2D Programmer's Guide

2. 2D PROGRAMMER'S GUIDE

2.1 2D Graphics Engine

The 2D graphics engine performs BitBLT (bit block transfer) operations on the frame buffer. During a BitBLT operation, a destination array of pixels in the frame buffer are replaced by a pixel-by-pixel combination of data from a source array of pixels in the frame buffer or host memory. These pixels are also replaced with data from a destination array of pixels in the frame buffer and data from a pattern. This is shown in Figure 2-1.

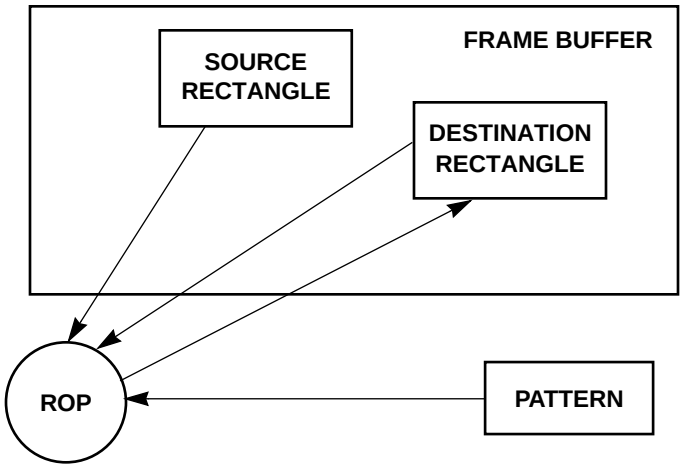

Figure 2-1. 2D Graphics Engine BitBLT

The pixel-by-pixel combination (called raster operation or ROP) is one of the 256 possible combinations of the source, data, and pattern using the NOT, OR, XOR, and AND logical operations. The pattern is an 8×8 color or monochrome pattern, or a solid color.

The 2D engine is programmed by Memory-Mapped registers that define drawing operations and the various parameters required for those operations. As shown in Figure 2-2, it consists of a control unit, a pixel path, and a frame buffer.

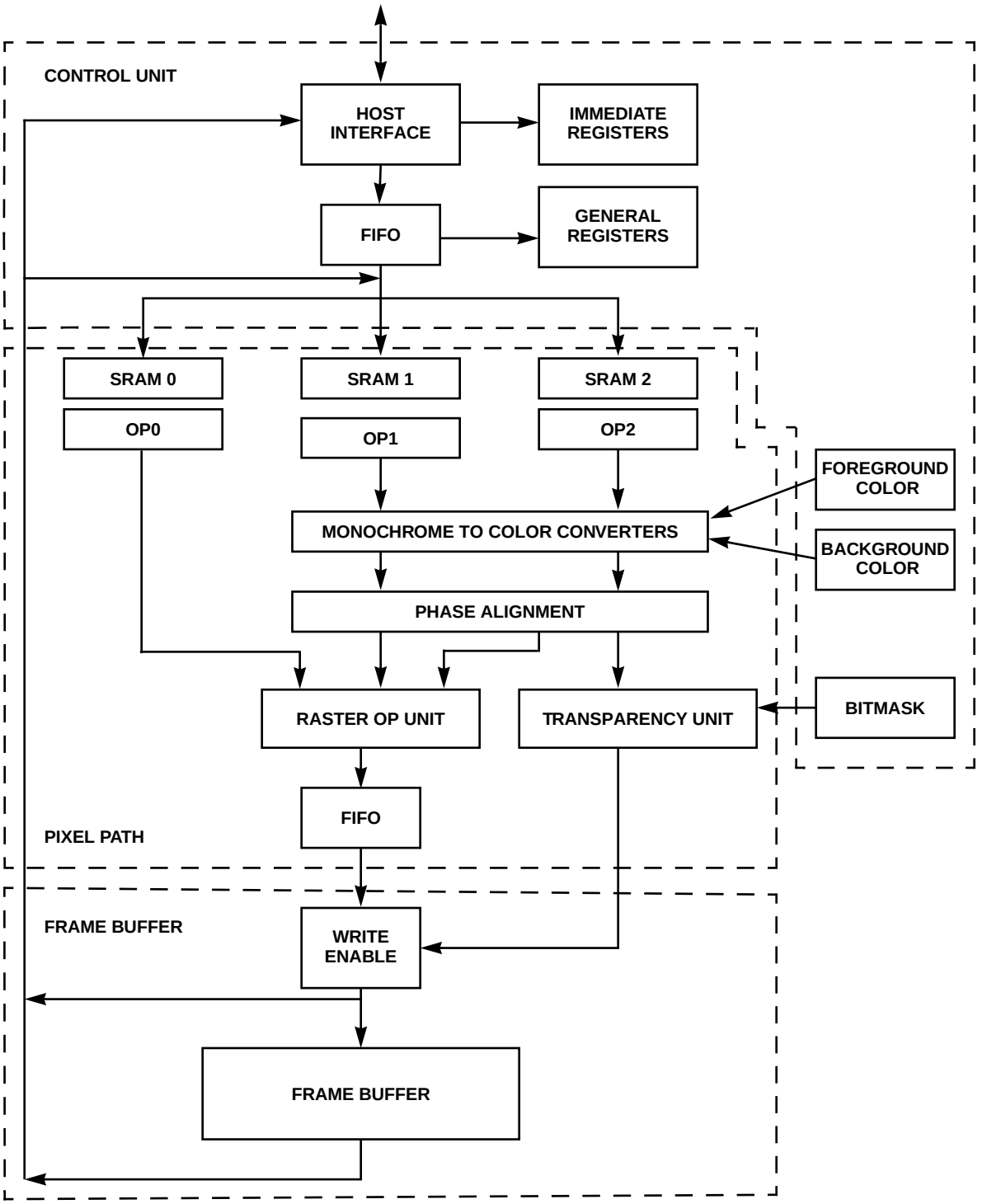

Figure 2-2. 2D Graphics Engine Model and Data Flow

The control unit contains the immediate and general registers, the drawing control and the command/data FIFO. The pixel path contains the three operand fetch units (OFU0, OFU1, OFU2), the ROPs unit, the transparency control, and the pixel FIFO. The frame buffer consists of 1, 2, 4, or 8 Mbytes of Rambus RDRAM memory.

Writes to the immediate registers take effect immediately and do not go through the write FIFO. These are used to read 2D engine status and write general control information. Writes to the general registers are queued through the 25-entry write FIFO and are used to set drawing parameters and initiate drawing operations. During a BitBLT operation, color pixel data is loaded into SRAM0. Color and/or monochrome pixel data is loaded into SRAM1 and SRAM2. Monochrome data is converted to color using the foreground and background color registers. Color pixel data is aligned with the destination. Then the three operands are combined in the ROPs unit to form the output pixel data that can be stored in the frame buffer, sent to the host or stored in one SRAM. If pixel transparency is enabled, SRAM2 is used as the transparency mask. For monochrome masks, the output pixel is written if the corresponding bit in SRAM2 is '1'. For color masks, the output pixel is written if the corresponding pixel in SRAM2 compares with the Background color. (The comparison can be programmed to be 'equal' or 'not-equal'.) SRAM0 is typically the destination operand, SRAM1 is typically the source operand, and SRAM2 is typically the pattern operand.

2.1.1 2D Frame Buffer

The 2D frame buffer is organized as a rectangular array of packed pixels, with pixel '0,0' at the upper left-hand corner and pixel 'xmax,ymax' at the lower right-hand corner. A rectangular portion of the frame buffer (the display rectangle) is visible on the display device. In the upper left-hand corner is pixel 'xs,ys' and in the lower right-hand corner is pixel 'xe,ye' ($0 \le xs < xe \le xmax$, $0 \le ys < ye \le ymax$). The display rectangle is shown in relation to the frame buffer in Figure 2-3. The display rectangle is typically aligned to the upper-left corner of the display buffer (xs = 0, ys = 0), but can be positioned anywhere on the frame buffer surface. Pixel sizes of 8, 16, 24, and 32 bits are supported. Pixel addresses given to the 2D engine are always specified in two dimensional 'x,y' coordinates.

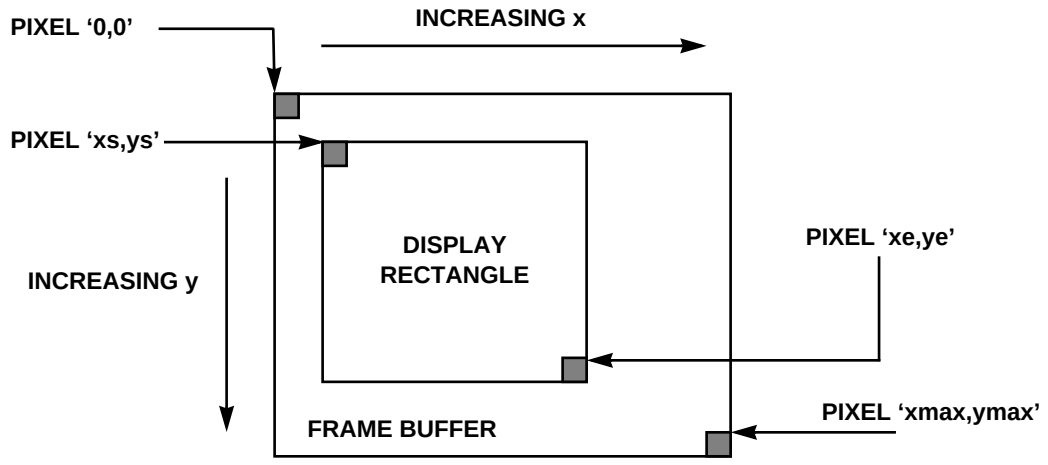

Figure 2-3. 2D Frame Buffer

2.1.2 Bit Swizzle

Bit swizzling is the end for end reversal of bits in a byte. For example, the swizzle of 10001101b is 10110001b. In some cases, monochrome glyphs (fonts or brushes) are presented to the programmer in a format that requires bit swizzle prior to being color expanded into the frame buffer. The CL-GD546X provides a mechanism for implementing bit swizzle during host-to-frame-buffer BLTs. Setting the CONTROL.SWIZ bit during a host-to-frame-buffer BLT causes the bytes within the HOSTDATA dwords to be swizzled. The byte order within the dwords remains the same. If byte-order reversal is also required, write the HOSTDATA through register memory aperture two or three.

2.1.3 Patterns

Patterns can be thought of as an ink-stamp on a roller. As the roller is moved left to right, the image is repeatedly copied onto the destination. At the end of a row, the roller is moved down one image and the process is repeated. Patterns are used when the source for a rectangular block of pixels repeats in x and y. This is achieved by having a small source rectangle repeatedly filling up the destination. When the left end of the pattern is reached during the BitBLT to the destination, the source pointer is reset to the right end of the pattern. When the bottom of the pattern is reached, the source pointer is reset to the top of the pattern. All patterns supported by hardware are 8×8 pixels. The point inside the pattern that is anchored to the upper-left corner of the display can be selected using the PATOFF register.

Conceptually, patterns should be thought of as 8×8 square pixel regions that can be tiled onto the screen. The upper-left corner of the pattern tile is aligned to the upper-left corner (0,0) of the frame buffer. The alignment is adjustable to any point within the tile by setting the x, y values in the PATOFF register.

Patterns are not stored in rectangular format in the frame buffer. They are stored linearly. A monochrome (1 bpp) pattern is stored in a single qword (64 bits). An 8-bpp color pattern is stored in 64 sequential bytes. A 16-bpp color pattern is stored in 128 sequential bytes. 24-bpp and 32-bpp color patterns are stored on two adjacent lines with half the pattern on each line. The first line contains the top four lines of the pattern and the second line contains the bottom four lines. The first line of a 24- or 32-bpp color pattern must be on an even scanline address in y.

For optimal performance, the software should align patterns to tiles. Crossing tile boundaries puts a penalty on memory performance. The architecture allows any pattern to fit within a single tile and the programmer concerned with optimal performance is encouraged to respect this constraint. See Chapter 5, "System Operation" for more information.

2.1.4 Monochrome-to-Color Expansion

Monochrome bitmaps can be converted to foreground and background colors, or to all ones and all zeroes (white on black). Strings of ones and zeroes are fed into the ROP unit and are output respectively as foreground color and background color. Source (OP1) or pattern (OP2) operands can convert monochrome pixel data to color. Typical use of monochrome-to-color conversion is for transferring font maps to characters on the screen, or for hatched brushes.

Foreground and background colors are selected by placing the appropriate values in the OP_opFGCOLOR and OP_opBGCOLOR registers and turning off DRAWDEF.SATn. With DRAWDEF.SAT.n on, the foreground color is all ones (–1) and the background is all zeroes (typically white on black). Refer to the *Laguna VisualMedia™ Accelerators Family — CL-GD546X Volume I (Hardware Reference Manual, Second Edition, September 1996)* for register information.

2.1.5 Transparency

Two types of transparencies can be generated; one using a monochrome input data stream, and another using a Color Comparison register. When transparency is turned on, OP2 fetches transparency mask pixels and makes the decision whether or not to enable writes to the destination based on the compare operation. OP2 can still be used in the raster operation.

Monochrome transparency causes the pixels mapped from a one to be written with the result of the current raster operation. This causes pixels mapped from a zero to retain their prior value. A typical use of this type of operation puts foreground solid colored fonts over an arbitrary existing background.

Color transparency compares an incoming data stream with a fixed OP_opBGCOLOR value on a pixel-by-pixel basis. It then writes the ROP result or retains the destination pixel based on the result of the comparison. Transparency masks can be aligned with source data, destination data, or any other region. Source-aligned color transparency allows the programmer to simulate chroma keying (also known as blue screening).

2.2 2D Graphics BitBLT Operations

This section guides the system programmer in the most effective use of the CL-GD546X 2D graphics engine for implementing display drivers, and special graphics and video application software. Methods for implementing typical operations are discussed and supplemented with tested examples.

2.2.1 Commonly Used BitBLT Control Registers

Table 2-1 illustrates the fields in the three most-used 2D Engine Control registers in the CL-GD546X.

Table 2-1. Primary 2D Engine Control Registers

BLTDEF

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
YDir	RESult			OP1= OP2			OP0, DST	OP1, SRC				OP2, PAT			
Dn =0 Up =1	FB =1 HO =2 SMO =4 SM1 =5 SM2 =6 SM1:2 =7			Off =0 On =1			SM =0 FB =1	Pattern Off =0 On =1	Color =0 Mono =1 Fill =1	SM =0 FB =1 HO =2 Fill =3		Pattern Off =0 On =1	Color=0 Mono=1 Fill =1	SM =0 FB =1 HO =2 Fill =3	

DRAWDEF

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Mono Saturate					Pixel Tag	Transparency		ROP							
OP1 Off =0 On =1	OP2 Off =0 On =1				GRX=0 Video=1	Opera-tion BG(=) =0 FG(!=) =1	Switch Off =0 On =1	Raster Operation Code 00.FF							

LNCNTL

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
		YUV 4:1:1 Average Con-trol				Chain	Auto		Graphics Pixel Format			Y_SH RINK	X_SH RINK	Y_INT ERPO-LATE	X_INT ERPO-LATE
		UV HOLD Off =0 On =1	LOW PASS Off =0 On =1			Off =0 On =1	BLT=0 Resize =1		8 bpp CLUT =0 16 bpp 1:5:5:5 =1 16 bpp 5:6:5 =2 YUV 4:2:2 =3 32 bpp aRGB = 4 24 bpp RGB = 5 AccuPak = 6 Reserved = 7			Shrink =1 Stretch = 0	Shrink =1 Stretch =0	Off =0 On =1	Off = 0 On =1

2.2.2 BitBLT Programming Overview

Standard BitBLT operations are used for moving rectangular blocks of pixels from one location to another. Source data comes from the Host, Frame Buffer, or operand SRAM caches. Independent of pixel size, the result data is written to the Host, SRAM, or the Frame Buffer as color (8, 16, 24, 32 bpp) data or byte data. Source and destination extents are the same with the exception of pattern sources, which are cyclic on eight pixel by eight-line boundaries. Stretch and shrink BitBLTs do not have the same source and destination extents. All 256 three operand raster operations are performed on any combination of the three source locations. Monochrome data, color data and transparency masking can all be combined in a single BitBLT operation. BitBLT programmers should carefully read the detailed descriptions of the 2D Graphics Engine registers, refer to the *Laguna VisualMedia™ Accelerators Family — CL-GD546X Volume I (Hardware Reference Manual, Second Edition, September 1996)* for register information.

Most graphics registers, when written in the CL-GD546X, are posted through the 2D engine command queue, which has an effective depth of 25-dword entries.

In PCI systems, checking the queue depth is not required since the bus architecture supports retries and waits while the queue is full.

Immediate registers are not posted through the queue and checking QFREE is never required. These registers are typically device initialization registers that are not used during a BitBLT operation. Each register write consumes one, two, or three entries in the queue. Each write uses one entry per 16 bits (word), plus one extra entry if it is a command.

BLTEXT registers consume three entries: two for the X/Y, and one for the implied command. Immediate registers bypass the queuing mechanism and consume no entries. The programmer does not have to check that a BitBLT operation is completed before programming the next operation. This is due to the queueing mechanism and the double buffering of internal registers in the CL-GD546X. The queue allows host and BitBLT operations to execute in parallel, increasing the overall system throughput.

Typically, a register is written as a 32-bit dword or portions of it are written as 16-bit WORDs. The programmer can choose to write 16-bit halves of the register when it is the only portion of an X/Y address or extent that is changing between operations in an inner loop. This is a convenient optimization for text operations when the Y extent remains constant for a long sequence of operations.

In addition to triggering the operation, BLTEXT registers set the extents of the BitBLT. Writing to one 16-bit half of the BLTEXT register sets an X or Y value. Writing to the other 16-bit half sets the corresponding Y or X value and starts the BLT operation. Be careful to write these register halves in the correct order so that the BLT is started by the second write, not the first. The register descriptions for these registers indicate which half of the register is the triggering write. BLTEXT registers are provided for initiation of the operation on X write (_XEX) or on the Y write (_EX).

Prior to each BitBLT operation, several registers are set up. The four most important registers are BITMASK, BLTDEF, LNCNTL, and DRAWDEF. The bit assignments of the BLTDEF, LNCNTL, and DRAWDEF registers appear in Table 2-1. Incorrect setup of these registers accounts for a large share of the problems that a programmer encounters. After these four registers are set up, the OP_opRDRAM Pointer registers are set up to point to source and destination operands and result locations. Following initialization of the Control registers, a 'command' BLTEXT register is written. It contains the destination X and Y extents and causes the desired operation to proceed.

The first register to consider when setting up for a BitBLT is the BITMASK register. It should be set to FFFFFFFFh to enable writing to all bits. Each bit in the BITMASK enables (1) or masks (0) the write to the corresponding bit that it aligns with during the memory write. All 32 bits in the BITMASK register are used, regardless of the width of pixels in the frame buffer. In 8 bits-per-pixel frame buffers, the 8-bit BITMASK must appear four times throughout the entire 32-bit register. In 16-bit systems, it appears twice. In 32-bit systems, one copy of the mask fills the 32-bit register. In 24 bits-per-pixel systems, BITMASK cycles in and out of alignment if any value other than FFFFFFFFh is used. This is because BITMASK is effectively dword aligned. Many drivers can be written to set BITMASK to FFFFFFFFh at initialization and do not need to be changed again.

The next register to set is the BLTDEF register. It has fields that specify the fetching properties of the Operand Fetch Units. The OP1 and OP2 fields are set to fetch data from the frame buffer, the host, or SRAM. Additionally, the fetch field can be set up for a 'fill' fetch, which always supplies the background color. Color or monochrome source data and pattern fetching is selected in the OP1 and OP2 fields. The OP0 fetch is selected from either SRAM or the frame buffer. The result of the operation is then selected by the RES field and can be set to the frame buffer, host, or SRAM. SRAM destinations are SRAM0, SRAM1, SRAM2, or SRAM1 and SRAM2 combined. Next, the Y direction for BitBLTs is set to either up the screen (decreasing Y) or down the screen (increasing Y). Finally, the BD_Same field should be set or cleared.

Setting BD_Same (bit 11) causes OP1 and OP2 data to be fetched based solely on OP2 operand pointers. OP1 X registers need to be set to the same value as OP2 X registers in this mode. When OP1 and OP2 use the same data, it is a useful optimization for reducing the amount of data that needs to be fetched for an operation. For example, host-to-frame-buffer transparent text, with identical font and transparency masks using OP1 for monochrome-to-color expansion and OP2 for transparency control, is an excellent use of BD_Same.

Overlapping BitBLTs that move down (increasing Y) the screen are performed with BLTDEF.YDIR set to up, and with the source and destination pointers anchored to the lower left corners of the rectangles. Overlapping BitBLTs that move up the screen are performed with BLTDEF.YDIR set to down, and with the source and destination pointers anchored to the upper left corners. In addition, the case of a purely horizontal, left to right BitBLT must be considered. Unless it is broken up into strips or cached through an intermediate buffer by software, this BitBLT results in vertical stripes.

The next register to set up is the DRAWDEF register. The three operand raster operation is set in the ROP field. Transparency (using OP2) is enabled by setting the transparency switch on and the type. The compare operation (foreground or background) is selected by the transparency operation field. The Pixel Tag field is used in 9-bit RDRAM systems to control the setting or clearing of the ninth bit during the BitBLT. If set, The ninth bit indicates to the back-end that the associated pixel is video and must be interpreted using the video format and depth fields. Finally, the monochrome saturate fields must be set or cleared. If OP1 and/or OP2 are fetching monochrome data and saturate is on, then ones are converted to all ones (FF, FFFF, FFFFFFFF) and zeroes are expanded to zeroes. If saturate is off, then monochrome data is converted to the foreground and background colors.

To perform a logical ternary raster operation (ROP), the DRAWDEF ROP field must be set to the 8-bit ROP code. This is provided by the application or some layer of software above the driver. The ROP code is computed by applying the desired logical operations to the operand constants, which are OP0 = AAh, OP1 = CCh, and OP2 = F0h. For example, OP1 data is copied to the result using code CCh, or OP0 data is ANDed to OP1 data using a ROP code of (OP0 and OP1) = (AAh and CCh) = 88h. A more complex example is a 'patterned stencil BitBLT' in which a pattern is copied to a rectangular region in the frame buffer through a monochrome stencil bitmap. Let the zeroes

in the stencil represent the stencil holes through which the pattern is applied, and let ones represent solid areas. This prevents the underlying data in the frame buffer from being changed. Make OP0 the existing destination pixel, make OP1 the monochrome stencil source, and make OP2 the pattern source. Where OP1 is a one, the result bit is taken from OP0. Where OP1 is a zero, the result is taken from OP2. In this example, the ROP code is computed as listed below.

```
OP1         CC          11001100              ones select OP0,
                                                zeroes select OP2
OP0         AA          10101010
OP2         F0          11110000
---         --          --------
RES         B8          10111000
```

ROP code B8 is represented in reverse polish Boolean as PSDPxax and in algebraic notation as PAT^SRC&(DST^PAT), where P = PAT = OP2, S = SRC = OP1, and D = DST = OP0.

The DRAWDEF and BLTDEF registers are written as a single 32-bit operation or as two 16-bit operations. Typically, the programmer composes the contents of a DRAWBLTDEF register setting for a given operation by combining various bitmap macros at compile time, or by masking the registers at run time to specify the BitBLT operation. Then, the programmer moves the 32-bit dword to the hardware. Some optimization of the driver is realized by leaving the DRAWDEF and BLTDEF registers in a standard configuration at the end of every operation. This allows subsequent operations to skip setting these registers. The most common setting for these registers is often frame-buffer-to-frame-buffer source copy using color source and destination data types.

If the BitBLT is doing monochrome-to-color conversion, set the Foreground and Background Color registers must be set. In 8 and 16 bits-per-pixel, the colors are replicated throughout the register. An 8-bit-per-pixel color of 37h is loaded as 37373737h, while a 16-bit-per-pixel color of 6A7Fh is loaded as 6A7F6A7Fh.

The LNCNTL register, for normal BitBLTs, should have the auto field set to '0', and the chain field set to off. If the frame buffer contains mixed-pixel formats, set the graphics pixel format field. The LNCNTL register is discussed in detail in the Chapter 4, "Video Programming".

After BITMASK, BLTDEF, DRAWDEF, and LNCNTL registers are set up, the operand pointers for fetching and storing the data need to be set up. If OP1 or OP2 are fetching monochrome data from the frame buffer, then OP{1|2}.opMRDRAM is used to point to the source data in the frame buffer. The Y portion (upper 16 bits) of the register contains the line-number index and the X portion (lower 16 bits) contains the bit index to the monochrome data. For instance, a monochrome glyph cached at byte 3 on the second line has a Y value of '1' and a X value of '24' loaded into the OP pointer. If monochrome data is fetched from SRAM, then the OP{1|2}_opMSRAM pointers should point to the bit offset in the appropriate SRAM. Monochrome data, fetched from SRAM, wraps around at the end of SRAM and continues fetching at the beginning. If monochrome data is fetched from the host, the phase alignment of the data (within the dwords fetched from the host), is indicated by the value programmed into the OP{1|2}_opMRDRAM register. For this operation, these registers serve the secondary purpose of phase control, which is not clearly indicated by their name.

Color data from the frame buffer is pointed to by the OP{0|1|2}_opRDRAM registers. Color data from the SRAM is pointed to by the OP{0|1|2}_opSRAM registers. This data is cyclic in that at the end of SRAM, the internal pointers cycle to the beginning of SRAM. Color data from the host is phase aligned by using the OP{1|2}_opRDRAM registers in a manner similar to the monochrome case. The registers point to the byte alignment (0, 1, 2, 3) within the dwords loaded from the host.

The OP0_opRDRAM register points to the result X/Y location in the frame buffer. The OP0_opRDRAM.pt.X register points to the result offset in SRAM. OP0_opRDRAM.pt.X indexes into the particular SRAM cache selected in the BLTDEF result field.

Operand Pointer registers do not read back with the same values that are written to them. The programmer must not make the assumption that they do. Color RDRAM pointers are byte converted on write and read back as byte offsets. Monochrome pointers (MRDRAM and MSRAM) read back as monochrome pointers. MRDRAM pointers are interpreted by the hardware as byte pointers for color operands and as bit pointers for monochrome operands. The Y part of pointers undergoes no conversion (reference the Graphics Accelerator Registers Chapter 10 for a detailed description).

Pattern data is always 8 lines by 8 pixels and anchored to the frame buffer (0,0) location. If OP1 and/or OP2 fetch patterned data, then the PATOFF register is set to align the pattern tile to the upper-left hand corner of the frame buffer. PATOFF contains X and Y fields that index into the pattern and define the point of alignment. Once a pattern alignment is set, all subsequent pattern BitBLTs are mutually aligned within the frame buffer. PATOFF applies to OP1 and OP2 concurrently. If two patterns are used and the programmer needs different relative alignments, then one pattern must be rotated in software prior to caching.

2.2.3 Monochrome-to-Color Conversion BitBLTs

Monochrome bitmaps consisting of one bit for each pixel are converted to foreground and background colors, or to all zeroes and all ones (black and white) during a monochrome BitBLT operation. Strings of ones and zeroes are fed into the graphics accelerator 2D engine and are output respectively as foreground color and background color. Source (OP1) or Pattern (OP2) operands fetch and convert monochrome bit streams to color. Typical use of monochrome-to-color conversion transforms font maps to characters on the screen or for painting two-color hatched brushes.

The saturate fields in the DRAWDEF register are used to select monochrome-to-color conversion (saturate off), or to select monochrome conversion to all ones and all zeroes (saturate on). Foreground and background colors for color converted bitmaps are selected by placing the appropriate values in the OP_opFGCOLOR and OP_opBGCOLOR registers and turning DRAWDEF.SATn off. Color values in the dword FGCOLOR and BGCOLOR registers are expanded to fill the full 32 bits in 8- and 16-bit cases. In a 24-bits-per-pixel case, the color value is loaded into the lower 24 bits of the register.

Monochrome-to-color conversion BitBLTs are sourced from the host, frame buffer or SRAM, and can use host, frame buffer, or SRAM as the result. Monochrome BitBLTs are combined with transparent BitBLTs to produce foreground only fonts. Monochrome data can be stretched in a two-step operation by first performing a monochrome saturate BitBLT, and then performing a replicate stretch on the result. Monochrome data is used as pattern source data, making for very efficient storage of two-color brushes or hatching patterns.

2.2.4 Transparent BitBLTs

Both monochrome and color bitmaps are used as transparent masks. The OP2 operand fetch unit fetches data and performs the transparent compare operation. It enables or masks write operations to the frame buffer based on the compare result.

Monochrome transparency operations are performed by enabling transparency and setting the transparent operation to foreground or background in DRAWDEF. Zero (background) bits in the monochrome mask suppress writes to their associated pixels in the frame buffer. Ones enable writes to the frame buffer, if monochrome foreground transparency is selected. If background transparency is selected, ones in the monochrome mask suppress writes to their associated pixels in the frame buffer and zeroes enable writes. The FGCOLOR and BGCOLOR registers do not need to be set as a part of monochrome transparency setup.

Color transparency is implemented by pointing OP2 to a color region, setting BGCOLOR to the transparent compare color, and selecting equal or not equal as the transparent compare operation. A typical operation is chroma keying or blue-screening where picture is rendered on top of a constant colored background. The constant background color is put into the BGCOLOR register as the compare color. The operation is then set to equal. If an OP2_Copy ROP (0F0h, PATCOPY) is selected and transparency is enabled, then the picture on the 'blue-screen' background copies to the destination region of the screen while the background is untouched. This is useful for animation provided that a save-under and restore operation are associated with the transparent BitBLT.

Operand fetch unit one fetches data written to the frame buffer. Operand fetch unit two selects data from a distinctly different monochrome transparency mask located elsewhere in off-screen memory. This method is used to mask video or graphics data into an arbitrarily shaped region. Monochrome bitmasking is used to implement occlusion of normally rectangular video regions by associating an equal-sized bitmask with the region, and enabling or disabling writes to the occluded video region by careful manipulation of the transparency bitmask.

Transparent BitBLTs are combined with color or monochrome patterned data and are used with any of the 256 available ROPs. Align off-screen transparent font caches with memory to minimize tile boundary crossings to minimize RDRAM page breaks and to optimize fetching performance.

2.2.5 Pattern BitBLTs

Patterns are used when the source for a rectangular block of pixels repeats in X and Y. This is produced by having a small source rectangle that is used repetitively to fill up the destination. When the left end of the pattern is reached during the BitBLT to the destination, the source pointer is reset to the right end of the pattern. When the bottom of the pattern is reached, the source pointer is reset to the top of the pattern. Patterns are 8×8 square pixel regions that are tiled onto the screen. The upper-left corner of the pattern tile is aligned to the upper-left corner (0,0) of the frame buffer. The alignment is adjusted to any point within the tile. This is done by setting the X, Y values in the PATOFF register. Pattern data can be monochrome or color sourced from the host.

Align dword for patterns caches stored in off-screen frame buffer memory. Pattern loading from the host is simplified by using the bpp independent MBitBLT, which is discussed later in this section.

Patterns are not stored in rectangular format in the frame buffer, they are stored linearly. A monochrome (1 bpp) pattern is stored in a single qword (64 bits). An 8-bpp color pattern is stored in 64 sequential bytes on a single line. A 16-bpp color pattern is stored in 128 sequential bytes, also on a single line. 24-bpp and 32-bpp color patterns are stored on two adjacent lines with half the pattern on each line. The even line contains the top half of the pattern. The next odd line contains the bottom half. Monochrome pattern maps are converted to foreground and background color. Otherwise they are saturated to black and white depending on the settings of the DRAWDEF.Saturate fields.

Both operand one (OP1) and operand two (OP2) can fetch pattern data from any source. OP2 can additionally perform transparent compare operations on pattern data.

For optimal performance, patterns should be tile-aligned. Crossing tile boundaries puts a penalty on memory performance by causing RDRAM page breaks. The architecture allows any pattern to fit within a single tile and the programmer, striving for optimal performance, is encouraged to take advantage of this feature. Monochrome, eight and sixteen bits-per-pixel patterns fit in a single SRAM cache, whereas 24- and 32-bits-per-pixel patterns are fetched multiple times yielding lower performance due to repeated pattern fetching.

2.2.6 Host BitBLTs

The BitBLT engine can source data from the host and write data to the host. Once a host BitBLT is initiated, host data must be written or read under program control, since it is not a bus master device.

Host BitBLTs are phase-aligned so that the fetched or stored data is dword-aligned. Since aligned X86-dword fetches are executed more quickly than non-aligned fetches, alignment yields higher performance. Given a non-aligned pointer to data on the host, the programmer can choose the next lower dword aligned address and use the OP{0|1|2}_opMRDRAM registers to select the byte offset (phase) within the dword where the first pixel begins. The OP{1|2}_opMRDRAM registers are used for host sourced BitBLTs to control the phase alignment of data within the dwords written to HOSTDATA, following the BLTEXT. The programmer should set OP{1|2}_opMRDRAM.pt.X to (0, 1, 2, 3) to select the proper phase of alignment for host-sourced BitBLTs. Host-destination BitBLTs operate in a similar manner, giving the programmer the ability to select the phase within qwords in the frame buffer. The number of host destination dwords is always a multiple of two. The CL-GD546X only supplies qwords for host destination BitBLTs. The programmer should set OP0_opMRDRAM.pt.X to (0, 1, 2, 3) to select the proper phase alignment for host destination BitBLTs. Phase-aligning host BitBLTs is strictly a performance enhancement. If it is not chosen, exercise caution and set the phase pointers to zero. This is important since expected data may not be aligned, and the number of dwords to supply may be calculated incorrectly.

BitBLTs to and from the host are performed by setting up the OP pointers, the phasing pointers described earlier, the Control registers (BLTDEF, DRAWDEF), and commanding the BitBLT by the BLTEXT registers. Care must be taken in properly calculating the number of HOSTDATA dwords to read or write. The general formula for the number of HOSTDATA_dword writes is:

$$H \times \left(\frac{W \times BytesPerPixel + 3 + PhaseInBytes}{4} \right) \qquad \textbf{Equation 2-1}$$

The general formula for the number of HOSTDATA_dword reads is:

$$H \times \left(\frac{W \times BytesPerPixel + 7 + PhaseInBytes}{8} \right) \qquad \textbf{Equation 2-2}$$

where

H	is the height in rows (scanlines) of the BitBLT.
W	is the width in pixels of each row and Bytes PerPixel = 1 (8 bpp), 2 (16 bpp), 3 (24 bpp), and 4 (32 bpp).

In 32-bpp modes only, the above formula can be simplified to H $\times$ W.

The RDQUEUE field in the STATUS register is a '1', if data is available for the host to read. The Swizzle bit in the CONTROL register is set to reverse the bit order within bytes during Host BitBLTs. This is useful for reversing the direction of font maps that are provided 'backwards' to the software. Be careful not to upset the state of other bits in the CONTROL register when programming the Swizzle bit.

2.2.7 Byte BitBLTs (MBitBLTs)

Byte BitBLTs are available at all pixel depths to simplify off-screen memory management, making large parts of typical drivers bit-per-pixel independent. Byte BitBLTs are also called monochrome BitBLTs or MBitBLTs, although they are not monochrome BitBLTs. Without MBitBLTs, the requirement to dword-align patterns in off-screen memory (while in packed 24-bpp formats); would be problematic since not all pixel X pointers point to dword-aligned boundaries.

To perform a MBitBLT, set up the BLTDEF and DRAWDEF registers as usual. Next, set up the OP{0|1|2}_opMRDRAM pointers with byte offsets. Finally, command the BitBLT extents with MBLTEXT registers. The MONOQW register must be set to the number of qwords that encompass the X extent of monochrome data (MONOQW = (Number_of_Bytes + 8) / 8).

2.3 Tips and Tricks

This section helps the programmer avoid common pitfalls and increase performance while programming the CL-GD546X.

- Align patterns on tile boundaries for optimal memory bandwidth use. Patterns must be aligned on word boundaries.
- Minimize the number of tile boundaries crossed in a given operation for optimal memory bandwidth use.
- Many software errors can be traced to improper setup of four registers: BITMASK, BLTDEF, DRAWDEF, and LNCNTL. These registers should be set up carefully.
- In some cases it may be optimal to verify that the write queue contains sufficient entries by reading QFREE prior to a macro operation. On the PCI bus, this only reduces retries when other devices are contending for the bus.
- Where multiple registers contain the same information, be certain that the Graphics Accelerator and Display Controller register settings match for any given mode.
- Operations fetching from SRAM are faster than operations fetching from RDRAM. If patterns are used several times in succession, cache them in SRAM.
- Be careful to feed HOSTDATA the proper number of dwords when doing a host-sourced BitBLT. Also, when doing a host-destination BitBLT (refer to Section 2.2.6), be careful to read the proper number of qwords.
- Set OP(1,2)_opRDRAM.pt.X to (0, 1, 2, 3) to select the proper phase of alignment for host-sourced BitBLTs.
- Set OP0_opRDRAM.pt.X to (0, 1, 2,..., 7) to select the proper phase alignment for host-destination BitBLTs and for SRAM result BitBLTs.
- SRAM pointers that are not qword aligned causes the other bytes in the containing SRAM qword to be written.

2.4 BitBLT Programming Examples

Several BitBLT programming examples are provided in the following sections. These examples contain the name of the register to be written, followed by the value to be written. Values are in decimal or hex. Register names follow the coding practice found in the LGREGS.H file and the register chapters of this manual.

Reading the Programming Examples

When reading or using the BitBLT programming examples, follow the instructions at the beginning of this chapter. HOSTDATA indicate data being written to the HOSTDATA port from the system CPU during a host to frame buffer or SRAM BitBLT. Following the BLTEXT command in these cases, HOSTDATA lines are provided with the proper number of dwords. One or more dwords are appended to a single HOSTDATA line. All HOSTDATA writes are 32-bit writes regardless of the numeric format in the example. Another special case is READHOSTDATA followed by the size (byte, word, dword) to be read back from a frame buffer or SRAM during the host BitBLT.

2.4.1 Software Cursor Programming Example

The CL-GD546X supports a 64 × 64 bit-mapped hardware cursor. The following example is for illustrative purposes only. It provides a method of software implementation for generating cursor-like objects of arbitrary size, color, and transparency properties.

A monochrome transparency mask is combined with a color cursor map. The color cursor is in the shape of a picture frame. The monochrome transparency mask is the frame's interior region. This type of operation could be used in a game for a target sighting device. The cursor color and transparency maps are loaded into off screen memory. The cursor area is then saved to off-screen memory and the cursor is painted. The off-screen save area is copied back to the cursor location. The cursor is moved and the cycle repeats. The example below shows the first two cycles of moving the cursor diagonally from upper left towards lower right. The delta X and delta Y movements are set to '1'. Redundant register writes are eliminated from the example.

```
# load 16 pixel x 16 line color cursor map from host to frame buffer.
BITMASK                  0xFFFFFFFF  # write mask -> all enable
BLTDEF                   0x1020      # op1_is_host, res_is_rdram
DRAWDEF                  0x00CC      # rop_op1_copy
CONTROL                  0x0400      # Swizzle on (Reverses Bit Order
                                     # with in HOSTDATA BYTES). Note that
                                     # this is 8 Bits-per-pixel.
OP0_opRDRAM.pt.X 0                   # BLT -> 0,256 off-screen memory
OP0_opRDRAM.pt.Y         256
OP1_opRDRAM.pt.X         0           # Set Host Data Transfer phase of 0
BLTEXT_EX.pt.X           16          # 16 pixel by 16 line color map
BLTEXT_EX.pt.Y           16
HOSTDATA                 0x52525252 0x52525252 0xAAAAAAAA 0xAAAAAAAA # line 1
HOSTDATA                 0x52525252 0x52525252 0xAAAAAAAA 0xAAAAAAAA # line 2
HOSTDATA                 0x00005252 0x00000000 0x00000000 0xAAAA0000 # line 3
HOSTDATA                 0x00005252 0x00000000 0x00000000 0xAAAA0000 # line 4
HOSTDATA                 0x00005252 0x00000000 0x00000000 0xAAAA0000 # line 5
HOSTDATA                 0x00005252 0x00000000 0x00000000 0xAAAA0000 # line 6
HOSTDATA                 0x00005252 0x00000000 0x00000000 0xAAAA0000 # line 7
HOSTDATA                 0x00005252 0x00000000 0x00000000 0xAAAA0000 # line 8
HOSTDATA                 0x00004444 0x00000000 0x00000000 0xFFFF0000 # line 9
HOSTDATA                 0x00004444 0x00000000 0x00000000 0xFFFF0000 # 10
HOSTDATA                 0x00004444 0x00000000 0x00000000 0xFFFF0000 # 11
HOSTDATA                 0x00004444 0x00000000 0x00000000 0xFFFF0000 # 12
HOSTDATA                 0x00004444 0x00000000 0x00000000 0xFFFF0000 # 13
HOSTDATA                 0x00004444 0x00000000 0x00000000 0xFFFF0000 # 14
HOSTDATA                 0x44444444 0x44444444 0xFFFFFFFF 0xFFFFFFFF # 15
HOSTDATA                 0x44444444 0x44444444 0xFFFFFFFF 0xFFFFFFFF # 16
#
# load a 16 bit x 16 line mono transparency map from host to frame # buffer.
```



```
OP0_opRDRAM.pt.X 16                 # BLT->16,256 BLT to off-screen memory
OP0_opRDRAM.pt.Y         256
BLTEXT_EX.pt.X           2          # mask width (in bytes)
BLTEXT_EX.pt.Y           16         # mask length (in lines)
HOSTDATA                 0xFFFF     # line 1 XXXXXXXXXXXXXXXX
HOSTDATA                 0xFFFF     # line 2 XXXXXXXXXXXXXXXX
HOSTDATA                 0x03c0     # line 3 XX            XX
HOSTDATA                 0x03c0     # line 4 XX            XX
HOSTDATA                 0x03c0     # line 5 XX            XX
HOSTDATA                 0x03c0     # line 6 XX            XX
HOSTDATA                 0x03c0     # line 7 XX            XX
HOSTDATA                 0x03c0     # line 8 XX            XX
HOSTDATA                 0x03c0     # line 9 XX            XX
HOSTDATA                 0x03c0     # line 10 XX            XX
HOSTDATA                 0x03c0     # line 11 XX            XX
HOSTDATA                 0x03c0     # line 12 XX            XX
HOSTDATA                 0x03c0     # line 13 XX            XX
HOSTDATA                 0x03c0     # line 14 XX            XX
HOSTDATA                 0xFFFF     # line 15 XXXXXXXXXXXXXXXX
HOSTDATA                 0xFFFF     # line 16 XXXXXXXXXXXXXXXX
# Setup FG & BG color for transparency engine.
# point to mono mask
OP2_opmRDRAM.pt.X 128
OP2_opmRDRAM.pt.Y 256
# BLT X extent is invariant in example.
BLTEXT_EX.pt.X                      16

#begin cursor movement loop
#SaveUnder from first location.
BLTDEF                              0x1111
DRAWDEF                             0x00CC
OP0_opRDRAM.pt.X 24
OP0_opRDRAM.pt.Y 256
OP1_opRDRAM.pt.X 0
OP1_opRDRAM.pt.Y 0
BLTEXT_EX.pt.Y                      16

#Paint Cursor.
BLTDEF                              0x1015
DRAWDEF                             0x01CC
OP0_opRDRAM.pt.X 0
OP0_opRDRAM.pt.Y 0
```

```
OP1_opRDRAM.pt.X 0
OP1_opRDRAM.pt.Y 256
BLTEXT_EX.pt.Y                          16

#Restore from SaveUnder to this location.
BLTDEF                                  0x1111
DRAWDEF                                 0x00CC
OP1_opRDRAM.pt.X 24
OP1_opRDRAM.pt.Y 256
BLTEXT_EX.pt.Y                          16

#SaveUnder from next location.
OP0_opRDRAM.pt.X 24
OP0_opRDRAM.pt.Y 256
OP1_opRDRAM.pt.X 1
OP1_opRDRAM.pt.Y 1
BLTEXT_EX.pt.Y                          16

#Paint Cursor to next location.
BLTDEF                                  0x1015
DRAWDEF                                 0x01CC
OP0_opRDRAM.pt.X 1
OP0_opRDRAM.pt.Y 1
OP1_opRDRAM.pt.X 0
OP1_opRDRAM.pt.Y 256
BLTEXT_EX.pt.Y                          16
#etc...
```


2.4.2 Font Load Programming Example

This example illustrates a simple move of a block of data from the host to the frame buffer. The host is caching a font (typically in off-screen memory) to use in a subsequent text BitBLT. The monochrome bitmap is treated as color data and moved to the frame buffer. Make sure that the proper number of dwords are written to the HOSTDATA register following the BLTEXT command. The number of dwords is the number of lines times the number of dwords required to contain all the data on a single line. For instance, a font that fills 5 bytes in width and 21 lines in height requires $2 \times 21 = 42$ dwords, since 5 bytes consume all of one and part of a second dword. For more information on calculating the number of dwords, reference the general formula in Section 2.2.6 of this chapter.

```
# load 8x8 mono font for letter "E" from host to frame buffer.
BITMASK                  0xFFFFFFFF  # allbitsenable
BLTDEF                   0x1020      # op1_is_host, res_is_rdram
DRAWDEF                  0x00CC      # rop_op1_copy
OP0_opRDRAM.pt.X         0           # BLT -> 0,0 cache font at 0,0
OP0_opRDRAM.pt.Y         0
OP1_opRDRAM.pt.X         0           # HostData phase of 0 (Use Byte at
                                     # Low Address as First Byte)
BLTEXT_EX.pt.X           1
BLTEXT_EX.pt.Y           8
HOSTDATA                 0x00000000
HOSTDATA                 0x0000003E
HOSTDATA                 0x00000002
HOSTDATA                 0x0000001E
HOSTDATA                 0x00000002
HOSTDATA                 0x00000002
HOSTDATA                 0x0000007E
HOSTDATA                 0x00000000
```

2.4.3 Text BitBLT, Foreground/Background Color Programming Example

The following text BitBLT example writes a character to the screen from a monochrome bitmap with ones appearing as foreground color, and zeroes appearing as background color. The bitmap is assumed to be in the frame buffer. Fill out the foreground and background colors in the Foreground Color and Background Color registers if they are 8 or 16 bpp (at 8 bpp, the color is repeated four times).

```
# do a mono to color foreground/background BLT mono font using the
# mono "E" font from prior example.
OP_OPBGCOLOR              0x22222222  # load colors
OP_OPFGCOLOR              0x00000000
BLTDEF                    0x1050      # res = RDRAM, op1/src = fb mono
DRAWDEF                   0x00CC      # rop = src copy
OP1_OPmRDRAM.pt.X 0x0
OP1_OPmRDRAM.pt.Y 0x0
OP0_OPRDRAM.pt.X 0x1
OP0_OPRDRAM.pt.Y 0x0
BLTEXT_EX                 0x00080008              # 8 x 8 character
```

2.4.4 Text BitBLT, Monochrome Font from Host Programming Example

This example illustrates how to write a character to the screen with the font cached on the host. This method of rendering characters can be simpler and faster than caching fonts in the frame buffer in some systems. If the monochrome font bitmap is provided by the function that is requesting the font to be rendered, then skipping the cache to off-screen eliminates unnecessary memory accesses.

```
# draw 8x8 mono font for letter "h" from host to framebuffer.
BITMASK                   0xFFFFFFFF              # allbitsenable
BLTDEF                    0x1060                  # op1_is_host_mono,
res_is_rdram
DRAWDEF                   0x00CC                  # rop_op1_copy
OP0_opRDRAM.pt.X          0                       # BLT -> 0,0
OP0_opRDRAM.pt.Y          0
OP1_opMRDRAM.pt.X 0                   # Set Host Phase of zero
OP_OPBGCOLOR              0xAAAAAAAA              # Set FG & BG colors
OP_OPFGCOLOR              0x55555555
CONTROL                   0x0400                  # Swizzle on
BLTEXT_EX.pt.X            8                       # Command BLTer to take data
BLTEXT_EX.pt.Y            8
HOSTDATA                  0x00000000
HOSTDATA                  0x00000040
HOSTDATA                  0x00000040
HOSTDATA                  0x00000040
HOSTDATA                  0x00000078
```



```
HOSTDATA                0x00000048
HOSTDATA                0x00000048
HOSTDATA                0x00000000

# draw 8x8 mono font for transparent letter "h" from host to frame
# buffer.
BLTDEF                  0x1866              # op1_is_host_mono,
                                            # op2_is_host_mono,
                                            # res_is_rdram, op1 = op2
DRAWDEF                 0x01CC              # rop_op1_copy, xpar
OP0_opRDRAM.pt.X        8                   # BLT -> 0,0
OP0_opRDRAM.pt.Y        8
OP2_opMRDRAM.pt.X 0                         # Set Host Phase of zero
OP1_opMRDRAM.pt.X 0
BLTEXT_EX.pt.X                  8           # Command BLTer to take data
BLTEXT_EX.pt.Y                  8
HOSTDATA                        0x00000000
HOSTDATA                        0x00000040
HOSTDATA                        0x00000040
HOSTDATA                        0x00000040
HOSTDATA                        0x00000078
HOSTDATA                        0x00000048
HOSTDATA                        0x00000048
HOSTDATA                        0x00000000
```

2.4.5 Text BitBLT, Transparent Background Programming Example

This example places an 8 × 8 character on the screen. The character is foreground color, and the background is transparent. Both the OP1 (SRC) and the OP2 (PAT) are pointed to the same font. While OP1 performs the foreground monochrome-to-color conversion, OP2 performs the transparent decision making.

```
# load 8x8 mono font for letter “h” from host to framebuffer.
BITMASK                  0xFFFFFFFF              # allbitsenable
BLTDEF                   0x1020                  # op1_is_host, res_is_rdram
DRAWDEF                  0x00CC                  # rop_op1_copy
OP1_opRDRAM.pt.X         0                       # host phase is 0
OP0_opRDRAM.pt.X         0                       # BLT -> 0,0
OP0_opRDRAM.pt.Y 0
BLTEXT_EX.pt.X           1
BLTEXT_EX.pt.Y           8
HOSTDATA                 0x00000000
HOSTDATA                 0x0000003E
HOSTDATA                 0x00000002
HOSTDATA                 0x0000001E
HOSTDATA                 0x00000002
HOSTDATA                 0x00000002
HOSTDATA                 0x0000007E
HOSTDATA                 0x00000000
# ---------------------------------------------------------------
# do a transparent mono to color BLT using the mono “E” font.
# FGC! = BGC for mono transparency to work.
OP_OPBGCOLOR             0x00000000
OP_OPFGCOLOR             0x01010101
BLTDEF                   0x1055                  # res = RDRAM op1 =
rdram_mono,
                                                 # op2 = rdram_mono
DRAWDEF                  0x01cc                  # src copy transp_op = “=“
OP2_OPmRDRAM.pt.X 0x0                            # point to mono mask (same as
font)
OP2_OPmRDRAM.pt.Y 0x0
OP1_OPmRDRAM.pt.X 0x0                            # point to font (same as mono
mask)
OP1_OPmRDRAM.pt.Y 0x0
OP0_OPRDRAM.pt.X         0x8                     # point to destination
OP0_OPRDRAM.pt.Y 0x8
BLTEXT_EX                0x00080008              # BLT it.
```


2.4.6 Simple Source Copy Programming Example

The following example is of a BitBLT moving a rectangular region from one location in the frame buffer to another location. Analyze the cases for overlapping BitBLTs. Overlapping BitBLTs that move down (increasing Y) the screen are performed with BLTDEF.YDIR set to up, and with the source and destination pointers anchored to the lower-left corners of the rectangles. Overlapping BitBLTs that move up the screen are performed with BLTDEF.YDIR set to down, and with the source and destination pointers anchored to the upper-left corners. Additionally, the programmer should consider the case of a purely horizontal, left-to-right BitBLT that has the destination overlapping the source. This BitBLT results in vertical stripes unless it is broken up into vertical strips of 128 bytes width or cached through an intermediate buffer.

```
BLTDEF                  0x1111              # result=fb, op1=op2=color fb
DRAWDEF                 0x00CC              # rop = srccpy (op1 copy)
OP1_opRDRAM.pt.X        60                  # source = 60,2
OP1_opRDRAM.pt.Y        2
OP0_opRDRAM.pt.X        88                  # result = 88,8
OP0_opRDRAM.pt.Y        08
BLTEXT_EX.pt.X          8                   # BLT size = 8x16
BLTEXT_EX.pt.Y          16
```

The variation below, of the example above, shows that OP2 is also used for 'source' copy BitBLTs. Do not confuse its 'Pat' name and assume it always patterns. Both OP1 and OP2 can pattern, but the respective pattern bits must be set in the BLTDEF register. This is NOT a pattern BitBLT. It is a simple rectangular move.

```
BLTDEF                  0x1111              # result=fb, op1=op2=color fb
DRAWDEF                 0x00F0              # rop = patcpy (op2 copy)
OP2_opRDRAM.pt.X        60                  # source = 60,2
OP2_opRDRAM.pt.Y        2
OP0_opRDRAM.pt.X        88                  # result = 88,8
OP0_opRDRAM.pt.Y        08
BLTEXT_EX.pt.X          8                   # BLT size = 8x16
BLTEXT_EX.pt.Y          16
```

2.4.7 Copy Frame Buffer-to-Host Programming Example

The following BitBLT example moves a rectangular block from the frame buffer into the host memory. To set up the BitBLT parameters, issue the BLTEXT command, and read the appropriate number of dwords from the HOSTDATA register. Make sure that the proper number of dwords are read from the HOSTDATA register. The number of dwords is the number of lines times the number of dwords required to contain all the data on a single line rounded up to the nearest qword. The 2D engine only supplies data in 8-byte units to the host. For instance, a font that fills 5 bytes in width and 21 lines in height requires $2 \times 21 = 42$ dwords, since 5 bytes consume all of one and part of a second dword. For more information on how to calculate the number of dwords, reference the general formula in Section 2.2.6.

```
# read 4x4 region from 8bpp frame buffer
BLTDEF                  0x2010                  # op1_is_rdram, res_is_host
DRAWDEF                 0x00CC                  # rop_op1_copy
OP0_opRDRAM.pt.X        0                       # host phase is 0
OP1_opRDRAM.pt.X        100                     # BLT source is 100,100
OP1_opRDRAM.pt.Y        100
BLTEXT_EX.pt.X          4
BLTEXT_EX.pt.Y          4
read HOSTDATA           DWORD       # 4 pixels at 8 bpp first Y line
read HOSTDATA           DWORD       # 4 pixels at 8 bpp first Y line
read HOSTDATA           DWORD       # 4 pixels at 8 bpp second Y line
read HOSTDATA           DWORD       # 4 pixels at 8 bpp second Y line
read HOSTDATA           DWORD       # 4 pixels at 8 bpp third Y line
read HOSTDATA           DWORD       # 4 pixels at 8 bpp third Y line
read HOSTDATA           DWORD       # 4 pixels at 8 bpp fourth Y line
read HOSTDATA           DWORD       # 4 pixels at 8 bpp fourth Y line
```


2.4.8 Color-Pattern BitBLT Programming Example

In this example, a color-pattern BitBLT tiles a rectangular region with a patterned source. OP fetches the pattern from the frame buffer. The result is written to another frame buffer location.

All patterns are 8 × 8 pixels in dimension. Eight- and sixteen-bpp patterns are stored linearly in adjacent memory locations, 24- and 32-bpp patterns are stored on two adjacent lines (same X, Ys differ by one). Align pattern sources to qword boundaries.

The pattern (0,0) is anchored to the frame buffer (0,0) and is moved with respect to this anchoring by setting the PATOFF(X,Y) register. PATOFF(X,Y) values of (0,0) anchors the upper-left corner of the pattern to (0,0) in the frame buffer. PATOFF(X,Y) values of (7,7) anchors the lower-right corner of the pattern to (0,0) in the frame buffer. Once a PATOFF value is selected, all subsequent patterned BitBLTs align with each other regardless of their destination addresses.

```
BITMASK                 0xFFFFFFFF              # enable all bits
OP_OPBGCOLOR            0x00000000              # make fg! = bg for any future
                                                # transparency to work
OP_OPFGCOLOR            0xFFFFFFFF
OP0_opRDRAM.pt.X        0x0                     # set all phases to 0
OP1_opRDRAM.pt.X        0x0
OP2_opRDRAM.pt.X        0x0
OP0_opSRAM              0x0
OP1_opSRAM              0x0
OP2_opSRAM              0x0
CONTROL                 0x0000                  # Swizzle off
# ---------------------------------------------------------------
# load 8x8 color pattern from host to frame buffer.
BLTDEF                  0x1020                  # op1_is_host, res_is_rdram
DRAWDEF                 0x00CC                  # rop_op1_copy
OP1_opRDRAM.pt.X        0x0                     # host source alignment
OP0_opRDRAM.pt.X        0x8                     # send result to 8,1 quad word
aligned
OP0_opRDRAM.pt.Y        0x1         # NOTE ONLY 8/16 bpp Patterns can go
                                                # on ODD scanlines
BLTEXT_EX.pt.X          64                      # pattern size is always 8x8
but BLTed
                                                # linear into 64x1 in 8 bpp
BLTEXT_EX.pt.Y          1
HOSTDATA                0x20202020 0x40202020   # This 8 bpp pattern is
                                                # several concentric
                                                # colored squares
HOSTDATA                0x00000020h 40000000h
HOSTDATA                0x55550020h 40005555h
HOSTDATA                0x01550020h 40005503h
HOSTDATA                0x02550020h 40005504h
HOSTDATA                0x55550020h 40005555h
```

```
HOSTDATA                0x00000020h 40000000h
HOSTDATA                0x40404020h 40404040h
BLTDEF                  0x1090                  # op1 color pat, op2 color pat
DRAWDEF                 0x00CC                  # rop = SRCCPY
PATOFF                  0x0000                  # no offset into pattern
OP0_opRDRAM.pt.X        0x22                    # destination is 22h, 22h
OP0_opRDRAM.pt.Y        0x22
OP1_opRDRAM.pt.X        0x8                     # color pat at 8,1
OP1_opRDRAM.pt.Y        0x1
BLTEXT_EX.pt.X          32                      # BLT size is 32x32
BLTEXT_EX.pt.Y          32
```

2.4.9 Monochrome-to-Color BitBLT Programming Example

The following is an example of a monochrome-to-color BitBLT.

```
# load 8x8 mono pattern for letter "h" from host to frame buffer.
BITMASK                 0xFFFFFFFF              # allbitsenable
BLTDEF                  0x1020                  # op1_is_host, res_is_rdram
DRAWDEF                 0x00cc                  # srccpy
CONTROL                 0x0400                  # Swizzle on
OP1_opRDRAM.pt.X        0                       # host source alignment
OP0_opRDRAM.pt.X        0                       # BLT -> 0,0
OP0_opRDRAM.pt.Y        0
BLTEXT_EX.pt.X          8                       # 8 byte BLT
BLTEXT_EX.pt.Y          1
HOSTDATA                0x40404000              # pattern data
HOSTDATA                0x00484878
CONTROL                 0x0000                  # Swizzle off
# ---------------------------------------------------------------
# do a mono pattern to color fg/bg BLT using the pattern letter "h"
# above.
OP_OPBGCOLOR            0xAAAAAAAA
OP_OPFGCOLOR            0x55555555
BLTDEF                  0x10D0                  # res=fb; op1=pat,mono,fb
DRAWDEF                 0x00CC                  # src copy
PATOFF                  0x0000
OP1_OPmRDRAM.pt.X 0x0
OP1_OPmRDRAM.pt.Y 0x0
OP0_OPRDRAM.pt.X 0x1
OP0_OPRDRAM.pt.Y 0x1
BLTEXT_EX.pt.X          0x0010
BLTEXT_EX.pt.Y          0x0010
```


2.4.10 Solid-Color-Fill Programming Example

This example fills a rectangular frame buffer region with the color specified in the BGCOLOR register.

```
#Example of a color fill
OP_OPBGCOLOR              0x22222222              # load fill color
BLTDEF                    0x1070                  # op1 color source
DRAWDEF                   0x00CC                  # rop = srccpy
OP0_opRDRAM.pt.X          0x0                     # result at 0,0
OP0_opRDRAM.pt.Y          0x0
BLTEXT_EX.pt.X            1024                    # size = 1024x768 (fill the
screen)
BLTEXT_EX.pt.Y            768


#Example of a blackness rop fill
BLTDEF                    0x1101                  # op1 color
DRAWDEF                   0x0000                  # rop = BLACKNESS, op1 src
must be
                                                  # SRAM
OP0_opRDRAM.pt.X          0x2                     # result at 2,2
OP0_opRDRAM.pt.Y 0x2
BLTEXT_EX.pt.X            32                      # size = 32x32 (blacken a
square
                                                              # region)
BLTEXT_EX.pt.Y            32
```

2.4.11 Copy Host to SRAM to Frame Buffer Programming Example

This example loads a pattern from the host into the OP1 SRAM. This operation improves performance by loading SRAM prior to a series of patterning operations. A pattern BitBLT from the frame buffer loads SRAM during its operation. Subsequent BitBLTs are then performed from SRAM without an explicit move into SRAM (note that if auto-BitBLTs are triggered, there is a possibility that SRAM contents are overwritten during the auto-BitBLT. This can trigger between the first and second BitBLTs, invalidating the data.)

```
# load 8x8 color pattern from host to SRAM
BLTDEF                  0x5020                  # op1_is_host, res_is_sram1
DRAWDEF                 0x00CC                  # rop_op1_copy
OP0_opRDRAM.pt.X        0                       # point to SRAM.(SRAM0 ptr is
used
                                                # for ALL result SRAMS)
OP1_opRDRAM.pt. X       0x0                     # Host Phase of zero
BLTEXT_EX.pt.X          64
BLTEXT_EX.pt.Y          1
HOSTDATA                0x20202020 0x40202020
HOSTDATA                0x00000020 0x40000000
HOSTDATA                0x55550020 0x40005555
HOSTDATA                0x01550020 0x40005503
HOSTDATA                0x02550020 0x40005504
HOSTDATA                0x55550020 0x40005555
HOSTDATA                0x00000020 0x40000000
HOSTDATA                0x40404020 0x40404040
# ---------------------------------------------------------------
# pattern the screen with the above color pattern from sram1
PATOFF                  0x0000
BLTDEF                  0x1080                  # res RDRAM, OP1 SRAM pat
DRAWDEF                 0x00CC                  # SRC copy
OP1_opSRAM              0
OP0_opRDRAM.pt.X        0x00
OP0_opRDRAM.pt.Y        0x00
BLTEXT_EX.pt.X          0x20
BLTEXT_EX.pt.Y          0x20
# ---------------------------------------------------------------
#BLT sram1 to sram2 (an unusual operation but what the heck)
BLTDEF                  0x6000                  # res sram2, OP1 SRAM (NOTE
pat bit
                                                # not required)
DRAWDEF                 0x00CC                  # SRC copy
OP0_opRDRAM.pt.X        0                       # load at offset 0 in SRAM2
 OP1_opSRAM             0                       # load from offset 0 in SRAM1
BLTEXT_EX.pt.X          64                      # move 64 bytes (8bpp)
```



```
BLTEXT_EX.pt.Y           1
# ------------------------------------------------------------------
# pattern the screen with the sram2 color pattern
PATOFF                   0000
BLTDEF                   0x1008                # res RDRAM, OP2 SRAM color
pattern
DRAWDEF                  0x00F0                # PAT copy
OP2_opSRAM               0
OP0_opRDRAM.pt.X 0x20
OP0_opRDRAM.pt.Y 0x20
BLTEXT_EX.pt.X           0x20
BLTEXT_EX.pt.Y           0x20
```

2.4.12 Transparent-Monochrome-Cursor Programming Example

This example shows how to use two separate monochrome maps to implement a cursor. The shape of the cursor is defined by one transparency mask and the two foreground and background colors are defined by the other mask. The resulting cursor is a two-color picture frame shape with a transparent center cutout. The monochrome masks are stored in the frame buffer with two host BLTs, and the cursor object is rendered from them.

```
# ----------------------------------------------------------------
# load 16 bit x 16line mono cursor map from host to frame buffer.
BITMASK                  0xFFFFFFFF              # write mask -> all enable
BLTDEF                   0x1020                  # op1_is_host, res_is_rdram
DRAWDEF                  0x00CC                  # rop_op1_copy
OP0_opRDRAM.pt.X         0                       # BLT -> 0,256 off-screen
memory
OP0_opRDRAM.pt.Y 256
OP1_opRDRAM.pt.X         0x0                     # Host Phase of zero
CONTROL                  0x0400                  # Swizzle on
BLTEXT_EX.pt.X           2                       # cursor width in bytes
BLTEXT_EX.pt.Y           16                      # cursor length in lines
HOSTDATA                 0x0000                  # line 1
HOSTDATA                 0x0100                  # line 2   1=Foregound color
X
HOSTDATA                 0x0300                  # line 3     0=Background
color XX
HOSTDATA                 0x0700                  # line 4                 XXX
HOSTDATA                 0x0f00                  # line 5                XXXX
HOSTDATA                 0x1f00                  # line 6               XXXXX
HOSTDATA                 0x3f00                  # line 7              XXXXXX
HOSTDATA                 0x7f00      # line 8                 XXXXXXX
HOSTDATA                 0xff01      # line 9                XXXXXXXX
HOSTDATA                 0xff03      # line 10              XXXXXXXXX
HOSTDATA                 0xff07      # line 11             XXXXXXXXXX
HOSTDATA                 0xff0f      # line 12            XXXXXXXXXXX
HOSTDATA                 0xff1f      # line 13           XXXXXXXXXXXX
HOSTDATA                 0xff3f      # line 14          XXXXXXXXXXXXX
HOSTDATA                 0xff7f      # line 15         XXXXXXXXXXXXXX
HOSTDATA                 0xffff      # line 16        XXXXXXXXXXXXXXX
#
# load a 16 bit x 16 line mono transparency map from host to frame # buffer.
#
BITMASK                  0xFFFFFFFF              # write mask -> all enable
BLTDEF                   0x1020                  #
Src(Op1)=Host,Dest=FB(Frame
                                                 # Buffer)
```



```
DRAWDEF                    0x00CC          # ROP = Copy
OP0_opRDRAM.pt.X 16                        # BLT -> 16,256 BitBLT to off-
screen
                                           # memory
OP0_opRDRAM.pt.Y 256
CONTROL                    0x0500          # Swizzle on, Fifo = 32
BLTEXT_EX.pt.X             2               # mask width (in bytes)
BLTEXT_EX.pt.Y             16              # mask length (in lines)
HOSTDATA                   0xffff          # line 1   XXXXXXXXXXXXXXXX
HOSTDATA                   0xffff          # line 2   XXXXXXXXXXXXXXXX
HOSTDATA                   0xffff          # line 3   XXXXXXXXXXXXXXXX
HOSTDATA                   0xffff          # line 4   XXXXXXXXXXXXXXXX
HOSTDATA                   0x0ff0          # line 5   XXXX        XXXX
HOSTDATA                   0x0ff0          # line 6   XXXX        XXXX
HOSTDATA                   0x0ff0          # line 7   XXXX        XXXX
HOSTDATA                   0x0ff0          # line 8   XXXX        XXXX
HOSTDATA                   0x0ff0          # line 9   XXXX        XXXX
HOSTDATA                   0x0ff0          # line 10  XXXX        XXXX
HOSTDATA                   0x0ff0          # line 11  XXXX        XXXX
HOSTDATA                   0x0ff0          # line 12  XXXX        XXXX
HOSTDATA                   0xffff          # line 13  XXXXXXXXXXXXXXXX
HOSTDATA                   0xffff          # line 14  XXXXXXXXXXXXXXXX
HOSTDATA                   0xffff          # line 15  XXXXXXXXXXXXXXXX
HOSTDATA                   0xffff          # line 16  XXXXXXXXXXXXXXXX
OP_OPFGCOLOR               0x55555555      # Pick a color. Foreground
                                           # color = Pink.
OP_OPBGCOLOR               0x44444444      # Ditto. Background color =
Yellow
#
DRAWDEF                    0x83CC          # Src(OP1)=Mono_Sat,
Transp=on,
                                           # ROP=Src copy
BLTDEF                     0x1055          #
Dest=FB,Src=Mono_FB,Pat=Mono_FB
OP2_opmRDRAM.pt.X 128                      # pattern (bits, lines)
OP2_opmRDRAM.pt.Y 256
OP1_opmRDRAM.pt.X 0                        # source (bits, lines)
OP1_opmRDRAM.pt.Y 256
OP0_opRDRAM.pt.X           32              # destination (pixels, lines)
OP0_opRDRAM.pt.Y           64              # Cursor_X, Cursor_Y
BLTEXT_EX.pt.X             16
BLTEXT_EX.pt.Y             16              # Do it
```

2.4.13 Color-Transparency BitBLTs Programming Example

Use this example to put a striped-color rectangle on the screen. Use SRAM as an example, followed by a color-pattern load. The color pattern is displayed and then BitBLTed on top of the original rectangular area twice, once with source transparency and again with destination transparency. These BitBLTs are useful for 'blue-screen' animation.

```
# Set the Copy to happen from Start of SRAM
OP0_opSRAM              0x00
OP1_opSRAM              0x00
OP2_opSRAM              0x00
OP_opBGCOLOR            0x55555555          # transparent compare color
is 55h
OP1_opRDRAM.pt.X        0x0                 # Host Phase of zero
OP0_opRDRAM.pt.X        0x00                # set SRAM offset for result
# ---------------------------------------------------------------
# Load the SRAM 1 from host
BLTDEF                  0x5020              # op1_is_host, res_is_sram
DRAWDEF                 0x00CC              # rop_op1_copy
BLTEXT_EX.pt.X          128
BLTEXT_EX.pt.Y          1
# HOSTDATA contains many 55's to illustrate color transparency in
# subsequent BLTs.
HOSTDATA                0xcbcb55c8 0xcf55cfcc
HOSTDATA                0xd3d355d0 0xd755d7d4
HOSTDATA                0xdbdb55d8 0xdf55dfdc
HOSTDATA                0xe3e355e0 0xe755e7e4
HOSTDATA                0x03035500 0x07550704
HOSTDATA                0x0b0b5508 0x0f550f0c
HOSTDATA                0x13135510 0x17551714
HOSTDATA                0x1b1b5518 0x1f551f1c
HOSTDATA                0x03035500 0x07550704
HOSTDATA                0x0b0b5508 0x0f550f0c
HOSTDATA                0x13135510 0x17551714
HOSTDATA                0x1b1b5518 0x1f551f1c
HOSTDATA                0xcbcb55c8 0xcf55cfcc
HOSTDATA                0xd3d355d0 0xd755d7d4
HOSTDATA                0xdbdb55d8 0xdf55dfdc
HOSTDATA                0xe3e355e0 0xe755e7e4
# ---------------------------------------------------------------
# Load the FB from SRAM with the above color data.
BLTDEF                  0x1000              # op1_is_sram, res_is_rdram
DRAWDEF                 0x00cc              # rop_op1_copy
OP0_opRDRAM.pt.X        0x10                # BLT to 10h, 10h
```



```
OP0_opRDRAM.pt.Y         0x10
BLTEXT_EX.pt.X           128                    # size is 128x20
BLTEXT_EX.pt.Y           20
# ---------------------------------------------------------------
# Load 8x8 color pattern from host to frame buffer.
BLTDEF                   0x1020                 # op1_is_host, res_is_rdram
DRAWDEF                  0x00CC                 # rop_op1_copy
OP0_opRDRAM.pt.X         0x00                   # BLT pattern to 0,0
OP0_opRDRAM.pt.Y         0x00
BLTEXT_EX.pt.X           64
BLTEXT_EX.pt.Y           1
# Pattern contains lots of 55's to illustrate color transparency
# later on.
HOSTDATA                 0x55555555 0x55555555
HOSTDATA                 0x22222220 0x33333333
HOSTDATA                 0x55552222 0x33335555
HOSTDATA                 0x55552222 0x33335555
HOSTDATA                 0x55552222 0x33335555
HOSTDATA                 0x55552222 0x33335555
HOSTDATA                 0x22222222 0x33333333
HOSTDATA                 0x55555555 0x55555555
# ---------------------------------------------------------------
# Pattern the screen with the above color pattern just to show the
# raw pattern.
PATOFF                   0000
BLTDEF                   0x1109                 # res RDRAM, OP2 COL PAT
DRAWDEF                  0x00F0                 # PAT copy
OP2_opRDRAM.pt.X 0x00
OP2_opRDRAM.pt.Y 0x0
OP0_opRDRAM.pt.X         0x02                   # put it at 2,2
OP0_opRDRAM.pt.Y 0x02
BLTEXT_EX.pt.X           0x30                   # make it 30Hx10H
BLTEXT_EX.pt.Y           0x10
# ---------------------------------------------------------------
# Illustrate source transparency. Source pixels that match 55 will
# not be written to result.
# Pattern a rectangle on the screen over top of the original color
# rectangle with the color pattern with color transparency on.
# Note that OP2 is used for both transparency and pattern.
PATOFF                   0000
BLTDEF                   0x1109                 # res RDRAM, OP2 COL PAT,
DRAWDEF                  0x01F0                 # PAT copy, trn=on / opn="="
```

```
OP2_opRDRAM.pt.X         0x00                  # Point to source color
pattern
                                               # containing 55's
OP2_opRDRAM.pt.Y 0x0
OP0_opRDRAM.pt.X         0x40                  # Point on top of existing
color
                                               # rectangle.
OP0_opRDRAM.pt.Y         0x14
BLTEXT_EX.pt.X           0x40                  # Paint it!
BLTEXT_EX.pt.Y           0x18
# ---------------------------------------------------------------
# Illustrate destination transparency. Destination pixels that match 55 will not
be written to.
PATOFF                   0000
BLTDEF                   0x1191                # res RDRAM; OP1 color pat;
OP2 color
DRAWDEF                  0x03CC                # srccpy; trn=on, op="!="
OP1_opRDRAM.pt.X         0x00                  # point to color pattern
source.
OP1_opRDRAM.pt.Y 0x00
OP2_opRDRAM.pt.X         0x14                  # point to "destination" for
                                               # transparent compare
OP2_opRDRAM.pt.Y 0x08
OP0_opRDRAM.pt.X         0x14                  # point to result destination
OP0_opRDRAM.pt.Y 0x08
BLTEXT_EX.pt.X           0x18                  # Paint it 18Hx20H
BLTEXT_EX.pt.Y           0x20
```


2.4.14 Monochrome-Pattern-Transparency Mask Programming Example

This BitBLT example puts several 8 × 8 patterns onto the destination region. The result is a four-square set of BitBLTs. The monochrome pattern is stored in the frame buffer in one 64-bit qword. With monochrome transparency, the monochrome bits directly control the writes to the frame buffer. With the transparency compare operation set to equal, any bit that is a one blocks the write to the frame buffer for the corresponding pixel. A zero bit allows the write. The foreground and background colors need only be set if the programmer is actually using the monochrome-to-color conversion.

```
BITMASK                  0xFFFFFFFF
OP_OPBGCOLOR             0x00000000
OP_OPFGCOLOR             0xFFFFFFFF
OP1_opRDRAM.pt.X 0x0
OP0_opSRAM               0x0
# ------------------------------------------------------------------
# Load 8x8 mono pattern for diagonal stripes from host to frame
# buffer.
BLTDEF                   0x1020                  # op1_is_host, res_is_rdram
DRAWDEF                  0x00CC                  # rop_op1_copy
OP0_opRDRAM.pt.X         0x0                     # BLT -> 10,0
OP0_opRDRAM.pt.Y         0x0                     # BLT -> 10,0
OP1_opRDRAM.pt.X         0x0                     # Host Phase of zero
BLTEXT_EX.pt.X           0x8
BLTEXT_EX.pt.Y           0x1
HOSTDATA                 0x88112244
HOSTDATA                 0x88112244
# ------------------------------------------------------------------
# Transparent tile the upper right square with a BLT down
BLTDEF                   0x109D                  # op1 color pat, op2 MONOPAT
DRAWDEF                  0x01F0                  # transp on, ROP=PATCPY
PATOFF                   0x0000
OP0_opRDRAM.pt.X 0x24
OP0_opRDRAM.pt.Y 0x02
OP2_opmRDRAM.pt.X 0x00                           # mono pat at 0,0
OP2_opmRDRAM.pt.Y 0x00
BLTEXT_EX.pt.X           32
BLTEXT_EX.pt.Y           32
# ------------------------------------------------------------------
# Transparent tile the lower right square with a BLT up
BLTDEF                   0x909D                  # op1 color pat, op2 MONOPAT
DRAWDEF                  0x01F0                  # transp on, ROP=PATCPY
PATOFF                   0x0000
OP0_opRDRAM.pt.X 0x24
OP0_opRDRAM.pt.Y         0x42                    # lower left dest
OP2_opmRDRAM.pt.X 0x00                           # mono pat at 0,0
OP2_opmRDRAM.pt.Y 0x00
BLTEXT_EX.pt.X           32
BLTEXT_EX.pt.Y           32
```

2.4.15 Byte BitBLT Using MBitBLT and BitBLT for Color-Fills Programming Example

In this example, a 16-bpp frame buffer demonstrates how to put four colored rectangles of pixel extent, 100 × 100 (byte extent 200 × 100 lines), onto the screen with mixed OP pointer types and BLTEXT types.

```
# ------------------------------------------------------------------
# FILL WITH PIXEL POINTERS AND PIXEL EXTENTS.  Goes to PP (0,0)
BLTDEF                  0x1070                  # op1 color fill
DRAWDEF                 0x00CC                  # rop = FILL
OP_OPBGCOLOR            0x11111111
OP0_opRDRAM.pt.X 0
OP0_opRDRAM.pt.Y 0
BLTEXT_EX.pt.X          100
BLTEXT_EX.pt.Y          100
# ------------------------------------------------------------------
# FILL WITH PIXEL POINTERS AND BYTE EXTENTS.  Goes to PP (100,200)
OP_OPBGCOLOR            0x44444444
OP0_opRDRAM.pt.X 100
OP0_opRDRAM.pt.Y 100
MBLTEXT_EX.pt.X         200                     # in 16 bpp, 100 pixels wide
is 200
                                                # bytes
MBLTEXT_EX.pt.Y         100
# ------------------------------------------------------------------
# FILL WITH BYTE POINTERS AND BYTE EXTENTS; GOES TO PP(200,200)
OP_OPBGCOLOR            0x88888888
OP0_opmRDRAM.pt.X 400                           # in 16 bpp, X pixel address
200 is
                                                # byte address 400
OP0_opmRDRAM.pt.Y 200
MBLTEXT_EX.pt.X         200                     # in 16 bpp, 100 pixels wide
is 200
                                                # bytes
MBLTEXT_EX.pt.Y         100
# ------------------------------------------------------------------
# FILL WITH BYTE POINTERS AND PIXEL EXTENTS.  GOES TO PP(300,300)
OP_OPBGCOLOR            0xCCCCCCCC
OP0_opmRDRAM.pt.X 600                           # in 16 bpp, X pixel address
300 is
                                                # byte address 600
OP0_opmRDRAM.pt.Y 300
BLTEXT_EX.pt.X          100
BLTEXT_EX.pt.Y          100
```


2.4.16 Byte BitBLT Using MBitBLT to Off-Screen Cache Programming Example

Using the MBitBLT from host, this example demonstrates loading a 56-byte auto-BitBLT record into the frame buffer at (0,1024). This is useful for loading data off-screen in a bit-per-pixel independent manner. Typical uses of this example are for font loading, pattern caching, auto-BitBLT record downloads, and cursor-bitmap loading.

```
BITMASK                  0xFFFFFFFF            # turn on all bits
BLTDEF                   0x1020                # OP1/host source to RDRAM
color
                                               # result
DRAWDEF                  0x00CC                # rop:SRCCOPY
OP0_opMRDRAM.pt.X 0x0000                       # point to off-screen
destination
                                               # location (0,1024)
OP0_opMRDRAM.pt.Y 0x0400
OP1_opRDRAM.pt.X         0x0000                # data is aligned in host
dwords
                                               # phase = 0
MBLTEXT_EX.pt.X          0x0038                # byte extents are 56X bu 1Y
MBLTEXT_EX.pt.Y          0x0001
# ------------------------------------------------------------------
# Auto BLT table 1:
HOSTDATA                 0x00000200
HOSTDATA                 0x100900F0
HOSTDATA                 0x00000000
HOSTDATA                 0xFFFFFFFF
HOSTDATA                 0x00000080
HOSTDATA                 0x00000000
HOSTDATA                 0x00000000
HOSTDATA                 0x00000000
HOSTDATA                 0x00000020
HOSTDATA                 0x00000000
HOSTDATA                 0x00180080
HOSTDATA                 0x00000000
HOSTDATA                 0x00000000
HOSTDATA                 0x00000000
# ------------------------------------------------------------------
```

2.5 Register Header Files

This section presents the implementation of a set of 'C' language data types and structures, header files, used for interfacing software to the CL-GD546X. The first file, lgtypes.h, defines several types used to create the registers data structure that follows in the second file, lgregs.h.

2.5.1 Header File — lgtypes.h

```
#ifndef _LGTYPES_H
#define _LGTYPES_H
/* Use #define here instead of typedef to make it easier on systems
where these same types are also defined elsewhere, i.e., it is possible to use
#ifdef, #undef, etc. on these types.
*/
#define byte unsigned char
#define word unsigned short
#define dword unsigned long
typedef unsigned long ul;
typedef unsigned short word;
typedef unsigned char byte;
typedef unsigned char boolean;
typedef struct PT {                                 // point
             WORD        X;
             WORD        Y;
} PT;
typedef struct LOHI {                               // low, high
             WORD        LO;
             WORD        HI;
} LOHI;
typedef union _reg32 {                              // 32 bit register
             DWORD       dw;
             DWORD       DW;
             PT          pt;
             PT          PT;
             LOHI        lh;
             LOHI        LH;
} REG32;
typedef struct LOHI16 {                             // low, high
             BYTE        LO;
             BYTE        HI;
} LOHI16;
typedef struct PT16 {                               // point
             BYTE        X;
             BYTE        Y;
```



```
} PT16;
typedef union _reg16 {                          // 16 bit register
             WORD        w;
             WORD        W;
             PT16        pt;
             PT16        PT;
             LOHI16      lh;
             LOHI16      LH;
} REG16;
#endif       /* _LGTYPES_H */
```

2.5.2 Header File — lgregs.h

This file contains the register-set mapping with all component offsets correctly matched to the CL-GD546X. It contains a record definition for the auto-BitBLT record using the types defined in lgtypes.h to construct the Graphics_Accelerator_Registers_Type. Bit-field # defines are given to simplify construction of control fields for selected registers.

```
#ifndef _LGREGS_
#define _LGREGS_
#include "lgtypes.h"
// CL-GD546X Graphics Accelerator Registers data type.
typedef struct GAR {
// Memory-mapped Registers
// Memory-mapped VGA Registers
             BYTE grCR0;                        //0x0
             BYTE grPADCR0[3];
             BYTE grCR1;                        //0x04
             BYTE grPADCR1[3];
             BYTE grCR2;                        //0x08
             BYTE grPADCR2[3];
             BYTE grCR3;                        //0x0C
             BYTE grPADCR3[3];
             BYTE grCR4;                        //0x010
             BYTE grPADCR4[3];
             BYTE grCR5;                        //0x014
             BYTE grPADCR5[3];
             BYTE grCR6;                        //0x018
             BYTE grPADCR[3];
             BYTE grCR7;                        //0x01C
             BYTE grPADCR7[3];
             BYTE grCR8;                        //0x020
             BYTE grPADCR8[3];
             BYTE grCR9;                        //0x024
```

```
BYTE grPADCR9[3];
BYTE grCRA;                         //0x028
BYTE grPADCRA[3];
BYTE grCRB;                         //0x02C
BYTE grPADCRB[3];
BYTE grCRC;                         //0x030
BYTE grPADCRC[3];
BYTE grCRD;                         //0x034
BYTE grPADCRD[3];
BYTE grCRE;                         //0x038
BYTE grPADCRE[3];
BYTE grCRF;                         //0x03C
BYTE grPADCRF[3];
BYTE grCR10;                        //0x040
BYTE grPADCR10[3];
BYTE grCR11;                        //0x044
BYTE grPADCR11[3];
BYTE grCR12;                        //0x048
BYTE grPADCR12[3];
BYTE grCR13;                        //0x04C
BYTE grPADCR13[3];
BYTE grCR14;                        //0x050
BYTE grPADCR14[3];
BYTE grCR15;                        //0x054
BYTE grPADCR15[3];
BYTE grCR16;                        //0x058
BYTE grPADCR16[3];
BYTE grCR17;                        //0x05C
BYTE grPADCR17[3];
BYTE grCR18;                        //0x060
BYTE grPADCR18[3];
BYTE grCR19;                        //0x064
BYTE grPADCR19[3];
BYTE grCR1A;                        //0x068
BYTE grPADCR1A[3];
BYTE grCR1B;                        //0x06C
BYTE grPADCR1B[0x74-0x6D];
BYTE grCR1D;                        //0x074
BYTE grPADCR1D[3];
BYTE grCR1E;                        //0x078
BYTE grPADCR1E[0x80-0x79];
BYTE grMISC;                        //0x080
```



```
                BYTE grPADMISC[3];
                BYTE grSRE;                         //0x084
                BYTE grPADSRE[3];
                BYTE grSR1E;                        //0x088
                BYTE grPADSR1E[3];
                BYTE grBCLK_Numerator;              //0x08C
                BYTE grPADBCLK_Numerator[3];
                BYTE grSR18;                        //0x090
                BYTE grPADSR18[3];
                BYTE grSR19;                        //0x094
                BYTE grPADSR19[3];
                BYTE grSR1A;                        //0x098
                BYTE grPADSR1A[0xA0-0x99];
                BYTE grPalette_Mask;                //0x0A0
                BYTE grPADPalette_Mask[3];
                BYTE grPalette_Read_Address;        //0x0A4
                BYTE grPADPalette_Read_Address[3];
#define grPalette_State_Read_Only grPalette_Read_Address
                BYTE grPalette_Write_Address;       //0x0A8
                BYTE grPADPalette_Write_Address[3];
                BYTE grPalette_Data;                //0x0AC
                BYTE grPADPalette_Data[0xB1-0xAD];
// Video Pipeline Registers
                BYTE grPalette_State;               //0x0B0
                BYTE grPADPalette_State[0xB4-0xB1];

                BYTE grExternal_Overlay;            //0x0B4
                BYTE grPADExternal_Overlay[0xB8-0xB5];

                BYTE grColor_Key;                   //0x0B8
                BYTE grPADColor_Key[0xBC-0xB9];
                BYTE grColor_Key_Mask;              //0x0BC
                BYTE grPADColor_Key_Mask[0xC0-0xBD];
                WORD grFormat;                      //0x0C0
                BYTE grPADFormat[0xCA-0xC2];
                BYTE grStop_BLT_3;                  //0x0CA
                BYTE grStart_BLT_3;                 //0x0CB
                WORD grX_Start_2;                   //0x0CC
                WORD grY_Start_2;                   //0x0CE
                WORD grX_End_2;                     //0x0D0
                WORD grY_End_2;                     //0x0D2
                BYTE grStop_BLT_2;                  //0x0D4
```

```
            BYTE grStart_BLT_2;                 //0x0D5
            BYTE grPADStop_BLT_2[0xDE-0xD6];
            BYTE grStop_BLT_1;                  //0x0DE
            BYTE grStart_BLT_1;                 //0x0DF
            WORD grCursor_X;                    //0x0E0
            WORD grCursor_Y;                    //0x0E2
            WORD grCursor_Preset;               //0x0E4
            WORD grCursor_Control;              //0x0E6
            WORD grCursor_Location;             //0x0E8
            WORD grDisplay_Threshold_and_Tiling;//0x0EA
            BYTE grPADDisplay_Thr[F0h-ECh];
            WORD grTest;                        //0x0F0
            WORD grTest_HT;                     //0x0F2
            WORD grTest_VT;                     //0x0F4
            BYTE        grPADTest_VT[0x100-0x00F6];
// V-PORT Registers
            WORD        grX_Start_Odd;          //0x100
            WORD        grX_Start_Even;         //0x102
            WORD        grY_Start_Odd;          //0x104
            WORD        grY_Start_Even;         //0x106
            WORD        grVport_Width;          //0x108
            BYTE        grVport_Height;         //0x10A
            BYTE grPADVport_Height;
            WORD        grVport_Mode;           //0x10C
            BYTE        grVportpad[0x180-0x10E];
// LPB Registers
            BYTE        grLPB_Data[0x1F8-0x180];//0x180
            BYTE        grPADLPB[0x1FC-0x1F8];
            WORD        grLPB_Config;           //0x1FC
            WORD        grLPB_Status;           //0x1FE
// Rambus Registers
// Rambus Registers for BIOS Simulation
            WORD        grRIF_CONTROL;          //0x200
            WORD        grRAC_CONTROL;          //0x202
            WORD        grRAMBUS_TRANS;         //0x204
            BYTE        grPADRAMBUS_TRANS[0x204-0x206];
            REG32       grRAMBUS_DATA;          //0x240
            BYTE        grPADRAMBUS_DATA[0x280-0x244];
// Serial Bus Registers
            WORD        grSerial_Bus;           //0x0280
            BYTE        grPADSerial_Bus[0x300-0x282];
// PCI Configuration Registers
```



```
                WORD        grVendor_ID;            //0x0300
                WORD        grDevice_ID;            //0x0302
                WORD        grCommand;              //0x0304
                WORD        grStatus;               //0x0306
                BYTE        grRevision_ID;          //0x0308
                BYTE        grClass_Code;           //0x0309
                BYTE        grPADClass_Code[0x30E-0x30A];
                BYTE        grHeader_Type;          //0x030E
                BYTE        grPADHeader_Type[0x310-0x30F];
                REG32       grBase_Address_0;       //0x0310
                REG32       grBase_Address_1;       //0x0314
                BYTE        grPADBase_Address_1[0x32C-0x318];
                WORD        grSubsystem_Vendor_ID;  //0x032C
                WORD        grSubsystem_ID;         //0x032E
                REG32       grExpansion_ROM_Base;   //0x0330
                BYTE        grPADExpansion_ROM_Base[0x33C-0x334];

                BYTE        grInterrupt_Line;       //0x033C
                BYTE        grInterrupt_Pin;        //0x033D
                BYTE        grPADInterrupt_Pin[0x3F8-0x33E];
                REG32       grVGA_Shadow;           //0x03F8
                REG32       grVS_Control;           //0x03FC
    // Graphics Accelerator Registers
    // 2D Engine Control Registers
                WORD        grSTATUS;               //0x400
                WORD        grCONTROL;              //0x402
                BYTE        grQFREE;                //0x404
                BYTE        grOFFSET_2D;            //0x405
                BYTE        grTIMEOUT;              //0x406
                BYTE        grTILE_CTRL;            //0x407
                REG32       grRESIZE_A_opRDRAM;     //408
                REG32       grRESIZE_B_opRDRAM;     //40C
                REG32       grRESIZE_C_opRDRAM;     //410
                BYTE        pad2[0x480-0x414];
                REG32       grCOMMAND;              //480
                BYTE        pad3[0x500-0x484];
                WORD        grMIN_Y;                //500
                WORD        grMAJ_Y;                //502
                WORD        grACCUM_Y;              //504
                BYTE        pad3A[0x508-0x506];
                WORD        grMIN_X;                //508
                WORD        grMAJ_X;                //50A
```

```
                WORD        grACCUM_X;              //50C
                REG16       grLNCNTL;               //50E
                WORD        grSTRETCH_CNTL;         //510
                WORD        grCHROMA_CNTL;          //512
                BYTE        pad3B[0x520-0x514];
                REG32       grOP0_opRDRAM;          //520
                REG32       grOP0_opMRDRAM;         //524
                WORD        grOP0_opSRAM;           //528
                REG16       grPATOFF;               //52A
                BYTE        pad4[0x540-0x52C];
                REG32       grOP1_opRDRAM;          //540
                REG32       grOP1_opMRDRAM;         //544
                WORD        grOP1_opSRAM;           //548
                WORD        grOP1_opMSRAM;          //54A
                BYTE        pad5[0x560-0x54C];
                REG32       grOP2_opRDRAM;          //560
                REG32       grOP2_opMRDRAM;         //564
                WORD        grOP2_opSRAM;           //568
                WORD        grOP2_opMSRAM;          //56A
                BYTE        pad6[0x580-0x56C];
                WORD        grSRCX;                 //580
                REG16       grSHRINKINC;            //582
                REG32       grDRAWBLTDEF;           //584
//              WORD        grDRAWDEF;              //584
#define grDRAWDEF grDRAWBLTDEF.LH.LO
//              WORD        grBLTDEF;               //586
#define grBLTDEF grDRAWBLTDEF.LH.HI
                REG16       grMONOQW;               //588
                BYTE        pad7[0x5e0-0x58A];
                REG32       grOP_opFGCOLOR;         //5E0
                REG32       grOP_opBGCOLOR;         //5E4
                REG32       grBITMASK;              //5E8
                WORD        grPTAG;                 //5EC
                BYTE        pad8[0x5F0 - 0x5EE];
                REG32       grCHROMA_LOWER;         /* 5F0 */
                REG32       grCHROMA_UPPER;         /* 5F4 */
                BYTE        pad8a [0x600 - 0x5F8];
                REG32       grBLTEXT_XEX;           //600
                REG32       grBLTEXTFF_XEX;         //604
                REG32       grBLTEXTR_XEX;          //608
                WORD        grBLTEXT_LN_EX;         //60C
                BYTE        pad9[0x620-0x60E];
```



```
                REG32       grMBLTEXT_XEX;          //620
                BYTE        pad9a[0x628-0x624];
                REG32       grMBLTEXTR_XEX;         //628
                BYTE        pad9b[0x700-0x62C];
                REG32       grBLTEXT_EX;            //700
                REG32       grBLTEXTFF_EX;          //704
                REG32       grBLTEXTR_EX;           //708
                BYTE        pad10[0x720-0x70C];
                REG32       grMBLTEXT_EX;           //720
                BYTE        pad10a[0x728-0x724];
                REG32       grMBLTEXTR_EX;          //728
                BYTE        pad10b[0x800-0x72C];
                                                    //sim
                DWORD       grHOSTDATA[800h];       //800 -> fff
}               Graphics_Accelerator_Registers_Type,
                        * pGraphics_Accelerator_Registers_Type, GAR;
// Status Register values
#define STATUS_FIFO_NOT_EMPTY                   0x0001
#define STATUS_PIPE_BUSY                        0x0002
#define STATUS_DATA_AVAIL                       0x8000
#define STATUS_IDLE ( STATUS_PIPE_BUSY | STATUS_FIFO_NOT_EMPTY )
// Control register values
#define WFIFO_SIZE_32                           0x0100
#define HOST_DATA_AUTO                          0x0200
#define SWIZ_CNTL                               0x0400
// bits 12:11 define tile size
#define TILE_SIZE_128                           0x0000
#define TILE_SIZE_256                           0x0800
#define TILE_SIZE_2048                          0x1800
// bits 14:13 define bits per pixel for graphics modes
#define CNTL_8_BPP                              0x0000
#define CNTL_16_BPP                             0x2000
#define CNTL_24_BPP                             0x4000
#define CNTL_32_BPP                             0x6000
// Tile_ctrl register
// bits 7:6 interleave memory
#define ILM_1_WAY                               0x00
#define ILM_2_WAY                               0x40
#define ILM_4_WAY                               0x80
// bits 5:0 define BYTE pitch of display memory in conjunction with
// TILE_SIZE
// from Control register
```

```
/*
* DRAWDEF contents
*/
#define DD_ROP                                  0x0000
#define DD_TRANS                                0x0100      /* transparent */
#define DD_TRANSOP                              0x0200
#define DD_PTAG                                 0x0400
#define DD_SAT_2                                0x4000
#define DD_SAT_1                                0x8000
// LN_CNTL fields
#define LN_XINTP_EN                             0x0001
#define LN_YINTP_EN                             0x0002
#define LN_XSHRINK                              0x0004
#define LN_YSHRINK                              0x0008
//These are the auto BLT control bits
#define LN_RESIZE                               0x0100
#define LN_CHAIN_EN                             0x0200
// These are the yuv411 output average control bits
#define      LN_LOWPASS                         0x1000
#define LN_UVHOLD                               0x2000
//This extracts the data format field from LNCNTL
#define LN_FORMAT                               0x00F0
#define LN_YUV_SHIFT                            0x4
#define LN_8BIT                                 0x0000
#define LN_RGB555                               0x0001
#define LN_RGB565                               0x0002
#define LN_YUV422                               0x0003
#define LN_24ARGB                               0x0004
#define LN_24PACK                               0x0005
#define LN_YUV411                               0x0006
// 7 - 15 are reserved
/*
* pmBLTDEF contents
*/
#define BD_OP2          0x0001                  /* start of OP2 field
                                                3:0 */
#define BD_OP1          0x0010                  /* start of OP1 field
                                                7:4 */
#define BD_OP0          0x0100                  /* start of OP0 field
                                                8:8 */
#define BD_TRACK_X      0x0200                  /* Track OP ptrs in X
                                                9:9 (when implemented) */
```



```
#define BD_TRACK_Y       0x0400                  /* Track OP ptrs in Y
                         10:10(when implemented) */
#define BD_SAME          0x0800                  /* common operand field
                                                 11:11 */
#define BD_RES           0x1000                  /* start of RES field
                                                 14:12 */
#define BD_YDIR          0x8000                  /* y direction bit 15:
                                                 */
/*  STRETCH_CNTL fields CL-GD546X / ( Laguna 1+ ) */
#define STR_XINTP_EN                 0x0001
#define STR_YINTP_EN                 0x0002
#define STR_XSHRINK                  0x0004
#define STR_YSHRINK                  0x0008

/* These are the autoblt control bits -- REVB chips (using stretch ctrl)*/
#define STR_RESIZE                   0x0040
#define STR_CHAIN_EN                 0x0080

/*  These are the yuv411 output average control bits */
#define STR_LOWPASS                  0x0010
#define STR_UVHOLD                   0x0020

/* This extracts the data format field from STRETCH_CTRL */
#define STR_SRC_FORMAT               0xF000
#define STR_SRC_SHIFT                12

#define STR_DEST_FORMAT              0x0F00
#define STR_DEST_SHIFT               8

#define STR_8BIT                     0x0
#define STR_RGB555                   0x1
#define STR_RGB565                   0x2
#define STR_RGB888                   0x3
#define STR_RGBA888                  0x4

#define STR_YUV411                   0x8
#define STR_YUV422                   0x9
#define STR_YUV444                   0xA
#define STR_YUVA444                  0xB

#define STR_YUV10                    0xC
#define STR_YUV12                    0xD
```

```
/* These are for the CHROMA_CNTL register */
#define CHROMA_ENABLE                0x8000

#define CHROMA_YUV_CV                0x4000
#define CHROMA_TAG_EN                0x1000

#define CHROMA_SEC_SL1               0x0800
#define CHROMA_SEC_SL0               0x0400
#define CHROMA_RGB_OA                0x0200
#define CHROMA_SEC_OA                0x0100

#define CHROMA_SEC_EN                0x0080
#define CHROMA_R_EN                  0x0040
#define CHROMA_G_EN                  0x0020
#define CHROMA_B_EN                  0x0010
#define CHROMA_SEC_IO                0x0008
#define CHROMA_R_IO                  0x0004
#define CHROMA_G_IO                  0x0002
#define CHROMA_B_IO                  0x0001
/*
* Field values for BD_OP? and BD_res.
* LL( grBLTDEF,                      (BD_OP1 * IS_HOST_MONO) +
*                                    (BD_OP2 * (IS_rdram + IS_PATTERN)) +
*                                    (BD_RES * IS_RDRAM) );
*/
#define IS_SRAM                      0x0000
#define IS_RDRAM                     0x0001
#define IS_HOST                      0x0002
#define IS_SOLID                     0x0007
#define IS_SRAM_MONO                 0x0004
#define IS_RDRAM_MONO                0x0005
#define IS_HOST_MONO                 0x0006
#define IS_PATTERN                   0x0008
#define IS_MONO                      0x0004
// these are for BD_RES only
#define IS_SRAM0                     0x0004
#define IS_SRAM1                     0x0005
#define IS_SRAM2                     0x0006
#define IS_SRAM12                    0x0007
// these are for BD_SAME
#define NONE                         0x0000
```



```
typedef struct autoblt_regs {
             REG16       STRETCH_CNTL;
             REG16       SHRINKINC;
             REG32       DRAWBLTDEF;
             REG32       FGCOLOR;
             REG32       BGCOLOR;
             REG32       OP0_opRDRAM;
             WORD        MAJ_Y;
             WORD        MIN_Y;
             REG32       OP1_opRDRAM;
             WORD        ACCUM_Y;
             REG16       PATOFF;
             REG32       OP2_opRDRAM;
             WORD        MAJ_X;
             WORD        MIN_X;
             REG32       BLTEXT;
             WORD        ACCUM_X;
             WORD        OP0_opSRAM;
             WORD        SRCX;
             WORD        OP2_opSRAM;
             REG32       NEXT_HEAD;              // XY address of next in
             REG32       CHROMA_LOWER;           // chain if LNCTL chain
             REG32       CHROMA_UPPER;           // set
             WORD        CHROMA_CNTL;
             WORD        reserved [3];
} autoblt_regs, *autoblt_ptr;
#endif                                           // _LGREGS_
```

2.6 2D Graphics Engine Initialization

This section discusses initialization of the CL-GD546X 2D Graphics Engine registers. It is for display-driver developers working on the adapter initialization code. The following documentation distinguishes between display mode initialization code (mode-switch code) and 2D engine initialization code.

The mode-switch code is typically implemented in the BIOS. This code handles the details of setting up the display mode.

The display initializes and manages 2D Engine registers. Some of these registers are set once by the programmer and never need to be set again. Others may need short-term changes from the 'normal' setting for specific operations.

The display-driver programmer must consider initializing the following registers: BITMASK, CONTROL, LINECTL, OFFSET_2D, TAG_MASK, and TIMEOUT.

BITMASK Register

The 2D engine initialization code manages the BITMASK register. This register is typically not modified by the mode-switch code. The BITMASK register is set to all ones, allowing writes to all bits of each dword. For certain specialized graphics operations, this value can be overridden. Be sure to re-enable all bits.

CONTROL Register

In the CONTROL register, the bpp and TILE_SIZE fields are set by the mode-switch code, and not the AUTO_BLT_EN and SWIZ_CNTL fields. The SWIZ_CNTL field is typically set to '0', and only set to '1' for operations that specifically require the swizzling of host data. Whether or not the programmer enables auto-BitBLTs by default is a design decision. Typically, auto-BitBLTs are enabled. Graphics operations that use SRAM for intermediate storage should disable auto-BitBLTs to prevent the contents of the SRAMs from being destroyed by auto-BitBLTs.

LINECTL Register

Most LINECTL register fields are set by the graphics operations that use them. Only the Graphics Pixel Format field is set by the mode-switch code to match the value in CONTROL.BPP. This field is normally set to match CONTROL.BPP for the benefit of stretch BitBLTs.

OFFSET_2D Register

The OFFSET_2D register typically does not have to be set, since most applications put the 2D frame buffer at '0,0'. However, the programmer should initialize and control this register appropriately for applications that require a non-zero start for the 2D frame buffer (for example, for double-buffered graphics).

TAG_MASK Register

In the TAG_MASK register, the PTMASK field is set to '1' when using 9-bit RDRAMS. The programmer can use the PTAG field in the DRAWDEF register to tag video data in the frame buffer.

TIMEOUT Register

The TIMEOUT register controls the behavior of the 2D engine during long transactions from the host interface. This is caused by trying to write to the command FIFO when there is no room. If the transaction times out, the data in the transfer is dropped and the bus is released. This causes erratic behavior in the display driver, but is more acceptable than locking up the bus. To circumvent the above problem, set the time-out delay to a large value and enable time-outs. In particular, set TIMEOUT.TIMEOUT to '1111', TIMEOUT.TIMEOUT_EN to '1' and TIMEOUT.TIMEOUT_X16 to '1'. This should only be required during driver development as a debug aid. Production drivers should have TIMEOUT disabled.

