3D*labs*®

PERMEDIA® 3

Errata and Alerts

Issue 5

Proprietary Notice

Email: info@3dlabs.com
Web: http://www.3dlabs.com

3D*labs* Ltd.
Meadlake Place
Thorpe Lea Road, Egham
Surrey, TW20 8HE
United Kingdom
Tel: +44 (0) 1784 470555
Fax: +44 (0) 1784 470699

3D*labs* K.K
Shiroyama JT Mori Bldg 16F
40301 Toranomon
Minato-ku, Tokyo, 105,
Japan
Tel: +81-3-5403-4653
Fax: +91-3-5403-4646

3D*labs* Inc.
480 Potrero Avenue
Sunnyvale, CA 94086,
United States
Tel: (408) 530-4700
Fax: (408) 530-4701

Change History

Document	Issue	Date	Change
157.3.0	1	29 Feb 99	First Issue.
157.3.0	2	19 Mar 99	Layout and typos corrected.
157.3.0	3	23 Apr 99	Programming alert - DMA
157.3.0	4	29 June 99	Programming alert - *CombineCaches*
157.3.0	5	30 Jan 2000	PEREN0017 - Horizontal display resolution

Contents

1 Introduction

The following stepping information is for the **PERMEDIA 3** Graphics Accelerator Chip. There is an erratum for each known problem containing a detailed description and suggested workarounds.

1.1 PERMEDIA 3 Associated Documentation

This document should be read in conjunction with:

- **PERMEDIA3** Architecture Overview, Issue 5
- **PERMEDIA3** Reference Guide, Issue 2

1.2 PERMEDIA 3 Identification

A **PERMEDIA 3** may be identified by two means. The first is the physical markings on the part itself, the second is by reading the Vendor ID, Device ID and Revision ID Registers in PCI Configuration Region. Please refer to the **PERMEDIA 3** Reference Guide for further details.

Part Marking	Vendor ID	Device ID	Revision ID
PERMEDIA 3	3D3Dh	000Ah	0001h

1.3 Timing Values

All timing values referred to in this document apply across the full range of operating conditions as specified in the **PERMEDIA 3** Reference Guide.

1.4 Software Drivers and Reference Designs

Please note that all 3Dlabs supplied software drivers and reference designs include the appropriate bug fixes and workarounds as described in this document.

2

Errata Summary

Errata No.	Type	Reference	Revision ID
PEREN001	Device Errata	Video Streams	0001h
PEREN002	Device Errata	YUV planar to packed through bypass	0001h
PEREN003	Device Errata	Memory frequency dependency	0001h
PEREN004	Device Errata	Host-in DMA	0001h
PEREN005	Device Errata	Video Overlay line length restriction	0001h
PEREN006	Device Errata	Video Overlay de-interlacing	0001h
PEREN007	Device Errata	Constant color spans	0001h
PEREN008	Device Errata	Write DMA frequency dependency	0001h
PEREN009	Device Errata	GPOut write DMA	0001h
PEREN0010	Device Errata	PCIAbortStatus register	0001h
PEREN0011	Device Errata	GPInFifo space	0001h
PEREN0012	Device Errata	Video unit line patched doubling	0001h
PEREN0013	Device Errata	RAMDAC pan	0001h
PEREN0014	Device Errata	RAMDAC cursor	0001h
PEREN0015	Device Errata	Render2D register	0001h
PEREN0016	Device Errata	Invalidate texture cache	0001h

3 Errata Details

3Dlabs have extensive experience and a proven track record in delivering high performance, high quality, ready-to-ship WHQL certified software drivers that extract the maximum performance from both the **PERMEDIA 3** processor and the entire system.

3.1 PEREN001 - Video Streams

3.1.1 Problem

Operation of input and output video streams is incorrect. Modes 1, 2, 3 and 4 of the VSConfiguration register mode field do not function correctly and should not be used. Access to the ROM (mode 0) and the flat panel display (mode 6) do function correctly and can be used.

3.1.2 Software Workaround

None

3.2 PEREN002 - YUV planar to packed through bypass

3.2.1 Problem

Data in the YUV planar format (the internal format of MPEG2 data) cannot be converted to packed YUV (the format used for processing) while being written to the framebuffer through the bypass.

3.2.2 Software Workaround

Planar data should be converted to packed before being written to the framebuffer. DirectX normally does the conversion so this errata should have no effect for DirectX systems.

3.3 PEREN003 - Memory frequency dependency

3.3.1 Problem

The memory clock should not be set to run faster than the graphics processor clock. The graphics core may not function correctly if the frequency of the memory clock is greater than the frequency of the graphics processor clock. The memory clock is controlled by the MClkControl register, and the graphics processor clock by KClkControl and associated PLL registers.

3.3.2 Software Workaround

Under normal conditions, the memory clock should be tied to the graphics processor clock. If slow speed memories are used, the memory clock may be run from, for example, an external clock with a frequency lower than the graphics processor clock. Care should be taken when using the power saving mode of the graphics processor clock as it may result in it having a lower frequency than the memory clock.

3.4 PEREN004 - Host-in DMA

3.4.1 Problem

The host-in lightweight DMA mechanism may function incorrectly. There are three problems with the DMAContinue command:

1. The DMAError interrupt may be raised incorrectly and should be ignored.
2. If a DMACount command of zero is sent before a DMAContinue command, the DMAContinue will be applied to the previous DMACount.
3. There is a double buffering mechanism which allows new DMAAddress and DMACount values to be loaded before the current pair have completed without taking any room in the input FIFO. DMAContinue commands following this should not take any room in the input FIFO but do, and may fill the FIFO in consequence.

3.4.2 Software Workaround

Check the FIFO space before sending lightweight DMA commands. A DMACount should never be sent with a value of zero.

3.5 PEREN005 - Video Overlay line length restriction

3.5.1 Problem

The VideoOverlay Unit generates incorrect output when either:

a) The width of the image into the zoom-filter (i.e. after any x-shrink has been applied) is not an exact multiple of 4 pixels.

b) The X Shrink and Zoom deltas yield a final X coordinate which is part-way through the final source pixel (i.e. adding the value of zoom-delta one more time does not step to the next source pixel).

3.5.2 Software Workaround

The VideoOverlay X delta values must be carefully chosen to avoid the above conditions. This can be achieved by the C function 'compute_ovr_params' listed below:

```
#define valid_width(w)      ((w & 3) == 0)
#define make_valid_width(w) ((w) & ~0x3)
#define width_step          4

/* Forward declarations */
static int compute_best_fit_delta(unsigned long *src_dimension,
                                  unsigned long  dest_dimension,
                                  unsigned long filter_adj,
                                  unsigned long int_bits,
                                  unsigned long *best_delta);
static int find_zoom(unsigned long  src_width,
                     unsigned long* shrink_width,
                     unsigned long  dest_width,
                     unsigned long* zoom_delta);


/* Function to compute overlay X deltas suitable for scaling from    */
/* src_width to dest_width. May adjust the source width slightly in  */
/* order to meet required destination width. Final adjusted width is */
/* returned in ovr_w                                                 */

static void compute_ovr_params(
  unsigned long  src_width,    unsigned long  dest_width,
  unsigned long *ovr_shrinkxd, unsigned long *ovr_zoomxd,
  unsigned long *ovr_w)
{
  unsigned long sx_adj = 0;
  const unsigned long fixed_one = 0x00001000;
  int zoom_ok;
  unsigned long adj_src_width;
  unsigned long exact_shrink_xd;
  unsigned long exact_zoom_xd;
```

```
adj_src_width = src_width + 1; /* +1 accounts for – below */

/*                                                                  */
/* Use the source and destination rectangle dimensions to compute */
/* delta values.                                                    */
/*                                                                  */

 unsigned long shrink_width;

 /* Step to next source width */
 adj_src_width--;

 /* Make a stab at the deltas for the current source width */

 /* Initially, the deltas are assumed to be 1, and the width due to */
 /* shrinking is therefore equal to src width                       */
 shrink_width    = adj_src_width;
 exact_shrink_xd = fixed_one;
 exact_zoom_xd   = fixed_one;

 /* Compute the shrink width and delta required */
 if (dest_width < adj_src_width) {
   /* Shrink */
   exact_shrink_xd =
     (unsigned long)((((float)(adj_src_width - sx_adj) /
                       (float)(dest_width)) * (1<<12)) + 0.999f);

   shrink_width =
     (unsigned long)((adj_src_width - sx_adj) /
                    ((float)(exact_shrink_xd) / (1<<12)));

 }

 /* Truncate shrink to valid width */
 if (!valid_width(shrink_width) && (shrink_width > 4)) {
   shrink_width = make_valid_width(shrink_width);

   exact_shrink_xd =
     (unsigned long)((((float)(adj_src_width - sx_adj) /
                      (float)(shrink_width)) * (1<<12)) + 0.999f);
 }

 /* Compute any zoom delta required */
 zoom_ok = 1;
 if (shrink_width < dest_width) {
   /* Make an attempt at a zoom-delta & shrink-width for this src width */
   zoom_ok = find_zoom(adj_src_width, &shrink_width, dest_width,
                       &exact_zoom_xd);

   /* Compute final shrink delta from returned shrink width */
   exact_shrink_xd =
    (unsigned long)((((float)(adj_src_width - sx_adj) /
                     (float)(shrink_width)) * (1<<12)) + 0.999f);
 }

 *ovr_zoomxd       = exact_zoom_xd;
 *ovr_shrinkxd     = exact_shrink_xd;
```

```
  *ovr_w            = adj_src_width;
}
```

```
/* Function to calculate a 12.12 delta value to provide scaling from     */
/* a src_dimension to the target dest_dimension.                         */
/* The dest_dimension is not adjustable, but the src_dimension may be    */
/* adjusted slightly, so that the delta yields a more accurate value for */
/* dest.                                                                 */
/* filter_adj should be set to 1 if linear filtering is going to be      */
/* enabled                                                               */
/* during scaling, and 0 otherwise.                                      */
/* int_bits indicates the number of bits in the scaled delta format      */
static int compute_best_fit_delta(unsigned long *src_dimension,
                                  unsigned long  dest_dimension,
                                  unsigned long filter_adj,
                                  unsigned long int_bits,
                                  unsigned long *best_delta) {

  int result = 0;
  float fp_delta;
  float delta;
  unsigned long delta_mid;
  unsigned long delta_down;
  unsigned long delta_up;
  float mid_src_dim;
  float down_src_dim;
  float up_src_dim;
  float mid_err;
  float mid_frac;
  int   mid_ok;
  float down_err;
  float down_frac;
  int   down_ok;
  float up_err;
  float up_frac;
  int   up_ok;
  int  itemp;

  /* The value at which a scaled delta value is deemed too large */
  const unsigned int max_scaled_int = (1 << (12+int_bits));

  /* Calculate an exact floating point delta */
  fp_delta = (float)(*src_dimension - filter_adj) / dest_dimension;

  /* Calculate the scaled representation of the delta */
  delta = (fp_delta * (1<<12));

  /* Truncate to max_int */
  if (delta >= max_scaled_int) {
    delta = (float)(max_scaled_int - 1); /* Just below overflow value */
  }

  /* Calculate the scaled approximation to the delta */
  delta_mid = (unsigned long)delta;

  /* Calculate the scaled approximation to the delta, less a 'bit' */
  /* But don't let it go out of range                              */
  delta_down = (unsigned long)delta;
  if (delta_down != 0) {
```

```
  delta_down --;
}

/* Calculate the scaled approximation to the delta, plus a 'bit' */
/* But don't let it go out of range                              */
delta_up = (unsigned long)delta;
if ((delta_up + 1) < max_scaled_int) {
  delta_up ++;
}
```

```
/* Recompute the source dimensions, based on the dest and deltas */
mid_src_dim =
  (((float)(dest_dimension - 1) * delta_mid) / (1<<12)) + filter_adj;

down_src_dim =
  (((float)(dest_dimension - 1) * delta_down) / (1<<12)) + filter_adj;

up_src_dim =
  (((float)(dest_dimension - 1) * delta_up)   / (1<<12)) + filter_adj;


/* Choose the delta which gives final source coordinate closest to  */
/* target, while giving a fraction 'f' such that (1.0 - f) <= delta */

mid_err   = fabs(mid_src_dim - *src_dimension);
itemp     = (unsigned long)mid_src_dim;
mid_frac  = mid_src_dim - itemp;
mid_ok    = ((1.0 - mid_frac) <= ((float)(delta_mid) / (1<<12)));

down_err  = fabs(down_src_dim - *src_dimension);
itemp     = (unsigned long)down_src_dim;
down_frac = down_src_dim - itemp;
down_ok   = ((1.0 - down_frac) <= ((float)(delta_down) / (1<<12)));

up_err    = fabs(up_src_dim - *src_dimension);
itemp     = (unsigned long)up_src_dim;
up_frac   = (up_src_dim - itemp);
up_ok     = ((1.0 - up_frac) <= ((float)(delta_up) / (1<<12)));

if (mid_ok && (!down_ok || (mid_err <= down_err)) &&
              (!up_ok   || (mid_err <= up_err))) {
  *best_delta = delta_mid;
  itemp = (unsigned long)((mid_src_dim + ((float)(delta_mid) / (1<<12))));
  *src_dimension = (unsigned long)(itemp - filter_adj);

  result = 1;
}
else if (down_ok && (!mid_ok || (down_err <= mid_err)) &&
                    (!up_ok  || (down_err <= up_err))) {
  *best_delta = delta_down;
  itemp = (unsigned long)((down_src_dim + ((float)(delta_down) / (1<<12))));
  *src_dimension = (unsigned long)(itemp - filter_adj);

  result = 1;
}
else if (up_ok && (!mid_ok  || (up_err <= mid_err)) &&
                  (!down_ok || (up_err <= down_err))) {
  *best_delta = delta_up;
  itemp = (unsigned long)((up_src_dim + ((float)(delta_up) / (1<<12))));
  *src_dimension = (unsigned long)(itemp - filter_adj);
  result = 1;
}
else {
  result = 0;
  *best_delta = delta_mid;
  itemp = (unsigned long)((mid_src_dim + ((float)(delta_mid) / (1<<12))));
```

```
      itemp = (unsigned long)((itemp - filter_adj) + 0.9999f);
      *src_dimension = (unsigned long)itemp;
    }


    return result;
  }
```

```
/* Find a suitable zoom delta for the given source                    */
/* the source image may be adjusted in width by as much as 8 pixels to */
/* acheive a match                                                    */

static int find_zoom(unsigned long  src_width,
                     unsigned long* shrink_width,
                     unsigned long  dest_width,
                     unsigned long* zoom_delta) {
  int zoom_ok;
  int zx_adj = 0;

  /* Find zoom for requested width */
  unsigned long trunc_width = make_valid_width(*shrink_width);
  zoom_ok = compute_best_fit_delta(&trunc_width, dest_width, zx_adj, 1,
                                   zoom_delta);

  /* If no zoom was matched for requested width, start searching up/down */
  if (!zoom_ok || (!valid_width(trunc_width))) {
    unsigned long up_width   = make_valid_width(trunc_width) + width_step;
    unsigned long down_width = make_valid_width(trunc_width) - width_step;

    int done_up = 0;
    int done_down = 0;
    do {
      /* Check upwards */
      zoom_ok = 0;
      if (up_width < dest_width) {
        unsigned long new_width = up_width;
        zoom_ok =
          compute_best_fit_delta(&new_width, dest_width, zx_adj, 1,
                                 zoom_delta);

        /* If the above call somehow adjusts width to invalid, */
        /* mark the delta invalid                              */
        if (!valid_width(new_width)) {
          zoom_ok = 0;
        }

        if (zoom_ok) {
          *shrink_width = new_width;
        }
        else {
          up_width += width_step;
        }
      }
      else
        done_up = 1;

      /* Check downwards */
      if (!zoom_ok && (down_width >= 4) && (down_width < src_width)) {
        unsigned long new_width = down_width;
        zoom_ok =
          compute_best_fit_delta(&new_width, dest_width, zx_adj, 1,
                                 zoom_delta);

        /* If the above call somehow adjusts width to invalid, */
```

```
            /* mark the delta invalid                                    */
            if (!valid_width(new_width)) {
              zoom_ok = 0;
            }

            if (zoom_ok) {
              *shrink_width = new_width;
            }
            else {
              down_width -= width_step;
            }
          }
        else
          done_down = 1;
      } while (!zoom_ok && (!done_up || !done_down));
    }

    return zoom_ok;
  }
```

3.6 PEREN006 - Video Overlay de-interlacing

3.6.1 Problem

The VideoOverlay does not generate the correct number of scanlines for odd video fields when bob-deinterlace is enabled. This causes image skewing on the display.

3.6.2 Software Workaround

None. Do not use bob-deinterlacing with the VideoOverlay unit.

3.7 PEREN007 - Constant color 8 and 32 bpp span rendering

3.7.1 Problem

Constant color span operations on 8bpp or 32bpp framebuffers do not work correctly and omit some pixels.

3.7.2 Software Workaround

A simple software workaround for 8bpp rendering is to avoid using spans and render pixels normally. A more optimal solution is to treat the framebuffer as 16bpp and render the interior pixels like this. The edge pixels where scanlines start on an odd byte boundary or end on an odd byte boundary will need to be filled in using normal i.e. non-span rendering.

The recommended software workaround for 32bpp rendering is to switch to 16 bit rendering and adjust the start and end pixels to cover the same number of memory words.

For example if the span for 32bpp starts at coordinate 100 and ends at 250, then in 16bpp rendering set the start to coordinate 200 and end to 500. The block color should be set up as for 32bpp. Other than the few cycle cost of changing the pixel depth, there is no loss of fill rate.

3.8 PEREN008 - Write DMA frequency dependency

3.8.1 Problem

The bypass write DMA controller is used to transfer data from the framebuffer to system memory. If the memory clock is operating at a higher frequency than the PCI clock, the DMA may not operate correctly. As the highest PCI clock is 66MHz, it is normal for the memory clock to be faster, so this DMA controller should not be used.

3.8.2 Software Workaround

Bypass write DMA should not be used. Instead, use the CPU to read from the framebuffer, or use the graphics processor to DMA data.

3.9 PEREN009 - GPOut write DMA

3.9.1 Problem

The GPOutDMA address register in region zero does not return the next DMA Address to be issued to the DMA arbiter when read. The PCIFeedbackCount register does not return the number of DWORDs transferred in the current DMA. This means that operations such as rectangular write DMA do not work correctly.

3.9.2 Software Workaround

Legacy output DMA, as used by PERMEDIA 2 drivers, should be used instead.

3.10 PEREN0010 - PCIAbortStatus register

3.10.1 Problem

After a PCI master Abort, the aborting address and DMA source can be read from the PCIAbortAddress register and PCIAbortStatus register in region zero. Once set, the PCIAbortStatus register cannot be cleared by writing to the register.

3.10.2 Software Workaround

None. A PCI Master abort is a potentially fatal occurrence. However, this will only occur due to a serious bug in the driver software.

3.11 PEREN0011 - GPInFifo space

3.11.1 Problem

The InFIFOSpace register can report erroneous space values when the space in the input FIFO is greater than 120 items.

3.11.2 Software Workaround

When the InFIFOspace register is read, the value must be clamped to a maximum of 120, before it is used.

3.12 PEREN0012 - Video unit line patched doubling

3.12.1 Problem

The video unit does not support line doubling with a patched framebuffer. The video unit supports line doubling for situations where the resolution of the display is so low that a monitor has trouble locking to it. The video unit also supports a patched framebuffer which can give better drawing performance on high resolution displays. These modes are not independent and should not be used at the same time.

3.12.2 Software Workaround

None. Do not attempt line doubling with a patched framebuffer. This is not serious as line doubling is typically only used with screen resolutions of width <= 512 pixels.

3.13 PEREN0013 - RAMDAC pan

3.13.1 Problem

The RAMDAC can pan to 64 bit resolution, but not to 32 bits. A RAMDAC is normally required to pan to an accuracy of four pixels. The RAMDAC can pan to the nearest 64 bits which is suitable for 32 and 16 bit displays, but not 8 bit displays. Attempting to pan to 32 bits may cause the horizontal sync to change position within the blank, which may result in the display shifting position.

3.13.2 Software Workaround

If the byte double mode (video unit, misc control register) is used, each 8 bit pixel is issued twice so to pan to 4 pixels only requires 64 bit accuracy.

3.14 PEREN0014 - RAMDAC cursor

3.14.1 Problem

Once enabled, the hardware cursor cannot be disabled by switching it off in the RDCursorMode register.

3.14.2 Software Workaround

Hide the cursor by moving it off-screen when it is not needed.

3.15 PEREN0015 - Render2D Register

3.15.1 Problem

The Render2D command does not always render the last pixel(s). This occurs under the following conditions:

- the Render2D command has the Operation field set to PatchOrderRendering,
- the rectangle width is less than the patch width in pixels (i.e. < 64 for 32 bpp, < 128 for 16 bpp or < 256 for 8 bpp)
- and the rectangle does not cross a patch boundary in X

The effect is that the last pixel(s) are not flushed out to memory. They will be by any subsequent rendering, but if no more rendering is done (for example while waiting for user input) then one or more pixels will not be visible on the screen.

3.15.2 Software Workaround

The simplest solution is to follow any Render2D command which might fall into this category with a ContinueNewSub (0) command which will do nothing, but as a side effect cause any pending pixels to be flushed out from the internal registers. If this proves to be too much of a performance burden then it is possible to do the flush only on an interrupt driven basis such as on a frame blank interrupt. This will flush the residual pixels 60 or more times per second, which is frequent enough to make the missing pixels invisible.

3.16 PEREN0016 - Invalidate Texture Cache

3.16.1 Problem

After sending an Invalidate Cache command, the texturing mapping hardware must wait for this to be fully processed before continuing with command processing. Failing to do so can result in the graphics processor locking up.

3.16.2 Software Workaround

One workaround is to send the Invalidate Cache command and then load a data value of 0 into the FogModeOr and TextureReadMode0Or registers. An alternative is to send WaitForCompletion (0) after the Invalidate Cache command.

3.17 PEREN0017 - Horizontal Display Resolution

3.17.1 Problem

The video unit imposes a 2048 pixel width constraint. Beyond this resolution the display is wrapped.

3.17.2 Software Workaround

None.

4

Alert Details

Alerts are part of 3Dlabs committment to providing comprehensive and useful information about chipset products. Alerts describe issues arising when the chip is used outside normal operating parameters and may be of interest to driver programmers.

4.1 ALERT001 - Control DMA/ Programmed IO Interaction Caution

4.1.1 Problem

The input fifo is not designed to cope with fast switching between Control DMA and writes to the input fifo. Normally, either one mechanism or the other (but not both) should be used. This advisory does not apply to mixing fifo/register space acesses and Hostin DMA.

4.1.2 Software Workaround

Where it is necessary to combine both write methods, ensure that the input fifo is completely empty after writing to the fifo/register space and before starting a new DMA transfer. The input FIFO must report 128 spaces available. Clamping to fewer than 128 spaces will produce unpredictable results.

4.2 ALERT002 - Texture Lockup when Cache Combining above 100MHz

4.2.1 Problem

When *CombineCaches* is enabled for texturing at frequencies above 100MHz, lockups may occur. When this happens check to see if the *CombineCaches* bit is set by reading back the **TextureFilterMode** register (refer to the *Permedia3 Reference Guide*).

4.2.2 Software Workaround

Since the Primary Cache Manager is the only texture read function which uses the *CombineCaches* bit, the workaround is to ensure that drivers do not set the *CombineCaches* bit in the **TextureReadMode0/1** and **TextureFilterMode** registers if clock speeds in excess of 100MHz are anticipated.

4.3 ALERT003 - Unable to see more than 32MB of memory in Extended Addressing when Interleave is enabled

4.3.1 Problem

When the *Interleave* and *AddressExtension* bits are set in **LocalMemControl**, only 32MB of memory are visible. Effectively, Bank 1 becomes a clone of Bank 0.

4.3.2 Software Workaround

Correct the bank addressing by incrementing the *BankAddress* bits by 1. For example, 64MB of memory can usually be used with the following settings:
LocalMemCaps: 0x30F413B8
LocalMemControl: 0x0800001A

4.4 ALERT004 - In striped mode, secondary chip does not swap correctly between front left and front right buffers

4.4.1 Problem

When the **ScreenBaseRight** register on the secondary chip in a two-chip configuration (Striped Mode) is loaded it fails to take effect unless followed by by a write to ScreenBase.

4.4.2 Software Workaround

Re-load the ScreenBase after the **ScreenBaseRight** register update.