Video Programming

4

4. VIDEO PROGRAMMING

This section describes the features of the CL-GD546X that are used to support video data display concurrently with high-resolution graphics. Specifically, it describes the programming of resize and auto-triggered BitBLTs that move and resize data from the video capture buffers onto the display surface. The hardware features of the CL-GD546X that support capturing of video data from an external source and the overlaying of graphics data with video data from an external source are covered in the *Laguna VisualMedia™ Accelerators Family — CL-GD546X Volume I (Hardware Reference Manual, Second Edition, September 1996)*.

In the CL-GD546X, the resize BitBLT is the primary feature used to support video display. This is required because video data is supplied in a format that is low in resolution compared with modern high-resolution displays (such as 352 × 240 pixels versus 1024 × 768). If the data is displayed in its native size, the resulting image is unacceptably small. The data is presented at a frame rate too low for a high-resolution monitor. If displayed directly, this results in torn or banded images since the display shows several partial images of the data in the process of overwriting the previous image. To solve the first problem of resolution, stretch the data to occupy a larger area of the screen. This is the stretch/resize BitBLT. To solve the second problem, synchronize the stretching and movement of the data to the display surface with the vertical refresh count of the display. This is the function of the auto-BitBLT.

The CL-GD546X supports two modes of resize BitBLTs. The first mode is CL-GD5462–compatible and is programmed using the LNCNTL register. This is for stretching data and writing the output in the same format as the input, as described in the following sections. The second mode is programmed by the STRETCH_CNTL register, this mode allows outputting the stretched data in a different format from the input data, as described in the following sections.

The CL-GD546X resize engine also contains a flexible chroma-key comparator. This is described in Section 4.5 on page 4-12.

4.1 Resize BitBLTs

The CL-GD546X supports arbitrary ratio resize operations independently in both axes. It also supports pixel replication or quarter phase interpolation independently in both axes. The resize operations are either stretches or shrinks, or a mixture of a stretch in one axis with a shrink in another (although this usage is unusual). The resize engine can deal with data in a number of different formats that are applicable to either video or graphics.

The core of the resize engine is a pair of DDA (discrete differential analyzer) engines that operate in the X and Y axes. Conceptually, it is easiest to think of the Y DDA engine fetching a scanline of data and then calling the X DDA engine, which stretches it horizontally to the required width after which the Y DDA engine then steps the output to the next scanline. It either fetches a new scanline of data or repeats the old scanline as required and continues.

To program a resize BitBLT a number of parameters must be calculated and loaded into the appropriate 2D Engine Control registers. The calculations are the same for both the manually-commanded resize BitBLT and the auto-triggered resize BitBLT.

4.1.1 Required Parameters

1) Destination address in X and Y pixel coordinates in the frame buffer. This is loaded into OP0_opRDRAM.

2) Source address in X and Y pixel coordinates are generally off-screen in the frame buffer. This is loaded into OP1_opRDRAM.

3) Data format (such as 8-bpp CLUT / RGB 5:6:5 / YUV 4:2:2, and so on) is loaded into LNCNTL bits 6:4. The codes are given in Table 4-1. X,Y shrink enable and interpolation enable bits must also be set in LNCNTL.

4) Source extent in pixels and bytes; the source X extent in BYTES is loaded into SRCX. The source extent in pixels in X and Y calculates the DDA parameters along with the destination extents.

5) Data source type and destination ROP (raster operation) are required for all 2D operations. The only ones supported for resize are SRCCOPY: RDRAM to RDRAM and SRCCOPY: RDRAM to RDRAM with transparency mask. These are:
DRAWDEF = 00CCh, BLTDEF = 1111h and
DRAWDEF = 01CCh, BLTDEF = 111nh where
n = 1 for OP2 transparency source of RDRAM color,
n = 9 for OP2 transparency source of RDRAM color pattern,
n = 5 for OP2 transparency source of RDRAM mono, and
n = Dh for OP2 transparency source of RDRAM mono pattern.

6) When shrinking, SHRINKINC must be set. This points to the next line per pixel for a minor step. For all shrinks below 2:1, the value is '1'. The Y value is always 'n' based ('1' for a 2:1 shrink). X is 'n–1' when interpolation is enabled ('0' for a 2:1 shrink), or is 'n' when interpolation is disabled.

7) Destination extent in pixels is loaded into BLTEXTR_EX last to trigger the BitBLT.

4.1.2 Optional Parameters

When using transparency, set the OP2 pointers and Foreground/Background Color registers appropriately.

Table 4-1. Data Format for Resize Operations — LNCNTL 6:4

8-bpp palletized 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 packed	5
AccuPak™ 4:1:1	6

To calculate the DDA parameters for a given axis, it is necessary to know if the operation is to be a stretch or a shrink, and if interpolation/averaging is required for this axis. The interpolation control for each axis is written to the LNCNTL register, as is the bit that indicates to the DDA if the operation is a shrink or a stretch. The following code fragments show the calculations to determine the register settings for stretches and shrinks.

```
/*****************************************************************
*
* FUNCTION NAME:Stretch()
*
* DESCRIPTION:    For stretches, determine values for ACCUM_?, MAJ_?,
*                 and MIN_? registers. These values are NOT
*                 independent of interpolated or replicated
*                 stretches.? = X or Y as appropriate.
*
*****************************************************************/

static void Stretch (long lSrc, long lDst, char chCoord)
{
      // lSRC is the source extent in pixels.
      // lDst is the destination extent in pixels.
      // chCoord is either 'X' or 'Y' to lable the axis being
      // calculated.
      short sCount;

      /* For interpolated stretches registers values differ from
      * values for
      * replicated stretches
      */
      if ((('X'==chCoord)&&(0==(XINTERP_ENABLE & usLnCntl))) ||
                  (('Y'==chCoord)&&(0==(YINTERP_ENABLE & usLnCntl))))
      {
                  /* Compute ACCUM_?, MAJ_? and MIN_? for replicated
                   * stretch
                   * MAJ_?             = width (for x) or height (for y) of
                   *                   destination
                   * MIN_?             = negative of width (for x) or
                   *                   height (for y) of source
                   * ACCUM_?           = MAJ_? - 1 - (Dst%Src / (stretch
                   *                   factor + 1))
                   */

                  lRegVals[MIN] = -lSrc;
                  lRegVals[MAJ] = lDst;
                  lRegVals[ACCUM] = lRegVals[MAJ] - 1 - ((lDst % lSrc)
                              / ((lDst / lSrc) + 1)) ;
      }
      else
      {
                  /* Compute ACCUM_?, MAJ_? and MIN_? for interpolated
                   * stretch
                   * Interpolated stretches use bits 13 & 14 of ACCUM_?
                   * to determine
                   * whether to use pixel A, 3/4 A + 1/4 B, 1/2 A +
                   * 1/2 B or 1/4 A + 3/4 B.
                   * To set DDA values appropriately there are three
                   * choices.
                   * 1)        Set MAJ_? to 32k and scale MIN_? to keep
                   *           ratio approximately correct
                   *           MAJ_?   = 32k
                   *           MIN_?   = (negative of ratio of src/dst)
                   *                   scaled up to 32k
                   *           ACCUM_? = MAJ_? - 1 - ( 1/2 * Absolute
```



```
                        *                   difference of MAJ and MIN)
                        * 2)         Scale both src and dst appropriately such
                        *            that the ratio of src/dst
                        *            is preserved exactly.
                        *            Note:   In the following, the division is
                        *                    performed first thus there is a
                        *                    possibility of a rounding error
                        *                    which shows the difference between
                        *                    options 1 & 2.
                        *            MAJ_?   = (32k / dst) * dst
                        *            MIN_?   = (32k / dst) * src
                        *            ACCUM_? = MAJ_? - 1 - ( 1/2 * Absolute
                        *                    difference of MAJ and MIN)
                        * 3)         Scale both SRC and DST in such a manner as to
                        *            force the last pixels output on a line to
                        *            match the last source pixels rather than the
                        *            last source interpolated with the pixel past
                        *            the end of the line.
                        * Option 3 is used here.
                        *            Note: Options 1 and both oversample Src data
                        *            and so will replicate last pixel in X and
                        *            interpolate past end of data in Y
                        */

#if 0                               // Option 1
                    lRegVals[MIN] = -((_32K * lSrc) / lDst);
                    lRegVals[MAJ] = _32K;
#else
#if 0                               // Option 2
                    lRegVals[MAJ] = ((_32K / lDst) * lDst);
                    lRegVals[MIN] = -((_32K / lDst) * lSrc);

#else                               // Option 3
                    lDst          =       lDst * 4;
                    lSrc          =       lSrc * 4 - 3;
                    lRegVals[MAJ] = ((_32K / lDst) * lDst);
                    lRegVals[MIN] = -((_32K / lDst) * lSrc);
#endif
#endif
                    lRegVals[ACCUM] = lRegVals[MAJ] - 1 -
                    ((lRegVals[MAJ] % -lRegVals[MIN]) /
                    ( lDst/lSrc + 1) );
          }

          /* Write out computed register values */

          for (sCount = ACCUM; sCount < MAX_REGS; sCount++)
          {
                    Regs[sCount].szRegName[Regs[sCount].sCharOffset] =
                    chCoord;
                    printf(szScnWordFormat, Regs[sCount].szRegName,
                    lRegVals[sCount]);
          }
          printf("\n");
}
```

```
/*****************************************************************
*
* FUNCTION NAME: Shrink()
*
* DESCRIPTION:    For shrinks, determine values for SHRINKINC,
*                 ACCUM_?, MAJ_?, and MIN_? registers. These values
*                 are independent of averaged or decimated shrinks.
*                 ? = X or Y as appropriate.
*
*****************************************************************/

static void Shrink ( long lSrc, long lDst, char chCoord )
{
      // lSRC is the source extent in pixels.
      // lDst is the destination extent in pixels.
      // chCoord is either 'X' or 'Y' to lable the axis being
      // calculated.
      short sCount;

      /* Set SHRINKINC value,
       *          for y, SHRINKINC = ratio of src/dst
       *          for x, SHRINKINC = ratio of src/dst if not
       *          interpolating SHRINKINC = (ratio of src/dst minus
       *                      one) if interpolating
       * low byte for x coordinate
       * high byte for y coordinate
       */

      if ('X' == chCoord)
      {
                  sShrinkInc |= (lSrc / lDst);
                  if (XINTERP_ENABLE & usLnCntl)
                              sShrinkInc--;
      }
      else
                  sShrinkInc |= (lSrc / lDst) << 8;

      /* Compute ACCUM_?, MAJ_? and MIN_? values
      * MAJ_? = width (for x) or height (for y) of destination
      * MIN_? = negative of the remainder of src/dst
      * ACCUM_? = MAJ_? - 1 - ( Src%Dst / (shrink factor + 1))
      */

      lRegVals[MIN]            = -(lSrc % lDst);
      lRegVals[MAJ]            =  lDst;
      lRegVals[ACCUM] =  lRegVals[MAJ] - 1 - ((lSrc % lDst)
                  / ((lSrc / lDst) + 1)) ;

      /* Write out computed register values */
      for (sCount = ACCUM; sCount < MAX_REGS; sCount++)
      {

      Regs[sCount].szRegName[Regs[sCount].sCharOffset] = chCoord;
      printf(szScnWordFormat, Regs[sCount].szRegName,
      lRegVals[sCount]);
      }
      printf("\n");
}
```


The following is an example of stretching a 352 × 240 YUV 4:2:2 image from off-screen at x = 0 y = 768 to on-screen at x = 0, y = 0 with interpolation enabled in both axes. This operation assumes that the frame buffer is set to display YUV 4:2:2 data (FORMAT = 0x01A1A).

```
#####################################################################
# Src extents x = 352, y = 240
# Dst extents x = 1024, y = 768
# X Interpolation on
# Y Interpolation on
LNCNTL            0x0033               # x interp =1,y interp =1
BLTDEF            0x1110
DRAWDEF           0x00CC
ACCUM_X           0x729A
MAJ_X             0x8000
MIN_X             0xD418
ACCUM_Y           0x7609
MAJ_Y             0x7800
MIN_Y             0xDA9E
OP1_OPRDRAM       0x03000000           # Source x=0 y=768
OP0_OPRDRAM       0x00000000           # Destination x=0 y=0
SRCX              0x02C0               # Src extent in bytes = 352 * 2
BLTEXTR_EX        0x03000400           # Dest extent x=1024, y=768
#####################################################################
```

4.2 Resize BitBLTs, Occlusion, and 9-bit Frame Buffers

This section describes how to use the CL-GD546X for several unique operations. Resize BitBLTs, used for stretching/shrinking graphics or video images, are described. Several methods for implementing partially occluded rectangular regions are presented. A method for loading macro tables of BitBLT operations is also discussed.

If the whole frame buffer display area is not used in video format, several techniques are available to display mixed graphics and video data. This depends upon whether the system is configured with 9-bit RDRAMs or whether the display region is occupied by video wholly unoccluded by other windows.

Take an example of a frame buffer that has 9-bit RDRAMs and an unoccluded display window:

```
DRAWDEF                   0x04CC      # Set bit 10 PTAG to 1 to write into
                                      # the 9th RDRAM bit.
```

This example assumes that the following registers are set:

```
FORMAT                    0x141A      # 16 bpp Graphics RGB 16 bpp Video
                                      # YUV422
PTAG                      0x0001      # enable TAG_MASK bit
```

Assuming that all non-video 2D graphics operations are performed with DRAWDEF[10] (PixelTag) set to ‘0’, this ensures that the RAMDAC can distinguish between graphics and video data based upon the ninth bit. The only restriction of this mode is that video data must be dword-aligned and occupy complete dwords in the frame buffer.

With a 9-bit frame buffer and a window that is partially occluded, use a monochrome transparency mask to suppress writing data to occluded portions of the display. The mask must be the same height as the destination rectangle, and the same width in bits as the destination is in pixels. Using the above example, this is 128-bytes wide and 768-scanlines high.

For displayed regions, a zero bit is set in the mask to correspond to both 16-bit subpixels of the dword in the frame buffer. For occluded regions, a one bit is set for both halves of the dword. This implies that the minimum width of an occlusion must be two pixels. This requirement is imposed by the RAMDAC requiring both the U and V values to calculate the corresponding RGB values to output to the monitor. Modify the previous example as:

```
DRAWDEF                 0x05CC              # Set Transparency ON
BLTDEF                  0x1115              # Set Mono Src for OP2
OP2_OPRDRAM             0x00002000          # Point to mono mask defined
in
                                            # offscreen memory to right
side of
                                            # screen
```

NOTE: To define a monochrome mask as tall as the screen in systems with limited off-screen memory, it may be necessary to increase the RDRAM pitch of the display area. Do this by changing the Tiles_per_line field of the TILE_CTRL register. This moves some or all of the off-screen memory from below the screen to the right edge of the screen. In this case, a 1024×768 display at 16-bpp with 2 Mbytes of RDRAM, does not leave enough memory to capture the video image. Therefore, the video display is unoccluded or limited to substantially less than a full-screen size. The programmer can provide the monochrome mask from the host, but this may prove tedious to manage and will not work with auto-BitBLTs.

An alternate solution is to decompose the displayable portion of the window into a series of rectangles, then BitBLT each separately.

Without a 9-bit frame buffer, there are two choices.

1) Display one unoccluded rectangle on the screen in video format.

2) Use a software codec to convert the video format from the captured-video format to the frame-buffer-graphics format, followed by a resize operation on the converted data.

Using the resize hardware for scaling and interpolating the image after conversion is faster than resizing the image in software.

To display a single unoccluded rectangle in the video format requires programming the X_START_2, Y_START_2, X_END_2, and Y_END_2 registers to define the upper-left and lower-right corners of the window. The RAMDAC displays any data read from this region in the video format programmed into the FORMAT register.

4.3 Displaying 16-bit Video in an 8-bit Frame Buffer

Normally, the depth of the frame buffer for video data and for graphics data must be the same. However, there is one special mode that the CL-GD546X supports — displaying captured 16-bit data in an 8-bit frame buffer. This allows high-quality video with an inexpensive, and fast, 8-bit frame buffer. This mode requires slightly different programming than normal modes due to the different data widths between the two data types.

The video-capture process is the same, the major difference is the way the RAMDAC is forced to manage the data. It takes two bytes of data to produce each pixel output by the RAMDAC. Since it is necessary for the RAMDAC to maintain the same output frequency, it performs automatic pixel doubling for each video pixel it outputs. This means that when the data is being resized into the frame buffer, it must shrink to only half the width in pixels that the display area appears to occupy. For example, a window of 800 pixels in the 8-bit graphics screen is only 400 video pixels wide. The RAMDAC then applies a second pixel replicate in X when the data is displayed. Dependent upon the actual width of the region being used to display the video data, it is possible that the resize operation can be close to a 1:1 ratio. In some cases it can be a shrink operation in the X axis and a stretch operation in the Y axis.

The following code example considers the previous example of stretching a 352 $\times$ 240 image into a 1024 $\times$ 768 display that is now set up for 8-bit graphics pixels.

```
FORMAT                    0x001A      # (If VCLK > 85 MHz [1280x1024 etc.]
                                      # this will be 401Ah)


# Src extents x = 352, y = 240
# Dst extents x = 512, y = 768
# X Interpolation on
# Y Interpolation on
LNCNTL                    0x0033      # x interp =1,y interp =1
BLTDEF                    0x1110
DRAWDEF                   0x00CC
ACCUM_X                   0x6BE7
MAJ_X                     0x8000
MIN_X                     0xA830
ACCUM_Y                   0x7609
MAJ_Y                     0x7800
MIN_Y                     0xDA9E
OP1_OPRDRAM               0x03000000              # Source x=0 y=768
OP0_OPRDRAM               0x00000000              # Destination x=0 y=0
SRCX                      0x02C0                  # Src extent in bytes = 352 *
2
BLTEXTR_EX                0x03000400              # Dest extent x=(512*2),
y=768
```

There is a very subtle point in the destination extent in X being 1024 rather than 512. Normally, when a BitBLT-extent or a color-pixel pointer is written, it is byte-converted to the graphics format frame depth (in this case 8 bpp).

NOTE: The 512 YUV 4:2:2 pixels take up 1024 bytes of space and the resize engine is expecting a byte count as it decrements the count by the size of the pixels it is manipulating.

It is necessary to multiply the pixel extent of the BitBLT in X, by the ratio of the two different sizes. In the previous example using a 16-bpp frame buffer, the X extent was also 1024, this would have been byte-converted to 2048, then decremented by two for every pixel to yield an output of 1024 16-bit pixels.

The resulting image is not exactly the same as if the operation were performed and displayed using a 16-bit frame buffer; the image is only partially interpolated in X. The visual result, however, is almost indistinguishable.

The only other differences are in occlusion masks: when using 9-bit RDRAMs for the frame buffer, masks are now set to be the true pixel width of the stretched object rather than the apparent width after the RAMDAC has performed the pixel doubling. In this case, the mask, is 512-bits wide rather than 1024-bits wide. It is necessary that both halves of the dword remain masked together, implying that the minimum apparent occlusion width is 4 pixels.

4.4 Format Conversion Resize BitBLTs

The resize BitBLT engine of the CL-GD546X can output data in a different format from the input. To do this, the engine must be programmed using the STRETCH_CNTL register rather than the LNCNTL register.

The STRETCH_CNTL register duplicates all features of the LNCNTL register, except the 3-bit-wide-format field. This has been replaced with two fields, each 4-bits wide: one for source format and the second for destination format. By using these fields it is possible to convert any input RGB data type to any other RGB data type for output. It is also possible to convert any of the YUV input types to any of the YUV output types, or to any of the RGB output types. The engine does not support conversion of RGB data to YUV data. It is not possible to convert 8-bpp CLUT (palettized) data to any other type or to convert any RGB or YUV format to 8-bpp CLUT data and derive meaningful results. Similarly enabling interpolation/averaging on 8-bpp data only results in a meaningful image if the data is grayscale.

The RDRAM pointers for the source and destination are best written by the OP{0|1}_opMRDRAM registers after they are converted into byte addresses. This is because the normal RDRAM pointers are byte-converted using the frame buffer depth, which may not match either format.

Similarly, the MBLTEXTR_EX register should be used in place of the BLTEXTR_EX register, with the X portion of the extent converted into bytes according to the number of bytes-per-pixel of the destination type. When calculating stretch-shrink ratios and the associated MAJ/MIN/ACCUM values, perform the calculations using the pixel extents of the source and destination.

NOTE: The SRCX register must be programmed normally, with the byte extent of the source data.

The following example code shows the register programming for stretching an off-screen 352 $\times$ 240, YUV 4:4:4 image to an on-screen, RGB 5:6:5 image. It is displayed in an 8-bpp, 1024 $\times$ 768 frame buffer using backend pixel doubling by the ninth bit data.

NOTE: For proper operation, a minimum of a 1280-byte memory pitch is required because the source image buffer is 1056-bytes wide.


```
# Src extents x = 352, y = 240
# Dst extents x = 512, y = 768
# X Interpolation on
# Y Interpolation on

CHROMA_CNTL                0000h
STRETCH_CNTL               A203h

BLTDEF                    1010h
DRAWDEF                   04CCh

ACCUM_X                   6BE7h
MAJ_X                     8000h
MIN_X                     A830h
ACCUM_Y                   7609h
MAJ_Y                     7800h
MIN_Y                      DA9Eh

OP1_OPMRDRAM               04000080H              # Src Buffer at x=128  y=1024
OP0_OPMRDRAM               00000000H              # Destination onscreen at x=0
y=0
SRCX                       0420h                  # Src extent = 352*3

MBLTEXTR_EX               03000400h               # Dst extent = 512*2

This assumes Format = 001ah
                          PTAG  = 0001h (9th bit TAG_MASK)
#(set bit 13 of drawdef to enable mask loading before writing this register)
                          External_Overlay = 01h
                          Start_BLT_2 = C0h       # enable video format for 9th
bit
                          Stop_BLT_2 = 00h        # not used
#(use video format for window / enable window)
```

4.5 Chroma-Key Programming for Resize BitBLTs

The chroma-key function in the CL-GD546X supports source color key and source color key space functions in Microsoft DirectDraw and DirectVideo. The resize engine includes a flexible chroma-key comparator that operates on the data flowing through the engine to determine, on a per-pixel basis, the destination where pixels are overwritten. In addition, if the output format has an alpha field, the comparator selects which of two possible alpha values will fill the output pixel.

The chroma-key comparator is controlled by the CHROMA_CNTL register. The default state is disabled for compatibility with the CL-GD5462. The comparator has two stages. The results of the primary stage can be used alone or be combined with the secondary stage using either an AND or OR operation.

The primary stage has three comparators that operate individually on the three 8-bit channels of the resize engine. Each comparator channel can be individually enabled. When the channel is enabled, the data it contains is compared against the corresponding data in the CHROMA_LOWER and CHROMA_UPPER registers. The comparator generates a true result:

- If the data falls between the lower and upper bounds contained in the CHROMA_LOWER and CHROMA_UPPER registers *and* if the _IO_ bit for the comparator is set to '1'.
- If the _IO_ bit is set to '0' and the data falls outside the range defined.

Otherwise the comparator returns a false result. The data comparison range is inclusive of the endpoints. The results of enabled comparators is combined using either an AND operation – if the RGB_OA bit of the CHROMA_CNTL register is set to '0' – or an OR operation if this bit is set to '1'.

The data in the channel is normalized so that the MSB of each unpacked RGB or YUV field is in bit 7 of the comparator channel. That is if the input type is RGB 5:6:5, the RED channel bits 7:3 contain the red data from bits 15:11 of the input word; bits 2:0 of the RED channel are '000'. Similarly, the GREEN channel bits 7:2 contain bits 10:5 of the input data, and the BLUE channel contain bits 4:0 of the input in bits 7:3, with the unused bits set to '0'. All YUV inputs have Y in the RED channel, U in the GREEN channel, and V in the BLUE channel.

The following code example is for 'blue screen' chroma keying. In this example, only pixels in the source, that are not equal to the background of intense blue, are written. Pixels are written if either RED or GREEN are between 24 and 255, or if BLUE is between 0 and 224. Note these values are the same for any RGB data source format.

```
CHROMA_CNTL             8277h
CHROMA_LOWER            00181800h
CHROMA_UPPER            00FFFFE0h
```

The secondary stage is a single, 8-bit comparator that takes its data from any of the three data channels. The channel is selected by the SEC_SEL field (bits 11:10) of the CHROMA_CNTL register. The secondary data source is compared against bits 31:24 of the CHROMA_LOWER and CHROMA_UPPER registers.

The comparator can be switched from using the data unpacked from the frame buffer to using the unpacked data after conversion from YUV to RGB. Selecting the data source is controlled by setting the DATA_SELECT bit (bit 14) of the CHROMA_CNTL register to '0' for the original data, or to '1' to select the data after conversion.

NOTE: Setting the DATA_SELECT bit to '1' on non-YUV data generates meaningless results.

The previous programming example changes as follows, if the data source format is YUV. Note that the comparison ranges are the same as before.

```
CHROMA_CNTL             C277h
CHROMA_LOWER            00181800h
CHROMA_UPPER            00FFFFE0h
```

When the engine is only processing 8-bpp CLUT data, channel 0 (BLUE) contains ‘good data’, therefore only the BLUE channel is enabled.

When the engine is outputting data that includes an alpha-channel field, such as RGB A:8:8:8, the source of the alpha channel data is OP_opFGCOLOR[31:24] during regular chroma-key operations. This field is also normalized, for example in RGB 1:5:5:5 mode, OP_opFGCOLOR[31] fills the alpha channel.

NOTE: Alpha channel data can only be written by the engine. It does not pass through the engine.

The resize engine can be used in conjunction with the chroma-key function to set the alpha channel data of the output image to one of two different values, depending on the results of the chroma comparator. Program the comparator for the comparison values, but set the CHROMA_EN bit of the CHROMA_CNTL register to ‘0’. This allows separating an image into two regions of different alpha values for further processing within the 3D engine.

If the CHROMA_EN bit of CHROMA_CNTL is set to ‘0’ but the CHROMA_TAG_EN bit is set to ‘1’, the engine then uses OP_opFGCOLOR[23:16] to supply the alpha channel value for those pixels that were written during regular operations, and OP_opFGCOLOR[31:24] for those pixels that were transparent (not written) during regular operation. If both bits are set to ‘0’, then OP_opFGCOLOR[31:24] is used for all output pixels processed by the resize engine.

4.6 Programming Resize BitBLT Using Auto-BitBLT

The auto-BitBLT feature of the CL-GD546X allows a number of operations to be performed as a single operation. The primary purpose is to ensure that resize operations write data to the display surface, when possible, without interfering with the smooth display of data from the display surface. This is to prevent image tearing or banding.

A secondary feature is that the resize BitBLT, associated with window 2, is triggered purely by the external video data source connected to the V-Port (for example, an MPEG decoder, or a TV-standard converter or similar device) without additional software intervention.

The auto-BitBLT table header can store or combine commonly used BitBLT operations into MACRO BitBLT operations. The auto-BitBLT headers are written into RDRAM. The CL-GD546X reads the header from RDRAM when it is triggered, and loads the internal registers from the appropriate fields.

The auto-BitBLT headers associated with each video display window point to the RESIZE_A_OPRDRAM, RESIZE_B_OPRDRAM and RESIZE_C_OPRDRAM pointers for windows 1, 2, and 3, respectively. The RESIZE pointers also contain bits to arm the auto-BitBLT (bit 30 = 1) or, if this is a MACRO BitBLT operation, to immediately trigger the auto-BitBLT (bit 31 = 1).

There is also a master auto-BitBLT trigger/arm enable control bit in the CONTROL register (bit 8 == 1). This bit *must* be set for any auto-BitBLT or MACRO BitBLT operations to work.

The structure definition example below shows the layout of an auto-BitBLT header.

NOTE: This format is for the CL-GD5464 only and is *not* compatible with the CL-GD5462 structure.

```
typedef struct autoblt_regs {
              REG16       STRETCH_CNTL;          // Offset 0
              REG16       SHRINKINC;             // Offset 2
              REG32       DRAWBLTDEF;            // Offset 4
              REG32       FGCOLOR;               // Offset 8
              REG32       BGCOLOR;               // Offset 12
              REG32       OP0_opRDRAM;           // Offset 16
              WORD        MAJ_Y;                 // Offset 20
              WORD        MIN_Y;                 // Offset 22
              REG32       OP1_opRDRAM;           // Offset 24
              WORD        ACCUM_Y;               // Offset 28
              REG16       PATOFF;                // Offset 30
              REG32       OP2_opRDRAM;           // Offset 32
              WORD        MAJ_X;                 // Offset 36
              WORD        MIN_X;                 // Offset 38
              REG32       BLTEXT;                // Offset 40
              WORD        ACCUM_X;               // Offset 44
              WORD        OP0_opSRAM;            // Offset 46
              WORD        SRCX;                  // Offset 48
              WORD        OP2_opSRAM;            // Offset 50
              REG32       NEXT_HEAD;             // Offset 52
              REG32       CHROMA_LOWER;          // Offset 56
              REG32       CHROMA_UPPER;          // Offset 60
              WORD        CHROMA_CNTL;           // Offset 64
} autoblt_regs, *autoblt_ptr;
```

To write an auto-BitBLT header into the frame buffer, it can be either directly written, using the mapped linear address that corresponds to the xy location in the frame buffer, or copied, using a HOST BitBLT as a 68-byte-wide by 1-scanline-high color data object. See the Chapter 2, “2D Programmer’s Guide” for details of Host BitBLTs (Section 2.2.6).

NOTE: Fields that are not used in the Resize BitBLT should be programmed to ‘0’.

The STRETCH_CNTL register used in the auto-BitBLT header has three fields that are programmed to ‘0’ when the STRETCH_CNTL register is used in a manually-commanded Resize BitBLT. The first field is STR_RESIZE. If this is an auto-BitBLT header used for a Resize BitBLT, bit 6 is set to ‘1’; this bit is set to ‘0’ if this auto-BitBLT header is being used for a regular BitBLT operation (such as a MACRO BitBLT). The second field is bit 7 (STR_CHAIN_EN) that indicates to the CL-GD546X to load the auto-BitBLT header pointed to by NEXT_HEAD at the completion of this operation and execute it immediately. This allows synthesizing complex operations for MACRO BitBLTs or other video special effects.

When using an auto-BitBLT header, note the difference in the way pointers and BitBLT extents are byte-converted compared from when the pointers and BitBLT extents are normally loaded. The byte-conversion is performed using the data size associated with the data type specified in the STRETCH_CNTL field, rather than the frame buffer depth specified in the CONTROL register.

While performing 16-bpp video operations into an 8-bpp frame buffer using auto-BitBLT tables, the X portion of a color pointer or BitBLT extent must be one-half the value used if specified manually. This does not apply to monochrome object pointers.

When loading and immediately using the auto-BitBLT or MACRO BitBLT tables, the RESIZE pointers and control bits that arm and trigger them are 'Immediate' registers that bypass the Command FIFO to the 2D engine. That is, if a HOST BitBLT loads the tables, it is necessary to check that the operation completes (STATUS [2:0] = 000) before writing the RESIZE pointer. If the tables are written into RDRAM by the direct linear frame buffer interface, issuing a read to the frame buffer ensures that the written data is completely flushed from the write FIFO and that it is safe to write the RESIZE pointer.

4.7 Programming Safe Times for Auto-BitBLTs

The safe time for an auto-BitBLT to occur is when no part of the previous image is displayed from the display surface while the corresponding section of the new image is being loaded.

To calculate a safe time for an auto-BitBLT, it is necessary to determine two things: the refresh rate of the display being driven, and the time to complete the update operation.

For example, a 1024 × 768 display that is refreshed at 75 Hz. This means that any part of the frame buffer is displayed at 13.3-ms intervals. If the operation is the full-screen stretch of the above examples (Section 4.3), this is likely to take about 16.3 ms. To calculate this figure, multiply the X extent of the destination in pixels by the Y extent of the destination in scanlines and divide by 48 million pixels-per-second.

NOTE: This is preliminary data assuming 62.5 MHz for MCLK and approximately 22% overhead for fetch/latency during resize operations. The actual figure depends upon the selected MCLK.

As the operation takes longer than the display refresh cycle, start the auto-BitBLT soon after the start of the display cycle so that when the next display refresh cycle starts reading, it is writing as close to the bottom of the display area as possible. As this operation takes approximately 25% longer than the display cycle, do not command the operation to start if the display cycle is more than 75% complete. This implies that the START_BLT_{1|2|3} can be programmed to 'C0h' and that STOP_BLT_{1|2|3} can be programmed to '09h' (bits 5:0 = 75% × 768 ÷ 64).

If the operation being performed takes a shorter period than a display cycle, ensure that the display count is far enough ahead of where the destination to start writing output data for the operation. This prevents an overwrite of data as the display refresh outputs to the display. If using the same display rate and performing a stretch into a 700 × 480 region located at a Y offset of 120 scanlines, then it takes approximately 7 ms to perform the operation. This is 50% of a display cycle, therefore the operation can only start when the vertical count reaches C3h {[480 + 120 – (50% × 768)] ÷ 64 rounded up}. Also, the time to not start the operation is when the display refresh count is lower than the result of this equation. Set STOP > START, therefore it is best to set STOP = C0h [768 ÷ 64 = 12].

NOTE: The calculation of resize times is, to a large extent, independent of the size of the source region that is, stretching a 160 × 120 region to 1024 × 768 requires the same amount of time (±1%) as it takes to stretch a 360 × 240 region to 1024 × 768.

4.8 Programming Considerations for Odd/Even Field Captured Data

When data is captured from a TV-standard source using a decoder device, it can be captured into two separate capture buffers for the Odd and Even frames. To do this, set the Double Buffering bit in the V-Port register to '1', and set the X_START_EVEN, X_START_ODD, Y_START_EVEN and Y_START_ODD registers to point to the respective buffers. If the RESIZE_B_OPRDRAM pointer is set to point to an even scanline address, the auto-BitBLT triggered (that is, when the captured frame is complete and the START_BLT_2 and STOP_BLT_2 considerations are met) is different for the two captured frames. The Odd frame (which is always the first one to be captured) triggers the auto-BitBLT at the RDRAM pointer. The Even frame triggers the auto-BitBLT from the odd scanline at RDRAM.pt.Y + 1. This allows implementation of several different data handling techniques.

The simplest technique is to resize the two separate frames alternately into the same display region. However, this may not be a viable solution if the display region is large, because the interlaced frames are arriving at 16.6-ms intervals, if the source is NTSC or 20-ms intervals from PAL/SECAM. Any resize operation that takes longer causes skipped frames. Any resize operation that can be safely triggered within a narrow timing window is likely to cause skipped frames.

This technique may not be the most visually appealing solution, especially if the display frame is frozen as only half of the available data for any given frame is used to construct the frozen image. A better solution is to combine the Odd and Even frames into a single, composite frame, then resize the composite frame onto the display surface. Although this solution requires a considerable amount of off-screen memory, it allows high-frame-rate monitors to display video images without tearing or banding at high-resolution.

To perform a replicate stretch of the Odd frame into an off-screen buffer that is the same width as the capture buffer but twice the height, combine the two frames. This doubles all scanlines. Assuming that the captured frame in this case is YUV 4:2:2 at 360 × 240, this operation takes 3.6 ms. The synchronization requirements are not as critical as the data is still off-screen.

When the Even frame completes, it triggers a two auto-BitBLT chain. The first auto-BitBLT performs a similar replicate stretch of the Even frame into the same buffer as the Odd frame, but uses a monochrome pattern as a transparency mask (pattern = FF00FF00FF00FF00h). This pattern allows only alternate scanlines of the output to be written into the off-screen buffer. On completion of the first auto-BitBLT of the even-frame auto-BitBLT chain, the CL-GD546X uses the NEXT_HEAD pointer to immediately load and execute a second Resize BitBLT.

The second auto-BitBLT stretches the 360 × 480 buffer onto the display surface at the required resolution. This auto-BitBLT performs the pixel interpolation required to smooth the display output. The timing requirements are calculated for the second part of this chain to complete without tearing the screen. The values programmed into the START_BLT_2 and STOP_BLT_2 registers are then offset to allow the 3.6-ms first operation to complete prior to the critical window.

For example, assume that the display is running 1280 × 1024 × 8-bpp at 75 Hz in a 2-Mbyte frame buffer. Display timing requires screen refresh every 13.3 ms. To stretch the data onto the entire screen surface requires a 640 × 1024 stretch using pixel doubling in the RAMDAC. This allows 1280 pixels to be displayed from only 1280 bytes of the YUV 4:2:2 source.

NOTE: There is insufficient memory in the system to allow 1280 × 1024 × 16-bpp. This requires more than 2.5 Mbytes.

The stretch operation takes 13.65 ms. Adding the 3.6 ms for the first half of the operation allows a total time of 17.25 ms for the even field auto-BitBLT. This is approximately 4 ms longer than the

refresh cycle for the display, or approximately 25%. The programmer can allow this operation to start anywhere in the first 75% of the display region by setting START_BLT_2 to ‘C0h’ and STOP_BLT_2 to ‘0Ch’ (75% × 1024 ÷ 64).

The following code example illustrates the above stretch operation. The frame buffer is configured to use a 2048-byte memory pitch with the off-screen buffers located at X = 1280.

```
#####################################################################
BLTDEF                  0x1020              # OP1/host -> RDRAM
                                            # color
DRAWDEF                 0x00CC              # rop:SRCCOPY
#--------------------------------------------------------------------
# auto BLT table 1: This is for Odd field capture, stretch the Odd
# frame into the Merge buffer.
OP0_opMRDRAM.pt.X       0x0500
OP0_opMRDRAM.pt.Y       0x03C0
OP1_opMRDRAM.pt.X       0x0000              # phase = 0
MBLTEXT_EX.pt.X         0x0044
MBLTEXT_EX.pt.Y         0x0001
HOSTDATA                0x00009940          # STRETCH_CNTL/SHRINKINC
HOSTDATA                0x101000CC          # DRAWBLTDEF
HOSTDATA                0x00000000          # OP_opFGCOLOR
HOSTDATA                0x00000000          # OP_opBGCOLOR
HOSTDATA                0x01E00280          # OP0_opRDRAM
HOSTDATA                0xFF1001E0          # MAJ_Y/MIN_Y
HOSTDATA                0x00000280          # OP1_opRDRAM
HOSTDATA                0x000001DF          # ACCUM_Y/PATOFF
HOSTDATA                0x00000000          # OP2_opRDRAM
HOSTDATA                0xFE980168          # MAJ_X/MIN_X
HOSTDATA                0x01E00168          # BLTEXT
HOSTDATA                0x00000167          # ACCUM_X/OP0_opSRAM
HOSTDATA                0x000002D0          # SRCX/OP2_opSRAM
HOSTDATA                0x00000000          # next head, Not used as
                                            # used as LN_CHAIN_EN = 0
HOSTDATA                0x00000000          # CHROMA
HOSTDATA                0x00000000          # NOT
HOSTDATA                0x00000000          # USED
```

```
#---------------------------------------------------------------
# auto BLT table 2: This is for Even field capture, Merge the Odd and
# Even frames in the Merge buffer.
OP0_opMRDRAM.pt.X       0x0500
OP0_opMRDRAM.pt.Y       0x03C1
OP1_opMRDRAM.pt.X       0x0000              # phase = 0
MBLTEXT_EX.pt.X         0x0044
MBLTEXT_EX.pt.Y         0x0001
HOSTDATA                0x000099C0          # STRETCH_CNTL/SHRINKINC
HOSTDATA                0x101D01CC          # DRAWBLTDEF
HOSTDATA                0x00000000          # OP_opFGCOLOR
HOSTDATA                0x00000000          # OP_opBGCOLOR
HOSTDATA                0x01E00280          # OP0_opRDRAM
HOSTDATA                0xFF1001E0          # MAJ_Y/MIN_Y
HOSTDATA                0x00F00280          # OP1_opRDRAM
HOSTDATA                0x000001DF          # ACCUM_Y/PATOFF
HOSTDATA                0x03C129C0          # OP2_opMRDRAM
HOSTDATA                0xFE980168          # MAJ_X/MIN_X
HOSTDATA                0x01E00168          # BLTEXT
HOSTDATA                0x00000167          # ACCUM_X/OP0_opSRAM
HOSTDATA                0x000002D0          # SRCX/OP2_opSRAM
HOSTDATA                0x03C20500          # next head
HOSTDATA                0xFF00FF00          # this is the
                                            # transparency pattern
HOSTDATA                0xFF00FF00          # second dword of
                                            # Xpar. pattern
HOSTDATA                0x00000000          # Chroma key not used. Note
                                            # CHROMA_UPPER and
CHROMA_LOWER
                                            # spaces used by transparency
                                            # pattern
```



```
#-----------------------------------------------------------------


# auto BLT table 3: This is for Even field capture, stretch merged
# frame to display
OP0_opMRDRAM.pt.X        0x0500
OP0_opMRDRAM.pt.Y        0x03C2
OP1_opMRDRAM.pt.X        0x0000                # phase = 0
MBLTEXT_EX.pt.X          0x0044
MBLTEXT_EX.pt.Y          0x0001
HOSTDATA                 0x00009943            # STRETCH_CNTL/SHRINKINC
HOSTDATA                 0x101004CC            # DRAWBLTDEF
HOSTDATA                 0x00000000            # OP_opFGCOLOR
HOSTDATA                 0x00000000            # OP_opBGCOLOR
HOSTDATA                 0x00000000            # OP0_opRDRAM
HOSTDATA                 0xC4188000            # MAJ_Y/MIN_Y
HOSTDATA                 0x01E00280            # OP1_opRDRAM
HOSTDATA                 0x00007D45            # ACCUM_Y/PATOFF
HOSTDATA                 0x00000000            # OP2_opMRDRAM
HOSTDATA                 0xBCA47800            # MAJ_X/MIN_X
HOSTDATA                 0x04000280            # BLTEXT
HOSTDATA                 0x00005DAD            # ACCUM_X/OP0_opSRAM
HOSTDATA                 0x000002D0            # SRCX/OP2_opSRAM
HOSTDATA                 0x00000000            # next head
HOSTDATA                 0x00000000
HOSTDATA                 0x00000000
HOSTDATA                 0x00000000
#################################################################
```

```
FORMAT                  0x401A          # set for 8 bpp frame
                                        # buffer and 16 bpp
                                        # video
VPORT                   0x006A          # CL-PX4072 attached in
                                        # 16 bit mode???
X_START_ODD             0x0500          # point to the Odd
                                        # capture buffer
Y_START_ODD             0x0000
X_START_EVEN            0x0500          # point to the Even
                                        # capture buffer
Y_START_EVEN            0x00F0
START_BLT_2             0xC0            # start at top of frame
STOP_BLT_2              0x0C            # do not start if > 75%
                                        # displayed
X_START_2               0x0000
Y_START_2               0x0000          # start of WINDOW 2
                                        # display (for 8 bit
                                        # systems)
X_STOP_2                0x0500
Y_STOP_2                0x0400          # end of WINDOW 2
                                        # display
RESIZE_B_opRDRAM        0x43C00500      # Arm the Auto BLTs.
```

