width Register to 9 - 1. REG[8018h] is set to 08h.

3. Program the BitBLT Height Register to 301 - 1. REG[801Ch] is set to 12Ch (300 decimal).

4. Program the BitBLT Foreground Color Register. REG[8024h] is set to F800h (Full intensity red in 16 bpp is F800h).

5. Program the BitBLT Operation Register to select Solid Fill. REG[8008h] bits 3-0 are set to 0Ch.

6. Program the BitBLT Color Format Register for 16 bpp operations. REG[8000h] bit 18 is set to 1.

7. Program the BitBLT Memory Offset Register to the ScreenStride in WORDS.

   BltMemoryOffset = ScreenStride ÷ 2
   = 320
   = 140h

   REG[8014h] is set to 0140h.

8. Program the BitBLT Destination/Source Linear Select bits for a rectangular BitBLT (BitBLT Destination Linear Select = 0, BitBLT Source Linear Select = 0).

   Start the BitBLT operation. REG[8000h] bit 0 is set to 1.

Note

The sequence of register setup is irrelevant as long as all required registers are programmed before the BitBLT is started.

9.2.5 Move BitBLT in a Positive Direction with ROP

The Move BitBLT is used to copy one area of display memory to another area in display memory.

The source and the destination areas of the BitBLT may be either rectangular or linear. Performing a rectangular to rectangular Move BitBLT creates an exact copy of one portion of video memory at the second location. Selecting a rectangular source to linear destination would be used to compactly store an area of displayed video memory into non-displayed video memory. Later, the area could be restored by performing a linear source to rectangular destination Move BitBLT.

The Move BitBLT in a Positive Direction with ROP is a self completing operation. Once the width, height and the source and destination start addresses are setup and the BitBLT is started the BitBLT engine will complete the operation automatically.

Should the source and destination areas overlap a decision has to be made as to whether to use a Positive or Negative direction so that source data is not overwritten by the move before it is used. Refer to Figure 9-1: to see when to make the decision to switch to the Move BitBLT in a Negative direction. When the destination area overlaps the original source area and the destination address is greater then the source address, use the Move BitBLT in Negative Direction with ROP.

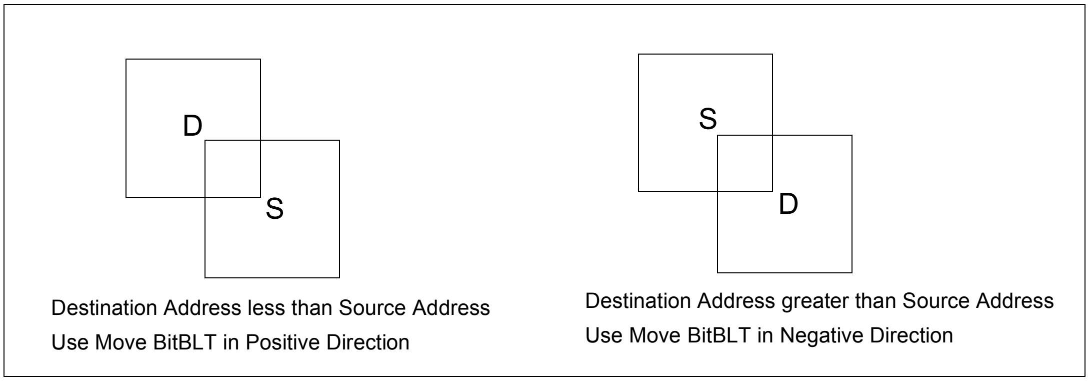

D
S
Destination Address less than Source Address
Use Move BitBLT in Positive Direction
S
D
Destination Address greater than Source Address
Use Move BitBLT in Negative Direction

Figure 9-1: Move BitBLT Usage

Example 12: Copy a 9 x 101 rectangle at the screen coordinates x = 100, y = 10 to screen coordinates x = 200, y = 20 using a 320x240 display at a color depth of 16 bpp.

1. Calculate the source and destination addresses (upper left corners of the source and destination rectangles), using the following formula.

   SourceAddress $= (y \times \text{ScreenStride}) + (x \times \text{BytesPerPixel})$
   $= (10 \times (320 \times 2)) + (100 \times 2)$
   $= 6600$
   = 19C8h

   DestinationAddress $= (y \times \text{ScreenStride}) + (x \times \text{BytesPerPixel})$
   $= (20 \times (320 \times 2)) + (200 \times 2)$
   $= 13200$
   = 3390h

   where:
   BytesPerPixel = 2 for 16 bpp
   ScreenStride = DisplayWidthInPixels × BytesPerPixel = 640 for 16 bpp

   Program the BitBLT Source Start Address Register. REG[800Ch] is set to 19C8h.

   Program the BitBLT Destination Start Address Register. REG[8010h] is set to 3390h.

2. Program the BitBLT Width Register to 9 - 1. REG[8018h] is set to 08h.

3. Program the BitBLT Height Register to 101 - 1. REG[801Ch] is set to 64h (100 decimal).

4. Program the BitBLT Operation Register to select the Move BitBLT in Positive Direction with ROP. REG[8008h] bits 3-0 are set to 2h.

5. Program the BitBLT ROP Code Register to select Destination = Source. REG[8008h] bits 19-16 are set to 0Ch.

6. Program the BitBLT Color Format Select bit for 16 bpp operations. REG[8000h] bit 18 is set to 1.

7. Program the BitBLT Memory Offset Register to the ScreenStride in WORDS.

   BltMemoryOffset $= \text{ScreenStride} \div 2$
   $= 320$
   = 140h

   REG[8014h] is set to 0140h.

8. Program the BitBLT Destination/Source Linear Select bits for a rectangular BitBLT (BitBLT Destination Linear Select = 0, BitBLT Source Linear Select = 0).

   Start the BitBLT operation. REG[8000h] bit 0 is set to 1.

Note
The sequence of register setup is irrelevant as long as all required registers are programmed before the BitBLT is started.

9.2.6 Move BitBLT in Negative Direction with ROP

The Move BitBLT in Negative Direction with ROP is similar to the Move BitBLT in Positive direction. Use this BitBLT operation when the source and destination BitBLT areas overlap and the destination address is greater then the source address. Refer to Figure 9-1: on page 82 to see when to make the decision to switch to the Move BitBLT in a Positive direction.

When using the Move BitBLT in Negative Direction it is necessary to calculate the addresses of the last pixels as opposed to the first pixels. This means calculating the addresses of the lower right corners as opposed to the upper left corners.

Example 13: Copy a 9 x 101 rectangle at the screen coordinates x = 100, y = 10 to screen coordinates X = 105, Y = 20 using a 320x240 display at a color depth of 16 bpp.

In the following example, the coordinates of the source and destination rectangles intentionally overlap.

1. Calculate the source and destination addresses (**lower right** corners of the source and destination rectangles) using the following formula.

   SourceAddress
   = ((y + Height - 1) × ScreenStride) + ((x + Width - 1) × BytesPerPixel)
   = ((10 + 101 - 1) × (320 × 2)) + ((100 + 9 - 1) × 2)
   = 70616
   = 113D8h

   DestinationAddress
   = ((Y + Height - 1) × ScreenStride) + ((X + Width - 1) × BytesPerPixel)
   = ((20 + 101 - 1) × (320 × 2)) + ((105 + 9 - 1) × 2)
   = 77026
   = 12CE2h

   where:
   BytesPerPixel = 2 for 16 bpp
   ScreenStride = DisplayWidthInPixels × BytesPerPixel = 640 for 16 bpp

   Program the BitBLT Source Start Address Register. REG[800Ch] is set to 113D8h.

   Program the BitBLT Destination Start Address Register. REG[8010h] is set to 12CE2h.

2. Program the BitBLT Width Register to 9 - 1. REG[8018h] is set to 08h.

3. Program the BitBLT Height Register to 101 - 1. REG[801Ch] is set to 64h (100 decimal).

4. Program the BitBLT Operation Register to select the Move BitBLT in Negative Direction with ROP. REG[8008] bits 3-0 are set to 3h.

5. Program the BitBLT ROP Code Register to select Destination = Source. REG[8008h] bits 19-16 are set to 0Ch.

6. Program the BitBLT Color Format Select bit for 16 bpp operations. REG[8000h] bit 18 is set to 1.

7. Program the BitBLT Memory Offset Register to the ScreenStride in WORDS.

$$\begin{aligned} \text{BltMemoryOffset} &= \text{ScreenStride} \div 2 \\ &= 320 \\ &= 140\text{h} \end{aligned}$$

   REG[8014h] is set to 0140h.

8. Program the BitBLT Destination/Source Linear Select bits for a rectangular BitBLT (BitBLT Destination Linear Select = 0, BitBLT Source Linear Select = 0).

   Start the BitBLT operation. REG[8000h] bit 0 is set to 1.

Note

The sequence of register setup is irrelevant as long as all required registers are programmed before the BitBLT is started.

9.2.7 Transparent Write BitBLT

Transparent Write BitBLTs are similar to the Write BitBLT with ROP with two differences; first, a specified color in the source data leaves the destination pixel untouched and second ROPs are not supported.

This operation is used to copy a bitmap image from system memory to the display buffer with one source color treated as transparent. Pixels of the transparent color are not transferred. This allows fast display of non-rectangular or masked images. For example, consider a source bitmap having a red circle on a blue background. By selecting the blue as the transparent color and using the Transparent Write BitBLT on the whole rectangle, the effect is a BitBLT of the red circle only.

During a Transparent Write BitBLT operation the BitBLT engine expects to receive a particular number of WORDs and it is the responsibility of the CPU to provide the required amount of data.

When performing BitBLTs at 16 bpp color depth the number of WORDS to be sent is the same as the number of pixels as each pixel is one WORD wide. The number of WORD writes the BitBLT engine expects is calculated using the following formula.

$$\begin{aligned} \text{WORDS} &= \text{Pixels} \\ &= \text{BitBLTWidth} \times \text{BitBLTHeight} \end{aligned}$$

When the color depth is 8 bpp the formula must take into consideration that the BitBLT engine accepts only WORD accesses and each pixel is one BYTE. This may lead to a different number of WORD transfers than there are pixels to transfer.

The number of WORD accesses is dependant on the position of the first pixel within the first WORD of each row. Is the pixel stored in the low byte or the high byte of the WORD? This aspect of the BitBLT is called phase and is determined as follows:

Source phase is 0 when the first pixel is in the low byte and the second pixel is in the high byte of the WORD. When the source phase is 0, bit 0 of the Source Start Address Register is 0. The Source Phase is 1 if the first pixel of each row is contained in the high byte of the WORD, the contents of the low byte are ignored. When the source phase is 1, bit 0 of the Source Start Address Register is set.

Depending on the Source Phase and the BitBLT Width, the last WORD may contain only one pixel. In this case it is always in the low byte. The number of WORD writes the BitBLT engine expects for 8 bpp color depths is shown in the following formula.

$$\text{WORDS} = ((\text{BitBLTWidth} + 1 + \text{SourcePhase}) \div 2) \times \text{BitBLTHeight}$$

Once the Transparent Write BitBLT begins, the BitBLT engine remains active until all pixels have been written. The BitBLT engine requires the correct number of WORDS to be sent from the local CPU before it ends the Transparent Write BitBLT operation.

Note

The BitBLT engine counts WORD writes made to the BitBLT register. This does not imply only 16-bit CPU instructions are acceptable. If a system is able to separate one DWORD write into two WORD writes **and** the CPU writes the low word before the high word, then 32-bit CPU instructions are acceptable. Otherwise, 16-bit CPU instructions are required.

Example 14: Write 100 x 20 pixels at the screen coordinates x = 25, y = 38 using a 320x240 display at a color depth of 8 bpp. Transparent color is high intensity blue (assume LUT Index 124).

1. Calculate the destination address (upper left corner of the screen BitBLT rectangle), using the formula:

$$\begin{aligned}\text{DestinationAddress} &= (y \times \text{ScreenStride}) + (x \times \text{BytesPerPixel})\\ &= (38 \times 320) + (25 \times 1)\\ &= 12185\\ &= \text{2F99h}\end{aligned}$$

   where:
   BytesPerPixel = 1 for 8 bpp
   ScreenStride = DisplayWidthInPixels × BytesPerPixel = 320 for 8 bpp

   Program the BitBLT Destination Start Address Register. REG[8010h] is set to 2F99h.

2. Program the BitBLT Width Register to 100 - 1. REG[8018h] is set to 63h (99 decimal).

3. Program the BitBLT Height Register to 20 - 1. REG[801Ch] is set to 13h (19 decimal).

4. Program the Source Phase in the BitBLT Source Start Address Register. In this example, the data is WORD aligned, so the source phase is 0. REG[800Ch] is set to 00h.

5. Program the BitBLT Operation Register to select Transparent Write BitBLT.
   REG[8008h] bits 3-0 are set to 4h.

6. Program the BitBLT Background Color Register to select transparent color.
   REG[8020h] is set to 7Ch (124 decimal).

7. Program the BitBLT Color Format Select bit for 8 bpp operations. REG[8000h] bit 18 is set to 0.

8. Program the BitBLT Memory Offset Register to the ScreenStride in WORDS.

   $$\begin{aligned}\text{BltMemoryOffset} &= \text{ScreenStride} \div 2\\ &= 320 \div 2\\ &= 160\\ &= \text{A0h}\end{aligned}$$

   REG[8014h] is set to 0A0h.

9. Calculate the number of WORDS the BitBLT engine expects to receive.

   $$\begin{aligned}\text{WORDS} &= ((\text{BLTWidth} + 1 + \text{SourcePhase}) \div 2) \times \text{BLTHeight}\\ &= (100 + 1 + 0) \div 2 \times 20\\ &= 1000\\ &= \text{3E8h}\end{aligned}$$

10. Program the BitBLT Destination/Source Linear Select bits for a rectangular BitBLT (BitBLT Destination Linear Select = 0, BitBLT Source Linear Select = 0).

    Start the BitBLT operation. REG[8004h] bit 0 returns a 1.

11. Prior to writing any data to the BitBLT FIFO, confirm the BitBLT FIFO is not full (REG[8004h] bit 4 returns a 0).
    If the BitBLT FIFO Not Empty Status (REG[8004h] bit 6) returns a 0, the FIFO is empty. Write up to 16 WORDS to the BitBLT data register area.
    If the BitBLT FIFO Not Empty Status (REG[8004h] bit 6) returns a 1 and the BitBLT FIFO Half Full Status (REG[8004h] bit 5) returns a 0 then you can write up to 8 WORDS.
    If the BitBLT FIFO Full Status returns a 1, do not write to the BitBLT FIFO until it returns a 0.

    The following table summarizes how many words can be written to the BitBLT FIFO.

Table 9-7: Possible BitBLT FIFO Writes

BitBLT Status Register (REG[8004h])			Word Writes Available
FIFO Not Empty Status	FIFO Half Full Status	FIFO Full Status	
0	0	0	16
1	0	0	8
1	1	0	less than 8
1	1	1	0 (do not write)

Note

The sequence of register setup is irrelevant as long as all required registers are programmed before the BitBLT is started.

9.2.8 Transparent Move BitBLT in Positive Direction

The Transparent Move BitBLT in Positive Direction combines the capabilities of the Move BitBLT with the ability to define a transparent color. Use this operation to copy a masked area of display memory to another area in display memory.

The source and the destination areas of the BitBLT may be either rectangular or linear. Performing a rectangular to rectangular Move BitBLT creates an exact copy of one portion of video memory at the second location. Selecting a rectangular source to linear destination would be used to compactly store an area of displayed video memory into non-displayed video memory. Later, the area could be restored by performing a linear source to rectangular destination Move BitBLT.

The transparent color is not copied during this operation, whatever pixel color existed in the destination will be there when the BitBLT completes. This allows fast display of non-rectangular images. For example, consider a source bitmap having a red circle on a blue background. By selecting the blue color as the transparent color and using the Transparent Move BitBLT on the whole rectangle, the effect is a BitBLT of the red circle only.

Note

The Transparent Move BitBLT is supported **only** in a positive direction.

Example 15: Copy a 9 x 101 rectangle at the screen coordinates x = 100, y = 10 to screen coordinates X = 200, Y = 20 using a 320x240 display at a color depth of 16 bpp. Transparent color is blue.

1. Calculate the source and destination addresses (upper left corners of the source and destination rectangles), using the formula:

$$\begin{aligned}\text{SourceAddress} &= (y \times \text{ScreenStride}) + (x \times \text{BytesPerPixel})\\ &= (10 \times (320 \times 2)) + (100 \times 2)\\ &= 6600\\ &= \text{19C8h}\end{aligned}$$

$$\begin{aligned}\text{DestinationAddress} &= (Y \times \text{ScreenStride}) + (X \times \text{BytesPerPixel})\\ &= (20 \times (320 \times 2)) + (200 \times 2)\\ &= 13200\\ &= \text{3390h}\end{aligned}$$

where:
BytesPerPixel = 2 for 16 bpp
ScreenStride = DisplayWidthInPixels × BytesPerPixel = 640 for 16 bpp

Program the BitBLT Source Start Address Register. REG[800Ch] is set to 19C8h.

Program the BitBLT Destination Start Address Register. REG[8010h] is set to 3390h.

2. Program the BitBLT Width Register to 9 - 1. REG[8018h] is set to 08h.
3. Program the BitBLT Height Register to 101 - 1. REG[801Ch] is set to 64h (100 decimal).
4. Program the BitBLT Operation Register to select the Transparent Move BitBLT in Positive Direction. REG[8008h] bits 3-0 are set to 05h.
5. Program the BitBLT Background Color Register to select blue as the transparent color. REG[8020h] is set to 001Fh (Full intensity blue in 16 bpp is 001Fh).
6. Program the BitBLT Color Format Register to select 16 bpp operations. REG[8000h] bit 18 is set to 1.
7. Program the BitBLT Memory Offset Register to the ScreenStride in WORDS.

   BltMemoryOffset = ScreenStride ÷ 2
   = 320
   = 140h

   REG[8014h] is set to 0140h.
8. Program the BitBLT Destination/Source Linear Select bits for a rectangular BitBLT (BitBLT Destination Linear Select = 0, BitBLT Source Linear Select = 0).

   Start the BitBLT operation. REG[8000h] bit 0 is set to 1.

Note

The order of register setup is irrelevant as long as all relevant registers are programmed before the BitBLT is initiated.

9.2.9 Pattern Fill BitBLT with ROP

The Pattern Fill BitBLT with ROP fills a specified area of display memory with a pattern. The pattern is repeated until the fill area is completely filled. The fill pattern is limited to an eight by eight pixel array and must be loaded to off-screen video memory before starting the BitBLT. The pattern can be logically combined with the destination using any of the 16 ROP codes, but typically the copy pattern ROP is used (ROP code 0Ch).

A pattern is defined to be an array of 8x8 pixels and the pattern data must be stored in consecutive bytes of display memory (64 consecutive bytes for 8 bpp color depths and 128 bytes for 16 bpp color depths). For 8 bpp color depths the pattern must begin on a 64 byte boundary, for 16 bpp color depths the pattern must begin on a 128 byte boundary.

This operation is self completing. Once the parameters have been entered and the BitBLT started the BitBLT engine will fill all of the specified memory with the pattern.

To fill an area using the pattern BitBLT, the BitBLT engine requires the location of the pattern, the destination rectangle position and size, and the ROP code. The BitBLT engine also needs to know which pixel from the pattern is the first pixel in the destination rectangle (the pattern start phase). This allows seamless redrawing of any part of the screen using the pattern fill.

Example 16: Fill a 100 x 150 rectangle at the screen coordinates x = 10, y = 20 with the pattern in off-screen memory at offset 3C000h using a 320x240 display at a color depth of 8 bpp. The first pixel (upper left corner) of the rectangle is the pattern pixel at x = 3, y = 4.

1. Calculate the destination address (upper left corner of the destination rectangle), using the formula:

   $$\begin{aligned}\text{DestinationAddress} &= (y \times \text{ScreenStride}) + (x \times \text{BytesPerPixel})\\ &= (20 \times 320) + (10 \times 1)\\ &= 6410\\ &= \text{190Ah}\end{aligned}$$

   where:
   BytesPerPixel = 1 for 8 bpp
   ScreenStride = DisplayWidthInPixels × BytesPerPixels = 320 for 8 bpp

   Program the BitBLT Destination Start Address Register. REG[8010h] is set to 190Ah.

2. Calculate the source address. This is the address of the pixel in the pattern that is the origin of the destination fill area. The pattern begins at offset 240K, but the first pattern pixel is at x = 3, y = 4. Therefore, an offset within the pattern itself must be calculated.

   $$\begin{aligned}&\text{SourceAddress}\\ &= \text{PatternOffset} + \text{StartPatternY} \times 8 \times \text{BytesPerPixel} + \text{StartPatternX} \times \text{BytesPerPixel}\\ &= 240\text{K} + (4 \times 8 \times 1) + (3 \times 1)\\ &= 240\text{K} + 35\\ &= 245795\\ &= \text{3C023h}\end{aligned}$$

   where:
   BytesPerPixel = 1 for 8 bpp
   Program the BitBLT Source Start Address Register. REG[800Ch] is set to 3C023h.

3. Program the BitBLT Width Register to 100 - 1. REG[8018h] is set to 63h (99 decimal).

4. Program the BitBLT Height Register to 150-1. REG[801Ch] is set to 95h (149 decimal).

5. Program the BitBLT Operation Register to select the Pattern Fill with ROP. REG[8008h] bits 3-0 are set to 6h.

6. Program the BitBLT ROP Code Register to select Destination = Source. REG[8008h] bits 19-16 are set to 0Ch.

7. Program the BitBLT Color Format Select bit for 8 bpp operations. REG[8000h] bit 18 is set to 0.

8. Program the BitBLT Memory Offset Register to the ScreenStride in WORDS.

   BltMemoryOffset = ScreenStride ÷ 2
   = 320 ÷ 2
   = 160
   = A0h

   REG[8014h] is set to 00A0h.

9. Program the BitBLT Destination/Source Linear Select bits for a rectangular BitBLT (BitBLT Destination Linear Select = 0, BitBLT Source Linear Select = 0).

   Start the BitBLT operation. REG[8000h] bit 0 is set to 1.

Note

The sequence of register setup is irrelevant as long as all required registers are programmed before the BitBLT is started.

9.2.10 Pattern Fill BitBLT with Transparency

This operation is very similar to the Pattern Fill BitBLT with the difference being that one color can be specified to be transparent. Whenever the Transparent color is encountered in the pattern data the destination is left as is. This operation is useful to create hatched or striped patterns where the original image shows through the hatching.

The requirements for this BitBLT are the same as for the Pattern Fill BitBLT the only change in programming is that the BitBLT Operation field of REG[8008h] must be set to 07h and the BitBLT Background color register, REG[8020h] must be set to the desired color.

Example 17: Fill a 100 x 150 rectangle at the screen coordinates x = 10, y = 20 with the pattern in off-screen memory at offset 3C000h using a 320x240 display at a color depth of 8 bpp. The first pixel (upper left corner) of the rectangle is the pattern pixel at x = 3, y = 4. Transparent color is blue (assumes LUT index 1).

1. Calculate the destination address (upper left corner of destination rectangle), using the formula:

   DestinationAddress = (y × ScreenStride) + (x × BytesPerPixel)
   = (20 × 320) + (10 × 1)
   = 6410
   = 190Ah

   where:
   BytesPerPixel = 1 for 8 bpp
   ScreenStride = DisplayWidthInPixels × BytesPerPixels = 320 for 8 bpp

   Program the BitBLT Destination Start Address Register. REG[8010h] is set to 190Ah.

2. Calculate the source address. This is the address of the pixel in the pattern that is the origin of the destination fill area. The pattern begins at offset 240K, but the first pattern pixel is at x = 3, y = 4. Therefore, an offset within the pattern itself must be calculated.

   SourceAddress
   = PatternOffset + StartPatternY × 8 × BytesPerPixel + StartPatternX × BytesPerPixel
   = 240K + (4 × 8 × 1) + (3 × 1)
   = 240K + 35
   = 245795
   = 3C023h

   where:
   BytesPerPixel = 1 for 8 bpp
   Program the BitBLT Source Start Address Register. REG[800Ch] is set to 3C0023h.

3. Program the BitBLT Width Register to 100 - 1. REG[8018h] is set to 63h (99 decimal).

4. Program the BitBLT Height Register to 150-1. REG[801Ch] is set to 95h (149 decimal).

5. Program the BitBLT Operation Register to select the Pattern Fill BitBLT with Transparency. REG[8008h] bits 3-0 are set to 7h.

6. Program the BitBLT Background Color Register to select transparent color. This example uses blue (LUT index 1) as the transparent color. REG[8020h] is set to 01h.

7. Program the BitBLT Color Format Select bit for 8 bpp operations. REG[8000h] bit 18 is set to 0.

8. Program the BitBLT Memory Offset Register to the ScreenStride in WORDS.

   BltMemoryOffset = ScreenStride ÷ 2
   = 320 ÷ 2
   = 160
   = A0h

   REG[8014h] is set to A0h.

9. Program the BitBLT Destination/Source Linear Select bits for a rectangular BitBLT (BitBLT Destination Linear Select = 0, BitBLT Source Linear Select = 0).

   Start the BitBLT operation. REG[8000h] bit 0 is set to 1.

Note

The sequence of register setup is irrelevant as long as all required registers are programmed before the BitBLT is started.

9.2.11 Move BitBLT with Color Expansion

The Move BitBLT with Color Expansion takes a monochrome bitmap as the source and color expands it into the destination. All bits set to one in the source are expanded to destination pixels of the selected foreground color. All bits set to zero in the source are expanded to pixels of the selected background color.

The Move BitBLT with Color Expansion is used to accelerate text drawing. A monochrome bitmap of a font, in off-screen video memory, occupies very little space and takes advantage of the hardware acceleration. Since the foreground and background colors are programmable, text of any color can be created.

The Move BitBLT with Color Expansion can move data from one rectangular area to another, or either the source or destination may be specified to be linear. Storing rectangular display data in linear format in off screen memory results in a tremendous space saving.

Example 18: Color expand a 9 x 16 rectangle using the pattern in off-screen memory at 3C000h and move it to the screen coordinates x = 200, y = 20. Assume a 320x240 display at a color depth of 16 bpp, Foreground color of black, and background color of white.

1. Calculate the destination and source addresses (upper left corner of the destination and source rectangles), using the formula.

   DestinationAddress = (y × ScreenStride) + (x × BytesPerPixel)
   = (20 × (320 × 2)) + (200 × 2)
   = 13200
   = 3390h

   where:
   BytesPerPixel = 2 for 16 bpp
   ScreenStride = DisplayWidthInPixels × BytesPerPixels = 640 for 16 bpp

   SourceAddress = 240K
   = 3C000h

   Program the BitBLT Destination Start Address Register. REG[8010h] is set to 3390h.

   Program the BitBLT Source Start Address Register. REG[800Ch] is set to 3C000h.

2. Program the BitBLT Width Register to 9 - 1. REG[8018h] is set to 08h.

3. Program the BitBLT Height Register to 16 - 1. REG[801Ch] is set to 0Fh.

4. Program the BitBLT ROP Code/Color Expansion Register. REG[8008h] bits 19-16 are set to 7h.

5. Program the BitBLT Operation Register to select the Move BitBLT with Color Expansion. REG[8008h] bits 3-0 are set to 0Bh.

6. Program the BitBLT Foreground Color Register to select black (in 16 bpp black = 0000h). REG[8024h] is set to 0000h.

7. Program the BitBLT Background Color Register to select white (in 16 bpp white = FFFFh). REG[8024h] is set to FFFFh.

8. Program the BitBLT Color Format Select bit for 16 bpp operations. REG[8000h] bit 18 is set to 1.

9. Program the BitBLT Memory Offset Register to the ScreenStride in WORDS.

$$\begin{aligned}\text{BltMemoryOffset} &= \text{ScreenStride} \div 2\\ &= 320\\ &= 140\text{h}\end{aligned}$$

   REG[8014h] is set to 0140h.

10. Program the BitBLT Destination/Source Linear Select bits for a rectangular BitBLT (BitBLT Destination Linear Select = 0, BitBLT Source Linear Select = 0).

    Start the BitBLT operation. REG[8000h] bit 0 is set to 1.

Note

The sequence of register setup is irrelevant as long as all required registers are programmed before the BitBLT is started.

9.2.12 Transparent Move BitBLT with Color Expansion

The Transparent Move BitBLT with Color Expansion is virtually identical to the Move BitBLT with Color Expansion. This operation expands bits set to one in the source bitmap to the foreground color in the destination. Bits set to zero in the source bitmap leave the corresponding destination pixel as is.

Setup and use this operation exactly as the Move BitBLT with Color Expansion.

9.2.13 Read BitBLT

This Read BitBLT increases the speed of transferring data from the video memory to system memory. This BitBLT complements the Write BitBLT and is typically used to save a part of the display buffer to the system memory. Once the Read BitBLT begins, the BitBLT engine remains active until all the pixels have been read.

During a Read BitBLT operation the BitBLT engine expects to send a particular number of WORDs to the CPU, and it is the responsibility of the CPU to read the required amount of data.

When performing BitBLT at 16 bpp color depth the number of WORDS to be sent is the same as the number of pixels to be transferred as each pixel is one WORD wide. The number of WORD writes the BitBLT engine expects is calculated using the following formula.

WORDS = Pixels
= BitBLTWidth × BitBLTHeight

When the color depth is 8 bpp the formula must take into consideration that the BitBLT engine accepts only WORD accesses and pixels are only one BYTE. This may lead to a different number of WORD transfers than there are pixels to transfer.

The number of WORD accesses is dependant on the position of the first pixel within the first WORD of each destination row. Is the pixel stored in the low byte or the high byte of the WORD? Read BitBLT phase is determined as follows:

Destination phase is 0 when the first pixel is in the low byte and the second pixel is in the high byte of the WORD. When the destination phase is 0, bit 0 of the Destination Start Address Register is 0. The destination phase is 1 if the first pixel of each destination row is contained in the high byte of the WORD, the contents of the low byte are ignored. When the destination phase is 1, bit 0 of the Destination Start Address Register is set.

Depending on the destination phase and the BitBLT width, the last WORD may contain only one pixel. In this case it is always in the low byte. The number of WORD writes the BitBLT engine expects for 8 bpp color depths is shown in the following formula.

WORDS = ((BitBLTWidth + 1 + DestinationPhase) ÷ 2) × BitBLTHeight

The BitBLT engine requires this number of WORDS to be sent from the local CPU before it will end the Write BitBLT operation.

Example 19: Read 100 x 20 pixels at the screen coordinates x = 25, y = 38 and save to system memory. Assume a display of 320x240 at a color depth of 8 bpp.

1. Calculate the source address (upper left corner of the screen BitBLT rectangle), using the formula.

   SourceAddress = (y × ScreenStride) + (x × BytesPerPixel)
   = (38 × 320) + (25 × 1)
   = 12185
   = 2F99h

   where:
   BytesPerPixel = 1 for 8 bpp
   ScreenStride = DisplayWidthInPixels × BytesPerPixels = 320 for 8 bpp

   Program the BitBLT Source Start Address Register. REG[800Ch] is set to 2F99h.

2. Program the BitBLT Width Register to 100 - 1. REG[8018h] is set to 63h (99 decimal).

3. Program the BitBLT Height Register to 20 - 1. REG[801Ch] is set to 13h (19 decimal).

4. Program the Destination Phase in the BitBLT Destination Start Address Register. In this example, the data is WORD aligned, so the destination phase is 0. REG[8010h] is set to 00h.

5. Program the BitBLT Operation to select the Read BitBLT. REG[8008h] bits 3-0 are set to 1h.

6. Program the BitBLT Color Format Select bit for 8 bpp operations. REG[8000h] bit 18 is set to 0.

7. Program the BitBLT Memory Offset Register to the ScreenStride in WORDS.

   $$\begin{aligned}\text{BltMemoryOffset} &= \text{ScreenStride} \div 2\\ &= 320 \div 2\\ &= 160\\ &= \text{A0h}\end{aligned}$$

   REG[8014h] is set to 0A0h.

8. Calculate the number of WORDS the BitBLT engine expects to receive.

   $$\begin{aligned}\text{WORDS} &= ((\text{BLTWidth} + 1 + \text{DestinationPhase}) \div 2) \times \text{BLTHeight}\\ &= (100 + 1 + 0) \div 2 \times 20\\ &= 1000\\ &= \text{3E8h}\end{aligned}$$

9. Program the BitBLT Destination/Source Linear Select bits for a rectangular BitBLT (BitBLT Destination Linear Select = 0, BitBLT Source Linear Select = 0).

   Start the BitBLT operation. REG[8004h] bit 0 returns a 1.

10. Prior to reading from the BitBLT FIFO, confirm the BitBLT FIFO is not empty (REG[8004h] bit 4 returns a 1). If the BitBLT FIFO Not Empty Status (REG[8004h] bit 6) returns a 1 and the BitBLT FIFO Half Full Status (REG[8004h] bit 5) returns a 0 then you can read up to 8 WORDS. If the BitBLT FIFO Full Status returns a 1, read up to 16 WORDS. If the BitBLT FIFO Not Empty Status returns a 0 (the FIFO is empty), do not read from the BitBLT FIFO until it returns a 1.

    The following table summarizes how many words can be read from the BitBLT FIFO.

Table 9-8: Possible BitBLT FIFO Reads

BitBLT Status Register (REG[8004h])			Word Reads Available
FIFO Not Empty Status	FIFO Half Full Status	FIFO Full Status	
0	0	0	0 (do not read)
1	0	0	up to 8
1	1	0	8
1	1	1	16

Note

The sequence of register initialization is irrelevant as long as all required registers are programmed before the BitBLT is started.

9.3 BitBLT Synchronization

A BitBLT operation can only be started if the BitBLT engine is not busy servicing another BitBLT. Before a new operation is started, software must confirm the BitBLT Busy Status bit (REG[8004h] bit 0) is set to zero. The status of this bit can either be tested **after** each BitBLT operation, or **before** each BitBLT operation.

Testing the BitBLT Status After

Testing the BitBLT Active Status after starting a new BitBLT is simpler and less prone to errors.

To test after each BitBLT operation, perform the following.

1. Program and start the BitBLT engine.
2. Wait for the current BitBLT operation to finish -- Poll the BitBLT Busy Status bit (REG[8004h] bit 0) until it returns a 0.
3. Continue with program execution.

Testing the BitBLT Status Before

Testing the BitBLT Active Status before starting a new BitBLT results in better performance, as both CPU and BitBLT engine can be running at the same time. This is most useful for BitBLTs that are self completing (once started they don’t require any CPU assistance). While the BitBLT engine is busy, the CPU can do other tasks. To test before each BitBLT operation, perform the following.

1. Wait for the current BitBLT operation to finish -- Poll the BitBLT Busy Status bit (REG[8004h] bit 0) until it returns a 0.
2. Program and start the new BitBLT operation.
3. Continue with program execution (CPU and BitBLT engine work independently).

This approach can pose problems when mixing CPU and BitBLT access to the display buffer. For example, if the CPU writes a pixel while the BitBLT engine is running and the CPU writes a pixel before the BitBLT finishes, the pixel may be overwritten by the BitBLT. To avoid this scenario, always assure no BitBLT is in progress before accessing the display buffer with the CPU, or don’t use the CPU to access the display buffer at all.

9.4 Known Limitations

The S1D13A05 BitBLT engine has the following limitations.

- The 2D Accelerator Data Memory Mapped register must not be accessed except during BitBLT operations. Read from the register only during Read BitBLT operations and write to the register only during Write and Color Expand BitBLTs. Accessing the register at any other time may result in S1D13A05 stopping to respond and the system to freeze.
- The Read and Write BitBLT operations are not available when the S1D13A05 is configured for the Redcap or Dragonball without DTACK host bus interfaces.
- A BitBLT operation cannot be terminated once it has been started.

9.5 Sample Code

Sample code demonstrating how to program the S1D13A05 BitBLT engine is provided in the files **A05src.zip**. This file is available on the internet at vdc.epson.com.

10 Programming the USB Controller

USB (Universal Serial Bus) is an external bus designed to ease the connection and use of peripheral devices. USB incorporates a host/client architecture in which the host initiates all data transactions and the client either receives or supplies data to the host.

USB offers the following features to the end user:

- Single plug type for all peripheral devices.
- Support for up to 127 simultaneous devices.
- Speeds up to 12 Megabits per second.
- “hot-plugging” peripherals.

The S1D13A05 USB controller supports revision 1.1 of the USB specification. The S1D13A05 USB controller handles many common USB tasks without requiring local processor intervention. For example, setup and data transfers are handled automatically by the S1D13A05 controller. The controller notifies the local CPU, through an interrupt, when data is ready to be read from the FIFO or when data has been transmitted to the host.

This section demonstrates how to program and use the S1D13A05 USB controller. Topics covered include:

- Basic concepts such as registers and interrupts
- Initialization and data transfers
- S1D13A05 USB known issues.

10.1 Registers and Interrupts

10.1.1 Registers

Configuration, interrupt notification, and data transfers are all done using the S1D13A05 USB registers. The USB registers are located 4000h bytes past the beginning of S1D13A05 address space and should be written/read using 16 bit accesses.

On most systems the start of S1D13A05 address space, is fixed by the system design. The S1D13A05 evaluation board uses a PCI interface, thus the start of S1D13A05 address space may vary from one session to the next. Example code is written using a pointer to the USB registers (pUSB). The USB examples do not show how to obtain the register address. For a description of how to get the register address when using the S1D13A05 evaluation board, refer to the function halAcquireController() in Section 11, “Hardware Abstraction Layer” on page 118.

10.1.2 Interrupts

The S1D13A05 uses an interrupt to notify the local CPU when a USB event, which requires servicing, occurs. Events, such as USB reset and data transfer notifications generate interrupts.

It is beyond the scope of this document to explain how to setup and configure the interrupt system for the variety of platforms the S1D13A05 supports. The examples and flowcharts assume there is one interrupt handling routine which will determine the cause of the interrupt and call the appropriate handler function. It is assumed the user understands the mechanics and architecture of their system well enough setup a routine which will receive an interrupt notification and determine the cause of the interrupt.

10.2 Initialization

Initialization describes the process of setting the registers state to enable the USB controller for use. There are two cases where the USB registers need to be initialized. When the system is powered up and the registers need to be prepared for first use. The second time the registers need to be initialized is after receiving a RESET request from the host controller.

Refer to Section 10.2.2, “USB Registers” on page 102 for an example of the register initialization sequence.

10.2.1 GPIO Setup

The S1D13A05 shares four lines between GPIO and USB use. Before **any** accesses are made to the USB section the GPIO lines **must** be configured. To set the GPIO lines write the binary value 0010xxxx-1101xxxx-00000000-xxxxxxxx (2xDx00xxh) to REG[64h], the GPIO Status and Control register.

Note

X’s represent a don’t care state. Depending on other system configuration (i.e. panel technology) certain don’t care bits may have to be set also. See the *S1D13A05 Hardware Functional Specification*, document number X40A-A-001, for more information regarding the bits in the GPIO Status and Control register.

10.2.2 USB Registers

The steps described below are typical of the startup of the S1D13A05 USB controller.

- registers are set to an initial value
- the S1D13A05 is connected to a USB host controller
- the host controller issues a RESET command
- the USB registers are re-initialized

As initialization for both steps are similar it is recommended that one routine perform the sequence. The following table depicts a typical register initialization sequence.

Table 10-1: USB Controller Initialization Sequence

Register		Value (hex)	Notes
REG[4040h]	USBFC INPUT CONTROL	40	Enable the USB differential input receiver and indicate we are a bulk transfer self powered device. (for ISOchronous mode, use 43h)
REG[4044]	PIN IO STATUS DATA	01	USBPUP must be set to enable the USB interface and registers. REG[4000h] to REG[403Ah] cannot be written until this bit is set.
REG[4000]	CONTROL	84	Enable the clocks and USB GPIO pins.
REG[4024]	EP3 RECEIVE FIFO STATUS	1C	Clear EP3 status.
REG[402C]	USB EP4 TX FIFO STATUS	1C	Clear EP4 status
REG[4032]	USB STATUS	7E	Clear EP2 valid bit
REG[4004]	INTERRUPT STATUS 0	FF	Clear any pending USB interrupts
REG[4010]	EP1 INDEX	00	Set EP1 index to zero
REG[4018]	EP2 INDEX	00	Set EP2 index to zero
ext REG[00] ext REG[01]	VENDOR ID MSB VENDOR ID LSB	?? ??	Provide appropriate vendor ID
ext REG[02] ext Reg[03]	PRODUCT ID MSB PRODUCT ID LSB	?? ??	Provide appropriate product ID
ext REG[0C]	FIFO CONTROL	01	Enable EP4 (FIFO) valid transfer mode.
REG[4002]	INT ENABLE 0	0A	Enable interrupts for EP1 and EP3
REG[4004]	INT STATUS 0	0A	Make sure any pending interrupts are cleared.
REG[4046	INTERRUPT CONTROL ENABLE 0	02	Enable RESET and endpoints notifications
REG[4048]	INTERRUPT CONTROL ENABLE 1	01	
REG[404A]	INTERRUPT CONTROL STATUS/CLEAR 0	7F	Clear ALL interrupt status...
REG[404C]	INTERRUPT CONTROL STATUS/CLEAR 1	7F	
REG[4000]	CONTROL	A4	Enable the USB port for use

The USB controller is ready for operation with the following configuration:

- Endpoint 1 (mailbox receive) is configured for bulk OUT and Endpoint 2 (mailbox transmit) is configured for interrupt IN. The functionality of these endpoints cannot be altered.
- Endpoint 3 (FIFO receive) is configured for bulk in and Endpoint 4 (FIFO transmit) is configured for bulk out. Endpoints 3 and 4 may also be configured for isochronous operation.

When the S1D13A05 is connected to a host controller, the host will issue a RESET command to the S1D13A05. In response to the RESET the S1D13A05 clears all USB registers in the range REG[4000h] to REG[403Ah]. The client software must respond to the reset and reprogram the USB registers. A host controller may issue a RESET at any time during operation.

After the S1D13A05 receives the RESET and re-initializes the registers, the host controller starts the USB SETUP phase. The SETUP sequence is handled entirely by the S1D13A05 USB controller. After the setup is complete the S1D13A3 is ready to begin transferring data.

Note

Prior to initializing the registers, host controller accesses are responded to with NAKs. After being configured, host controller accesses will be handled in the normal way.

Note

A Vendor ID can be obtained through the USB Implementers Forum at http://www.usb.org.

10.3 Data Transfers

The S1D13A05 USB requires very little local CPU assistance during data transfers. For the most part data transfers from the host involve reading a FIFO data register when notified of that the transfer is complete or writing a FIFO register and setting a ’ready’ bit to send data to the host.

The following sections expand on the data transfer mechanism.

10.3.1 Receiving Data from the Host - the OUT command

Data transferred from the host to the S1D13A05 is directed to either EndPoint 1 (the mailbox) or EndPoint 3 (the FIFO). When the data packet has been successfully received the S1D13A05 generates an interrupt.

On receipt of the interrupt the local CPU examines the masked interrupt status registers REG[404Eh] and REG[4050h] to determine the source of the interrupt. If the interrupt came from bit 0 of the Negative Interrupt Masked Status register, REG[4050h], the next step is to examine REG[4004] to determine the exact cause of the interrupt.

Endpoint 1 - Mailbox Receive

If the cause of the interrupt is determined to be EndPoint 1 (REG[4004h] bit 1 = 1), then the data is read from the EndPoint 1 data register (REG[4012h]). The following figure shows the procedure for the CPU to read the mailbox register.

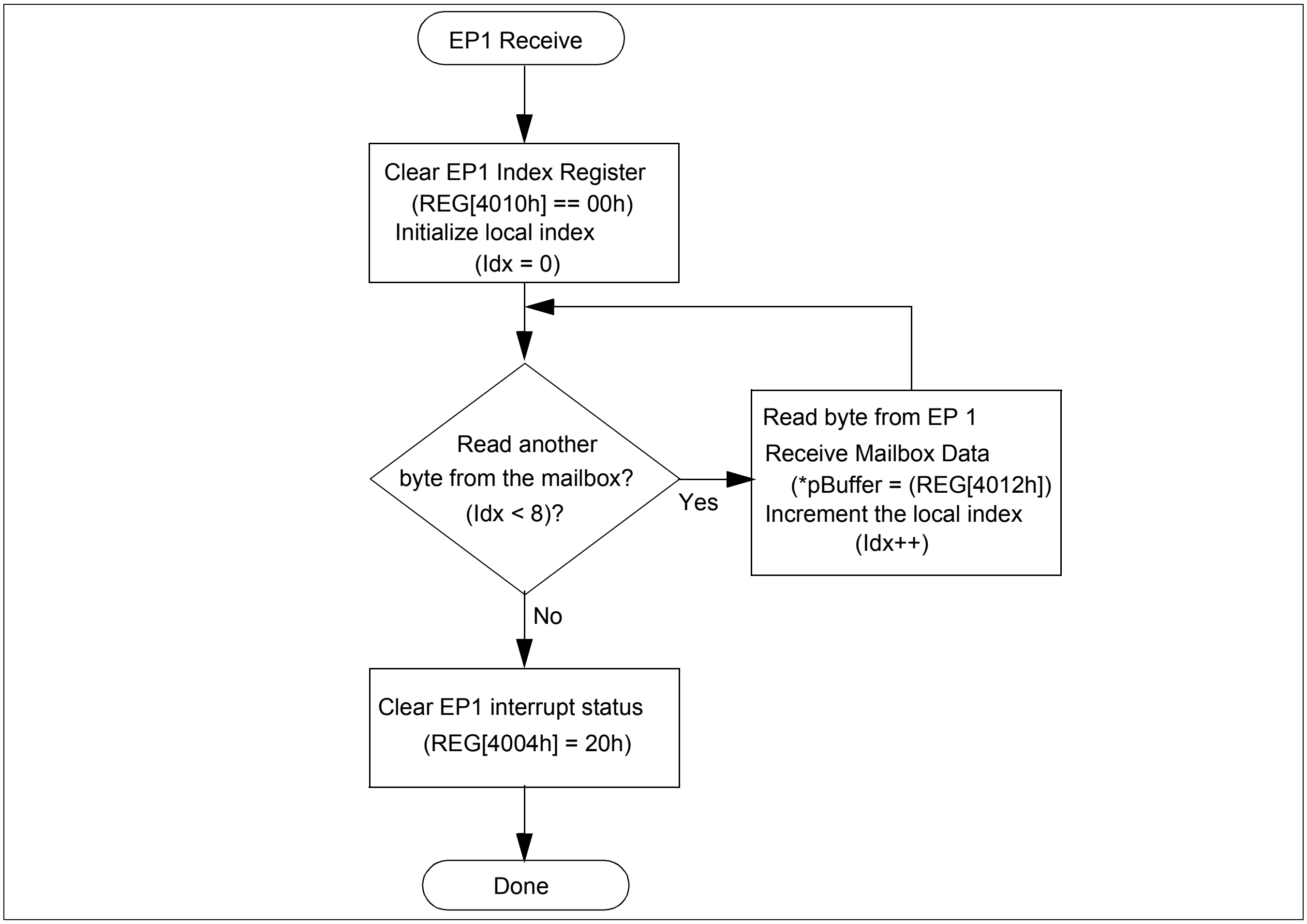


Figure 10-1: Endpoint 1 Data Reception

Note

In this diagram reference is made to two pseudo-variables:

Idx is an integer used as a loop counter

pBuffer is a pointer to eight bytes of memory to store the EP1 data

Endpoint 3 - FIFO Receive

If the cause of the interrupt is determined to be EndPoint 3, REG[4004h] bit 3 = 1b, then the host controller has sent data to EndPoint 3. Figure 10-2: shows the procedure for reading data from EndPoint 3.

An EndPoint 3 interrupt is generated when the number of bytes in the receive FIFO equal the value in the Receive FIFO Almost Full Threshold register (REG[403Ah], Index[06h]). The default value is sixty bytes. On systems where bulk transfers are used, the default value for the receive FIFO threshold should be satisfactory.

Systems with slow processors, high interrupt service latency, or configured for isochronous operation may have to decrease this value to allow the CPU time to begin reading data before the data transfer overflows the FIFO.

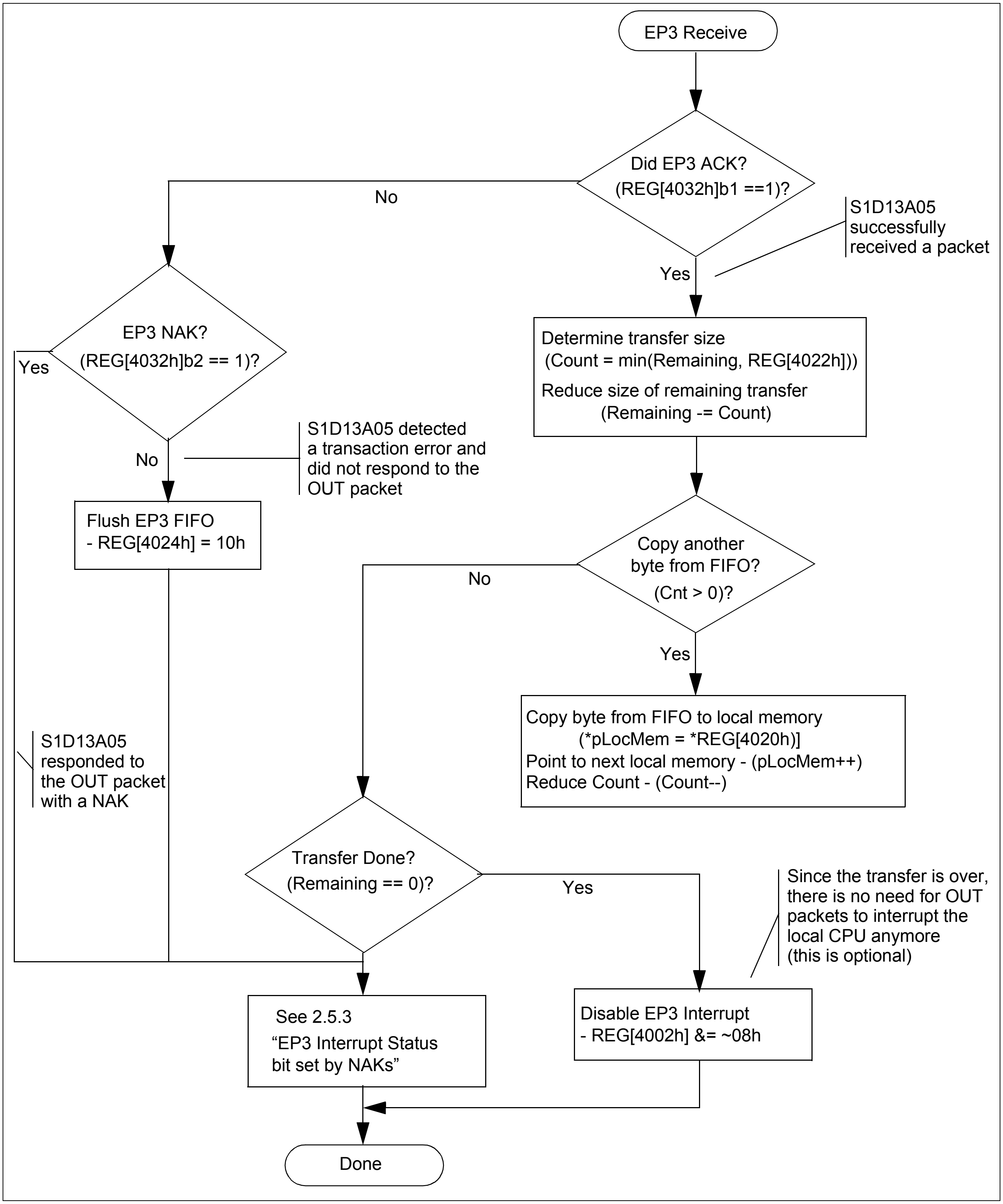

EP3 Receive
Did EP3 ACK?
(REG[4032h]b1 ==1)?
No
Yes
S1D13A05 successfully received a packet
EP3 NAK?
(REG[4032h]b2 == 1)?
Yes
No
S1D13A05 detected a transaction error and did not respond to the OUT packet
Determine transfer size
(Count = min(Remaining, REG[4022h]))
Reduce size of remaining transfer
(Remaining -= Count)
Flush EP3 FIFO
- REG[4024h] = 10h
Copy another byte from FIFO?
(Cnt > 0)?
No
Yes
Copy byte from FIFO to local memory
(*pLocMem = *REG[4020h)]
Point to next local memory - (pLocMem++)
Reduce Count - (Count--)
S1D13A05 responded to the OUT packet with a NAK
Transfer Done?
(Remaining == 0)?
Yes
Since the transfer is over, there is no need for OUT packets to interrupt the local CPU anymore (this is optional)
See 2.5.3
“EP3 Interrupt Status bit set by NAKs”
Disable EP3 Interrupt
- REG[4002h] &= ~08h
Done

Figure 10-2: Endpoint 3 Data Reception

10.3.2 Sending Data to the Host - the IN command

Data transfers to the host controller occur when the host issues an IN command. The data comes from EndPoint 2 (the mailbox) or EndPoint 4 (the FIFO). The data transfer is handled automatically by the S1D13A05 and requires no CPU assistance.

Data transfers, from the S1D13A05 to the host controller, are performed by writing the data into either EndPoint 2 (mailbox) or EndPoint 4 (FIFO) data registers. After writing the data to the registers a control bit indicating that mailbox or FIFO data is valid is set.

Endpoint 2 - Mailbox Transmit

Figure 10-3: shows the logical flow for sending data to the host controller using EndPoint 2, the mailbox.

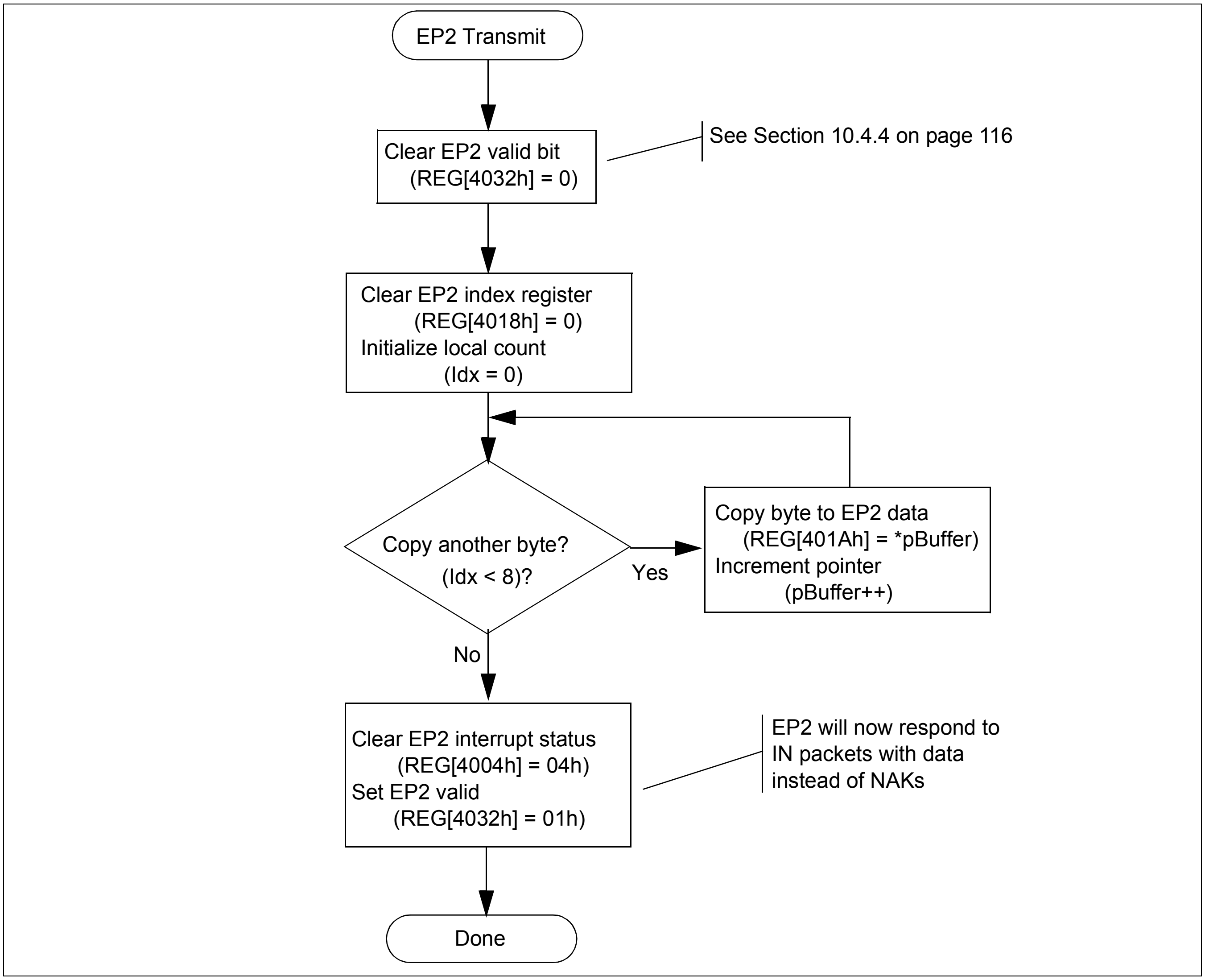

EP2 Transmit
Clear EP2 valid bit (REG[4032h] = 0)
See Section 10.4.4 on page 116
Clear EP2 index register (REG[4018h] = 0)
Initialize local count (Idx = 0)
Copy another byte? (Idx < 8)?
Yes
Copy byte to EP2 data (REG[401Ah] = *pBuffer)
Increment pointer (pBuffer++)
No
Clear EP2 interrupt status (REG[4004h] = 04h)
Set EP2 valid (REG[4032h] = 01h)
EP2 will now respond to IN packets with data instead of NAKs
Done

Figure 10-3: EndPoint 2 Data Transmission

Note

In this diagram reference is made to two pseudo-variables:

Idx is an integer used as a loop counter

pBuffer is a pointer to eight bytes of memory to send to the host

Endpoint 4 - Data Transmit

Transferring data to the host controller using the FIFO controller has additional overhead as this routine must run tests to ensure error free data transmission.

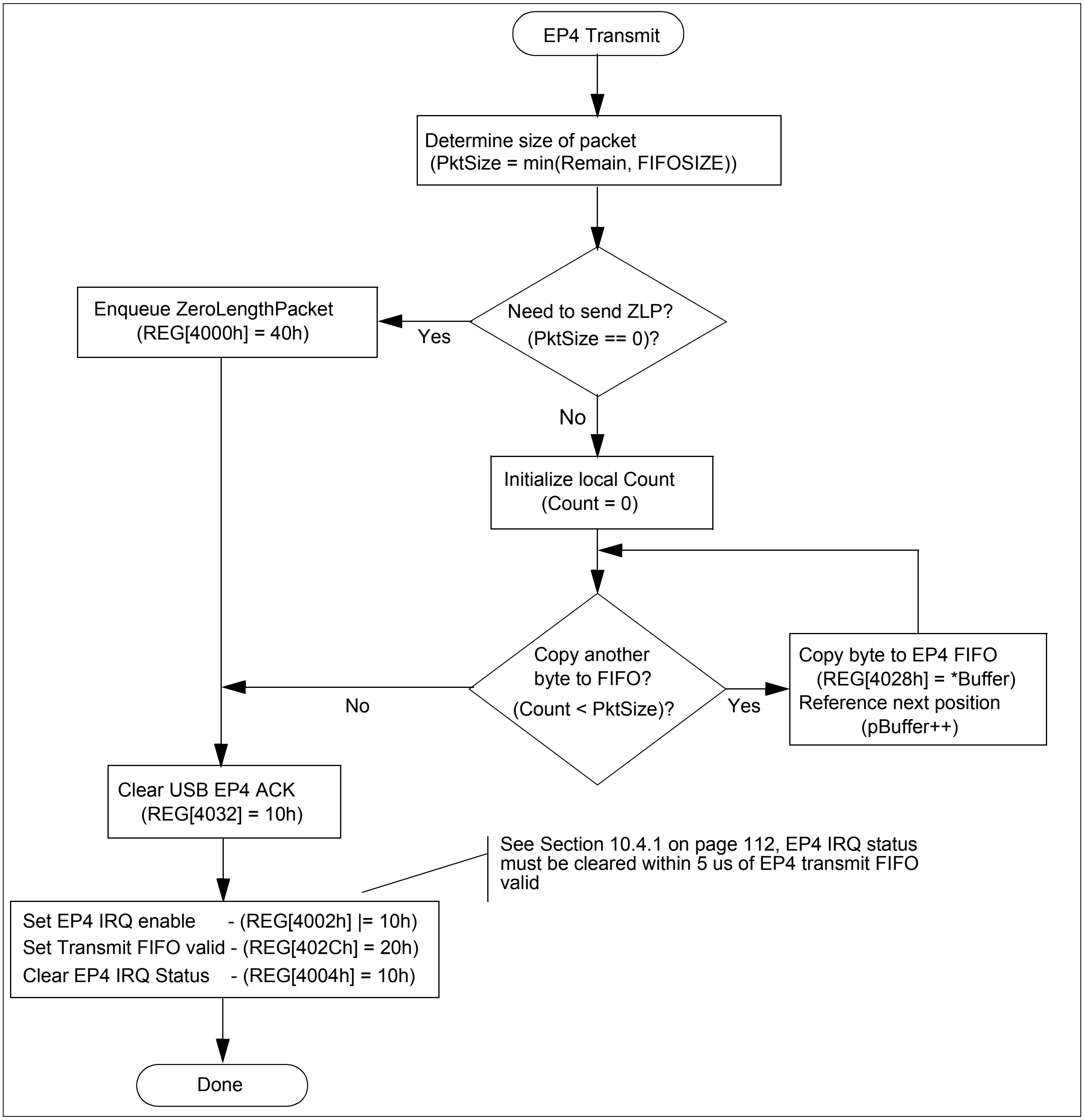


Figure 10-4: Endpoint 4 Data Transmission

Note

In this example there are three variables:

PktSize is an integer containing the number of bytes to transfer in this packet
Count is an integer used for local loop control
pBuffer is a pointer to an array of at least FIFOSIZE bytes.

To ensure the host controller receives the packet error free, an interrupt handler for EndPoint 4 must be configured and the flow control as shown in the following diagram must be implemented.

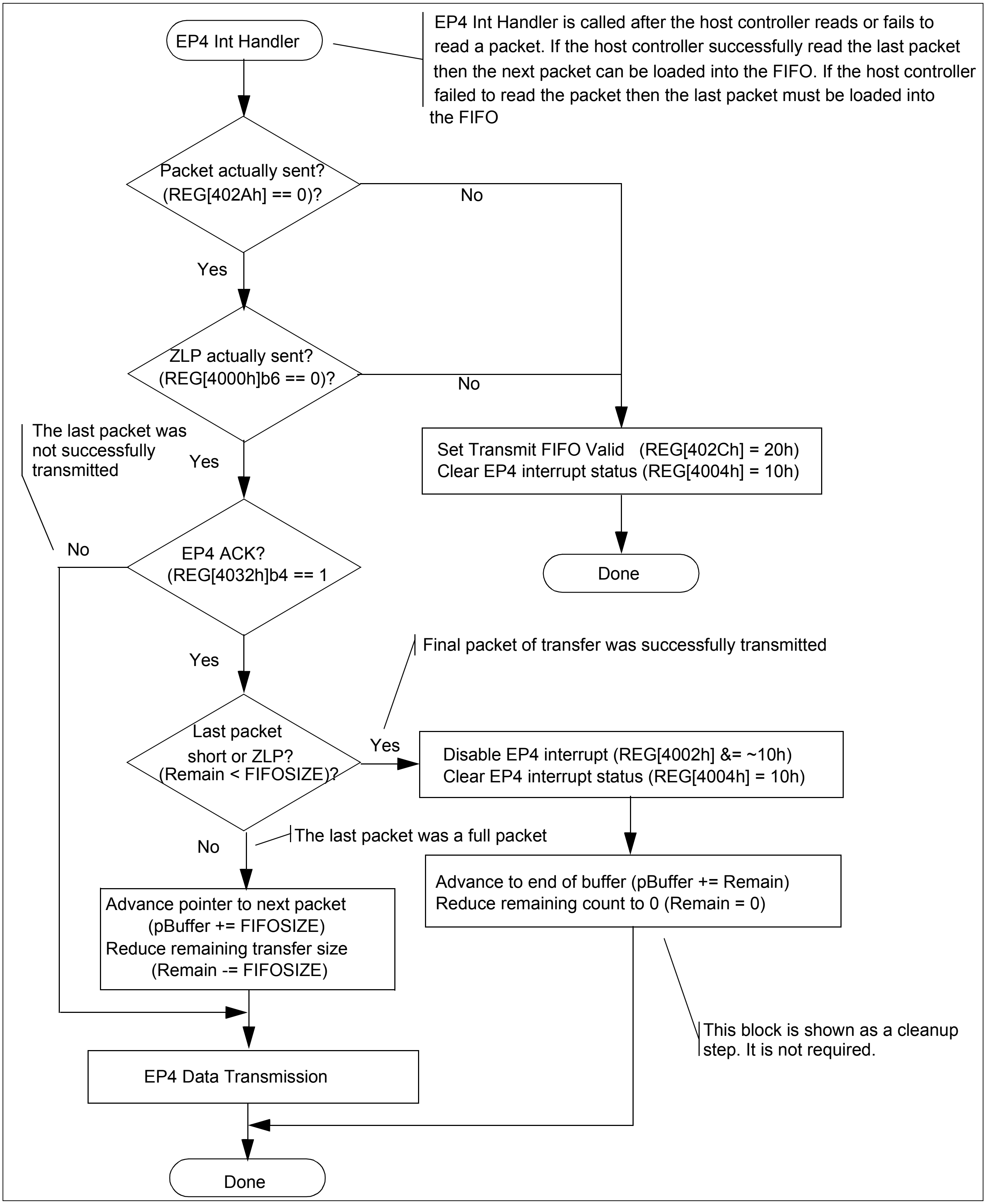

EP4 Int Handler
EP4 Int Handler is called after the host controller reads or fails to read a packet. If the host controller successfully read the last packet then the next packet can be loaded into the FIFO. If the host controller failed to read the packet then the last packet must be loaded into the FIFO
Packet actually sent? (REG[402Ah] == 0)?
No
Yes
ZLP actually sent? (REG[4000h]b6 == 0)?
No
Set Transmit FIFO Valid (REG[402Ch] = 20h)
Clear EP4 interrupt status (REG[4004h] = 10h)
Done
The last packet was not successfully transmitted
Yes
No
EP4 ACK? (REG[4032h]b4 == 1
Yes
Final packet of transfer was successfully transmitted
Last packet short or ZLP? (Remain < FIFOSIZE)?
Yes
Disable EP4 interrupt (REG[4002h] &= ~10h)
Clear EP4 interrupt status (REG[4004h] = 10h)
No
The last packet was a full packet
Advance to end of buffer (pBuffer += Remain)
Reduce remaining count to 0 (Remain = 0)
Advance pointer to next packet (pBuffer += FIFOSIZE)
Reduce remaining transfer size (Remain -= FIFOSIZE)
This block is shown as a cleanup step. It is not required.
EP4 Data Transmission
Done

Figure 10-5: Endpoint 4 Interrupt Handling

Note

In the diagram the variables:

pBuffer is a pointer to the local memory buffer containing the data to be transferred to the host controller

Remain is an integer tracking the number of bytes still to be sent.

10.4 Known Issues

This section presents known issues with USB transfers when using the S1D13A05 USB controller.

10.4.1 EP4 NAK Status not set correctly in USB Status Register

The EP4 NAK status bit is not set in the USB Status Register (REG4032h]) when the S1D13A05 responds to an IN request on EP4 with a NAK. As a result, a local CPU receiving an “EP4 Packet Transmitted” interrupt may mistakenly believe a bus error occurred in the most recently transmitted packet.

Work Around

Disable the EP4 Packet Transmitted interrupt when no data is queued for transmission to the local CPU. The basic flow is:

In Chip Initialization Code

Do not enable ‘EP4 Packet Transmitted’ bit in Interrupt Enable Register 0 (REG[4002h]).

When Local Side Wishes to Send Data

1. Put data to transmit in FIFO.
2. Enable ‘EP4 Packet Transmitted’ bit in Interrupt Enable Register 0.
3. Set FIFO Valid (if using FIFO Valid Mode == TRUE). See Section 10.4.2 on page 113 for more information on setting the FIFO Valid.
4. Clear ‘EP4 Packet Transmitted’ status bit in Interrupt Status Register 0 (REG[4004]).

Note

Step 4 is time-critical. It must be performed within 5 µs after Step 3.

In Packet Transmitted Interrupt Routine

Disable ‘EP4 Packet Transmitted’ bit in Interrupt Enable Register 0.

10.4.2 Write to EP4 FIFO Valid bit cleared by NAK

After the local CPU sets EP4 FIFO Valid (in Endpoint 4 FIFO Status Register, REG[402Ch]), the S1D13A05 will erroneously clear the EP4 valid bit if the S1D13A05 is concurrently sending a NAK handshake in response to a previous IN token to EP4.

Work Around

The work-around is in the 'EP4 Packet Transmitted' interrupt routine. It requires the interrupt routine to know whether the recently queued packet was a zero-length packet or not, so that must be stored as a bit when the packet was loaded into the FIFO. On entry to the 'EP4 Packet Transmitted' interrupt routine:

For a non-zero-length Packet

Check the FIFO count. If it is non-zero, this error occurred. In that case, set FIFO Valid again, clear the interrupt status bit, and exit the interrupt routine.

For a zero-length Packet

Check the Software EOT bit (in Control Register, REG[4000h]). If it is set, the FIFO Valid write failed. In that case, set FIFO Valid again, clear the interrupt status bit, and exit the interrupt routine

10.4.3 EP3 Interrupt Status bit set by NAKs

When receiving Bulk OUT packets from a Host PC, the S1D13A05 "Endpoint 3 Interrupt Status" interrupt typically is used to notify the peripheral firmware that a packet has been received. This bit also serves as the "Receive FIFO Valid" bit, so additional packets addressed to Endpoint 3 are NAKed until this status bit is cleared. Once cleared, however, it may become set by another packet which is NAKed by the S1D13A05, causing the Receive FIFO to become "Valid" again. The Host PC may immediately attempt to re-transmit the NAKed packet. The firmware should be written to prevent a cycle in which the FIFO is "Valid" each time that the Host PC sends an OUT packet.

The following rules govern the S1D13A05's behavior regarding packets received on Endpoint 3:

Rule A. At the end of a received OUT token to EP3 (and before the data is received), the S1D13A05 decides to NAK the packet if the "EP3 Interrupt Status" bit is set, and will therefore throw away data received.

Rule B. At the end of a received packet (including one which is NAKed), the S1D13A05 sets the "EP3 Interrupt Status" bit.

Rule C. Local firmware should clear the "EP3 Interrupt Status" bit after reading all bytes out of the EP3 Receive FIFO.

The following figure shows how a repeating cycle of NAKed OUT packets may occur.

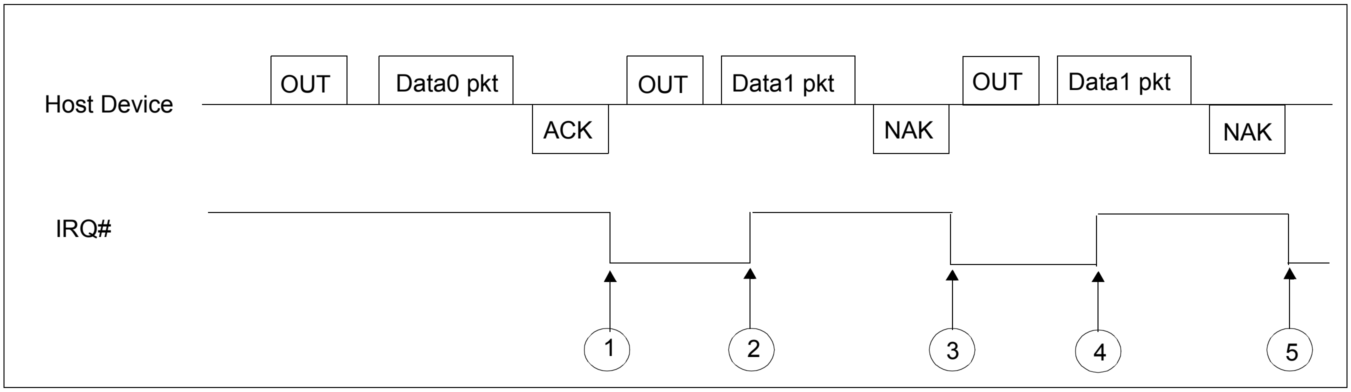


Figure 10-6: Firmware Looping Continuously on Received OUT packets

At Point 1, the EP3 Interrupt activates because a packet has been received. In response, the firmware reads the bytes out of the packet and clears the interrupt at Point 2. A second packet is already being received at Point 2, and the S1D13A05 has already decided to NAK this packet due to Rule A. At point 3, the S1D13A05 has NAKed the packet and asserts the Interrupt status bit.

Again, the local firmware responds to the interrupt, and seeing it is only a “NAK” interrupt, clears the interrupt condition at Point 4. However, the Host PC has begun to retry the second packet already, so the packet will again get NAKed due to Rule B. This cycle could continue until something changes the flow of OUT packets – for instance, an SOF at the beginning of the next frame, or packet traffic directed at another device or endpoint.

Work Around

The normal program flow for a packet which the S1D13A05 NAKs is as follows:

1. S1D13A05 asserts IRQ# after NAKing a received packet on EP3.
2. Local CPU is interrupted, enters interrupt routine.
3. Local CPU reads Interrupt Status Register 0 (REG[4004h]) and sees “EP3 Packet Received” interrupt bit.
4. Local CPU reads USB Status Register (REG[4032h]) and sees “NAK” bit set.
5. Local CPU clears Interrupt Status Register 0 (REG[4004h]) “EP3 Packet Status” interrupt bit.
6. Local CPU clears USB Status Register (REG[4032]) “NAK” bit.

The technique for avoiding this potential pitfall depends on the speed of the peripheral CPU. The critical timing parameter is the time from the S1D13A05 asserting IRQ# to the firmware clearing the “EP3 Packet Received” bit in Interrupt Status Register 0.

For a Fast CPU

A CPU which can clear the Interrupt Status Register 0 bit within 10 msec after the S1D13A05 asserts the IRQ# signal requires no extra code to prevent the potential cycling. In this case, the CPU is fast enough to clear a NAKed packet's Interrupt Status Register 0 bit before another packet can be received.

For a Slow CPU

A CPU which can't meet the timing requirements for a fast CPU above will require some additional firmware to eliminate the potential for this cycle. After successfully receiving a packet on Endpoint 3 and emptying received data out of the FIFO, the firmware should follow the flow in the following figure.

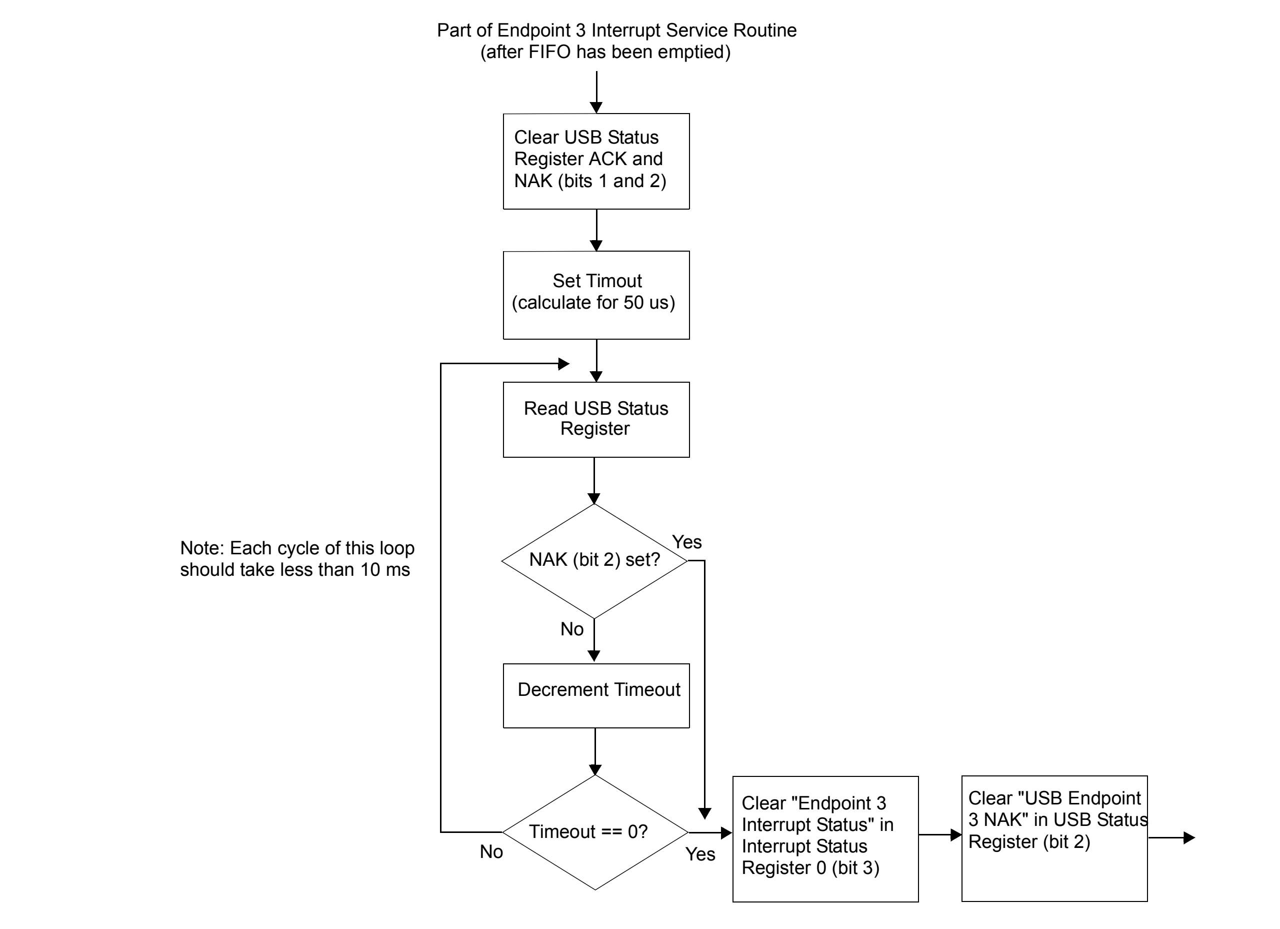

Part of Endpoint 3 Interrupt Service Routine
(after FIFO has been emptied)
Clear USB Status Register ACK and NAK (bits 1 and 2)
Set Timout (calculate for 50 us)
Read USB Status Register
Note: Each cycle of this loop should take less than 10 ms
NAK (bit 2) set?
Yes
No
Decrement Timeout
Timeout == 0?
No
Yes
Clear "Endpoint 3 Interrupt Status" in Interrupt Status Register 0 (bit 3)
Clear "USB Endpoint 3 NAK" in USB Status Register (bit 2)

Figure 10-7: Endpoint 3 Program Flow for Slow CPU

10.4.4 "EP2 Valid Bit" in USB Status can be erroneously set by firmware

"Endpoint 2 Valid" is the only bit in USB Status which is not written as a "Yes/CLR" bit. Therefore, the firmware must do a read-modify-write sequence when clearing other bits in Interrupt Status Register 0 (REG[4004h]), to preserve the state of "Endpoint 2 Valid". However, this read-modify-write could lead to erroneously setting the EP2 Valid bit if the following sequence occurs with "EP2 Valid" set True:

1. Firmware reads Interrupt Status Register 0 to do a read-modify-write
2. Data from EP2 is sent to Host PC, causing S1D13A05 to clear EP2 Valid
3. Firmware writes modified value to Interrupt Status Register 0

In this case, the firmware has set EP2 Valid in Step 3 after it was cleared by the Host PC, erroneously validating EP2 for the next IN token from the Host.

Work Around

First, the firmware should do the read-modify-write operation as described above anytime it is modifying bits in "USB Status".

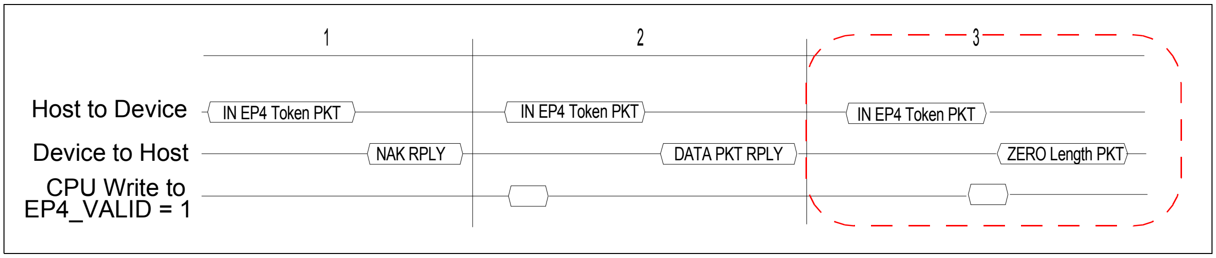
Second, when the firmware recognizes an interrupt for "EP2 Packet Transmitted", it should immediately write a '0' to USB Status Register. This will clear the EP2 Valid bit in the unlikely event that it was erroneously set during a read-modify-write operation.

10.4.5 Setting EP4 FIFO Valid bit while NAKing IN token

Bit 5 of REG[402Ch] indicates to the S1D13A05 controller when data in the endpoint 4 FIFO is ready to be transferred to the host computer. Changing the state of this bit at certain times may generate an error.

When the S1D13A05 USB controller receives an endpoint 4 IN request and endpoint 4 is not ready to transmit data (REG[402Ch] bit 5 = 0), the response is a NAK packet. If endpoint 4 is toggled to a ready to transmit state just before a NAK response packet is sent, the controller may erroneously send a zero length packet instead. When this happens, the data toggle state will be incorrectly set for the next endpoint 4 data transmit.

The following timing diagram shows the error occurring in section 3.



This unexpected occurrence of a zero length packet may cause file system handling errors for some operating systems.

Work Around

There are two software solutions for this occurrence.

Disable USB Receiver before setting the EP4 FIFO Valid bit

The first solution involves disabling the USB receiver to avoid responding to an EP4 IN packet. During the time the USB receiver is disabled the EP4 FIFO Valid bit is set.

When the local CPU is ready to send data on endpoint 4 the steps to follow are:

2. Disable the USB differential input receiver (REG[4040h] bit 6 = 0)
3. Wait a minimum of 1µs. If needed, delays may be added
4. Enable the EP4 FIFO Valid bit (REG[402Ch] bit 5 = 1)
5. Clear the EP4 Interrupt status bit (REG[4004h] bit 4 = 1)
6. Enable the USB differential input receiver (REG[4040h] bit 6 = 1)

Note
Steps 1 through 5 are time critical and must be performed in less than 6 µs.

Note
To comply with “EP4 NAK Status not set correctly in USB Status register”, steps 3 and 4 must be completed within 5 µs of each other. For further information on “EP4 NAK Status not set correctly in USB Status register”, see Section 10.4.1, “EP4 NAK Status not set correctly in USB Status Register” .

EP4 FIFO Valid bit set after NAK and before the next IN token

The second solution is to wait until immediately after the USB has responded to an IN request with a NAK packet before setting the transmit FIFO valid bit. This solution is recommended only for fast processors.

When the local CPU is ready to send data on endpoint 4, it must first detect that a NAK packet has been sent. This is done by reading the EP4 Interrupt Status bit (REG[4004h] bit 4). If the EP4 FIFO Valid bit was not set, the EP4 Interrupt Status bit is set only if a NAK packet has been sent. When the local CPU detects the NAK it must immediately set the EP4 FIFO Valid bit (before responding to the next IN token).

After filling the EP4 FIFO the steps to follow before setting the EP4 FIFO Valid bit are:

1. Clear the EP4 Interrupt Status bit (REG[4004h] bit 4)
2. Read the EP4 Interrupt Status bit (REG[4004h] bit 4) until it is set
3. Set the EP4 FIFO Valid bit (REG[402Ch] bit 5 = 1)

The setting of the EP4 FIFO Valid bit is time critical. The EP4 FIFO Valid bit must be set within 3 µs after the EP4 Interrupt Status has been set internally by the S1D13A05.

11 Hardware Abstraction Layer

11.1 Introduction

The S1D13A05 Hardware Abstraction Layer (HAL) is a collection of routines intended to simplify the programming for the S1D13A05 evaluation board. Programmers can use the HAL to assist in rapid software prototyping for the S1D13A05 evaluation board.

The HAL routines are divided into discrete functional blocks. The functions for startup and clock control offer specific support for the S1D13A05 evaluation board, while other routines demonstrate memory and register access techniques. For a complete list, see Table 11-1:, “HAL Library API” .

11.2 API for the HAL Library

The following table lists the functions provided by the S1D13A05 HAL library.

Table 11-1: HAL Library API

Function	Description
Startup	
halAcquireController	This routine loads the driver required to access the S1D13A05, locates the and returns the address of the controller.
halInitController	Initializes the controller for use. This includes setting the programmable clock and initializing registers as well as setting the lookup table and clearing video memory.
Memory Access	
halReadDisplay8	Reads one byte from display memory
halReadDisplay16	Reads one word from display memory
halReadDisplay32	Reads one double word from display memory
halWriteDisplay8	Writes one byte to display memory
halWriteDisplay16	Writes on word to display memory
halWriteDisplay32	Writes on double word to display memory
Register Access	
halReadReg8	Reads one byte from a control register
halReadReg16	Reads one word from a control register
halReadReg32	Reads one dword from a control register
halWriteReg8	Writes one byte to a control register
halWriteReg16	Writes one word to a control register
halWriteReg32	Writes one dword to a control registers
Clock Support	
halSetClock	Programs the ICD2061A Programmable Clock Generator.
halGetClock	Returns the frequency of the requested ICD2061A clock
Miscellaneous	
halGetVersionInfo	Returns a standardized startup banner message
halGetLastError	Returns the numerical value of the last error and optionally an ASCII string describing the error

Table 11-1: HAL Library API

halInitLUT	This routine sets the LUT to uniform values for color/mono panels at all color depths

11.2.1 Startup Routines

There are two routines dedicated to startup and initializing the S1D13A05. Typically these two functions are the first two HAL routines a program will call.

The startup routines locate the S1D13A05 controller and initialize HAL data structures. As the name suggests, the initialization routine prepares the S1D13A05 for use. Splitting the startup functionality allows programs to start and locate the S1D13A05 but delay or possibly never initialize the controller.

Boolean halAcquireController(UInt32 * pMem, UInt32 * pReg)

Description: This routine initializes data structures and initiates the link between the application software and the hardware. When the S1D13A05 HAL is used this routine must be the first HAL function called.

On PCI platforms, the routine attempts to load the S1D13xxx driver. If the driver loads successfully, then a check is made for the existence of an S1D13A05 evaluation board.

Parameters:

pMem Pointer to an unsigned 32-bit integer which will receive the offset to the first byte of display memory. The offset may be cast to a pointer to access display memory.

pReg Pointer to an unsigned 32-bit integer which will receive the offset to the first byte of register space. The offset may be cast to a pointer and to access S1D13A05 registers.

On Win32 systems the returned offsets correspond to a linear addresses within the callers address space.

Return Value:

TRUE (non-zero) if the routine is able to locate an S1D13A05.
pMem will contain the offset to the first byte of display memory.
pRegs will contain the address of the first 13A05 control register.

FALSE (zero) if an S1D13A05 is not located.
pMem and pRegs will be undefined.
If additional error information is required call halGetLastError().

Note

1. This routine **must** be called before any other HAL routine is called.
2. For programs written for the S1D13A05 evaluation board, an application may call this routine to obtain pointers to the registers and display memory and then perform all S1D13A05 accesses directly.
3. This routine does not modify S1D13A05 registers or memory.

Boolean halInitController(UInt32 Flags)

Description: This routine performs the initialization portion of the startup sequence.
Initialization of the S1D13A05 evaluation board consists of several steps:
- Program the ICD2061A clock generator
- Set the initial state of the control
- Set the LUT to its default value
- Clear video memory

All display memory and nearly every control register can or will be affected by the initialization.

Any, or all, of the initialization steps may be bypassed according to values contained in the Flags parameter. This allows for conditional run-time changes to the initialization.

Parameters: Flags contains initialization specific information. The default action of the HAL is to perform all initialization steps. Flags contain specific instructions for bypassing certain initialization steps. The values for Flags are:

fDONT_SET_CLOCKS
Setting this flag causes initialization to skip programming the ICD2061A clock generator. Normally the clock on the S1D13A05 is programmed to configured values during initialization.

fDONT_INIT_REGS
Bypass register initialization. Normally the initialization process sets the register values to a known state. Setting this flag bypasses this step.

fDONT_INIT_LUT
Bypass look-up table initialization.

fDONT_CLEAR_MEM
The final step of the initialization process is to clear video display memory. Setting this flag will bypass this step.

Return Value:

TRUE	(non-zero) if the initialization was successful.
FALSE	(zero) if the HAL was unable to initialize the S1D13A05 If additional error information is required call halGetLastError()

11.2.2 Memory Access

The S1D13A05 HAL includes six memory access functions. The primary purpose of the memory access functions is to demonstrate how to access display memory using the C programming language. Most programs that need to access memory will bypass the HAL and access memory directly.

UInt8 halReadDisplay8(UInt32 Offset)

Description: Reads and returns the value of one byte of display memory.

Parameters: Offset — A 32 bit offset to the byte to be read from display memory

Return Value: The value of the byte at the requested offset.

UInt16 halReadDisplay16(UInt32 Offset)

Description: Reads and returns the value of one word of display memory.

Parameters: Offset — A 32 bit byte offset to the word to be read from display memory
To prevent system slowdowns and possibly memory faults, Offset should be a word multiple.

Return Value: The value of the word at the requested offset.

UInt32 halReadDisplay32(UInt32 Offset)

Description: Reads and returns the value of one dword of display memory.

Parameters: Offset — A 32 bit byte offset to the dword to be read from display memory.
To prevent system slowdowns and possibly memory faults, Offset should be a dword multiple.

Return Value: The value of the dword at the requested offset.

void halWriteDisplay8(UInt32 Offset, UInt8 Value, UInt32 Count)

Description: Writes a byte into display memory at the requested address.

Parameters:

Offset	A 32 bit byte offset to the byte to be written to display memory.
Value	The byte value to be written to display memory.
Count	The number of times to repeat Value in memory. By including a count (or loop) value this function can efficiently fill display memory.

Return Value: Nothing.

void halWriteDisplay16(UInt32 Offset, UInt16 Value, UInt32 Count)

Description: Writes a word into display memory at the requested offset.

Parameters:

Offset — a 32 bit byte offset to the byte to be written to display memory. To prevent system slowdowns and possibly memory faults, Offset should be a word multiple.

Value — the word value to be written to display memory.

Count — the number of times to repeat the Value in memory. By including a count (or loop) value this function can efficiently fill display memory.

Return Value: Nothing.

void halWriteDisplay32(UInt32 Offset, UInt32 Value, UInt32 Count)

Description: Writes a dword into display memory at the requested offset.

Parameters:

Offset — A 32 bit byte offset to the byte to be written to display memory. To prevent system slowdowns and possibly memory faults, Offset should be a dword multiple.

Value — The dword value to be written to display memory.

Count — The number of times to repeat the Value in memory. By including a count (or loop) value this function can efficiently fill display memory.

Return Value: Nothing.

11.2.3 Register Access

The S1D13A05 HAL includes six register access functions. The primary purpose of the register access functions is to demonstrate how to access the S1D13A05 control registers using the C programming language. Most programs that need to access the registers will bypass the HAL and access the registers directly.

UInt8 halReadReg8(UInt32 Index)

Description: Reads and returns the contents of one byte of an S1D13A05 register at the requested offset. No S1D13A05 registers are changed.

Parameters:

Index — 32 bit offset to the register to read. Index is zero based from the beginning of register address space. (e.g. if Index == 04h then the Memory Clock Configuration register will be read and if Index == 8000h then the BitBLT Control Register will be read)

Return Value: The value read from the register.

Use caution in selecting the index and when interpreting values returned from halReadReg8() to ensure the correct meaning is given to the values. Changing between big endian and little endian will move relative register offsets.

UInt16 halReadReg16(UInt32 Index)

Description: Reads and returns the contents of one word of an S1D13A05 register at the requested offset. No S1D13A05 register are changed.

Parameters: Index — 32 bit offset to the register to read. Index is zero based from the beginning of register address space. (e.g. if Index == 04h then the Memory Clock Configuration register will be read and if Index == 8000h then the BitBLT Control Register will be read)

Return Value: The word value read from the register.

Use caution in determining the index and interpreting the values returned from halReadReg16() to ensure the correct meaning is given to the values. Changing between big and little endian will move relative register offsets resulting in different values.

UInt16 halReadReg32(UInt32 Index)

Description: Reads and returns the dword value of an S1D13A05 register at the requested offset. No S1D13A05 register are changed.

Parameters: Index — 32 bit offset to the register to read. Index is zero based from the beginning of register address space. (e.g. if Index == 04h then the Memory Clock Configuration register will be read and if Index == 8000h then the BitBLT Control Register will be read)

Return Value: The dword value read from the register.

void halWriteReg8(UInt32 Index, UInt8 Value)

Description: Writes an 8 bit value to the register at the requested offset.

Parameters: Index — 32 bit offset to the register to write. Index is zero based from the beginning of register address space. (e.g. if Index == 04h then the Memory Clock Configuration register will be written to and if Index == 8000h then the BitBLT Control Register will be written to)

Value — The byte value to write to the register. Changing between big and little endian will move relative register offsets. Use caution in interpreting the index and values to write to registers using the halWriteReg8() function to ensure that register are programmed correctly.

Return Value: Nothing.

void halWriteReg16(UInt32 Index, UInt16 Value)

Description: Writes a 16 bit value to the S1D13A05 register at the requested offset.

Parameters:

Index 32 bit byte offset to the register to write. Index is zero based from the beginning of register address space. (e.g. if Index == 04h then the Memory Clock Configuration register will be written to and if Index == 8000h then the BitBLT Control Register will be written to)

Value The word value to write to the register.

Return Value: Nothing.

Changing between big and little endian will move relative register offsets. Use caution in interpreting the index and values to write to registers using the halWriteReg8() function to ensure that register are programmed correctly.

void halWriteReg32(UInt32 Index, UInt32 Value)

Description: Writes a 32 bit value (dword) to the register at the requested offset.

Parameters:

Index 32 bit byte offset to the register to write. Index is zero based from the beginning of register address space. (e.g. if Index == 04h then the Memory Clock Configuration register will be written to and if Index == 8000h then the BitBLT Control Register will be written to)

Value The dword value to write to the register.

Return Value: Nothing.

11.2.4 Clock Support

To maximize flexibility, S1D13A05 evaluation boards include a programmable clock. The following HAL routines provide support for the programmable clock.

Boolean halSetClock(UInt32 ClkiFreq, UInt32 Clki2Freq)

Description: This routine program the ICD2061A programmable clock generator to the specified frequency.

Parameters:

ClkiFreq The desired frequency, in Hz, for CLKI.

Clki2Freq The desired frequency, in Hz, for CLKI2.

dwFrequency The desired frequency (in Hz).

Return Value: TRUE (non-zero) if the function was successful in setting the clock.

FALSE (zero) if there was an error detected while trying to set the clock.
If additional error information is required call halGetLastError().

UInt32 halGetClock(CLOCKSELECT Clock)

Description: Returns the frequency of the clock input identified by 'Clock'.

Parameters: Clock Indicates which clock to read. This value can be CLKI or CLKI2.

Return Value: The frequency, in Hz, of the requested clock.

11.2.5 Miscellaneous

The miscellaneous function are an assortment of routines, determined to be beneficial to a number of programs and hence warranted being included in the HAL.

void halGetVersionInfo(const char * szProgName, const char * szDesc, const char * szVersion, char * szRetStr, int nLength)

Description: This routine creates a standardized startup banner by merging program and HAL specific information. The newly formulated string is returned to the calling program for display.

The final formatted string will resemble:

13A05PROGRAM - Internal test and diagnostic program - Build: 1234 [HAL: 1234]
Copyright (c) 2000,2001 Epson Research and Development, Inc.
All Rights Reserved.

Parameters:

szProgName	Pointer to an ASCIIZ string containing the name of the program. (e.g. “PROGRAM”)
szDesc	Pointer to an ASCIIZ string containing a description of what this program is intended to do. (e.g. “Internal test and diagnostic program”)
szVersion	Pointer to an ASCIIZ string containing the build info for this program. This should be the revision info string as updated by VSS. (e.g. “$Revision: 30 $”)
szRetStr	Pointer to a buffer into which the product and version information will be formatted into.
nLength	Total number of bytes in the string pointed to by szRetStr. This function will write nLength or fewer bytes to the buffer pointed to by szRetStr.

Return Value: Nothing.

int halGetLastError(char * ErrMsg, int MaxSize)

Description: This routine retrieves the last error detected by the HAL.

Parameters: ErrMsg When halGetLastError() returns ErrMsg will point to the textual error message. If ErrMsg is NULL then only the error code will be returned.

MaxSize Maximum number of bytes, including the final '\0' that can be placed in the string pointed to by ErrMsg.

Return Value: The numerical value of the internal error number.

HALEXTERN void halInitLUT(void)

Description: To standardize the appearance of test and validation programs, it was decided the HAL would have the ability to set the lookup table to uniform values.

The routine cracks the color depth and display type to determine which LUT values to use and proceeds to write the LUT entries.

Parameters: None

Return Value: Nothing.

12 Sample Code

Example source code demonstrating programming the S1D13A05 using the HAL library is available on the internet at vdc.epson.com.

13 Change Record

X40A-G-003-06 Revision 6.1 - Issued: March 28, 2018

- updated Sales and Technical Support Section
- updated some formatting

X40A-G-003-06 Revision 6.0 - Issued: July 20, 2010

- section 10.4.3 EP3 Interrupt Status bit set by NAKs - correct typo in figure 10-7, Endpoint 3 Program Flow for Slow CPU, change "50 ms" to "50 us"

14 Sales and Technical Support

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

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

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

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