Intel® OpenSource HD Graphics Programmer's Reference Manual (PRM)
Volume 4 Part 1: Subsystem and Cores – Shared Functions (Ivy Bridge)

For the 2012 Intel® Core™ Processor Family

May 2012

Revision 1.0

NOTICE:
This document contains information on products in the design phase of development, and Intel reserves the right to add or remove product features at any time, with or without changes to this open source documentation.

Creative Commons License

You are free to Share — to copy, distribute, display, and perform the work

Under the following conditions:

Attribution. You must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work).

No Derivative Works. You may not alter, transform, or build upon this work.

Contents

1. Subsystem Overview

1.1 Introduction

The subsystem consists of an array of *execution units* (*EUs*, sometimes referred to as an arrray of *cores*) along with a set of *shared functions* outside the EUs that the EUs leverage for I/O and for complex computations. Programmers access the subsystem via the 3D or Media pipelines.

EUs are general-purpose programmable cores that support a rich instruction set that has been optimized to support various 3D API shader languages as well as media functions (primarily video) processing.

Shared functions are hardware units which serve to provide specialized supplemental functionality for the EUs. A shared function is implemented where the demand for a given specialized function is insufficient to justify the costs on a per-EU basis. Instead a single instantiation of that specialized function is implemented as a stand-alone entity outside the EUs and shared among the EUs.

Invocation of the shared functionality is performed via a communication mechanism called a *message*. A message is a small self-contained packet of information created by a kernel and directed to a specific shared function. Messages are dispatched to the shared function under software control via the send instruction. This instruction identifies the contents of the message and the GRF register locations to direct any response.

The message construction and delivery mechanisms are general in their definition and capable of supporting a wide variety of shared functions.

1.2 Subsystem Topology

The subsystem is organized as an array of EUs, and a set of functions that are shared among all of the EUs. (The EU array is further divided into rows with each row having its own first level instruction cache and Extended Math shared function, though this aspect of the implemented topology is not exposed to software). The Sampler, DataPort, URB and Message Gateway functions are shared among the entire array of EUs.

1.3 Execution Units (EUs)

Each EU is a vector machine capable of performing a given operation on as many as 16 pieces of data of the same type in parallel (though not necessarily on the same instant in time). In addition, each EU can support a number of execution contexts called *threads* that are used to avoid stalling the EU during a high-latency operation (external to the EU) by providing an opportunity for the EU to switch to a completely different workload with minimal latency while waiting for the high-latency operation to complete.

For example, if a program executing on an EU requires a texture read by the sampling engine, the EU may not necessarily idle while the data is fetched from memory, arranged, filtered and returned to the EU. Instead the EU will likely switch execution to another (unrelated) thread associated with that EU. If that thread encounters a stall, the EU may switch to yet another thread and so on. Once the Sampler result arrives back at the EU, the EU can switch back to the original thread and use the returned data as it continues execution of that thread.

The fact that there are multiple EU cores each with multiple threads can generally be ignored by software. There are some exceptions to this rule: e.g., for

- thread-to-thread communication (see *Message Gateway*, *Media*)
- synchronization of thread output to memory buffers (see *Geometry Shader*).

In contrast, the internal SIMD aspects of the EU are very much exposed to software.

This volume will not deal with the details of the EUs.

1.4 Thread Dispatching

When the 3D and Media pipelines send requests for thread initiation to the Subsystem, the thread Dispatcher receives the requests. The dispatcher performs such tasks as arbitrating between concurrent requests, assigning requested threads to hardware threads on EUs, allocating register space in each EU among multiple threads, and initializing a thread's registers with data from the fixed functions and from the URB. This operation is largely transparent to software.

1.5 Shared Functions

In general, a shared function has the ability to receive messages at its input, perform some specialized amount of work for each, and if required, generate output back to the message's originating execution unit (Message Gateway may generate output to a target execution unit specified by the message).

To uniquely identify shared functions, each is assigned a unique 4-bit identifier code called its 'Function ID'. This ID is specified in the 'send' instruction's 32b <desc> field of each message. Function ID assignments are listed in the *Graphics Processing Engine* chapter of this specification.

Each shared function may support one or more related operations within itself. For example an Extended Math shared function may support operations such as reciprocal, sine, cosine, and/or others. These are generically referred to as sub-functions. The communication method as to which sub-function is desired is typically contained in the 16b 'function-control' field of the 'send' instruction <desc> field. Alternatively, a function may choose to define sub-function encodings in-band within message payload, or in the case of a single function shared-function, the function code may be implied. The architecture, in no way interprets the sub-function code and the actual implementation choice is left to the function itself.

The Shared Function units included in the Subsystem are as follows (refer to the chapters devoted to each of these functions):

- Extended Math function
- Sampling Engine function
- DataPort function
- Message Gateway function
- Unified Return Buffer (URB)
- Thread Spawner (TS)
- Null function

The **Extended Math** function acts as an extension of the math functions already available inside the EUs. Certain functions such as inverse, square root, exponentiation, etc., require significant hardware resources to implement and are used infrequently enough that it is inefficient to implement them separately in each EU. The EUs therefore send the operands for these operations along with the

operation to be performed to the Extended Math function which computes and returns the result to the requesting EU.

The **Sampling Engine** acts a (read-only) I/O port on behalf of the EUs, translating texture coordinates (and/or structure references) to memory addresses, reading texels and/or other data from memory, and in the case of texels, combining and filtering them according to programmed state. The resulting pixel and/or other data are then returned to the requesting EU.

The **Data Port** function acts as another I/O port on behalf of the EUs. It is both a read and a write port, and the only way for the Graphics Processing Engine to write results (e.g., images) back to memory. The Data Port contains the render and depth caches which receive the newly rendered pixels and write them out to memory when necessary. They also permit previously rendered objects to be read back efficiently by the Graphics Processing Engine in order to blend them with other rendered objects and test for visibility of newly rendered objects. Finally, the Data Port also provides read access constant buffers (arrays of constants in memory.)

The **Message Gateway** allows a thread to communicate (send a message to) another thread. A key is used to connect the sender and receiver threads, and a simple gateway protocol is used to send messages. This is primarily intended for media where a parent/child thread model is sometimes used and requires parent and child threads to synchronize and efficiently share information. It is not intended to be used by 3D graphics rendering threads.

The **Unified Return Buffer** (URB) is a single set of registers that EU threads use to return result data for future fixed functions and their threads to make use of. Individual entries in the buffer are “owned” by a given fixed function but a mechanism is provided where other fixed functions (those that follow) can read the data placed there by another fixed function. The buffer is considered a “Shared Function” since EUs need to be able to write result data to it using messages. In general, EU threads write their final results either to memory via the Data Port or to the URB for re-use by subsequent EU threads or certain 3D pipeline fixed-function units (CLIP, GS).

The **Thread Spawner** (TS) is a Shared Function that acts as a conduit for dispatching kernel-software-generated threads, one thread can request another thread to be dispatched by sending a request to the TS. TS is unique as it is also a Fixed Function in the media pipeline for dispatching threads originated from Video Front End fixed function.

The **Null** shared function is supported to allow the broadcast of certain information (e.g, End Of Thread) without invoking any other operation or response.

1.5.1 Message Payload Containing a Header

For most shared functions, the first register of the message payload contains the *header payload* of the message (or simply the *message header*). Consequently, the rest of the message payload is referred to as the *data payload*.

Messages to Extended Math do not have a header and only contain data payload. Those messages may be referred to as header-less messages. Messages to Gateway combine the header and data payloads in a single message register.

1.5.2 Writebacks

Some messages generate return data as dictated by the ‘function-control’ (opcode) field of the ‘send’ instruction (part of the <desc> field). The execution unit and message passing infrastructure do not interpret this field in any way to determine if writeback data is to be expected. Instead explicit fields in the ‘send’ instruction to the execution unit the starting GRF register and count of returning data. The execution unit uses this information to set in-flight bits on those registers to prevent execution of any

instruction which uses them as an operand until the register(s) is(are) eventually written in response to the message. If a message is not expected to return data, the ‘send’ instruction’s writeback destination specifier (<post_dest>) must be set to ‘null’ and the response length field of <desc> must be 0 (see ‘send’ instruction for more details).

The writeback data, if called for, arrives as a series of register writes to the GRF at the location specified by the starting GRF register and length as specified in the ‘send’ instruction. As each register is written back to the GRF, its in-flight flag is cleared and it becomes available for use as an instruction operand. If a thread was suspended pending return of that register, the dependency is lifted and the thread is allowed to continue execution (assuming no other dependency for that thread remains outstanding).

1.5.3 Message Delivery Ordering Rules

All messages between a thread and an individual shared function are delivered in the ordered they were sent. Messages to different shared functions originating from a single thread may arrive at their respective shared functions out of order.

The writebacks of various messages from the shared functions may return in any order. Further individual destination registers resulting from a single message may return out of order, potentially allowing execution to continue before the entire response has returned (depending on the dependency chain inherent in the thread).

1.5.4 Execution Mask and Messages

The Architecture defines an Execution Mask (EMask) for each instruction issued. This 16b bit-field identifies which SIMD computation channels are enabled for that instruction. Since the ‘send’ instruction is inherently scalar, the EMask is ignored as far as instruction dispatch is concerned. Further the execution size has no impact on the size of the ‘send' instruction’s implicit move (it is always 1 register regardless of specified execution size).

The 16b EMask is forwarded with the message to the destination shared function to indicate which SIMD channels were enabled at the time of the ‘send’. A shared function may interpret or ignore this field as dictated by the functionality it exposes. For instance, the Extended Math shared function observes this field and performs the specified operation only on the operands with enabled channels, while the DataPort writes to the render cache ignore this field completely, instead using the pixel mask included in-band in the message payload to indicate which channels carry valid data.

1.5.5 End-Of-Thread (EOT) Message

The final instruction of all threads must be a ‘send’ instruction which signals ‘End-Of-Thread’ (EOT). An EOT message is one in which the EOT bit is set in the ‘send’ instruction’s 32b <desc> field. When issuing instructions, the EU looks for an EOT message, and when issued, shuts down the thread from further execution and considers the thread completed.

Only a subset of the shared functions can be specified as the target function of an EOT message, as shown in the table below.

Target Shared Functions supporting EOT messages	**Target Shared Functions not supporting EOT messages**
Null, DataPortWrite, URB, MessageGateway, ThreadSpawner	DataPortRead, Sampler

Both the fixed-functions and the thread dispatcher require EOT notification at the completion of each thread. The thread dispatcher and fixed functions in the 3D pipeline obtain EOT notification by snooping

all message transmissions, regardless of the explicit destination, looking for messages which signal end-of-thread. The Thread Spawner in the media pipeline does not snoop for EOT. As it is also a shared function, all threads generated by Thread Spawner must send a message to Thread Spawner to explicity signal end-of-thread.

The thread dispatcher, upon detecting an end-of-thread message, updates its accounting of resource usage by that thread, and is free to issue a new thread to take the place of the ended thread. Fixed functions require end-of-thread notification to maintain accounting as to which threads it issued have completed and which remain outstanding, and their associated resources such as URB handles.

Unlike the thread dispatcher, fixed-functions discriminate end-of-thread messages, only acting upon those from threads which they originated, as indicated by the 4b fixed-function ID present in R0 of end-of-thread message payload. This 4b field is attached to the thread at new-thread dispatch time and is placed in its designated field in the R0 contents delivered to the GRF. Thus to satisfy the inclusion of the fixed-function ID, the typical end-of-thread message generally supplies R0 from the GRF as the first register of an end-of-thread message.

As an optimization, an end-of-thread message may be overload upon another “productive” message, saving the cost in execution and bandwidth of a dedicated end-of-thread message. Outside of the end-of-thread message, most threads issue a message just prior to their termination (for instance, a Dataport write to the framebuffer) so the overloaded end-of-thread is the common case. The requirement is that the message contains R0 from the GRF (to supply the fixed-function ID), and that destination shared function be either (a) the URB; (b) the Read or Write Dataport; or, (c) the Gateway, as these functions reside on the O-Bus. In the case where the last real message of a thread is to some other shared function, the thread must issue a separate message for the purposes of signaling end-of-thread to the “null” shared function.

1.5.6 Performance

The Architecture imposes no requirement as to a shared function’s latency or throughput. Due to this as well as factors such as message queuing, shared bus arbitration, implementation choices in bus bandwidth, and instantaneous demand for that function, the latency in delivering and obtaining a response to a message is non-deterministic. It is expected that an implementation has some notion of fairness in transmission and servicing of messages so as to keep latency outliers to a minimum.

Other factors to consider with regard to performance:

- A thread may choose to have multiple messages under construction in non-overlapping registers in the MRF at the same time.
- Multiple messages are allowed to be enqueued for transmission at the same time, so long as their MRF payload registers do not overlap.
- Messages may rely on the MRF registers being maintained across a send message, thus constructing subsequent messages overlaid on portions of a previous message,
- Software prefetching techniques may be beneficial for long latency data fetches (i.e. issue a load early in the thread for data that is required late in the thread).

1.5.7 Message Description Syntax

All message formats are defined in terms of DWords (32 bits). The message registers in all cases are 256 bits wide, or 8 DWords. The registers and DWords within the registers are named as follows, where n is the register number, and d is the DWord number from 0 to 7, from the least significant DWord at bits [31:0] within the 256-bit register to the most significant DWord at bits [255:224], respectively. For

writeback messages, the register number indicates the offset from the specified starting destination register.

Dispatch Messages: **R**n.d

Dispatch messages are sent by the fixed functions to dispatch threads. See the fixed function chapters in the *3D and Media* volume.

SEND Instruction Messages: **M**n.d

These are the messages initiated by the thread via the SEND instruction to access shared functions. See the chapters on the shared functions later in this volume.

Writeback Messages: **W**n.d

These messages return data from the shared function to the GRF where it can be accessed by thread that initiated the message.

The bits within each DWord are given in the second column in each table.

1.5.8 Message Errors

Messages are constructed via software, and not all possible bit encodings are legal, thus there is the possibility that a message may be sent containing one or more errors in its descriptor or payload contents. There are two points of error detection in the message passing system: (a) the message delivery subsystem is capable of detecting bad FunctionIDs and some cases of bad message lengths; (b) the shared functions contain various error detection mechanisms which identify bad sub-function codes, bad message lengths, and other misc errors. The error detection capabilities are specific to each shared function. The execution unit hardware itself does not perform message validation prior to transmission.

In both cases, information regarding the erroneous message is captured and made visible through MMIO registers, and the driver notified via an interrupt mechanism . The set of possible errors is listed in *Message Errors* with the associated outcome.

Error Cases

Error	Outcome
Bad Shared Function ID	The message is discarded before reaching any shared function. If the message specified a destination, those registers will be marked as in-flight, and any future usage by the thread of those registers will cause a dependency which will never clear, resulting in a hung thread and eventual time-out.
Unknown opcode Incorrect message length	The destination shared function detects unknown opcodes (as specified in the 'send' instructions <desc> field), and known opcodes where the message payload is either too long or too short, and threats these cases as errors. When detected, the shared function latches and makes available via MMIO registers the following information: the EU and thread ID which sent the message, the length of the message and expected response, and any relevant portions of the first register (R0) of the message payload. The shared function alerts the driver of an erroneous message through and interrupt mechanism , then continues normal operation with the subsequent message.
Bad message contents in payload	Detection of bad data is an implementation decision of the shared function. Not all fields may be checked by the shared function, so an erroneous payload may return bogus data or no data at all. If an erroneous value is detected by the shared function, it is free to discard the message and continue with the subsequent message. If the thread was expecting a response, the destination registers specified in the associated 'send' instruction are never cleared potentially resulting in a hung thread and time-out.
Incorrect response length	Case: too few registers specified – the thread may proceed with execution prior to all the data returning from the shared function, resulting in the thread operating on bad data in the

Error	Outcome
	GRF. Case: too many registers specified – the message response does not clear all the registers of the destination. In this case, if the thread references any of the residual registers, it may hand and result in an eventual time-out.
Improper use of End-Of-Thread (EOT)	Any ‘send’ instruction which specifies EOT must have a ‘null’ destination register. The EU enforces this and, if detected, will not issue the ‘send’ instruction, resulting in a hung thread and an eventual time-out. The ‘send’ instruction specifies that EOT is only recognized if the <desc> field of the instruction is an immediate. Should a thread attempt to end a thread using a <desc> sourced from a register, the EOT bit will not be recognized. In this case, the thread will continue to execute beyond the intended end of thread, resulting in a wide range of error conditions.
Two outstanding messages using overlapping GRF destinations ranges	This is not checked by HW. Due to varying latencies between two messages, and out-of-order, non-contiguous writeback cycles, the outcome in the GRF is indeterminate; may be the result from the first message, or the result from the second message, or a combination of both.

2. Sampling Engine

The Sampling Engine provides the capability of advanced sampling and filtering of surfaces in memory.

The sampling engine function is responsible for providing filtered texture values to the Core in response to sampling engine messages. The sampling engine uses SAMPLER_STATE to control filtering modes, address control modes, and other features of the sampling engine. A pointer to the sampler state is delivered with each message, and an index selects one of 16 states pointed to by the pointer. Some messages do not require SAMPLER_STATE. In addition, the sampling engine uses SURFACE_STATE to define the attributes of the surface being sampled. This includes the location, size, and format of the surface as well as other attributes.

Although data is commonly used for "texturing" of 3D surfaces, the data can be used for any purpose once returned to the execution core.

The following table summarizes the various subfunctions provided by the Sampling Engine. After the appropriate subfunctions are complete, the 4-component (reduced to fewer components in some cases) filtered texture value is provided to the Core in order to complete the *sample* instruction.

Subfunction	Description
Texture Coordinate Processing	Any required operations are performed on the incoming pixel's interpolated internal texture coordinates. These operations may include: cube map intersection.
Texel Address Generation	The Sampling Engine will determine the required set of texel samples (specific texel values from specific texture maps), as defined by the texture map parameters and filtering modes. This includes coordinate wrap/clamp/mirror control, mipmap LOD computation and sample and/or miplevel weighting factors to be used in the subsequent filtering operations.
Texel Fetch	The required texel samples will be read from the texture map. This step may require decompression of texel data. The texel sample data is converted to an internal format.
Texture Palette Lookup	For streams which have "paletted" texture surface formats, this function uses the "index" values read from the texture map to look up texel color data from the texture palette.
Shadow Pre-Filter Compare	For shadow mapping, the texel samples are first compared to the 3rd (R) component of the pixel's texture coordinate. The boolean results are used in the texture filter.
Texel Filtering	Texel samples are combined using the filter weight coefficients computed in the Texture Address Generation function. This "combination" ranges from simply passing through a "nearest" sample to blending the results of anisotropic filters performed on two mipmap levels. The output of this function is a single 4-component texel value.
Texel Color Gamma Linearization	Performs optional gamma decorrection on texel RGB (not A) values.
Denoise/ Deinterlacer	Performs denoise and deinterlacing functions for video content ()
8x8 Video Scaler	Performs scaling using an 8x8 filter ()
Image Enhancement Filter / Video Signal Analysis	Image Enhancement functions for video content ()

2.1 Texture Coordinate Processing

The Texture Coordinate Processing function of the Sampling Engine performs any operations on the texture coordinates that are required before physical addresses of texel samples can be generated.

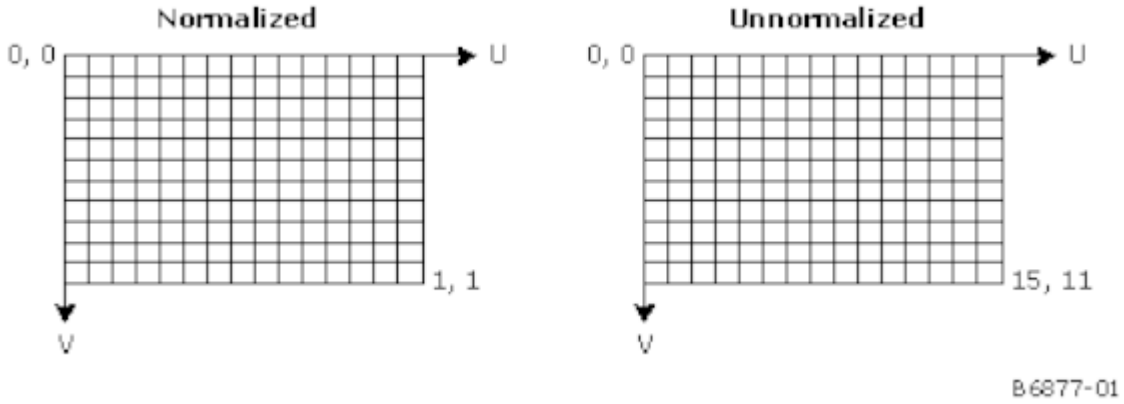
2.1.1 Texture Coordinate Normalization

A texture coordinate may have *normalized* or *unnormalized* values. In this function, unnormalized coordinates are normalized.

Normalized coordinates are specified in units relative to the map dimensions, where the origin is located at the upper/left edge of the upper left texel, and the value 1.0 coincides with the lower/right edge of the lower right texel . 3D rendering typically utilizes normalized coordinates.

Unnormalized coordinates are in units of texels and have not been divided (normalized) by the associated map's height or width. Here the origin is the located at the upper/left edge of the upper left texel of the base texture map. Unnormalized coordinates delivered to the sampling engine are only supported with the "ld" type messages.

Normalized vs. Unnormalized Texture Coordinates



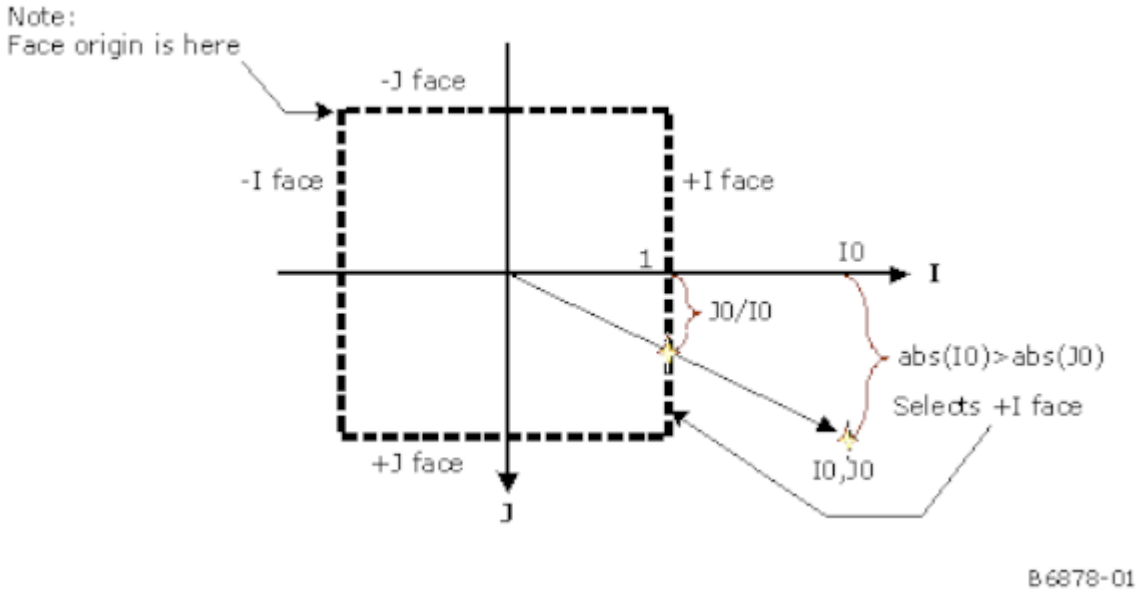
2.1.2 Texture Coordinate Computation

Cartesian (2D) and homogeneous (projected) texture coordinate values are projected from (interpolated) screen space back into texture coordinate space by dividing the pixel's S and T components by the Q component. This operation is done as part of the pixel shader kernel in the Core.

Vector (cube map) texture coordinates are generated by first determining which of the 6 cube map faces (+X, +Y, +Z, -X, -Y, -Z) the vector intersects. The vector component (X, Y or Z) with the largest absolute value determines the proper (major) axis, and then the sign of that component is used to select between the two faces associated with that axis. The coordinates along the two minor axes are then divided by the coordinate of the major axis, and scaled and translated, to obtain the 2D texture coordinate ([0,1]) within the chosen face. Note that the coordinates delivered to the sampling engine must already have been divided by the component with the largest absolute value.

An illustration of this cube map coordinate computation, simplified to only two dimensions, is provided below:

Cube Map Coordinate Computation Example



2.2 Texel Address Generation

To better understand texture mapping, consider the mapping of each object (screen-space) pixel onto the textures images. In texture space, the pixel becomes some arbitrarily sized and aligned quadrilateral. Any given pixel of the object may “cover” multiple texels of the map, or only a fraction of one texel. For each pixel, the usual goal is to sample and filter the texture image in order to best represent the covered texel values, with a minimum of blurring or aliasing artifacts. Per-texture state variables are provided to allow the user to employ quality/performance/footprint tradeoffs in selecting how the particular texture is to be sampled.

The Texel Address Generation function of the Sampling Engine is responsible for determining how the texture maps are to be sampled. Outputs of this function include the number of texel samples to be taken, along with the physical addresses of the samples and the filter weights to be applied to the samples after they are read. This information is computed given the incoming texture coordinate and gradient values, and the relevant state variables associated with the sampler and surface. This function also applies the texture coordinate address controls when converting the sample texture coordinates to map addresses.

2.2.1 Level of Detail Computation (Mipmapping)

Due to the specification and processing of texture coordinates at object vertices, and the subsequent object warping due to a perspective projection, the texture image may become *magnified* (where a texel covers more than one pixel) or *minified* (a pixel covers more than one texel) as it is mapped to an object. In the case where an object pixel is found to cover multiple texels (texture minification), merely choosing one (e.g., the texel sample nearest to the pixel’s texture coordinate) will likely result in severe aliasing artifacts.

Mipmapping and texture filtering are techniques employed to minimize the effect of undersampling these textures. With mipmapping, software provides *mipmap levels,* a series of pre-filtered texture maps of decreasing resolutions that are stored in a fixed (monolithic) format in memory. When mipmaps are provided and enabled, and an object pixel is found to cover multiple texels (e.g., when a textured object is

located a significant distance from the viewer), the device will sample the mipmap level(s) offering a texel/pixel ratio as close to 1.0 as possible.

The device supports up to 14 mipmap levels per map surface, ranging from 8192 x 8192 texels to a 1 X 1 texel. Each successive level has ½ the resolution of the previous level in the U and V directions (to a minimum of 1 texel in either direction) until a 1x1 texture map is reached. The dimensions of mipmap levels need not be a power of 2.

Each mipmap level is associated with a *Level of Detail (LOD)* number. LOD is computed as the approximate, $\log_2$ measure of the ratio of texels per pixel. The highest resolution map is considered LOD 0. A larger LOD number corresponds to lower resolution mip level.

The *Sampler[]BaseMipLevel* state variable specifies the LOD value at which the minification filter vs. the magnification filter should be applied.

When the texture map is magnified (a texel covers more than one pixel), the base map (LOD 0) texture map is accessed, and the magnification mode selects between the nearest neighbor texel or bilinear interpolation of the 4 neighboring texels on the base (LOD 0) mipmap.

2.2.1.1 Base Level Of Detail (LOD)

The per-pixel LOD is computed in an implementation-dependent manner and approximates the $\log_2$ of the texel/pixel ratio at the given pixel. The computation is typically based on the differential texel-space distances associated with a one-pixel differential distance along the screen x- and y-axes. These texel-space distances are computed by evaluating neighboring pixel texture coordinates, these coordinates being in units of texels on the base MIP level (multiplied by the corresponding surface size in texels). The q coordinates represent the third dimension for 3D (volume) surfaces, this coordinate is a constant 0 for 2D surfaces.

The ideal LOD computation is included below.

$$LOD(x,y) = \log_2[\rho(x,y)]$$

where:

$$\rho(x,y) = \max\left\{\sqrt{\left(\frac{\partial u}{\partial x}\right)^2 + \left(\frac{\partial v}{\partial x}\right)^2 + \left(\frac{\partial q}{\partial x}\right)^2}, \sqrt{\left(\frac{\partial u}{\partial y}\right)^2 + \left(\frac{\partial v}{\partial y}\right)^2 + \left(\frac{\partial q}{\partial y}\right)^2}\right\},$$

2.2.1.2 LOD Bias

A biasing offset can be applied to the computed LOD and used to artificially select a higher or lower miplevel and/or affect the weighting of the selected mipmap levels. Selecting a slightly higher mipmap level will trade off image blurring with possibly increased performance (due to better texture cache reuse). Lowering the LOD tends to sharpen the image, though at the expense of more texture aliasing artifacts.

The LOD bias is defined as sum of the *LODBias* state variable and the *pixLODBias* input from the input message (which can be non-zero only for sample_b messages). The application of LOD Bias is unconditional, therefore these variables must both be set to zero in order to prevent any undesired biasing.

Note that, while the LOD Bias is applied prior to clamping and min/mag determination and therefore can be used to control the min-vs-mag crossover point, its use has the undesired effect of actually changing the LOD used in texture filtering.

2.2.1.3 LOD Pre-Clamping

The LOD Pre-Clamping function can be enabled or disabled via the *LODPreClampEnable* state variable. Enabling pre-clamping matches OpenGL semantics, while disabling it matches .

After biasing and/or adjusting of the LOD , the computed LOD value is clamped to a range specified by the (integer and fractional bits of) *MinLOD* and *MaxLOD* state variables prior to use in Min/Mag Determination.

MaxLOD specifies the lowest resolution mip level (maximum LOD value) that can be accessed, even when lower resolution maps may be available. Note that this is the only parameter used to specify the number of valid mip levels that be can be accessed, i.e., there is no explicit “number of levels stored in memory” parameter associated with a mip-mapped texture. All mip levels from the base mip level map through the level specified by the integer bits of *MaxLOD* must be stored in memory, or operation is UNDEFINED.

MinLOD specifies the highest resolution mip level (minimum LOD value) that can be accessed, where LOD==0 corresponds to the base map. This value is primarily used to deny access to high-resolution mip levels that have been evicted from memory when memory availability is low.

MinLOD and *MaxLOD* have both integer and fractional bits. The fractional parts will limit the inter-level filter weighting of the highest or lowest (respectively) resolution map. For example if *MinLOD* is 4.5 and *MipFilter* is LINEAR, LOD 4 can contribute only up to 50% of the final texel color.

2.2.1.4 Min/Mag Determination

The biased and clamped LOD is used to determine whether the texture is being minified (scaled down) or magnified (scaled up).

The *BaseMipLevel* state variable is subtracted from the biased and clamped LOD. The *BaseMipLevel* state variable therefore has the effect of selecting the “base” mip level used to compute Min/Map Determination. (This was added to match OpenGL semantics). Setting *BaseMipLevel* to 0 has the effect of using the highest-resolution mip level as the base map.

If the biased and clamped LOD is non-positive, the texture is being magnified, and a single (high-resolution) miplevel will be sampled and filtered using the *MagFilter* state variable. At this point the computed LOD is reset to 0.0. Note that LOD Clamping can restrict access to high-resolution miplevels.

If the biased LOD is positive, the texture is being minified. In this case the *MipFilter* state variable specifies whether one or two mip levels are to be included in the texture filtering, and how that (or those) levels are to be determined as a function of the computed LOD.

2.2.1.5 LOD Computation Pseudocode

This section illustrates the LOD biasing and clamping computation in pseudocode, encompassing the steps described in the previous sections. The computation of the initial per-pixel LOD value *LOD* is not shown.

Bias:S4.8

MinLod:U4.8

MaxLod:U4.8

Base:U4.1

MIPCnt:U4

SurfMinLod: U4.8

ResMinLod: U4.8

AdjMaxLod = min(MaxLod, MIPCnt)

AdjMinLod = min(MinLod, MIPCnt)

AdjPR_minLOD = ResMinLod – SurfMinLod

AdjMinLod = max(AdjMinLod, AdjPR_minLOD)

Out_of_Bounds = AdjPR_minLOD > MIPCnt

```
if (sample_b)
        LOD += Bias + bias_parameter
else if (sample_l or ld)
        LOD = Bias + lod_parameter
else
        LOD += Bias
PreClamp = LODPreClampEnable
If (PreClamp)
                LOD = min(LOD, MaxLod)
                LOD = max(LOD, MinLod)
MagMode = (LOD - Base <= 0)
MagClampMipNone = 1
If ((MagMode && MagClampMipNone) or MipFlt = None)
        LOD = 0
        LOD = min(LOD, ceil(AdjMaxLod))
        LOD = max(LOD, floor(AdjMinLod))
else if (MipFlt = Nearest)
         LOD = min(LOD, AdjMaxLod)
         LOD = max(LOD, AdjMinLod)
        LOD = min(LOD, AdjMaxLod)
        LOD = max(LOD, AdjMinLod)
        LOD +=0.5
        LOD = floor(LOD)
else// MipFlt = Linear
        LOD = min(LOD, AdjMaxLod)
        LOD = max(LOD, AdjMinLod)
```

TriBeta = frac(LOD)

LOD_0 = floor(LOD)

LOD_1 = LOD_0 + 1

if (!lod)// “LOD” message type

Lod += SurfMinLod

If Out_of_Bounds is true, LOD is set to zero and instead of sampling the surface the texels are replaced with zero in all channels, except for surface formats that don’t contain alpha, for which the alpha channel is replaced with one. These texels then proceed through the rest of the pipeline.

Errata: Out of Bound true on surface format that doesn’t contain alpha will be forced to 0 instead of 1.0 for the case the filet type is Anisotropic.

Errata: when AdjPR_minLOD > MIPCnt and MIPFILTER_LINEAR texel values will not force to zero.

2.2.1.5.1 Inter-Level Filtering Setup

The *MipFilter* state variable determines if and how texture mip maps are to be used and combined. The following table describes the various mip filter modes:

***MipFilter* Value**	**Description**
MIPFILTER_NONE	Mipmapping is DISABLED. Apply a single filter on the highest resolution map available (after LOD clamping).
MIPFILTER_NEAREST	Choose the nearest mipmap level and apply a single filter to it. Here the biased LOD will be rounded to the nearest integer to obtain the desired miplevel. LOD Clamping may further restrict this miplevel selection.
MIPFILTER_LINEAR	Apply a filter on the two closest mip levels and linear blend the results using the distance between the computed LOD and the level LODs as the blend factor. Again, LOD Clamping may further restrict the selection of miplevels (and the blend factor between them).

When minifying and MIPFILTER_NEAREST is selected, the computed LOD is rounded to the nearest mip level.

When minifying and MIPFILTER_LINEAR is selected, the fractional bits of the computed LOD are used to generate an inter-level blend factor. The LOD is then truncated. The mip level selected by the truncated LOD, and the next higher (lower resolution) mip level are determined.

Regardless of *MipFilter* and the min/mag determination, all computed LOD values (two for MIPFILTER_LINEAR, otherwise one) are then unconditionally clamped to the range specified by the (integer bits of) *MinLOD* and *MaxLOD* state variables.

2.2.2 Intra-Level Filtering Setup

Depending on whether the texture is being minified or magnified, the *MinFilter* or *MagFilter* state variable (respectively) is used to select the sampling filter to be used within a mip level (intra-level, as opposed to any inter-level filter). Note that for volume maps, this selection also applies to filtering between layers.

The processing at this stage is restricted to the selection of the filter type, computation of the number and texture map coordinates of the texture samples, and the computation of any required filter parameters. The filtering of the samples occurs later on in the Sampling Engine function.

The following table summarizes the intra-level filtering modes.

Sampler[]Min/MagFilter value	Description
MAPFILTER_NEAREST	Supported on all surface types. The texel nearest to the pixel's U,V,Q coordinate is read and output from the filter.
MAPFILTER_LINEAR	Not supported on buffer surfaces. The 2, 4, or 8 texels (depending on 1D, 2D/CUBE, or 3D surface, respectively) surrounding the pixel's U,V,Q coordinate are read and a linear filter is applied to produce a single filtered texel value.
MAPFILTER_ANISOTROPIC	Not supported on buffer or 3D surfaces. A projection of the pixel onto the texture map is generated and "subpixel" samples are taken along the major axis of the projection (center axis of the longer dimension). The outermost subpixels are weighted according to closeness to the edge of the projection, inner subpixels are weighted equally. Each subpixel samples a bilinear 2x2 of texels and the results are blended according to weights to produce a filtered texel value.
MAPFILTER_MONO	Supported only on 2D surfaces. This filter is only supported with the monochrome (MONO8) surface format. The monochrome texel block of the specified size surrounding the pixel is selected and filtered.

2.2.2.1 MAPFILTER_NEAREST

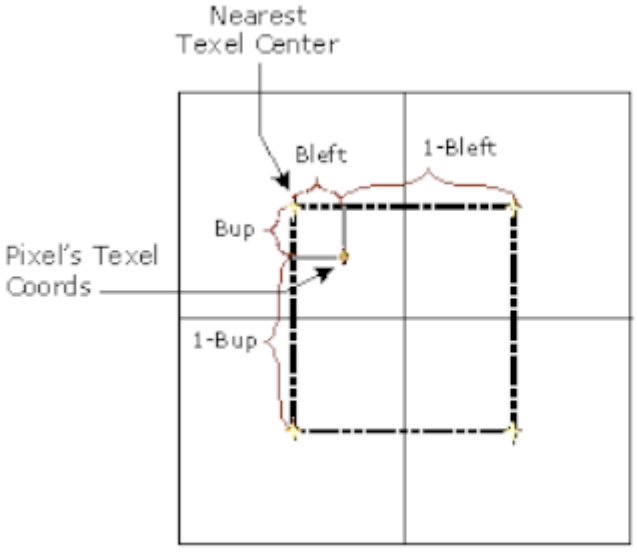
When the MAPFILTER_NEAREST is selected, the texel with coordinates nearest to the pixel's texture coordinate is selected and output as the single texel sample coordinates for the level.

2.2.2.2 MAPFILTER_LINEAR

The following description indicates behavior of the MIPFILTER_LINEAR filter for 2D and CUBE surfaces. 1D and 3D surfaces follow a similar method but with a different number of dimensions available.

When the MAPFILTER_LINEAR filter is selected on a 2D surface, the 2x2 region of texels surrounding the pixel's texture coordinate are sampled and later bilinearly filtered.

Bilinear Filter Sampling



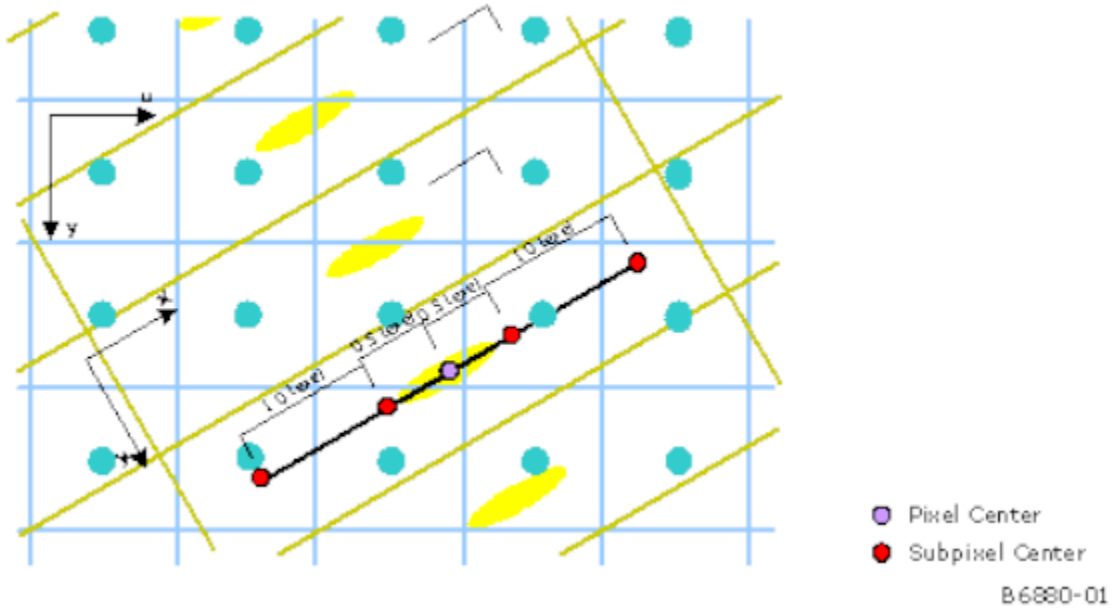
B6879-01

The four texels surrounding the pixel center are chosen for the bilinear filter. The filter weights each texel's contribution according to its distance from the pixel center. Texels further from the pixel center receive a smaller weight.

2.2.2.3 MAPFILTER_ANISOTROPIC

The MAPFILTER_ANISOTROPIC texture filter attempts to compensate for the anisotropic mapping of pixels into texture map space. A possibly non-square set of texel sample locations will be sampled and later filtered. The *MaxAnisotropy* state variable is used to select the maximum aspect ratio of the filter employed, up to 16:1.

The algorithm employed first computes the major and minor axes of the pixel projection onto the texture map. LOD is chosen based on the minor axis length in texel space. The anisotropic "ratio" is equal to the ratio between the major axis length and the minor axis length. The next larger even integer above the ratio determines the anisotropic number of "ways", which determines how many subpixels are chosen. A line along the major axis is determined, and "subpixels" are chosen along this line, spaced one texel apart, as shown in the diagram below. In this diagram, the texels are shown in light blue, and the pixels are in yellow.



Each subpixel samples a bilinear 2x2 around it just as if it was a single pixel. The result of each subpixel is then blended together using equal weights on all interior subpixels (not including the two endpoint subpixels). The endpoint subpixels have lesser weight, the value of which depends on how close the "ratio" is to the number of "ways". This is done to ensure continuous behavior in animation.

2.2.2.4 MAPFILTER_MONO

When the MAPFILTER_MONO filter is selected, a block of monochrome texels surrounding the pixel sample location are read and filtered using the kernel described below. The size of this block is controlled by **Monochrome Filter Height** and **Width** (referred to here as N_v and N_u, respectively) state. Filters from 1x1 to 7x7 are supported (not necessarily square).

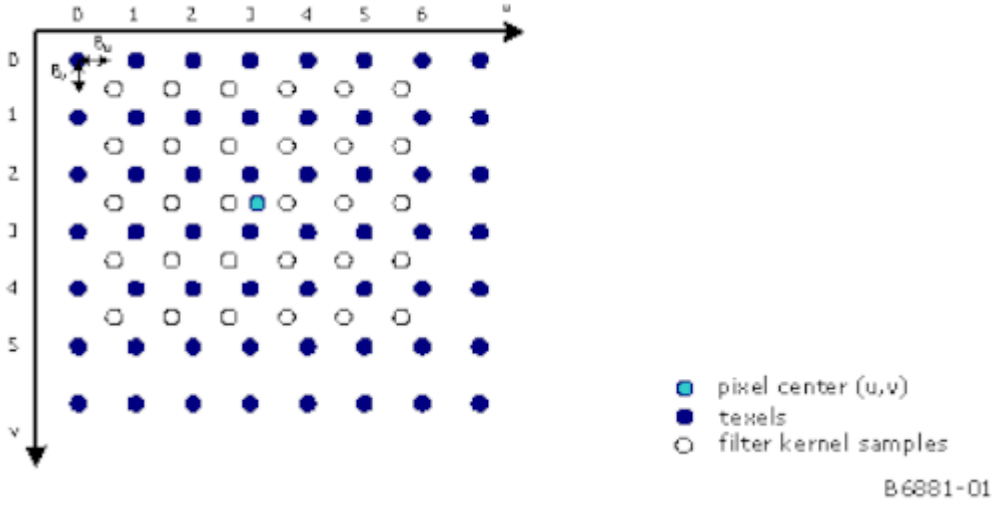
The figure below shows a 6x5 filter kernel as an example. The footprint of the filter (filter kernel samples) is equal to the size of the filter and the pixel center lies at the exact center of this footprint. The position of the upper left filter kernel sample (u_f, v_f) relative to the pixel center at (u, v) is given by the following:

$$u_f = u - \frac{N_u}{2}$$

$$v_f = v - \frac{N_v}{2}$$

β_u and β_v are the fractional parts of u_f and v_f, respectively. The integer parts select the upper left texel for the kernel filter, given here as $T_{0,0}$.

Sampling Using MAPFILTER_MONO



The formula for the final filter output F is given by the following. Since this is a monochrome filter, each texel value (T) is a single bit, and the output F is an intensity value that is replicated across the color and alpha channels.

$$S = \frac{1}{N_u * N_v}$$

$$F = \left[(1-\beta_u)(1-\beta_v) \sum_{i=0}^{N_u-1} \sum_{j=0}^{N_v-1} T_{i,j} + \beta_u (1-\beta_v) \sum_{i=1}^{N_u} \sum_{j=0}^{N_v-1} T_{i,j} + (1-\beta_u)\beta_v \sum_{i=0}^{N_u-1} \sum_{j=1}^{N_v} T_{i,j} + \beta_u \beta_v \sum_{i=1}^{N_u} \sum_{j=1}^{N_v} T_{i,j} \right] * S$$

2.2.3 Texture Address Control

The *[TCX,TCY,TCZ]ControlMode* state variables control the access and/or generation of texel data when the specific texture coordinate component falls outside of the normalized texture map coordinate range [0,1).

Note: For **Wrap Shortest** mode, the setup kernel has already taken care of correctly interpolating the texture coordinates. Software will need to specify TEXCOORDMODE_WRAP mode for the sampler that is provided with wrap-shortest texture coordinates, or artifacts may be generated along map edges.

TC[X,Y,Z] Control	Operation
TEXCOORDMODE_CLAMP	Clamp to the texel value at the edge of the map.
TEXCOORDMODE_CLAMP_BORDER	Use the texture map's border color for any texel samples falling outside the map. The border color is specified via a pointer in SAMPLER_STATE.
TEXCOORDMODE_HALF_BORDER	Similar to CLAMP_BORDER except texels outside of the map are clamped to a value halfway between the edge texel and the border color.
TEXCOORDMODE_WRAP	Upon crossing an edge of the map, repeat at the other side of the map in the same dimension.
TEXCOORDMODE_CUBE	Only used for cube maps. Here texels from adjacent cube faces can be sampled along the edges of faces. This is considered the highest quality mode for cube environment maps.
TEXCOORDMODE_MIRROR	Similar to the wrap mode, though reverse direction through the map each time an edge is crossed. INVALID for use with unnormalized texture coordinates.
TEXCOORDMODE_MIRROR_ONCE	Similar to the wrap mode, though reverse direction through the map each time an edge is crossed. INVALID for use with unnormalized texture coordinates.

Separate controls are provided for texture TCX, TCY, TCZ coordinate components so, for example, the TCX coordinate can be wrapped while the TCY coordinate is clamped. Note that there are no controls provided for the TCW component as it is only used to scale the other 3 components before addressing modes are applied.

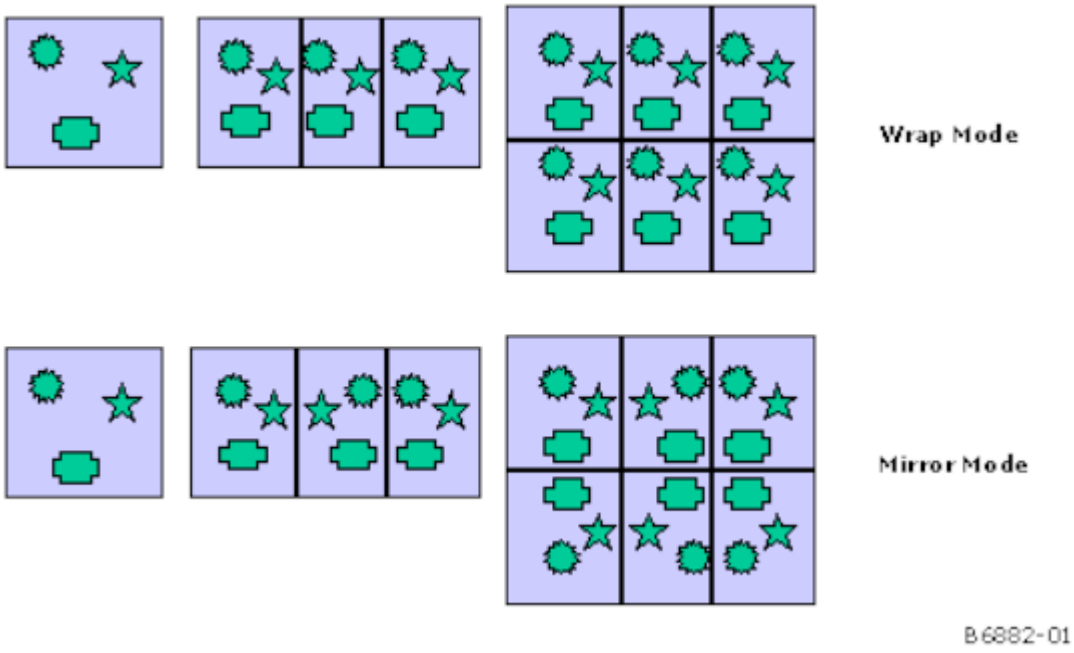
Maximum Wraps/Mirrors

The number of map wraps on a given object is limited to 32. Going beyond this limit is legal, but may result in artifacts due to insufficient internal precision, especially evident with larger surfaces. Precision loss starts at the subtexel level (slight color inaccuracies) and eventually reaches the texel level (choosing the wrong texels for filtering).

2.2.3.1 TEXCOORDMODE_MIRROR Mode

TEXCOORDMODE_MIRROR addressing mode is similar to Wrap mode, though here the base map is flipped at every integer junction. For example, for U values between 0 and 1, the texture is addressed normally, between 1 and 2 the texture is flipped (mirrored), between 2 and 3 the texture is normal again, and so on. The second row of pictures in the figure below indicate a map that is mirrored in one direction and then both directions. You can see that in the mirror mode every other integer map wrap the base map is mirrored in either direction.

Texture Wrap vs. Mirror Addressing Mode



2.2.3.2 TEXCOORDMODE_WRAP Mode

In TEXCOORDMODE_WRAP addressing mode, the integer part of the texture coordinate is discarded, leaving only a fractional coordinate value. This results in the effect of the base map ([0,1)) being continuously repeated in all (axes-aligned) directions. Note that the interpolation between coordinate values 0.1 and 0.9 passes through 0.5 (as opposed to WrapShortest mode which interpolates through 0.0).

2.2.3.3 TEXCOORDMODE_MIRROR_ONCE Mode

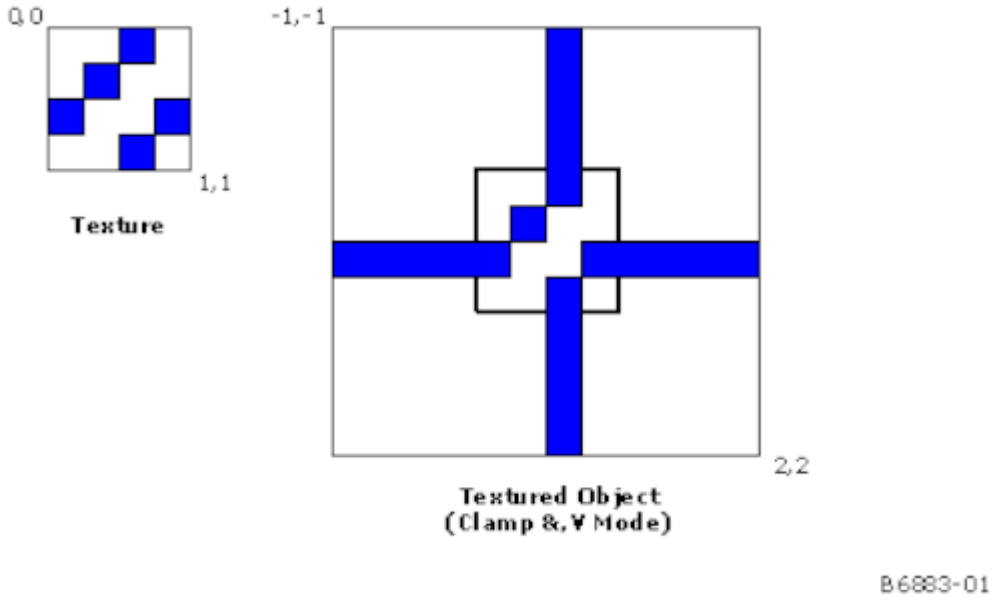
The TEXCOORDMODE_MIRROR_ONCE addressing mode is a combination of Mirror and Clamp modes. The absolute value of the texture coordinate component is first taken (thus mirroring about 0), and then the result is clamped to 1.0. The map is therefore mirrored once about the origin, and then clamped thereafter. This mode is used to reduce the storage required for symmetric maps.

2.2.3.4 TEXCOORDMODE_CLAMP Mode

The TEXCOORDMODE_CLAMP addressing mode repeats the “edge” texel when the texture coordinate extends outside the [0,1) range of the base texture map. This is contrasted to TEXCOORDMODE_CLAMPBORDER mode which defines a separate texel value for off-map samples. TEXCOORDMODE_CLAMP is also supported for cube maps, where texture samples will only be obtained from the intersecting face (even along edges).

The figure below illustrates the effect of clamp mode. The base texture map is shown, along with a texture mapped object with texture coordinates extending outside of the base map region.

Texture Clamp Mode



2.2.3.5 TEXCOORDMODE_CLAMPBORDER Mode

For non-cube map textures, TEXCOORDMODE_CLAMPBORDER addressing mode specifies that the texture map's border value *BorderColor* is to be used for any texel samples that fall outside of the base map. The border color is specified via a pointer in SAMPLER_STATE.

2.2.3.6 TEXCOORDMODE_CUBE Mode

For cube map textures TEXCOORDMODE_CUBE addressing mode can be set to allow inter-face filtering. When texel sample coordinates that extend beyond the selected cube face (e.g., due to intra-level filtering near a cube edge), the correct sample coordinates on the adjoining face will be computed. This will eliminate artifacts along the cube edges, though some artifacts at cube corners may still be present.

2.3 Texel Fetch

The Texel Fetch function of the Sampling Engine reads the texture map contents specified by the texture addresses associated with each texel sample. The texture data is read either directly from the memory-resident texture map, or from internal texture caches. The texture caches can be invalidated by the **Sampler Cache Invalidate** field of the MI_FLUSH instruction or via the **Read Cache Flush Enable** bit of PIPE_CONTROL. Except for consideration of coherency with CPU writes to textures and rendered textures, the texture cache does not affect the functional operation of the Sampling Engine pipeline.

When the surface format of a texture is defined as being a compressed surface, the Sampler will automatically decompress from the stored format into the appropriate [A]RGB values. The compressed texture storage formats and decompression algorithms can be found in the *Memory Data Formats* chapter. When the surface format of a texture is defined as being an index into the texture palette (format names includiong "Px"), the palette lookup of the index determines the appropriate RGB values.

2.3.1 Texel Chroma Keying

ChromaKey is a term used to describe a method of effectively removing or replacing a specific range of texel values from a map that is applied to a primitive, e.g., in order to define transparent regions in an RGB map. The Texel Chroma Keying function of the Sampling Engine pipeline conditionally tests texel samples against a "key" range, and takes certain actions if any texel samples are found to match the key.

2.3.1.1 Chroma Key Testing

ChromaKey refers to testing the texel sample components to see if they fall within a range of texel values, as defined by *ChromaKey[][High,Low]* state variables. If each component of a texel sample is found to lie within the respective (inclusive) range and ChromaKey is enabled, then an action will be taken to remove this contribution to the resulting texel stream output. Comparison is done separately on each of the channels and only if all 4 channels are within range the texel will be eliminated.

The Chroma Keying function is enabled on a per-sampler basis by the *ChromaKeyEnable* state variable.

The *ChromaKey[][High,Low]* state variables define the tested color range for a particular texture map.

2.3.1.2 Chroma Key Effects

There are two operations that can be performed to "remove" matching texel samples from the image. The *ChromaKeyEnable* state variable must first enable the chroma key function. The *ChromaKeyMode* state variable then specifies which operation to perform on a per-sampler basis.

The *ChromaKeyMode* state variable has the following two possible values:

KEYFILTER_KILL_ON_ANY_MATCH: Kill the pixel if any contributing texel sample matches the key

KEYFILTER_REPLACE_BLACK: Here the sample is replaced with (0,0,0,0). .

The Kill Pixel operation has an effect on a pixel only if the associated sampler is referenced by a sample instruction in the pixel shader program. If the sampler is not referenced, the chroma key compare is not done and pixels cannot be killed based on it.

2.4 Shadow Prefilter Compare

When a *sample_c* message type is processed, a special shadow-mapping precomparison is performed on the texture sample values prior to filtering. Specifically, each texture sample value is compared to the "ref" component of the input message, using a compare function selected by *ShadowFunction*, and described in the table below. Note that only single-channel texel formats are supported for shadow mapping, and so there is no specific color channel on which the comparison occurs.

ShadowFunction	**Result**
PREFILTEROP_ALWAYS	0.0
PREFILTEROP_NEVER	1.0
PREFILTEROP_LESS	(texel < ref) ? 0.0 : 1.0
PREFILTEROP_EQUAL	(texel == ref) ? 0.0 : 1.0
PREFILTEROP_LEQUAL	(texel <= ref) ? 0.0 : 1.0
PREFILTEROP_GREATER	(texel > ref) ? 0.0 : 1.0
PREFILTEROP_NOTEQUAL	(texel != ref) ? 0.0 : 1.0
PREFILTEROP_GEQUAL	(texel >= ref) ? 0.0 : 1.0

The binary result of each comparison is fed into the subsequent texture filter operation (in place of the texel's value which would normally be used).

Software is responsible for programming the "ref" component of the input message such that it approximates the same distance metric programmed in the texture map (e.g., distance from a specific light to the object pixel). In this way, the comparison function can be used to generate "in shadow" status for each texture sample, and the filtering operation can be used to provide soft shadow edges.

Programming Notes:

- Refer to the Surface Formats table in the section SURFACE_STATE for most messages for the specific surface formats that are supported with shadow mapping.

2.5 Texel Filtering

The Texel Filtering function of the Sampling Engine performs any required filtering of multiple texel values on and possibly between texture map layers and levels. The output of this function is a single texel color value.

The state variables *MinFilter*, *MagFilter*, and *MipFilter* are used to control the filtering of texel values. The *MipFilter* state variable specifies how many mipmap levels are included in the filter, and how the results of any filtering on these separate levels are combined to produce a final texel color. The *MinFilter* and *MagFilter* state variables specify how texel samples are filtered within a level.

2.6 Texel Color Gamma Linearization

This function is supported to allow pre-gamma-corrected texel RGB (not A) colors to be mapped back into linear (gamma=1.0) gamma space prior to (possible) blending with, and writing to the Color Buffer. This permits higher quality image blending by performing the blending on colors in linear gamma space.

This function is enabled on a per-texture basis by use of a surface format with "_SRGB" in its name. If enabled, the pre-filtered texel RGB color to be converted from gamma=2.4 space to gamma=1.0 space by applying a ^(1/2.4) = ^0.4167 exponential function.

2.7 Multisampled Surface Behavior

The ld message has added an additional parameter for sample index (si) to support unfiltered loading from a multisampled surface.

The sampleinfo message returns specific parameters associated with a multisample surface. The resinfo message returns the height, width, depth, and MIP count of the surface (in units of *pixels*, not samples).

Any of the other messages (sample*, LOD, load4) used with a (4x) multisampled surface would sample a surface with double the height and width as indicated in the surface state. Each pixel position on the original-sized surface is replaced with 2x2 samples that have the following arrangement:

sample 0	sample 2
sample 1	sample 3

This behavior is useful when implementing the multisample resolve operation by selecting MAPFILTER_LINEAR and rendering a full-screen rectangle half the size in each dimension of the source texture map (multisampled surface). If pixel offsets are set correctly, each pixel is the average of the four underlying samples.

2.7.1 Multisample Control Surface

Three new messages have been defined for the sampling engine, *ld_mcs*, *ld2dms*, and *ld2dss*. A pixel shader kernel sampling from an multisampled surface using an MCS must first sample from the MCS surface using the *ld_mcs* message. This message behaves like the *ld* message, except that the surface is defined by the MCS fields of SURFACE_STATE rather than the normal fields. The surface format is effectively R8_UINT for 4x surfaces and R32_UINT for 8x surfaces, thus data is returned in unsigned integer format. Following the *ld_mcs*, the kernel issues a *ld2dms* message to sample the surface itself. The integer value from the MCS surface is delivered in the mcs parameter of this messages.

Since *sample* is no longer supported on multisampled surfaces, the multisample resolve must be done using *ld2dms*. For surfaces with **Multisampled Surface Storage Format** set to MSFMT_MSS and **MCS Enable** set to enabled, an optimization is available to enable higher performance for compressed pixels. The *ld2dss* message can be used to sample from a particular sample slice on the surface. By examining the MCS value, software can determine which sample slices to sample from. A simple optimization with probable large return in performance is to compare the MCS value to zero (indicating all samples are on sample slice 0), and sample only from sample slice 0 using *ld2dss* if MCS is zero. Sample slice 0 is the pixel color in this case. If MCS is not zero, each sample is then obtained using *ld2dms* messages and the results are averaged in the kernel after being returned. Refer to the multisample storage format in the GPU Overview volume for more details.

2.8 Denoise/Deinterlacer

The Denoise/Deinterlacer function takes a 4:2:0 or 4:2:2 video stream and first applies a denoise filter to it and then deinterlace it.

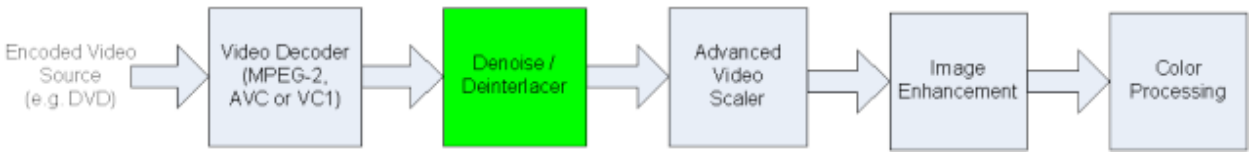
The denoise filter is applied before the deinterlacer. The denoise filter detects and tries to minimize noise in the input field, while the deinterlacer takes a field consisting of every other lines converts a field into a frame. This block also gathers statistics for a global noise estimate made in software at the end of the frame which is used in following frames to tune the denoise filter and image enhancement filter.

The deinterlacer takes the top and bottom fields of each frame and converts them into two individual frames. This block also gathers statistics for a film mode detector in software run at the end of the frame. If the film mode detector for the previous frame concludes that the input is progressive rather than interlaced then the fields will be put together in the best order rather than being interlaced.

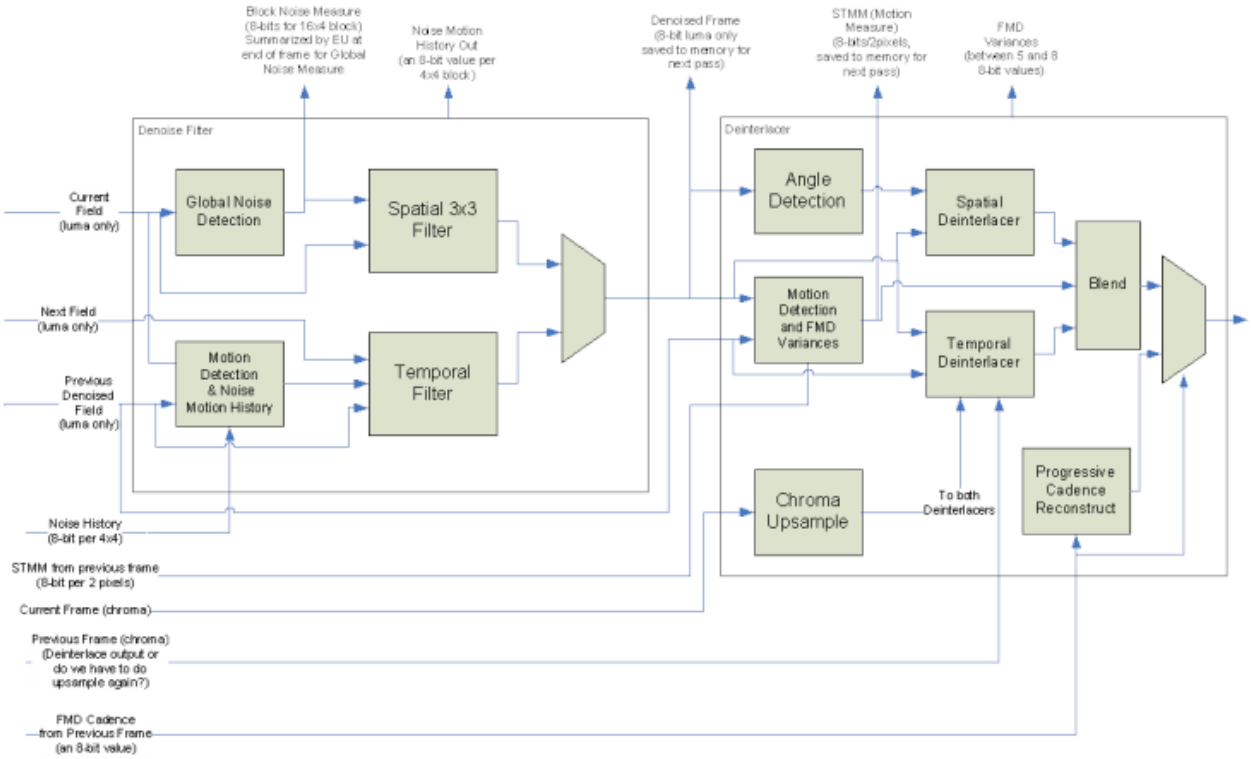
2.8.1 Introduction

2.8.1.1 Overview

This diagram shows how the Denoise/Deinterlacer fits in with the other functions of the video pipe. This is only one possible usage model, other models are possible.



Block Diagram



2.8.1.2 Features

- **Denoise Filter** – detects noise and motion and filters the block with either a temporal filter when little motion is detected or a spatial filter. Noise estimates are kept between frames and blended together. Since the filter is before the deinterlacer it works on individual fields rather than frames. This usually improves the operation since the deinterlacer can take a single pixel of noise and spread it to an adjacent pixel, making it harder to remove. The denoise filter works the same whether deinterlacing or progressive cadence reconstruction is being done.
- **Block Noise Estimate** (BNE) – part of the Global Noise Estimate (GNE) algorithm, this estimates the noise over the entire block. The GNE will be calculated at the end of the frame by combining all the BNEs. The final GNE value is used to control the denoise filter for the next frame.
- **Film Mode Detection** (FMD) Variances – FMD determines if the input fields were created by sampling film and converting it to interlaced video. If so the deinterlacer is turned off in favor of reconstructing the frame from adjacent fields. Various sum-of-absolute differences are calcluated per block. The FMD algorithm is run at the end of the frame by looking at the variances of all blocks for both fields in the frame.
- **Deinterlacer** – Estimates how much motion is occuring across the fields. Low motion scenes are reconstructed by averaging pixels from fields from nearby times (temporal deinterlacer), while high motion scenes are reconstructed by interpolating pixels from nearby space (spatial deinterlacer).
- **Progressive Cadence Reconstruction** – If the FMD for the previous frame determines that film was converted into interlaced video, then this block reconstructs the original frame by directly putting together adjacent fields.
- **Chroma Upsampling** – If the input is 4:2:0 then chroma will be doubled vertically to convert to 4:2:2. Chroma will then either go through it's own version of the deinterlacer or progressive cadence reconstruction.

When DI is enabled, the output for a 16x4 block is sent to the EU for further processing and writing to memory. When DI is disabled and DN enabled the output for a 16x8 block is sent to the EU.

Formats supported are:

NV12 is supported for hardware video decode.

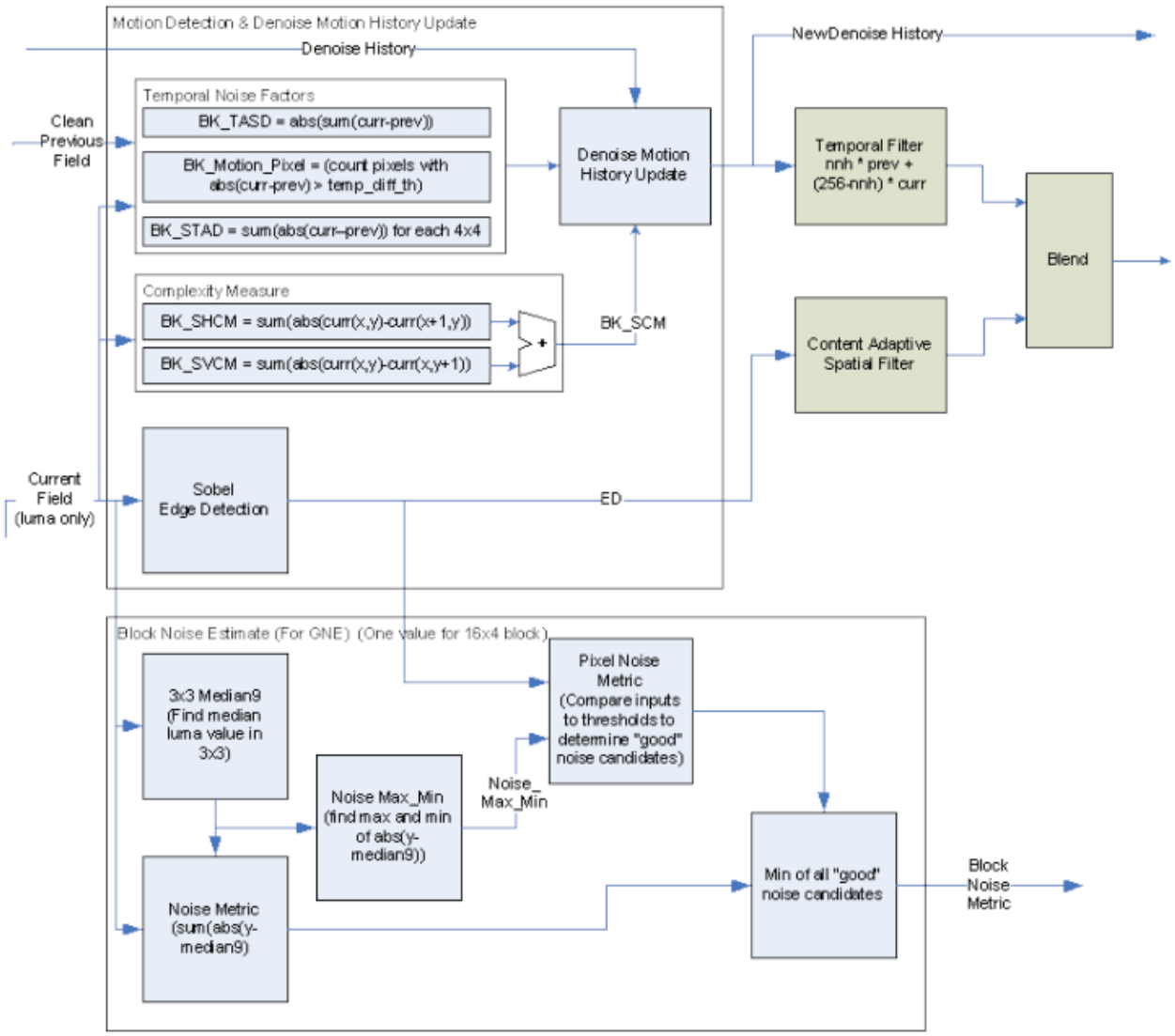
UYVY, YUY2 and NV12 are required for WHQL.

YV12 and I420 are supported for software video decode.

IMC3 and IMC4 are supported as internal temporary formats.

NV11 and P208 are not supported, since they have been removed from the WHQL logo requirement.

2.8.2 Denoise Algorithm



2.8.2.1 Motion Detection and Noise History Update

This block detection motion for the denoise filter, which it then combines with motion detected in the past in the same part of the screen. The Denoise History is both saved to memory and also used to control the temporal denoise filter.

The block calculates a number of values for updating the Denoise History. One value is calculated per 4x4 block (pixels from both fields, interleaved):

Block Sum of Temporal Absolute Difference:

BK_STAD = $\sum_{x=0}^{3}\sum_{y=0}^{3} abs(curr(x,y) - prev(x,y))$

Where curr(x,y) and prev(x,y) are lumas from the current and previous field. The previous field should have already been run through the denoise filter.

Count of motion pixels: increment BK_Motion_Pixel for every pixel in the 4x4 for which: (abs(curr(x,y) – prev(x,y)) >= temporal_diff_th.

Absolute Sum of Temporal Difference sums the differences without the initial absolute value, so that random motions will tend to cancel out:

BK_TASD = $abs(\sum_{x=0}^{3}\sum_{y=0}^{3}(curr(x,y) - prev(x,y)))$

Sum of Complexity Measure looks for differences in the spatial domain:

BK_SHCM = $\sum_{x=0}^{2}\sum_{y=0}^{3} abs(curr(x,y) - curr(x+1,y))$ // sum of 12 pixel pairs

BK_SVCM = $\sum_{x=0}^{3}\sum_{y=0}^{2} abs(curr(x,y) - curr(x,y+1))$ // sum of 12 pixel pairs

BK_SCM = BK_SHCM + BK_SVCM

Denoise Motion History Update (for an 8-bit motion history):

```
if (BK_STAD>dnmh_stad_th) or (BK_Motion_Pixel > dnmh_mp_th) { // Motion Block
   motion_block = 1;
   if (denoise_history >= 128)
new_denoise_history = denoise_history / 2;
   else
new_denoise_history = 0;
} else {  // static block
   motion_block = 0;
   if (denoise_history < 128)
new_denoise_history = 128;
   else if (denoise_history < dnmh_history_max)
new_denoise_history = denoise_history + dnmh_delta; // default value 8 for delta
   else
new_denoise_history = denoise_history;
```

```
   if ((BK_TASD > dnmh_tasd_th) and (BK_SCM < dnmh_scm_th))
new_denoise_history = 128;
}
```

2.8.2.2 Temporal Filter

For each pixel we need to filter we look at the noise history for the associated 4x4.

temporal_denoised = (new_denoise_history * prev(x,y) + (256 – new_denoise_history) * curr(x,y) +128) >> 8

2.8.2.3 Context Adaptive Spatial Filter

For each pixel in the local 3x3, compare it's luma to the lumas of the pixel to be filtered. Each pixel for which the absolute difference is less than or equal to good_neighbor_th is marked as a "good neighbor":

The filtered pixel is then equal to:

spatial_denoised = ∑ Good_neighbor luma / num_good_neighbors

The divide is implemented as a multiply by a table lookup:

spatial_denoised = ((∑Good_neighbor luma + (num_good_neighbors >>1)) * gn_q_table[num_good_neighbors-1]) >> 11

Note: The number of good neighbors varies from 1 to 9 since the center pixel is always good. Gn_q_table provides the reciprocal:

gn_q_table[9] = {2048, 1024, 682, 512, 409, 341, 292, 256, 227};

2.8.2.4 Denoise Blend

The denoise blend combines the temporal and spatial denoise outputs.

First we check to see if the temporal is out of the local range, if so we use the average of the denoised and the local limit instead:

```
if (temporal_denoised >= block_max)
    temporal_denoised=(temporal_denoised+block_max)>>1;
if (temporal_denoised < block_min)
    temporal_denoised=(temporal_denoised+block_min)>>1;
```

Where block_max and block_min are the largest and smallest luma values in the local 3x3 (can be shared with BNE calculation).

Next we decide between using the spatial and temporal denoise output:

```
    t_diff = abs(curr(x,y) – prev(x,y);
    if (t_diff < temporal_diff_th) {
        if (motion_block==1)
            denoise_out = spatial_denoised;
        else {
            if (t_diff < temp_diff_low)
```

```
            denoise_out=temporal_denoised;
        else {
            denoise_out=
              (spatial_denoised*(t_diff-temp_diff_low) +
               temporal_denoised*(temporal_diff_th-t_diff)+
               (temporal_diff_th-temp_diff_low)/2
               ) * q_table[temporal_diff_th-temp_diff_low-1]) >> 10;
    }
} else {
    denoise_out = spatial_denoised;
}
```

Motion_block is defined in section *Denoise Algorithm* above. T_diff can be limited to 6-bits to minimize the multipler gates required in the blend. A divide is eliminated by providing the reciprocal of the divisor in the q_table which is defined:

q_table[16] = {1024,512,341,256,205,171,146,128,114,102,93,85,79,73,68,64}

The following restrictions also apply:

1. Temporal_diff_th – temp_diff_low is limited in the state variable definition to the range 16 to 1.
2. Since t_diff<temporal_diff_th; (t_diff – temp_diff_low) is less than 16
3. Since t_diff>=temp_diff_low; (temporal_diff_th-t_diff) is less then or equal to 16.

The precision needed for spatial_denoised*(t_diff-temp_diff_low) is 8-bit times 4-bits to produce 12-bits. The other multiply is 8 by 5 to produce 13-bits; the extra bit is needed for 16. The multiplier to implement the divide will be a 13-bit times the 11-bit number out of q_table, but this could be reduced by implementing a 13x9 bit multiplier with the top 2 bits controlling a mux since the only table entries that use them are 1024 and 512.

2.8.3 Block Noise Estimate (part of Global Noise Estimate)

Edge detection is done on every pixel in the 16x4 (DI enabled) or 16x8 (DN only) by estimating a gradient on the 3x3 neighborhood of pixels in the current field. The calculation only uses a multiply of 2, so shifts and add are all that is needed. Currently only vertical and horizontal edges are detected, 45 degrees is a potential improvement.

Hrz Edge = abs(c(x-1,y-1) +2*c(x,y-1) +c(x+1,y-1) –c(x-1,y+1) –2*c(x,y+1) –c(x+1,y+1))

Vrt Edge = abs(c(x-1,y-1) +2*c(x-1,y) +c(x-1,y+1) –c(x+1,y-1) –2*c(x+1,y) –c(x+1,y+1))

The Hrz_Edge and Vrt_Edge are added together and if the sum is greater than bne_edge_th then an edge is detected:

ED = (Hrz_Edge +Vrt_Edge) >> 3

- median9 – the median of the 9 luma values for the 3x3 neighborhood pixels is used. Median5, the median of the pixels above/below/right/left/center may be satisfactory as a lower gate count solution.
- for each pixel luma “y” in 3x3: noise_metric = sum(y – median9)
- noise_min = min(abs(y-median9)) - min of all 9 ys in 3x3

- noise_max = max(abs(y-median9)) – max of all 9 ys in 3x3
- noise_min_max = noise_max(x,y) – noise_min(x,y)
- pixel_noise_metric = noise_metric if (ED(x,y) < bne_edge_th) and (noise_max_min(x,y) < bne_nn_th) block_noise_estimate = min of all pixel_noise_metrics that pass the if test in the 16x4 (use 255 if no pixels pass the test)

If the block_noise_estimate is less than 255 then it is added to a sum gathered across the entire frame. The summation will need to be 23-bits wide to be able to sum 8-bit values for all 32,400 blocks in a 1920x1080 frame. In addition, there will be a count of the number of blocks in the sum. The data will be written to memory at the end of the frame. Two sets of counters are needed to support 2 simultaneous streams. The streams are distinguished by the dndi_stream_id state variable in the DI state.

The per block block_noise_estimate is also sent to the EU in the output message for possible use by the video encoder.

2.8.4 Deinterlacer Algorithm

The overall goal of the motion adaptive deinterlacer is to convert an interlaced video stream made of fields of alternating lines into a progressive video stream made of frames in which every line is provided.

If there is no motion in a scene, then the missing lines can be provided by looking at the previous or next fields, both of which have the missing lines. If there is a great deal of motion in the scene, then objects in the previous and next fields will have moved, so we can't use them for the missing pixels. Instead we have to interpolate from the neighboring lines to fill in the missing pixels. This can be thought of as interpolating in time if there is no motion and interpolating in space if there is motion.

This idea is implemented by creating a measure of motion on a per 2 pixel basis called the Spatial-Temporal Motion Measure (STMM). If this measure shows that there is little motion in an area around the pixels, then the missing pixels are created by averaging the pixel values from the previous and next frame. If the STMM shows that there is motion, then the missing pixels are filled in by interpolating from neighboring lines with the Spatial Deinterlacer (SDI). The two different ways to interpolate the missing pixels are blended for intermediate values of STMM to prevent sudden transitions.

The Deinterlacer uses two frames for reference. The current frame contains the field that we are deinterlacing. The reference frame is the closest frame in time to the field that we are deinterlacing – if we are working on the 1st field then it is the previous frame, if it is the 2nd field then it is the next frame.

2.8.4.1 Spatial-Temporal Motion Measure

This algorithm combines a complexity measure with a estimate of motion. This prevents high complexity scenes from incorrectly causing motion to be detected. It is calculated for a set of pixels 2 wide by 1 high.

Complexity is measured in the vertical and horizontal directions with the SVCM and SHCM. For each set of 2 pixels which need to be interpolated, a window of pixels is used that is 4 wide and 5 high - +/-1 pixel in X and +/- 2 pixels in Y. The pixels values are taken from both the current and previous field - for example, if we are deinterlacing the top field then lines y+2,y, and y-2 will come from the top field; while line y+1 and y-1 will come from the bottom field.

Spatial vertical complexity measure (SVCM) is a sum of all the differences in the vertical direction for a window around the current pixels. If we take x,y=0,0 as the left pixel of our 2x1 then:

$$\text{SVCM} = \sum_{x=0}^{1}\sum_{y=0}^{2} abs(c(x,y) - c(x,y-2))$$

Where c(x,y) is the luma value at that x,y location in the current frame. Note that we are skipping by 2 in the Y direction to ensure that the compares are only done with lines from the same field.

Spatial horizontal complexity measure (SHCM) is a sum of differences in the horizontal direction.

SHCM = $\sum_{x=-1}^{1}\sum_{y=-1}^{y=1} abs(c(x,y)-c(x+1,y))$

The vertical edge complexity measure (VECM) is a sum of difference in the horizontal direction similar to SHCM, but uses different pixels from the window.

VECM = $\left(\left(\sum_{y=-2}^{y=2} abs(c(x,y)-c(x+1,y))\right) * \text{vecm\_mul}\right)$ >>5

Temporal Difference Measure (TDM) is a measure of differences between pairs of fields with the same lines. It uses filtered versions of c(x,y) from the current frame and r(x,y) from the reference frame (either the previous or next frame).

The filter used is a cross filter which uses the pixels above, below, to the right and to the left of the needed pixel in the same field. When denoise filter is enabled, the filter input c(x,y) is a denoised pixel only if -2<=y<=6 for dndi_topfirst=1, and -3<=Y<=5 for dndi_topfirst=0. Note that r(x,y) is a denoised pixel regardless of y.

c'(x,y) = (2*c(x,y) +c(x-1,y) +c(x+1,y) +2*c(x,y-2) +2*c(x,y+2)) >> 3 (Done for both c(x,y) and r(x,y))

TDM = $\sum_{x=-1}^{2}\sum_{y=-2}^{2} abs(c'(x,y)-r'(x,y))$

STMM is then calculated by :

STMM = ((TDM >>tdm_shift1)<<tdm_shift2) / (SCM >> 4) + stmm_c2)

where SCM = max(0, SVCM+SHCM-VECM). Tdm_shift1 is used to quantize the STMM result, while Tdm_shift2 is used to set the STMM range. Tdm_shift1 can range from 4 to 6; since TDM has 13 bits this results in between 9 and 7 bits of precision. Tdm_shift2 can range from 6 to 8, producing a value between 17 and 13 bits, of which only 9-bits are non-zero. The divide can be implemented by a 8-bit reciprocal table followed by an 9 -bit x 8-bit multiply by the TDM value, which finally produces an output of 8-bits.

```
STMM is then smoothed with an exponential moving average with the STMM saved from the
previous field:
 if (STMM > stmm_md_th)
 STMM2 = (stmm_trc1 * STMM_s + (256-stmm_trc1)*STMM) / 256
 else
STMM2 = (stmm_trc2 * STMM_s + (256-stmm_trc2)*STMM) / 256
```

with state variables stmm_trc1 (typical value 64), stmm_trc2 (typical value 200), and stmm_md_th.

This process prevent sudden changes in STMM, though STMM over a certain value uses a smaller smoothing constant (c1) which allows it to change faster. STMM2 is stored to memory to be read as STMM_s by the next frame.

One final step is used to prevent sudden drops in STMM in the horizontal direction – taking the maximum of the STMM on the right and left sides:

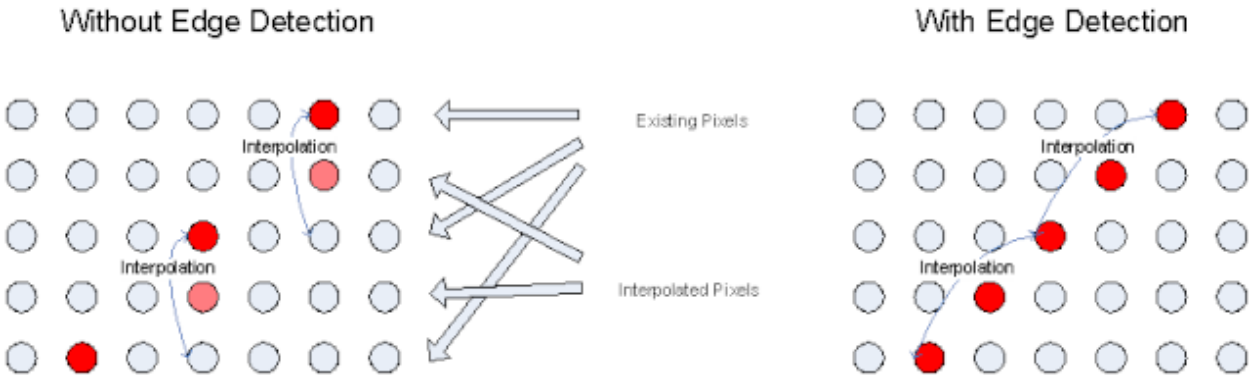
STMM3(x) = max (STMM2(x-2), STMM2(x), STMM2(x+2))

The resulting STMM3 will be used as a blending factor between the spatial and temporal deinterlacer.

2.8.4.2 Spatial Deinterlacer Angle Detection

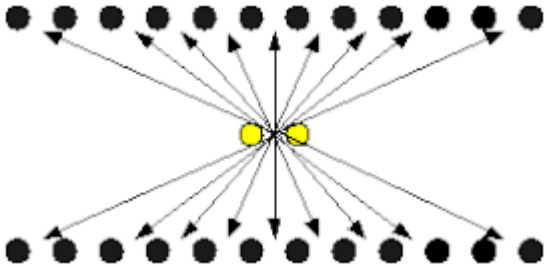
Deciding the best pixels to interpolate in the current field is the job of the spatial deinterlacer. The simplest method would be to interpolate directly from the pixels above and below the missing pixels, but this can look bad; edges and lines particularly look jagged with this solution.

A better solution is to detect the direction of edges in the pixel neighborhood and interpolate along the edge direction.



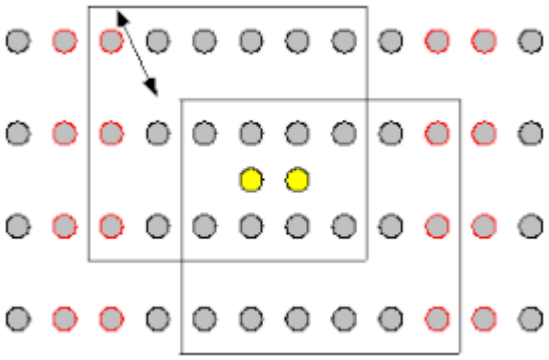
Edge detection is done per 2 pixels to lower the compute needed (may change in this implementation depending on quality). Edge detection is done by taking a window of pixels around the pixels of interest and comparing with a window offset in the direction being tested. The more simularity between the windows the more likely it is that the movement is in the direction of an edge.

We test 9 different directions to pick the best edge: vertical, +/-45°, +/-27°, +/-18° and +/-11 degrees. The window offset for 45° x+/-1, likewise the offset of 27° is x+/-2, 18° is x+/-3, and 11° is x+/-5. X+4 is not used because the gap between 18° and 11° is too small to make it worth checking.



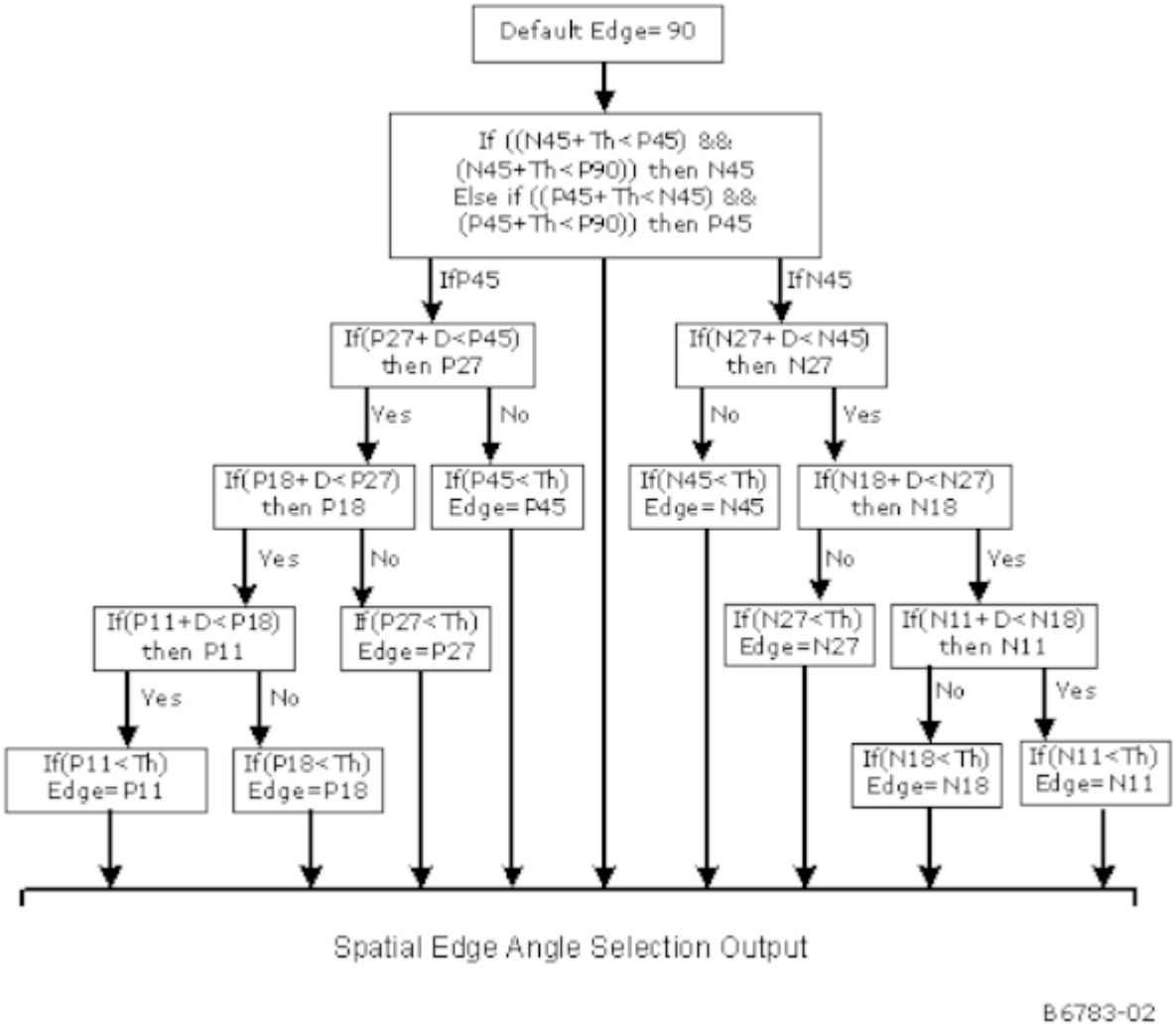
Use x,y=0,0 for the left pixel of the pair that we want to interpolate, and xoffset is the offset described in the above paragraph. The equation for each angle checked is:

AngleCost_6x3 = $\sum_{x=-2}^{3} \sum_{y=-2,0,2} abs(n(x+xoffset, y+1) - n(x-xoffset, y-1))$



The above picture illustrates the 45 degree angle compution – taking the sum-of-absolute differences of the two 6x3 blocks around the 2 pixels that need an angle estimated. Each block is offset by 1 in Y and X in opposite direction. The offset in X is larger for the other angles, of course. Angle detection requires up to 7 pixels (offset of 5 plus 2 to get all the pixels in the 6x3) on the right and left of the output block, requiring the input to the deinterlacer from the denoise to be 16 + 7 + 7, or 30 pixels.

Once we have all the angle values, the final decision is done by comparing them with each other. In the following diagram N45 indicates the AngleCost_6x3 for -45°, likewise P27 is the value for +27°, etc. Th and D are constants used to fine tune the algorithm.



Any missing arcs in the above diagram use the default edge of 90 degrees; for example if the lower left box has P11 >= Th then the default will be used.

2.8.4.3 Angle Robustness Check

Three special checks are made to eliminate incorrect angle detection.

Fallback Mode 1

Moving regions with fine details can confuse the angle detection. This fallback mode will detect fine details and fall back to 90 degrees if they are detected.

$$\text{SUM\_H1(x,y)} = \sum_{s=-2}^{3} abs(c(x+s,y) - c(x+s+1,y))$$

This sum is similar to SHCM, but over a horizontal line of -2 to +3 only.

$$\text{SUM\_H2(x,y)} = \max_{s=-2,-1,\ldots,3} (abs(c(x-2,y) - c(x+s,y)) + abs(c(x+s,y) - c(x+4,y))$$

if (SUM_H1(y-1) + SUM_H1(y+1) > SUM_H2(y-1) + SUM_H2(y+1) + sdi_t1 &&

SUM_H1(y-1) + SUM_H1(y+1) >= sdi_t2)Then use 90 degree

The final decision for each pixel is done using the sums from above and below the current Y.

Fallback Mode 2

Sometimes the 6x3 angle detection window makes mistakes due to pixels on the edge of the window. Adding a check using a 2x1 window fixes these problems:

If(AngleCost_6x3(90 degree) + (AngleCost_2x1(90 degree)<<3) <

AngleCost_6x3(best angle) + ((AngleCost_2x1(best angle) + sdi_angle2x1)<<3)) then use 90 degree

AngleCost_2x1 is the same as AngleCost_6x3 with a much smaller window:

AngleCost_2x1 = $\sum_{x=0}^{1} abs(n(x+xoffset, y+1) - n(x-xoffset, y-1))$

AngleCost_2x1 can be collected during the calculation of AngleCost_6x3.

Horizontal Median

One final step is used to prevent sudden angle changes – the angle detected for the pixel pair is compared to the angle detected for the pixels to the right and left and the median of the 3 is the angle finally used:

angle_final(x) = median3(angle(x-2), angle(x), angle(x+2))

2.8.4.4 Spatial Deinterlacer Interpolation

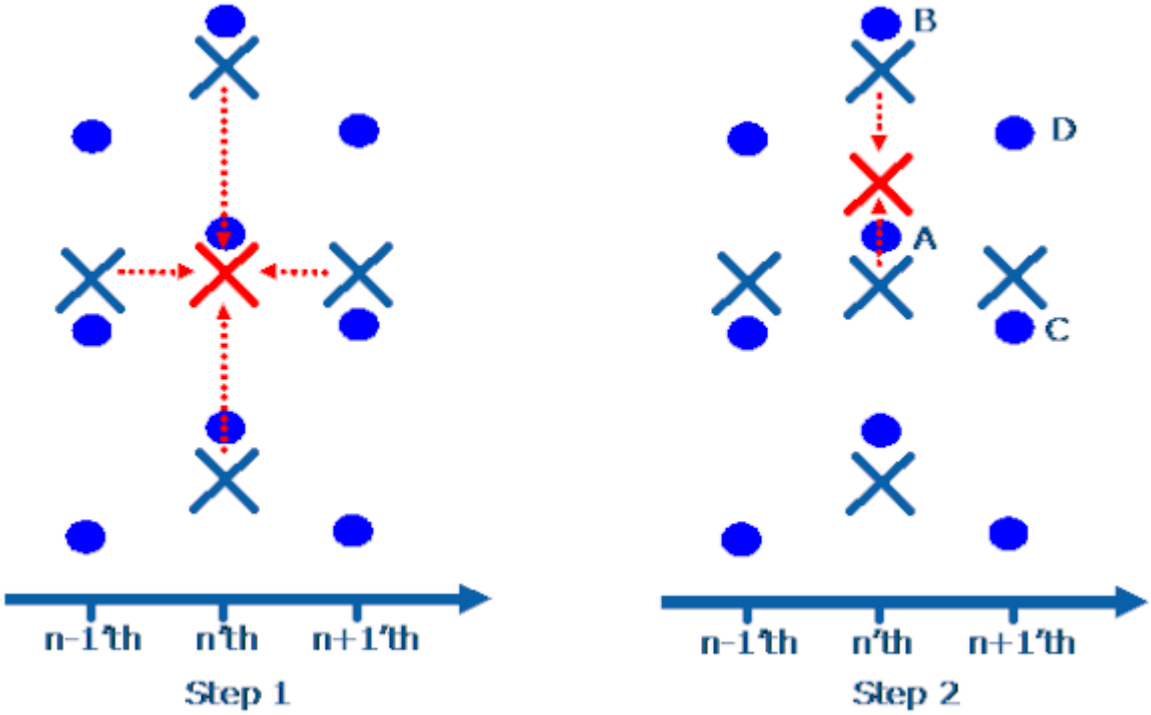
Once the best angle is picked, the interpolation is done on a per pixel basis. Both the chroma and luma need to be interpolated (see section *Chroma Up Sampler* for chroma). Only 422 output is needed, so there will be a chroma pair for each 2 lumas. The interpolation itself is very simple: take a pixel from the line above and the line below along one of the 9 possible angles, and average the 8-bit luma and chroma values to get the result pixel. We will do 2 lumas per clock to get enough performance.

2.8.4.5 Chroma Up-Sampler

The DN/DI block supports 4:2:0, 4:1:1 and 4:2:2 inputs, but only outputs 4:2:2. For 4:2:0 and 4:1:1 the chroma needs to be up-sampled to 4:2:2 before interpolation.

The 4:2:0 input has chroma at ¼ the rate of the luma; ½ in the horizontal and ½ in the vertical directions. The output needs to be 4:2:2, where chroma is ½ the rate of luma; ½ the horizontal but the same in the vertical direction. Then chroma can be de-interlaced in the vertical direction. For luma we are working with 16x4 blocks, so for chroma we will have 8x2 in 4:2:0 and 8x4 in 4:2:2.

The 4:2:0 to 4:2:2 conversion requires doubling the chroma in the vertical direction to match the luma:



The chroma is doubled by a simple interpolation in both time and space. In the following equations, pixel locations are specified as u(field, x_location, y_location). Field=n would be from the current field, n-1 is from the previous field, and n+1 is from the next field. The Cr and Cb X and Y values are ½ the luma values to map to the smaller area.

temporal_cr = (cr(n-1,x,y) + cr(n+1,x,y)) / 2// Simple average in time

spatial_cr = (cr(n,x,y-1) + cr(n,x,y+1)) / 2// Simple average in vertical space

if (STMM3 < stmm_min)

  new_cr = temporal_cr

else if (STMM > stmm_max)

  new_cr = spatial_cr

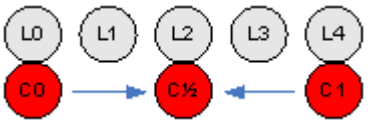
else

  new_cr = ((STMM3 – stmm_min) * spatial_cr + (stmm_max - STMM3) * temporal_cr) >> stmm_shift

Note that this simple chroma interpolation is not correct, since the chroma sample position is ¼ of a pixel different between 420 and 422. The polyphase filter in the scaler will be used to correct this inprecision by modifying the filter coefficients in software.

For performance a single Cr and Cb has to be produce per clock in this stage to match the 2 pixel per clock performance goal.

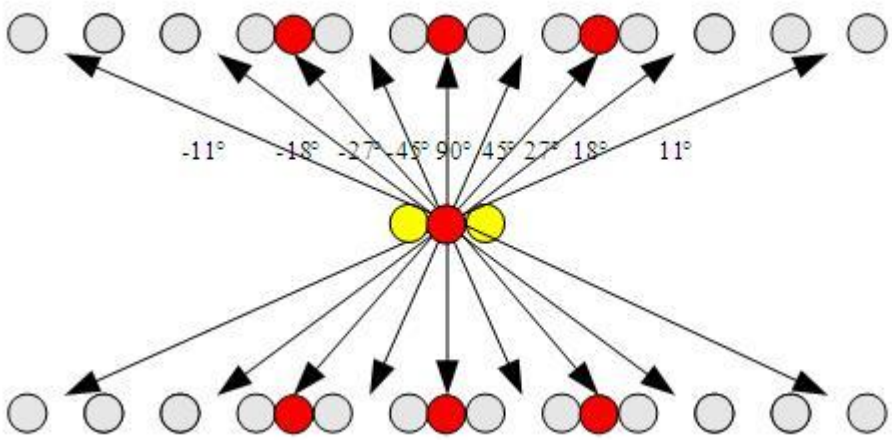
4:1:1 also has chroma at ¼ the rate of luma; ¼ in the horizontal direction and the same in the vertical direction. To convert to 4:2:2 we need to double the chroma horizontally. This will be done by averaging the chromas to the right and left to produce the new chroma.



The above diagram shows how the existing chroma values (both U and V) are averaged between C0 and C1 to produce the new C½. C0 is the chroma asociated with lumas L0 through L3, while C1 is associated with L4 through L7.

2.8.4.6 Chroma Deinterlace

The next step is to do the deinterlacing. Chroma uses the output of the luma angle decision, but reduces the number of angles. The actual spatial deinterlace algorithm is a little different for chroma, since there are only 1 chroma per 2 lumas: some of the chromas are missing and must be filled in.



The diagram shows the chromas used in red. Only 90°, -27° and 27° are directly available. The chromas for +/-45° are derived by a simple average of the 90° and 27° chromas. +/-18° and +/-11° both use the chroma for +/-27°.

2.8.4.6.1 Static Image Fallback Mode

This algorithm has a problem with static images – alternate fields use different luma angle detections and can select different angles, causing noticable flicker. Rather than calculating a separate set of angles for chroma, we instead will blend with STMM so that a static image will use 90 degrees.

```
if (STMM3 < stmm_min)
   chroma_sdi = chroma90degree
else if (STMM > stmm_max)
   chroma_sdi = chroma_3angle
else
   chroma_sdi = (chroma90degree * (stmm_max – STMM3) + chroma_3angle * (STMM3 – stmm_min)) >> stmm_shift
```

2.8.4.7 Temporal Deinterlacer and Final Deinterlacer Blend

The temporal deinterlacer is a simple average between the previous and next field; when deinterlacing the 1st field of current the average will be between the 2nd field of previous and the 2nd field of current.

The interpolation between spatial and temporal:

```
if (STMM3 < stmm_min)
  deinterlace_out = tdi;
else if (STMM3 > stmm_max)
  deinterlace_out = sdi;
else
 deinterlace_out = (sdi * (STMM3 – stmm_min) + tdi * (stmm_max – STMM3)) >> stmm_shift
```

2.8.4.8 Progressive Cadence Reconstruction

When the FMD for the previous frame indicates that a progressive mode is being used rather than interlaced, the luma and chroma will be taken from adjacent fields rather than spatially interpolated. The exact fields needed depend on state variables written to memory by a thread at the end of the previous frame. The thread will use the FMD variances written to memory via CSunit on the flush at the end of a frame.

Since we are deinterlacing 2 fields at a time – one from the previous frame and one from the current frame (see section *Implementation Overview*) we will need a state variable which says how each one should be put together. In each case there are only two possibilities – either the field should be put together with the matching field in the same frame or it should be put together with the adjacent field in the other frame.

If we are deinterlacing the 2nd field from frame N and the 1st field from frame N+1, then the FMD decision (which is made on frame boundaries) will be from frame N-1.

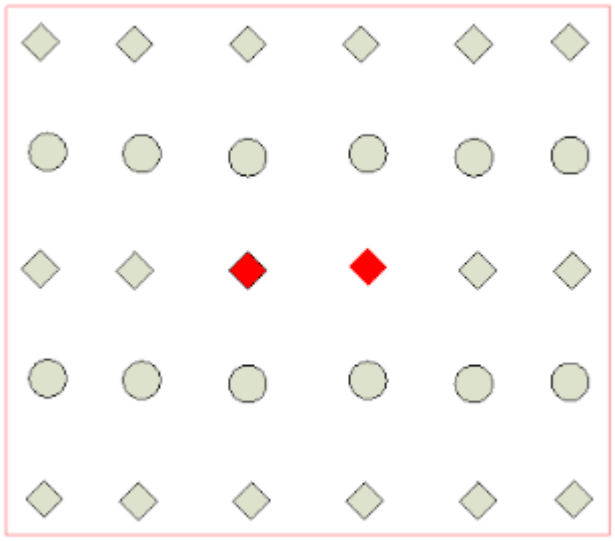
Chroma is reconstructed the same as luma – only the first step of doubling chroma is done in the chroma upsampling block for the two needed fields.

2.8.4.9 Motion Search

Motion will be estimated independently for each horizontal pair of pixels in the 16x4 block. The area around each pixel pair will be compared to areas in adjacent fields with different X/Y offsets. 16 different offsets, or motion vectors, will be examined in this order:

Y= -2, X = -1, 0, 1
Y = 0, X = -6, -5, -4, -3, -2, 2, 3, 4, 5, 6
Y = 2, X = -1, 0, 1

The area to be compared around the pixel pair is a 6 wide by 5 high window - 2 pixels on right and left and 2 lines above and below. The lines above and below are from both fields, so a total of 3 lines from the same field and 2 lines from the complement field are compared to lines in 2 fields from an adjacent frame.



The motion estimation equation for a pixel pair is:

$$SAD = \sum_{i=x-w}^{x+w+1} \sum_{j=y-h}^{y+h} \left| p_{ref}(i + M_x, j + M_y) - p_{curr}(i, j) \right|$$

$$(h = 2 \; and \; w = 2)$$

Mx, My is the motion vector offset being tested, and x,y is the location of the leftmost pixel of the pair. The motion vector with the smallest SAD is kept as the best motion estimate; if two motion vectors have the same SAD then the last one tested will be kept.

2.8.4.10 Robustness Checks

The motion estimate output goes through 2 checks to make sure it is not an aberration – a smoothness check and a consistency check.

2.8.4.10.1 Consistency Check

The consistency check is done per pixel and makes sure that the pixels we are interpolating for MC have a lower delta than the ones that would be interpolated for spatial DI:

$$\left| P_{cur\_opp}(x - Edge, y - 1) - P_{cur\_opp}(x + Edge, y + 1) \right| > \left| P_{DI}(x, y) - P_{DI\_cur}(x, y) \right|$$

$$\&\& \left| P_{DI}(x, y) - P_{DI\_cur}(x, y) \right| < MC\_pixel\_consistency\_TH(default: 25)$$

Here Edge is the delta found by SDI which corresponds to the best angle. ***MC_pixel_consistency_TH*** (U6)is a state parameter.

P_{DI_cur} is defined as: (same definition as in the motion compensation section)

- If(Mx%2 == 0 && (My/2)%2 == 0)

$$P_{IM\_cur}(x,y) = P_{cur\_same}(x - M_x/2, y - M_y/2);$$

- If(Mx%2 == 1 && (My/2)%2 == 0)

$$P_{IM\_cur}(x,y) = \begin{cases} \text{AVG}\left(P_{cur\_same}(x - M_x/2, y - M_y/2), P_{cur\_same}(x - M_x/2 - 1, y - M_y/2)\right); & \text{if } (M_x \geq 0) \\ \text{AVG}\left(P_{cur\_same}(x - M_x/2, y - M_y/2), P_{cur\_same}(x - M_x/2 + 1, y - M_y/2)\right); & \text{if } (M_x < 0) \end{cases}$$

- If(Mx%2 == 0 && (My/2)%2 == 1)

$$P_{IM\_cur}(x,y) = \text{AVG}\left(P_{cur\_same}(x - M_x/2, y - M_y/2 - 1), P_{cur\_same}(x - M_x/2, y - M_y/2 + 1)\right);$$

If(Mx%2 == 1 && (My/2)%2 == 1)

$$P_{IM\_cur}(x,y) = \begin{cases} \text{AVG}\begin{pmatrix} P_{cur\_same}(x - M_x/2, y - M_y/2 - 1), P_{cur\_same}(x - M_x/2 - 1, y - M_y/2 - 1), \\ P_{cur\_same}(x - M_x/2, y - M_y/2 + 1), P_{cur\_same}(x - M_x/2 - 1, y - M_y/2 + 1) \end{pmatrix}; & if\ (M_x \geq 0) \\ \text{AVG}\begin{pmatrix} P_{cur\_same}(x - M_x/2, y - M_y/2 - 1), P_{cur\_same}(x - M_x/2 + 1, y - M_y/2 - 1), \\ P_{cur\_same}(x - M_x/2, y - M_y/2 + 1), P_{cur\_same}(x - M_x/2 + 1, y - M_y/2 + 1) \end{pmatrix}; & if\ (M_x < 0) \end{cases}$$

2.8.4.10.2 Smoothness Check

The smoothness check compares the motion vector found for neighboring pixel pairs. The neighbors are different for different locations to make sure it stays within the local 4x4. Each pixel pair has 3 sets of comparison with neighbor pixel pair within the 4 by 4: 2 sets of X/Y comparisons for the vertical direction and one set of X/Y comparisons for the horizontal direction.

For lines 1 and 2 in the 16x4:

$$If\left(abs\left(MV_x(x,y) + MV_x(x,y+1)\right)\right) <= smooth\_mv\_th$$
$$AND\ abs\left(MV_y(x,y) + MV_y(x,y+1)\right) <= smooth\_mv\_th$$
$$AND\left(abs\left(MV_x(x,y) - MV_x(x,y+2)\right)\right) <= smooth\_mv\_th$$
$$AND\ abs\left(MV_y(x,y) - MV_y(x,y+2)\right) <= smooth\_mv\_th$$

Where ***smooth_mv_th***(U2) is a state parameter.

This equation ensures that the pixel pair 1 and 2 lines below have motion vector X and Y components (MVx & MVy) that are within a threshold of the best motion vector for the current pixel pair. The compares with y+1 use "+" rather than "-" since they are comparing motion vectors in the opposite field, which have motion vectors pointing in the opposite direction, since they are using the current field as their reference. For example, if the current pixel has a motion vector of (4,2), the motion vector of x,y+1 would be the same if it is (-4,-2).

For lines 3 and 4 in the 16x4:

$$If\left(abs\left(MV_x(x,y) + MV_x(x,y-1)\right)\right) <= smooth\_mv\_th$$
$$AND\ abs\left(MV_y(x,y) + MV_y(x,y-1)\right) <= smooth\_mv\_th$$
$$AND\left(abs\left(MV_x(x,y) - MV_x(x,y-2)\right)\right) <= smooth\_mv\_th$$
$$AND\ abs\left(MV_y(x,y) - MV_y(x,y-2)\right) <= smooth\_mv\_th$$

For pixel pairs with the first pixel location x%4 == 0 (low X in the 4x4):

$$If\left(abs\left(MV_x(x,y)-MV_x(x+2,y)\right)\right)<=smooth\_mv\_th$$
$$AND\ abs\left(MV_y(x,y)-MV_y(x+2,y)\right)<=smooth\_mv\_th$$

For pixel pairs with the first pixel location x%4 != 0 (high X in 4x4):

$$If\left(abs\left(MV_x(x,y)-MV_x(x-2,y)\right)\right)<=smooth\_mv\_th$$
$$AND\ abs\left(MV_y(x,y)-MV_y(x-2,y)\right)<=smooth\_mv\_th$$

When all 3 comparisons pass the threshold, the smoothness check is passed.

2.8.4.11 Motion Comp

The MCDI output is an average done per pixel on pixels chosen from adjacent field.

There are 4 different equations depending on the motion vector (Mx, My):

If (Mx%2 ==0) && (My == 0) then $P_{DI}(x,y)=P_{ref\_same}(x+M_x/2,y+M_y/2);$

If (Mx%2 ==1) && (My == 0) then

$$P_{DI}(x,y)=\begin{cases}\text{AVG}\left(P_{ref\_same}(x+M_x/2,y+M_y/2),P_{ref\_same}(x+M_x/2+1,y+M_y/2)\right); & \text{if } (M_x\geq 0)\\ \text{AVG}\left(P_{ref\_same}(x+M_x/2,y+M_y/2),P_{ref\_same}(x+M_x/2-1,y+M_y/2)\right); & \text{if } (M_x<0)\end{cases}$$

If (Mx%2==0) && abs(My) == 2 then

$$P_{DI}(x,y)=\text{AVG}\left(P_{ref\_same}(x+M_x/2,y+M_y/2-1),P_{ref\_same}(x+M_x/2,y+M_y/2+1)\right);$$

If (Mx%2==1) & abs(My) == 2 then

$$P_{DI}(x,y)=\begin{cases}\text{AVG}\begin{pmatrix}P_{ref\_same}(x+M_x/2,y+M_y/2-1),P_{ref\_same}(x+M_x/2+1,y+M_y/2-1),\\ P_{ref\_same}(x+M_x/2,y+M_y/2+1),P_{ref\_same}(x+M_x/2+1,y+M_y/2+1)\end{pmatrix}; & if\ (M_x\geq 0)\\ \text{AVG}\begin{pmatrix}P_{ref\_same}(x+M_x/2,y+M_y/2-1),P_{ref\_same}(x+M_x/2-1,y+M_y/2-1),\\ P_{ref\_same}(x+M_x/2,y+M_y/2+1),P_{ref\_same}(x+M_x/2-1,y+M_y/2+1)\end{pmatrix}; & if\ (M_x<0)\end{cases}$$

For all these equations, if more vareties of My are used than -2,0,2 then we need to use (My/2)%2==0) instead of My==0, and (My/2)%2==1 instead of abs(My)==2.

2.8.4.12 Merge with TDI & SDI

The MADI equation used in Gen6 was:

if (STMM3 < stmm_min)

deinterlace_out = tdi;

else if (STMM3 > stmm_max)

deinterlace_out = sdi;

Else

deinterlace_out = ((STMM3 – stmm_min) * sdi + (stmm_max - STMM3) * tdi) >> stmm_shift

Where STMM3 is a measure of the complexity of the scene and how much motion is in it.

The equation with MCDI is:

if (STMM3 < stmm_min)

Deinterlace_out = tdi;

else if (STMM3 > stmm_max)

deinterlace_out = DItemp;

else

deinterlace_out = ((STMM3 – stmm_min) * DItemp + (stmm_max - STMM3) * tdi) >> stmm_shift

Where DItemp is defined below:

Content Adaptive Thresholding:

We denote the best_ME_SAD as the minimal SAD value for the MV candidates. Best_ME_SAD and Best_SAD_Angle_cost are measured based on the block of pixels. The new control equation with MCDI is calculated per pixel:

If ((best_ME_SAD <= ***CAT_TH1***)

If (Consistency check is passed && Smoothness check is passed)

DItemp = MCDI;

Else

DItemp = sdi;

Else if (***CAT_TH1***<best_ME_SAD < CAT_TH2****30***) {

If (Consistency check is passed && Smoothness check is passed) AND

(SDI_angle =90 degree) AND

(best_ME_SAD + ***SAD_Tight_TH*30*** < Best_SAD_Angle_cost*2) AND

{(MCDI==median3(MCDI, $P_{curr\_opp}(x, y-1)$, $P_{curr\_opp}(x, y+1)$) ||

(Min[abs(MCDI - $P_{curr\_opp}(x, y-1)$), abs(MCDI - $P_{curr\_opp}(x, y+1)$)] <

NeighborPixel_TH)}

DItemp = MCDI;

Else

DItemp = sdi;

} Else

DItemp = sdi

Where ***CAT_TH1***(U2, default = 0), ***SAD_Tight_TH*** (U4, default=5) and ***NeighborPixel_TH***(U4, default=10) are state parameters. CAT_TH2 is a content adaptive value dependent on SCM. SCM = SHCM+SVCM from the spatial complexity measurement.

If (SCM < SCM_A)

CAT_TH2 = ***SAD_THA***;

Else if (SCM > SCM_B)

CAT_TH2 = ***SAD_THB***;

Else

CAT_TH2 = SCM / ***CAT_slope;***

Where ***CAT_slope*** (U4: default value 10). ***SAD_THA*** (U4, default 5) and ***SAD_THB*** (U4, default 10) are state parameters, and SCM_A and SCM_B are derived parameters:

SCM_A = ***CAT_slope * SAD_THA***;// 4-bit * 4-bit to produce 8-bit value

SCM_B = ***CAT_slope * SAD_THB***;// 4-bit * 4-bit to produce 8-bit value

2.8.5 Field Motion Detector

The Field Motion Detector is generated in either the EU or in the driver with a set of differences gathered across entire fields. It is used to detect when a non-interlaced source like a film has been converted to interlaced video – in this case there will be pairs of fields which can be put back together to make frames rather than interpolating. The variances for the block are sent to the VSCunit to be summed across the entire frame. The results are available in MMIO registers.

2.8.5.1 Simple Differences

The first set of variances are simply a sum of absolute pixel differences. The equations are done for every pixel with an even y coordinate:

variance[0] += Diff_cTpT = (c(x,y) – p(x,y)) ^ 2; – difference between pixels from the top fields of the current and previous frame.

variance[1] += Diff_cBpB = (c(x,y+1) – p(x,y+1)) ^ 2; – difference between pixels from the bottom fields of the current and previous frame.

variance[2] += Diff_cTcB = (c(x,y) – c(x,y+1)) ^ 2; – difference between pixels from the top field and bottom field in the current frame.

variance[3] += Diff_cTpB = (c(x,y) – p(x,y+1)) ^ 2; – difference between pixels from the top field of the current frame and bottom field of previous frame.

variance[4] += Diff_cBpT = (c(x,y+1) – p(x,y)) ^ 2; – difference between pixels from the bottom field of the current frame and top field of previous frame.

The variances summed for each 16x4 block are divided by 16 before adding them to the sum for the frame to make sure the frame-level sum fits in a 32-bit register.

2.8.5.2 Counter Variances

The rest of the variances are counters for variance conditions as described in the following code:

```
// Same field difference of the current frame
diff_cTcT = (c(x,y) - c(x,y+2)) ^ 2;
```

```
diff_cBcB = (c(x,y-1) - c(x,y+1)) ^ 2;
// Same field difference of the previous frame
diff_pTpT = (p(x,y) - p(x,y+2)) ^ 2;
diff_pBpB = (p(x,y-1) - p(x,y+1)) ^ 2;
// Same field vertical smoothness of the current frame
diff_cT = ABS(c(x,y) - c(x,y-2)) + ABS(c(x,y) - c(x,y+2)) - ABS(c(x,y-2) + c(x,y+2));
diff_cB = ABS( c(x,y+1) - c(x,y-1) ) + ABS( c(x,y+1) - c(x,y+3) ) -
ABS( c(x,y-1) + c(x,y+3) );
if( diff_cTpT + diff_cBpB > fmd_tdiff ) {// if moving pixels,

   // Fine tears for cadence detection except 2-2 detection
   if( diff_cTcB > diff_cTcT + diff_cBcB)variance[5]++;
   elsevariance[6]++;
   // Find tears for 2-2 cadence detection
   if( diff_cT < fmd_vdiff1 && diff_cB < fmd_vdiff1) {// if fields are vertically
smooth,
variance[7]++;// total moving pixels
// Find tears. (1st condition is to exclude very small variations)
if(diff_cTcB >=fmd_vdiff2 && diff_cTcB > diff_cTcT + diff_cBcB) TEAR1(x,y) = 1
if(diff_cTpB >=fmd_vdiff2 && diff_cTpB > diff_cTcT + diff_pBpB) TEAR_2(x,y) = 1
if(diff_cBpT>=fmd_vdiff2 && diff_cBpT > diff_pTpT + diff_cBcB) TEAR_3(x,y) = 1
   }
}
```

2.8.5.3 Tear Variances

The all 3 TEAR_N variables are compared to neighbors to eliminate strays:

if(TEAR_N(x-1,y) == 0 &&

TEAR_N(x+1,y) == 0 &&

TEAR_N(x,y-2) == 0 &&

TEAR_N(x,y+2) == 0)TEAR_N(x,y) = 0;where N=1,2,3.

variance[8] = sum of TEAR1(x,y)

variance[9] = sum of TEAR_2(x,y)

variance[10] = sum of TEAR_3(x,y)

if (variance[8] > variance[9] && variance[8] > variance[10])

variance[7] = variance[8] = variance[9] = variance[10] = 0

if (variance[8] < fmd_thr_tear) variance[8] = 0

if (variance[9] < fmd_thr_tear) variance[9] = 0

if (variance[10] < fmd_thr_tear) variance[10] = 0

The variances are summed for each block across the frame. The accumulators may require 24-bit adders if the differences are 8-bits and there can be 128 (horizontally) * 256 (vertically) of them. The sums are written to memory at the end of the frame.

Two sets of FMD variances are needed to support 2 simultaneous streams. The streams are distinguished by the dndi_stream_id state variable in the DI state.

A-Stepping Erratum: TEAR_N compute doesn't follow the equation above. Two signals were missing, thus, it is incorrectly calculated as the following. Without the added protection of the N=-2 & N=4 collection of feature, the robustness of 2:2 detection suffers.

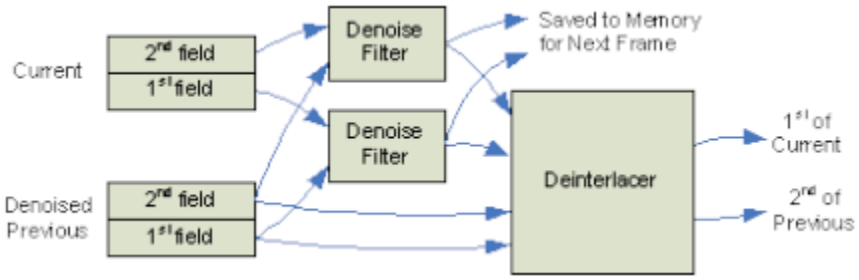
if(TEAR_N(x-1,y) == 0 &&

TEAR_N(x+1,y) == 0 &&)TEAR_N(x,y) = 0;where N=1,2,3.

2.8.6 Implementation Overview

2.8.6.1 Input and Output Frames

Two frames are needed to do deinterlacing, but for any two frames, two fields can be deinterlaced, doubling the output for the same input bandwidth. This also allows the denoise filter to only filter a frame once.



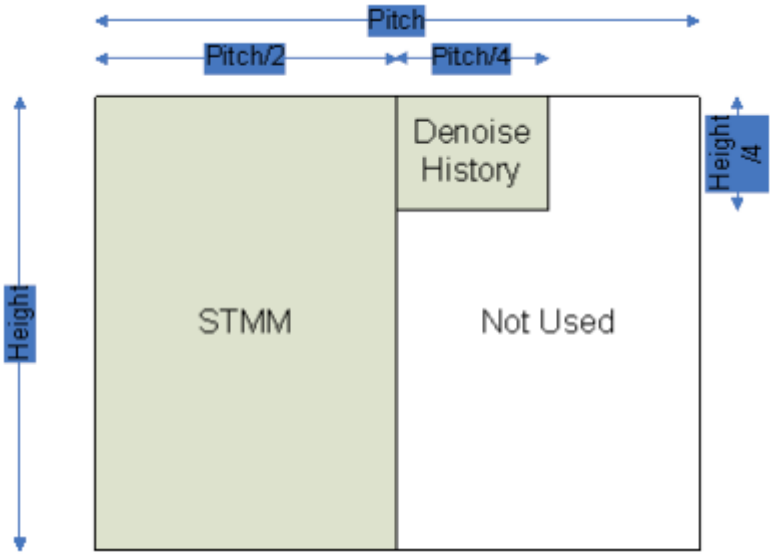
The above picture shows that two frames are read in, called current and previous. The two fields of the next frame are denoised using adjacent fields. The 2nd field of previous can be deinterlaced using current as the reference, and the 1st field of current can be deinterlaced using previous as reference.

Since we are producing 2 16x4 outputs, and the performance goal is to output 2 pixels per clock, we have 64 clocks to run 2 denoise filters and 2 deinterlacers.

The fields are referred to as 1st and 2nd because either the top or bottom field can be the first in the sequence depending on a state variable.

2.8.6.2 Statistics Surface Memory Format

The statistics memory page is used to store both STMM and Denoise history. The STMM and Denoise history are stored in separate areas addressed by a single base address pointer:



The STMM for any pixel pair is addressed by:

STMM_X = pixelX / 2

STMM_Y = pixelY

The Denoise History for any 4x4 block is addressed by

DH_X = Pitch/2 + pixelX/4

DH_Y = pixelY/4

Where the pixelX/Y comes from the address of the left pixel for STMM and the upper-left pixel for the Denoise History. The Pitch is from the surface state.

The read and write surfaces for each frame must be separate, since any individual block will not know if the neighbor blocks have been updated yet. This can be implemented as a ping-pong buffer pair with the write surface for each frame becoming the read surface for the next.

2.8.6.3 First Frame Special Case

The first frame in the sequence is a special case for both denoise and deinterlace. Only data from the current frame address is read, the previous frame, clean previous, statistics and control addresses are ignored. Behavior for each function is as follows:

1. Denoise – The denoise filter needs to use the spatial filter, since there is no previous frame from which to do a temporal filter.
    a. The Denoise Motion History is not read.
    b. The blend between the temporal and spatial is forced to 100% spatial.
    c. The Denoise Motion History output values are written to 0.
1. BNE – The Block Noise Estimate only uses current frame values and so works normally.
2. Deinterlacer – Only the 1st field of the current frame frame is deinterlaced in this case – the 2nd of previous does not exist.
    a. The spatial deinterlacer is used to produce the output.
    b. The STMM input values are not read.

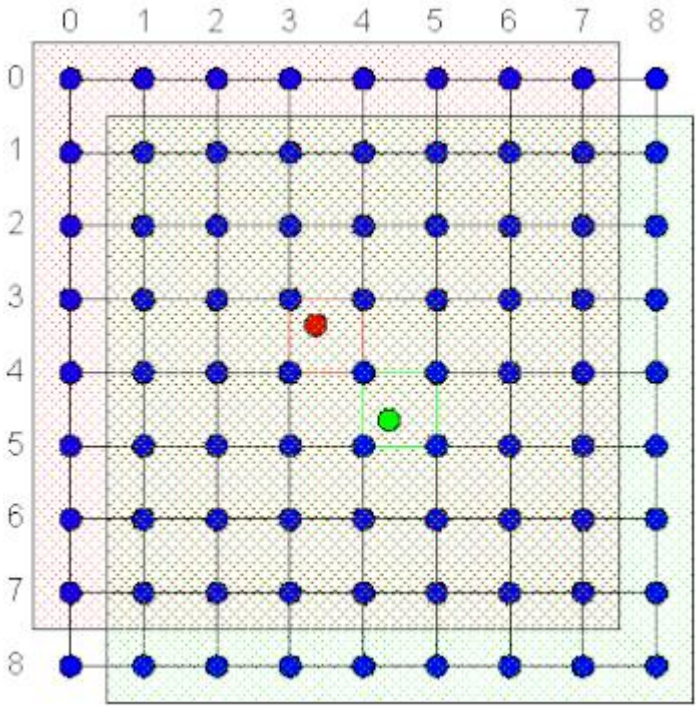
c. The STMM output values are written as a the maximum 255 value so that the next frame is correctly told that spatial deinterlacing was used in this frame.

3. FMD – variances between the top and bottom of the current field should be output correctly. Variances that read from the previous field should indicate a maximum difference.
4. Progressive Cadence Reconstruction – the FMD input is not read, so always assume interlaced.

2.9 Adaptive Video Scaler

The adaptive video scaler consists of a pair of filters. The sharp filter is an 8x8 and the smooth filter is bilinear. The results of the two filters are alpha blended together using an alpha factor determined separately from an algorithm that examines the pixel values in the each vector.

There are a total of four different coefficient tables with two in each direction. For both directions is it possible to use either of the two tables that are assigned to it or use both at once with one table for the Y and the other table for the U/V. The coefficients are programmable by software and loaded via a new command streamer instruction. The coefficients are considered to be nonpipelined state, with a full pipeline flush being required before a new set of coefficients is loaded.



The above diagram shows two pixels (red and green) mapped onto a texture map, with the texel centers blue. The red/green boxes around the pixels indicate the area where the pixel would choose the same 8x8 footprint for its filter, while the large transparent box indicates the footprint for each pixel.

The u/v addresses for each pixel (in texel space) are as follows:

red pixel: u=3.3, v=3.3 (betau=0.3, betav=0.3)

green pixel: u=4.3, v=4.7 (betau=0.3, betav=0.7)

The integer u/v address of the upper left pixel of the footprint is a function of the pixel u/v address as follows:

u(UL) = floor(u(pix)) – 3

v(UL) = floor(v(pix)) – 3

When the 8x8 filter is selected, the 8x8 texel block surrounding the pixel sample point is selected. The blend factors "beta" (horizontal and vertical) are determined by the relative distance between the pixel center and the nearest 4 texels (2x2). The betas are first truncated to 5 bits (*i*).

The beta value is used to look up two sets of 8 coefficients, one set of 8 for horizontal (called $K_h0..7$), and one set of 8 for vertical (called $K_v0..7$).

2.9.1 Filtering Operations

There are two separate filters, sharp and smooth, which are blended in an adaptive manner.

2.9.1.1 Sharp

The following formula is used to compute the filtered texture color for the sharp filter:

$R0 = T00*K_h0 + T01*K_h1 + T02*K_h2 + T03*K_h3 + T04*K_h4 + T05*K_h5 + T06*K_h6 + T07*K_h7$

$R1 = T10*K_h0 + T11*K_h1 + T12*K_h2 + T13*K_h3 + T14*K_h4 + T15*K_h5 + T16*K_h6 + T17*K_h7$

$R2 = T20*K_h0 + T21*K_h1 + T22*K_h2 + T23*K_h3 + T24*K_h4 + T25*K_h5 + T26*K_h6 + T27*K_h7$

$R3 = T30*K_h0 + T31*K_h1 + T32*K_h2 + T33*K_h3 + T34*K_h4 + T35*K_h5 + T36*K_h6 + T37*K_h7$

$R4 = T40*K_h0 + T41*K_h1 + T42*K_h2 + T43*K_h3 + T44*K_h4 + T45*K_h5 + T46*K_h6 + T47*K_h7$

$R5 = T50*K_h0 + T51*K_h1 + T52*K_h2 + T53*K_h3 + T54*K_h4 + T55*K_h5 + T56*K_h6 + T57*K_h7$

$R6 = T60*K_h0 + T61*K_h1 + T62*K_h2 + T63*K_h3 + T64*K_h4 + T65*K_h5 + T66*K_h6 + T67*K_h7$

$R7 = T70*K_h0 + T71*K_h1 + T72*K_h2 + T73*K_h3 + T74*K_h4 + T75*K_h5 + T76*K_h6 + T77*K_h7$

$F' = R0*K_v0 + R1*K_v1 + R2*K_v2 + R3*K_v3 + R4*K_v4 + R5*K_v5 + R6*K_v6 + R7*K_v7$

F_sharp = Clamp F' to [0.0, 1.0)

where:

- T*rc* is the texel color in row *r* ([0..3]) and column *c* ([0..3]) of the 8x8 array of neighboring texel colors
- F_sharp is the final output color of the sharp filter.

2.9.1.2 Smooth

The following formula is used to compute the filtered texture color for the smooth filter:

F_smooth = (T33 * (1-betaU) + T34 * betaU) * (1-betav) + (T43 * (1-betaU) + T44 * betaU) * betav

2.9.1.3 Adaptive Filtering

The adaptive filter only supports RGB or YUV packed formats. For YUV formats, the alpha value is determined only by the Y channel (green), with this alpha value being applied to all three channels. For the RGB formats the alpha value is determined based on an average of all three channels with G having double the weight as the other channels.

Each horizontal or vertical filter has 8 texels input which feeds into an eight tap filter. On the center two there is a linear blend using the betaV. Then using the Y channel an adaptive part weight is calculated

and the two filters are alpha blended. The adaptive part calculated on the Y channel is used on all three channels. Only the 8 MSBs are used in these calculations.

The adaptive part is done to classify a pixel as prone to ringing or not. This is done by analyzing the 8 Y samples from the interpolation window ($Wy_0 \ldots Wy_7$).

2.10 Image Enhancement Filter and Video Signal Analysis

The IEF module takes in the YUV 444 color space with 10 bit components.

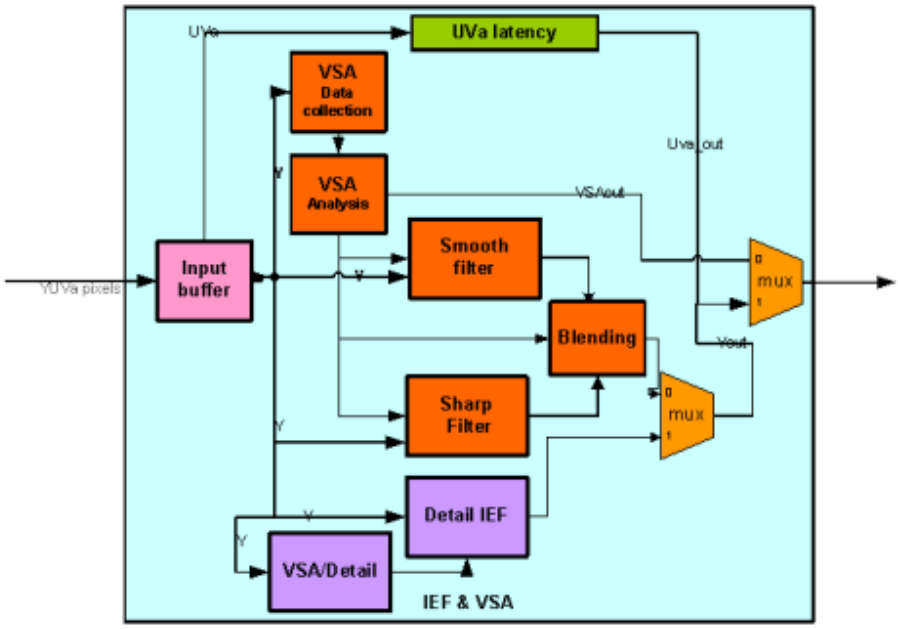
The IEF can be configured to operate either detail filtering or smooth filtering. A 3x3 and a 5x5 programmable filter are involved to achieve detail-enhanced or smooth effect.

VSA – Video Signal Analysis – analyzes the local Y environment of each pixel and outputs several values that describe its nature (smooth, detailed, sharpening). Those values will be used by the IEF to decide how the filter should be applied at each pixel location.

IEF – Image Enhancement Filter – The operations this filter performs are detail filter, smoothing and sharpening on the Y component, according to the VSA outputs.

The IEF throughput is 2 pixels per clock.

Block Diagram



2.10.1 Detail Filter Algorithm

2.10.1.1 VSA for Detail Filter

VSA in IEF aims to analyze the local property of the content and it is achieved with the usage of Sobel edge detection. For example, below detection kernel can be used to analyze 3x3 neighborhoods.

$$E\_h = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} \quad E\_v = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

Statistics from Sobel filtering outputs are taken for setting different weights to the detail added.

*EM(x) = |NH9(x) * E_h| + |NH9(x) * E_v| // where the input is 10 bits, EM is 6 bits (*CLIP((|NH9(x) * E_h| + |NH9(x) * E_v|+4) >> 3 , 0 ,63))

If (EM(x) > **Strong_Edge_Threshold**) local_adjust = **Strong_Edge_Weight** // local_adjust is 3bits

Else if (EM(x) > **Weak_Edge_Threshold**) local_adjust = **Regular_Weight**

Else local_adjust = **Non_Edge_Weight**

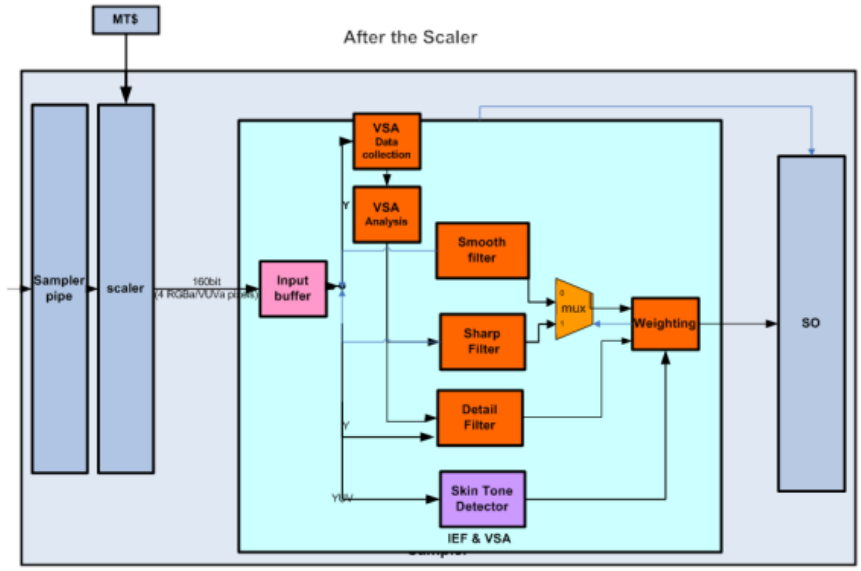
The **Strong_Edge_Threshold, Weak_Edge_Threshold, Strong_Edge_Weight, Non_Edge_Weight** and **Regular_Weight** are the pipelined state variables to be specified by driver.
Strong_Edge_Threshold & **Weak_Edge_Threshold** are 6-bit length variables.

2.10.1.2 Detail IEF

A programmable 3x3 and a 5x5 filter are provided to extract the detail information of the image. The detail extracted is the combination of the convolution results from the 3x3 and the 5x5 filter. The amount of detail added upon each pixel is determined by the weighting obtained from the VSA.

2.10.2 Skin-Tone Tuned IEF

The operation of IEF can be enhanced with the skin tone detection (STD). The STD unit detects the skin-tone like colors and passes a grade of skin tone color to the IEF. The skin tone operation on the YUV space, and the detected skin tone score will be recorded as a 5-bit number as per-pixel basis. The IEF modifies the pixel value by adjusting the delta signal according to the detected skin tone score.



2.10.2.1 Skin Tone Detection

2.10.2.1.1 STD - Detection in the (U,V) sub-space

The STD operates on digital images in the YUV color space. In these spaces, the skin-tone region is represented by the ellipse in the (U,V) subspace (chroma components), by a trapeze membership function in the Y direction (luma component) and by a piece-wise linear classifier in the (V,Y) subspace.

The detection in each sub-space outputs a likelihood score (i.e., det_UV, det_Y, and det_VY) representing how likely a pixel being a skin-tone pixel in that sub-space. Each score is represented with 5 bits, and the final skin-tone detection score SkinToneFactor is taken as the minimum of (det_UV, det_Y, det_VY).

2.10.3 Video Analytics Functions – Functional Description

2.10.3.1 Convolve

The CONVOLVE instruction performs a convolution on the source matrix, using the specified kernel, and stores the result in the destination matrix. The source input can either be 8bit UINT or 16bit UINT/SINT.

The floating point coefficient (i.e., each element within H' mentioned in Sec. 1.3.1) could be scaled up by before being loaded as the fixed point kernel for the convolve. The final result would need to be scaled down by the same amount before clamping the result to the fixed point 16 bit format. This scaling is helpful in getting more precision and hence near to accurate result of the floating point math. The scaling supported is only in powers of 2, and this could be found as follows:

Assuming all coefficients will be less than the value 8. In case it is greater they would need to be scaled down such that the max value is lesser than 8 before doing the following calculation. The scale up of the resultant convolve value then would need to be done in the EUs appropriately. In case all coefficients are

less than 1/2^7, then coefficients can be upscaled by driver and later the result from the convolve operation will be down scaled by EU.

If((MAX(ABS(C[i])) <= 8) && (MAX(ABS(C[i])) >= 1/2^7)

Scale = Max_power_of_2(8/MAX(ABS(C[i])))

C[i] = 2^(scale) * C[i]

Where the following are the functions:

MAX – Is the max of all the coefficients

Max_power_of_2 – Finds the Max power of 2 lower or equal to the input value. The Max_power_of_2 output cannot be greater than 512.

CONVOLVE(*src, kernel, scale, dest*)

Src - 8bit UINT/ 16bit SINT/ 16bit UINT

Kernel - 16bit S3.12

Dest – 16bit SINT

Scale – Range 0 to 10. The final result is shifted by this amount.

Pseudo Code:

```
// Example for kernel_widthxkernel_height below:
CONVOLVE src, kernel, dest
{
  ((
        real tmp = 0;
        for (m = 0; m < kernel_width; m++)
        {
              for (n = 0; n < kernel_height; n++)
              {
                    int ii = i + m – ((kernel_width-1)>>1);
                    int jj = j + n – ((kernel_height-1)>>1);

                    if (ii < 0) {
                          if(clamp) ii = 0;
                          else ii = -ii - 1; // mirror
                    }
                    else if (ii > src.width - 1) {
                          if(clamp) ii = src.width – 1;
```

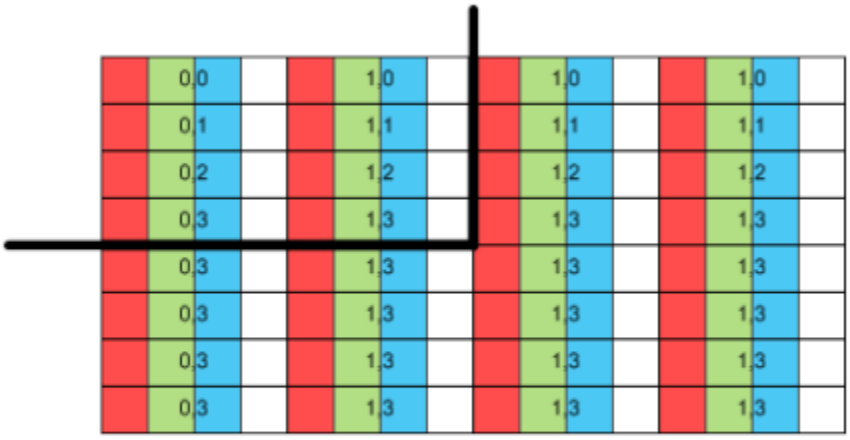
```
                    else ii = 2*src.width - ii - 1; //mirror
                }
                if (jj < 0) {
                    if(clamp) jj = 0;
                    else jj = -jj - 1; // mirror
                }
                else if (jj > src.height - 1) {
                    if(clamp) jj = src.height – 1;
                    else jj = 2*src.height - jj - 1; //mirror
                }
                // handle matrix sizes smaller than the kernel, use null value in place of src[ii,jj]
                if (ii < 0 || ii > src.width - 1 || jj < 0 || jj > src.height - 1)
                    val = 0;
                else
                    val = src[jj,ii];

                //Note: that the driver does the kernel flipping already and hardware does the
                following operation:
                tmp += val * kernel[n,m];
            }
        }
        tmp = tmp <<1;
        tmp = INT(tmp) >> scale
        tmp = (tmp + 1)>>1;
        dest[i,j] = Clamp(tmp, -2^16, 2^16-1);
  ))
}
```

Even though the convolve is suppose to do kernel flipping, the kernel is actually flipped by driver, and the hardware does not do any kernel flipping and operates using the kernel directly. Also the kernel is always starting from (0,0) and depending on the kernel width and height the appropriate coefficients are picked from the kernel coefficient stored.

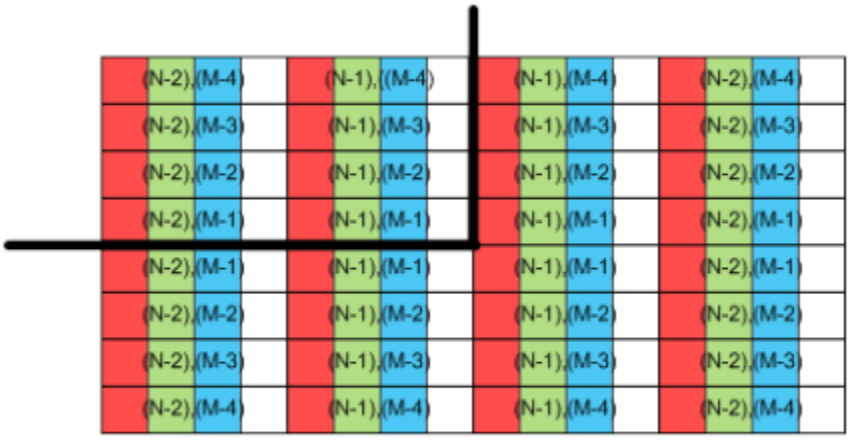
2.11 Mirror pixel at boundary edges for Media (sample_8x8 messages)

Presently we support only Clamp mode for sample_8x8 messages. This is to extend mirror mode as supported for 3D messages for sample_8x8 messages. The restriction here would be that the surface width is in multiples of DWords in native L1 storage format. The following are the surfaces which would need to be covered here:

1. 32bpp format in Memory and L1 (for AVS only and sample_unorm):
    a. 8: R10G10B10A2_UNORM
    b. 9: R8G8B8A8_UNORM
    c. 13: A8Y8U8V8_UNORM
    d. 14: B8G8R8A8_UNORM

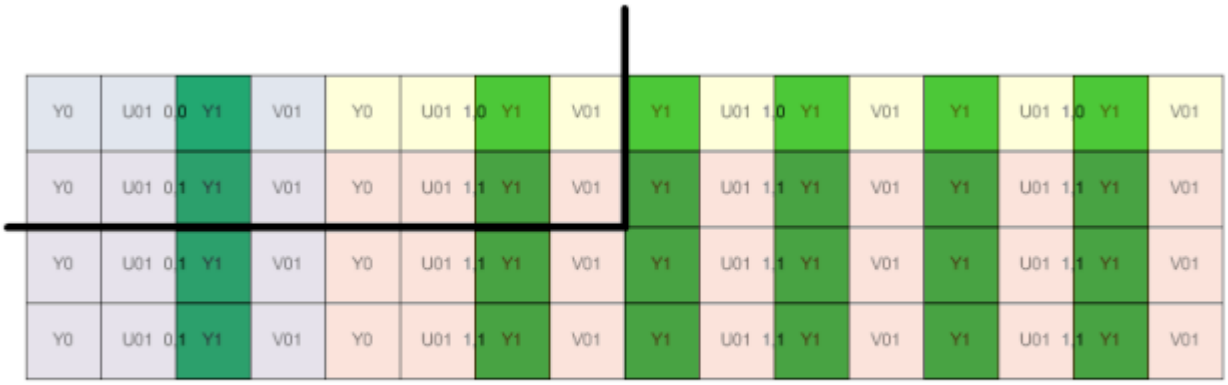


Clamp Shown for right and bottom boundary (32bpp)

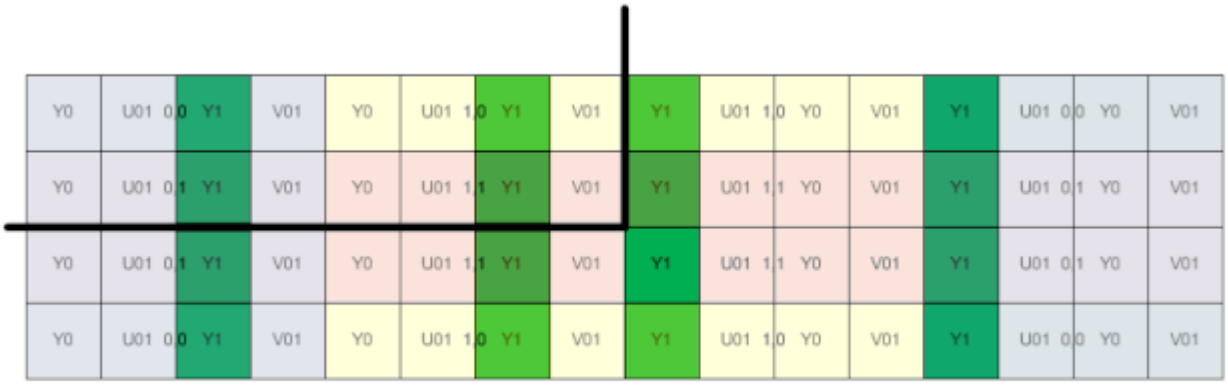


Mirror Shown for right and bottom boundary (32bpp)

2. 16bpp format in Memory and 16bpp in L1 (for AVS only and sample_unorm):
    a. 0: YCRCB_NORMAL
    b. 1: YCRCB_SWAPUVY
    c. 2: YCRCB_SWAPUV
    d. 3: YCRCB_SWAPY
    e. 10: R8B8_UNORM (CrCb)



Clamp Shown for right and bottom boundary (16bpp)



Mirror Shown for right and bottom boundary (16bpp)

3. 8bpp format in Memory and 16bpp in L1 (for AVS only):

The pixel when read out from L1 would be replicated as what is shown above in the diagram for 16bpp.

   a. 4: PLANAR_420_8
   b. 11: R8_UNORM (Cr/Cb)
   c. 12:Y8_UNORM

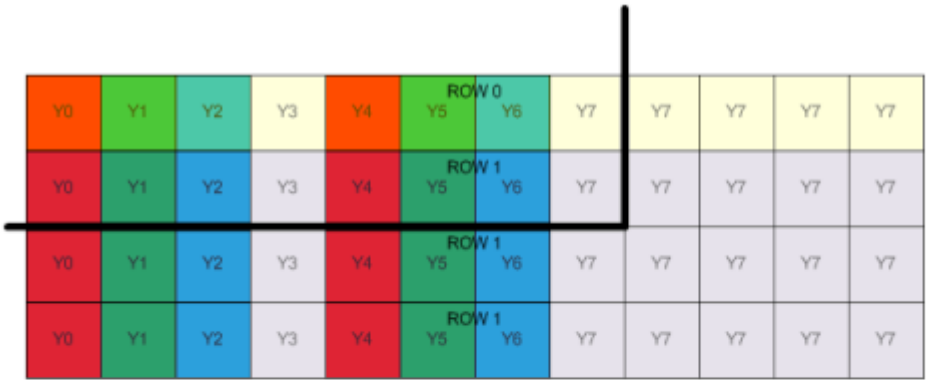
4. 64bpp format in Memory and L1 (for AVS only):

15: R16G16B16A16

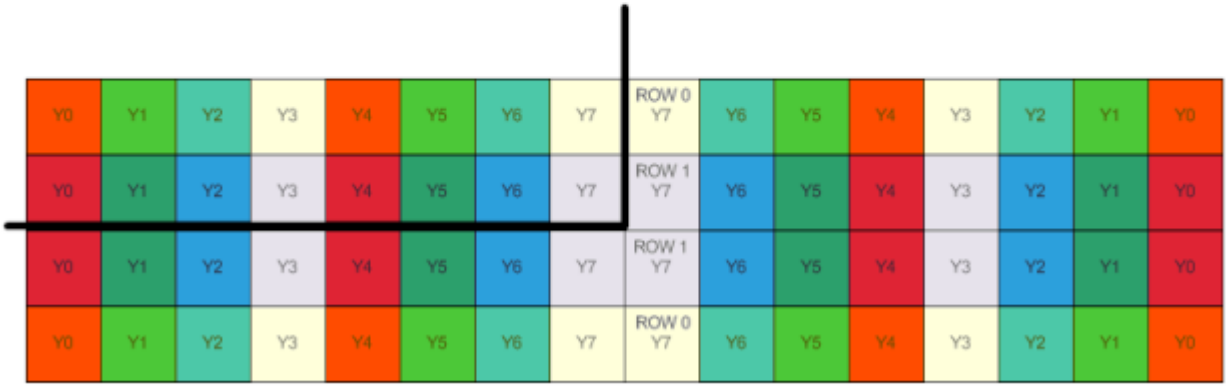
This would be similar to the 32bpp except each pixel is 64bpp instead.

5. 8bpp format in Memory and L1 (for VA only):
   a. 5: PLANAR_Y8_UNORM

Each cell in the below figure is 8bits.



Clamp Shown for right and bottom boundary (8bpp VA)



Mirror Shown for right and bottom boundary (8bpp VA)

6. 16bpp format in Memory and L1 (for VA only):

Same as the above except each cell is 16bpp.

   a. 7:PLANAR_Y16_UNORM
   b. 6 : PLANAR_Y16_SNORM

7. 1bpp format (Boolean) in Memory and 32bpp in L1(for VA only) :

Will not support Mirror on this surface. Clamp will still happen on vertical direction. Clamp on horizontal direction will not be taken care of in sampler for 1bpp format, but will be done in AVS unit.

   a. 16: PLANAR_Y32_UNORM

2.11.1 Restriction when Mirror mode is enabled for Sample_8x8 messages

When Function=AVS, ChromaKey is not supported with Mirror mode. In case ChromaKey needs to be enabled, then the Address control needs to be Clamp mode only.

2.11.1.1 For AVS

For AVS scaling, the following are the restrictions on the input image size:

Image Width > MAX((19*deltaU_nn + 139*ddu_nn + 7), 32)

Image Height > MAX((19*deltaV_nn + 139*ddv_nn + 7), 32)

The non-normalized input co-ordinate should be in the following range:

-width < (U_nn+2*deltaU_nn+3*ddu_nn) < (2*width – U – 17*deltaU_nn – 136*ddu_nn – 7)

-height < (V_nn+2*deltaV_nn+3*ddv_nn) < (2*height – 17*deltaV_nn – 136*ddv_nn – 7)

Where

U_nn = U_normaized * width

V_nn = V_normaized * height

deltaU_nn = deltaU_normaized * width

deltaV_nn = deltaV_normaized * height

ddU_nn = ddU_normaized * width

ddV_nn = ddV_normaized * height

2.11.1.2 For VA

For VA message (other than AVS scaling mode) the restriction is that the minimum input image size should be 32x32. The normalized input co-ordinate should not be in the range of -1 to 2 not inclusive.

2.12 State

2.12.1 BINDING_TABLE_STATE

BINDING_TABLE_STATE		
Default Value:	0x00000000	
The binding table binds surfaces to logical resource indices used by shaders and other compute engine kernels. It is stored as an array of up to 256 elements, each of which contains one dword as defined here. The start of each element is spaced one dword apart. The first element of the binding table is aligned to a 32-byte boundary.		
DWord	**Bit**	**Description**
0	31:5	**Surface State Pointer** Format: SurfaceStateOffset[31:5] This 32-byte aligned address points to a surface state block. This pointer is relative to the **Surface State Base Address**.
	4:0	**Reserved** Format: MBZ

2.12.2 SURFACE_STATE

The surface state is stored as individual elements, each with its own pointer in the binding table. Each surface state element is aligned to a 32-byte boundary.

Surface state defines the state needed for the following objects:

- texture maps (1D, 2D, 3D, cube) read by the sampling engine
- buffers read by the sampling engine
- constant buffers read by the data cache via the data port
- render targets read/written by the render cache via the data port
- streamed vertex buffer output written by the render cache via the data port
- media surfaces read from the texture cache or render cache via the data port
- media surfaces written to the render cache via the data port

2.12.2.1 SURFACE_STATE for most messages

RENDER_SURFACE_STATE

Exists If: (MessageType != 'Deinterlace') && (MessageType != 'Sample_8x8')

Default Value: 0x00000000, 0x00000000, 0x00000000, 0x00000000, 0x00000000, 0x00000000, 0x00000000, 0x00000000

This is the normal surface state used by all messages that use SURFACE_STATE except deinterlace and sample_8x8.

DWord	Bit	Description
0	31:29	**Surface Type** Project: All Format: U3 enumerated type This field defines the type of the surface.

Value	Name	Description	Project
0h	SURFTYPE_1D	Defines a 1-dimensional map or array of maps	All
1h	SURFTYPE_2D	Defines a 2-dimensional map or array of maps.	All
2h	SURFTYPE_3D	Defines a 3-dimensional (volumetric) map.	All
3h	SURFTYPE_CUBE	Defines a cube map or array of cube maps.	All
4h	SURFTYPE_BUFFER	Defines an element in a buffer.	All
5h	SURFTYPE_STRBUF	Defines a structured buffer surface.	All
6h	Reserved		All
7h	SURFTYPE_NULL	Defines a null surface.	All

Programming Notes

A null surface is used in instances where an actual surface is not bound. When a write message is generated to a null surface, no actual surface is written to. When a read message (including any sampling engine message) is generated to a null surface, the result is all zeros. Note that a null surface type is allowed to be used with all messages, even if it is not specificially indicated as supported. All of the remaining fields in surface state are ignored for null surfaces, with the following exceptions: Width, Height, Depth, LOD, and Render Target View Extent fields must match the depth buffer's corresponding state for all render target surfaces, including null.All sampling engine and data port messages support null surfaces with the above behavior, even if not mentioned as specifically supported, except for the following: Data Port Media Block Read/Write messages. The Surface Type of a surface used as a render target (accessed via the Data Port's Render Target Write message) must be the same as the Surface Type of all other render targets and of the depth buffer (defined in 3DSTATE_DEPTH_BUFFER), unless either the depth buffer or render targets are

	SURFTYPE_NULL.
28	**Surface Array**

Project:	All
Format:	Enable

This field, if enabled, indicates that the surface is an array.

If this field is enabled, the Surface Type must be SURFTYPE_1D, SURFTYPE_2D, or SURFTYPE_CUBE. If this field is disabled and Surface Type is SURFTYPE_1D, SURFTYPE_2D, or SURFTYPE_CUBE, the Depth field must be set to zero.

27	**Reserved**

Project:	All
Format:	MBZ

26:18	**Surface Format**

Project:	All
Format:	SURFACE_FORMAT

Specifies the format of the surface or element within this surface. Refer to the table in section 1.12.4.1.2 for the formats supported and their encodings.

Programming Notes

YUV (YCRCB) surfaces used as render targets can only be rendered to using 3DPRIM_RECTLIST with even X coordinates on all of its vertices, and the pixel shader cannot kill pixels. If Number of Multisamples is set to a value other than MULTISAMPLECOUNT_1, this field cannot be set to the following formats: any format with greater than 64 bits per element, if Number of Multisamples is MULTISAMPLECOUNT_8, any compressed texture format (BC*), and any YCRCB* format.

This field cannot be a YUV (YCRCB*) format if the **Surface Type** is SURFTYPE_BUFFER or SURFTYPE_STRBUF.

17:16	**Surface Vertical Alignment**

Format:	U2 enumerated type

Description	Project
For Sampling Engine Uncompressed and Render Target Surfaces: This field specifies the vertical alignment requirement for the surface. Refer to the “Memory Data Formats” chapter for details on how this field changes the layout of the surface in memory. This field applies to surface formats other than compressed formats. For other surfaces this field is ignored.	
A value of 1 is not supported for formats YCRCB_NORMAL (0x182), YCRCB_SWAPUVY (0x183), YCRCB_SWAPUV (0x18f), or YCRCB_SWAPY (0x190).	

Value	Name	Description
2h-3h	Reserved	Reserved

Programming Notes	Project
This field is intended to be set to VALIGN_4 if the surface was rendered as a depth buffer, for a	

multisampled (4x) render target, or for a multisampled (8x) render target, since these surfaces support only alignment of 4. Use of VALIGN_4 for other surfaces is supported, but uses more memory. This field must be set to VALIGN_4 for all tiled Y Render Target surfaces.	
Value of 1 is not supported for format YCRCB_NORMAL (0x182), YCRCB_SWAPUVY (0x183), YCRCB_SWAPUV (0x18f), YCRCB_SWAPY (0x190)	
If Number of Multisamples is not MULTISAMPLECOUNT_1, this field must be set to VALIGN_4.	

Errata	Description	Project
	VALIGN_4 is not supported for surface format R32G32B32_FLOAT.	

15 **Surface Horizontal Alignment**

Project:	All
Format:	U2 enumerated type

U2 enumerated type

For Sampling Engine Uncompressed and Render Target Surfaces: This field specifies the horizontal alignment requirement for the surface. Refer to the “Memory Data Formats” chapter for details on how this field changes the layout of the surface in memory. This field applies to surface formats other than compressed formats. For other surfaces, this field is ignored.

Value	Name	Description	Project
0h	HALIGN_4	Horizontal alignment factor j = 4	All
1h	HALIGN_8	Horizontal alignment factor j = 8	All

Programming Notes

This field is intended to be set to HALIGN_8 only if the surface was rendered as a depth buffer with Z16 format or a stencil buffer, since these surfaces support only alignment of 8. Use of HALIGN_8 for other surfaces is supported, but uses more memory.

This field must be set to HALIGN_4 if the Surface Format is BC*.

This field must be set to HALIGN_8 if the Surface Format is FXT1.

14 **Tiled Surface**

Project:	All
Format:	U1 enumerated type

This field specifies whether the surface is tiled.

Value	Name	Description	Project
0h	FALSE	Linear surface	All
1h	TRUE	Tiled surface	All

Programming Notes

Linear surfaces can be mapped to Main Memory (uncached) or System Memory (cacheable, snooped). Tiled surfaces can only be mapped to Main Memory. The corresponding cache(s)

must be invalidated before a previously accessed surface is accessed again with an altered state of this bit. If Surface Type is SURFTYPE_BUFFER, this field must be FALSE (because buffers are supported only in linear memory). If Surface Type is SURFTYPE_NULL, this field must be TRUE.

13 **Tile Walk**

Project:	All
Format:	U2 enumerated type

This field specifies the type of memory tiling (XMajor or YMajor) used to tile this surface. See Memory Interface Functions for details on memory tiling and restrictions.

Value	Name	Description	Project
0b	TILEWALK_XMAJOR	X major tiling.	All
1b	TILEWALK_YMAJOR	Y major tiling.	All

Programming Notes

Refer to Memory Data Formats for restrictions on TileWalk direction for the various buffer types. (Of particular interest is the fact that YMAJOR tiling is not supported for display/overlay buffers). The corresponding caches must be invalidated before a previously accessed surface is accessed again with an altered state of this bit.This field is ignored when the surface is linear.

Errata	Description	Project
	Set Tile Walk to TILEWALK_XMAJOR if Tiled Surface is False.	

12 **Vertical Line Stride**

Project:	All
Format:	U1 in lines to skip between logically adjacent lines

For 2D non-array surfaces accessed via the Sampling Engine or Data Port: Specifies the number of lines (0 or 1) to skip between logically adjacent lines and supports interleaved (field) surfaces as textures.

For other surfaces, Vertical Line Stride must be zero.

Programming Notes

This bit must not be set if the surface format is a compressed type (BCn*).

If this bit is set on a sampling engine surface, the mip mode filter must be set to MIPFILTER_NONE.

11 **Vertical Line Stride Offset**

Project:	All
Format:	U1 in lines of initial offset (when Vertical Line Stride == 1)

For 2D non-array Surfaces accessed via the Sampling Engine or Data Port: Specifies the offset of the initial line from the beginning of the buffer. Ignored when Vertical Line Stride is 0.

For other surfaces, Vertical Line Stride Offset must be zero.

10 **Surface Array Spacing**

Project:	All
Format:	U1 enumerated type

For 1D Array, 2D Array, Cube, and 2D Multisampled Surfaces: This field specifies whether space is reserved between array slices for additional LODs beyond LOD 0. Refer to the "Memory Data Formats" chapter for details on how this field changes the QPitch equation used to determine spacing between array slices in memory. For other surfaces, this field is ignored.

Value	Name	Description	Project
0h	ARYSPC_FULL	Memory space between array slices is reserved for all possible LOD's.	All
1h	ARYSPC_LOD0	Memory space is optimized for surfaces which contain only LOD 0.	All

Programming Notes

If Multisampled Surface Storage Format is MSFMT_MSS and Number of Multisamples is *not* MULTISAMPLECOUNT_1, this field must be set to ARYSPC_LOD0.

9 **Reserved**

Project:	All
Format:	MBZ

8 **Render Cache Read Write Mode**

Project:	All
Format:	U1 enumerated type

For Surfaces accessed via the Data Port to Render Cache: This field specifies the way Render Cache treats a write request. If clear, Render Cache allocates a write-only cache line for a write miss. If set, Render Cache allocates a read-write cache line for a write miss. For Surfaces accessed via the Sampling Engine or Data Port to Texture Cache or Data Cache: This field is reserved and MBZ.

Value	Name	Description	Project
0h		Allocating write-only cache for a write miss	All
1h		Allocating read-write cache for a write miss	All

Programming Notes

This field is provided for performance optimization for Render Cache read/write accesses (from EU's point of view).

7:6 **Media Boundary Pixel Mode**

Project:	All
Format:	U2 enumerated type

For 2D Non-Array Surfaces accessed via the Data Port Media Block Read Message: This field enables control of which rows are returned on vertical out-of-bounds reads using the Data Port Media Block Read Message. In the description below, frame mode refers to Vertical Line Stride = 0, field mode is Vertical Line Stride = 1 in which only the even or odd rows are addressable.

The frame refers to the entire surface, while the field refers only to the even or odd rows within the surface. For other surfaces this field is reserved and MBZ.

Value	Name	Description	Project
0h	NORMAL_MODE	the row returned on an out-of-bound access is the closest row in the frame or field. Rows from the opposite field are never returned.	All
1h	Reserved		All
2h	PROGRESSIVE_FRAME	the row returned on an out-of-bound access is the closest row in the frame, even if in field mode.	All
3h	INTERLACED_FRAME	In field mode, the row returned on an out-of-bound access is the closest row in the field. In frame mode, even out-of-bound rows return the nearest even row while odd out-of-bound rows return the nearest odd row.	All

5:0 **Cube Face Enables**

Project:	All
Format:	U6 bit mask of enables

For SURFTYPE_CUBE Surfaces accessed via the Sampling Engine: Bits 5:0 of this field enable the individual faces of a cube map. Enabling a face indicates that the face is present in the cube map, while disabling it indicates that that face is represented by the texture map's border color. Refer to Memory Data Formats for the correlation between faces and the cube map memory layout. Note that storage for disabled faces must be provided. For other surfaces this field is reserved and MBZ.

Value	Name
1xxxxxb	-X face
x1xxxxb	+X face
xx1xxxb	-Y face
xxx1xxb	+Y face
xxxx1xb	-Z face
xxxxx1b	+Z face

Programming Notes

When TEXCOORDMODE_CLAMP is used when accessing a cube map, this field must be programmed to 111111b (all faces enabled). This field is ignored unless the Surface Type is SURFTYPE_CUBE.

1 31:0 **Surface Base Address**

Project:	All
Format:	GraphicsAddress[31:0]

Specifies the byte-aligned base address of the surface.

Programming Notes

For SURFTYPE_BUFFER render targets, this field specifies the base address of first element of the surface. The surface is interpreted as a simple array of that single element type. The address must be naturally-aligned to the element size (e.g., a buffer containing R32G32B32A32_FLOAT elements must be 16-byte aligned)

For SURFTYPE_BUFFER non-rendertarget surfaces, this field specifies the base address of the first element of the surface, computed in software by adding the surface base address to

the byte offset of the element in the buffer.

Mipmapped, cube and 3D sampling engine surfaces are stored in a "monolithic" (fixed) format, and only require a single address for the base texture.

The Base Address for linear render target surfaces and surfaces accessed with the typed surface read/write data port messages must be element-size aligned, for non-YUV surface formats, or a multiple of 2 element-sizes for YUV surface formats. Other linear surfaces have no alignment requirements (byte alignment is sufficient).

Linear depth buffer surface base addresses must be 64-byte aligned. Note that while render targets (color) can be SURFTYPE_BUFFER, depth buffers cannot.

Tiled surface base addresses must be 4KB-aligned. Note that only the offsets from Surface Base Address are tiled, Surface Base Address itself is not transformed using the tiling algorithm.

For tiled surfaces, the actual start of the surface can be offset from the Surface Base Address by the X Offset and Y Offset fields.

Certain message types used to access surfaces have more stringent alignment requirements. Please refer to the specific message documentation for additional restrictions.

2

31:30 **Reserved**

Project:	All
Format:	MBZ

29:16 **Height**

Project:	All
Format:	U14

This field specifies the height of the surface. If the surface is MIP-mapped, this field contains the height of the base MIP level. For buffers, this field specifies a portion of the buffer size.

Value	Name	Description
0		SURFTYPE_1D: must be zero
[0,16383]		SURFTYPE_2D: height of surface – 1 (y/v dimension)
[0,2047]		SURFTYPE_3D: height of surface – 1 (y/v dimension)
[0,16383]		SURFTYPE_CUBE: height of surface – 1 (y/v dimension)
[0,16383]		SURFTYPE_BUFFER/STRBUF: contains bits [20:7] of the number of entries in the buffer – 1

Programming Notes

For typed buffer and structured buffer surfaces, the number of entries in the buffer ranges from 1 to 2^{27}. For raw buffer surfaces, the number of entries in the buffer is the number of bytes which can range from 1 to 2^{30}. After subtracting one from the number of entries, software must place the fields of the resulting 27-bit value into the Height, Width, and Depth fields as indicated, right-justified in each field. Unused upper bits must be set to zero.If Vertical Line Stride is 1, this field indicates the height of the field, not the height of the frameThe Height of a render target must be the same as the Height of the other render targets and the depth buffer (defined in 3DSTATE_DEPTH_BUFFER), unless Surface Type is SURFTYPE_1D or SURFTYPE_2D with Depth = 0 (non-array) and LOD = 0 (non-mip mapped).

If this surface in memory is accessed with Vertical Line Stride set to both 0 and 1, this field must be an even value when Vertical Line Stride is 0.

If Media Pixel Boundary Mode is not set to NORMAL_MODE, this field must be an even value.

15:14 **Reserved**

Project:	All
Format:	MBZ

13:0 **Width**

Project:	All
Format:	U14-1

This field specifies the width of the surface. If the surface is MIP-mapped, this field specifies the width of the base MIP level. The width is specified in units of pixels or texels. For buffers, this field specifies a portion of the buffer size.

For surfaces accessed with the Media Block Read/Write message, this field is in units of DWords except when used for IECP and the output surface format is NV12 (R16_UNORM), this field is in units of Words.

Value	Name	Description
[0, 16383]		SURFTYPE_1D: width of surface – 1 (x/u dimension)
[0, 16383]		SURFTYPE_2D: width of surface – 1 (x/u dimension)
[0, 2047]		SURFTYPE_3D: width of surface – 1 (x/u dimension)
[0, 16383]		SURFTYPE_CUBE: width of surface – 1 (x/u dimension)
[0, 127]		SURFTYPE_BUFFER/STRBUF: contains bits [6:0] of the number of entries in the buffer – 1

Programming Notes

For surface types other than SURFTYPE_BUFFER or STRBUF The Width specified by this field must be less than or equal to the surface pitch (specified in bytes viathe Surface Pitch field).For cube maps, Width must be set equal to the Height.For MONO8 textures, Width must be a multiple of 32 texels.The Width of a render target must be the same as the Width of the other render target(s) and the depth buffer (defined in 3DSTATE_DEPTH_BUFFER), unless Surface Type is SURFTYPE_1D or SURFTYPE_2D with Depth = 0 (non-array) and LOD = 0 (non-mip mapped).The Width of a render target with YUV surface format must be a multiple of 2.For SURFTYPE_BUFFER: The low two bits of this field must be 11 if the Surface Format is RAW (the size of the buffer must be a multiple of 4 bytes).

3 31:21 **Depth**

Project:	All
Format:	U11

This field specifies the total number of levels for a volume texture or the number of array elements allowed to be accessed starting at the Minimum Array Element for arrayed surfaces. If the volume texture is MIP-mapped, this field specifies the depth of the base MIP level. For buffers, this field specifies a portion of the buffer size.

Value	Name	Description
[0,2047]		SURFTYPE_1D: number of array elements – 1
[0,2047]		SURFTYPE_2D: number of array elements – 1

[0,2047]		SURFTYPE_3D: depth of surface – 1 (z/r dimension)
[0,2047]		SURFTYPE_CUBE: number of array elements – 1 [see programming notes for range]
[0,1023]		SURFTYPE_BUFFER: contains bits [30:21] of the number of entries in the buffer – 1 for Surface Format RAW.
[0,127]		SURFTYPE_BUFFER: Contains bits [27:21] of the number of entries in the buffer – 1 for other surface formats.
[0,63]		SURFTYPE_STRBUF: contains bits [26:21] of the number of entries in the buffer – 1

Programming Notes

The Depth of a render target must be the same as the Depth of the other render target(s) and of the depth buffer (defined in 3DSTATE_DEPTH_BUFFER).For SURFTYPE_CUBE:For Sampling Engine Surfaces, the range of this field is [0,340], indicating the number of cube array elements (equal to the number of underlying 2D array elements divided by 6). For other surfaces, this field must be zero.For SURFTYPE_BUFFER: The range of this field is [0,63] unless the Surface Format is RAW and Surface Ptich is 1 byte.

Errata	Description	Project
	Errata: For SURFTYPE_CUBE sampling engine surfaces, the range of this field is limited to [0,85].	
	Errata: If Surface Array is enabled, and Depth is between 1024 and 2047, an incorrect array slice may be accessed if the requested array index in the message is greater than or equal to 4096.	

20:18 **Reserved**

Format:	MBZ

17:0 **Surface Pitch**

Project:	All
Format:	U18 pitch in (#Bytes – 1)

This field specifies the surface pitch in (#Bytes - 1).For surfaces of type SURFTYPE_BUFFER and SURFTYPE_STRBUF, this field indicates the size of the structure.

Value	Name	Description
[0,2047]		For surfaces of type SURFTYPE_BUFFER: representing [1B, 2048B]
[0,2047]		For surfaces of type SURFTYPE_STRBUF: representing [1B, 2048B]
[0,262143]		For other linear surfaces: representing [1B, 256KB]
[511,262143]		For X-tiled surface: representing [512B, 256KB] = [1 tile, 512 tiles]
[127,262143]		For Y-tiled surfaces: representing [128B, 256KB] = [1 tile, 2048 tiles]

Programming Notes

For linear render target surfaces and surfaces accessed with the typed data port messages, the pitch must be a multiple of the element size for non-YUV surface formats. Pitch must be a multiple of 2 * element size for YUV surface formats. For linear surfaces with Surface Type of SURFTYPE_STRBUF, the pitch must be a multiple of 4 bytes.For other linear surfaces, the pitch can be any multiple of bytes.For tiled surfaces, the pitch must be a multiple of the tile width.

RENDER_SURFACE_STATE

Errata	Description	Project
	Maximum pitch supported is 128KB for all surfaces	

4 | 31 | **Reserved**

Project:	All
Exists If:	[Surface Type] != SURFTYPE_STRBUF
Format:	MBZ

30:29 | **Render Target Rotation**

Project:	All
Exists If:	[Surface Type] != SURFTYPE_STRBUF
Format:	U2 enumerated type

For Render Target Surfaces: This field specifies the rotation of this render target surface when being written to memory.For Other Surfaces:This field is ignored.

Value	Name	Description	Project
0h	RTROTATE_0DEG	No rotation (0 degrees)	All
1h	RTROTATE_90DEG	Rotate by 90 degrees	All
2h	Reserved		All
3h	RTROTATE_270DEG	Rotate by 270 degrees	All

Programming Notes

Rotation is not supported for render targets of any type other than simple, non-mip-mapped, non-array 2D surfaces. The surface must be using tiled with X major.Width and Height fields apply to the dimensions of the surface before rotation.For 90 and 270 degree rotated surfaces, the Height (rather than the Width) must be less than or equal to the Surface Pitch (specified in bytes).For 90 and 270 degree rotated surfaces, the actual Height and Width of the surface in pixels (not the field value which is decremented) must both be even.Rotation is supported only for surfaces with the following surface formats: B5G6R5_UNORM, B5G6R5_UNORM_SRGB, R8G8B8[A|X]8_UNORM, R8G8B8[A|X]8_UNORM_SRGB, B8G8R8[A|X]8_UNORM, B8G8R8[A|X]8_UNORM_SRGB, B10G10R10[A|X]2_UNORM, B10G10R10A2_UNORM_SRGB, R10G10B10A2_UNORM, R10G10B10A2_UNORM_SRGB, R16G16B16A16_FLOAT, R16G16B16X16_FLOAT.Rotation is not supported for typed UAV messages

31:27 | **Reserved**

Project:	All
Exists If:	[Surface Type] == SURFTYPE_STRBUF
Format:	MBZ

28:18 | **Minimum Array Element**

Project:	All
Exists If:	[Surface Type] != SURFTYPE_STRBUF
Format:	U11

For Sampling Engine, Render Target, and Typed 1D and 2D Surfaces:This field indicates the minimum array element that can be accessed as part of this surface. This field is added to the delivered array index before it is used to address the surface.For Render Target 3D Surfaces:This field indicates the minimum ‘R’ coordinate on the LOD currently being rendered to. This field is added to the delivered array index before it is used to address the surface.For Sampling Engine Cube Surfaces:This field indicates the minimum array element in the

underlying 2D surface array that can be accessed as part of this surface (the cube array index is multipled by 6 to compute this value, although this field is not restricted to only multiples of 6). This field is added to the delivered array index before it is used to address the surface.
For Other Surfaces:This field must be set to zero.

Value	Name	Description
[0,2047]		1D/2D/cube surfaces
[0,2047]		3D surfaces

Errata	Description	Project
	If Number of Multisamples is not MULTISAMPLECOUNT_1, this field must be set to zero if this surface is used with sampling engine messages.	

17:7 **Render Target View Extent**

Project:	All
Exists If:	[Surface Type] != SURFTYPE_STRBUF
Format:	U11

For Render Target 3D Surfaces:This field indicates the extent of the accessible ‘R’ coordinates minus 1 on the LOD currently being rendered to.For Render Target 1D and 2D Surfaces:This field must be set to the same value as the Depth field.For Other Surfaces:This field is ignored.

Value	Name	Description
[0,2047]		to indicate extent of [1,2048]

6 **Multisampled Surface Storage Format**

Project:	All
Exists If:	[Surface Type] != SURFTYPE_STRBUF
Format:	U1 enumerated type

This field indicates the storage format of the multisampled surface.

Value	Name	Description	Project
0h	MSFMT_MSS	Multsampled surface was/is rendered as a render target	All
1h	MSFMT_DEPTH_STENCIL	Multisampled surface was rendered as a depth or stencil buffer	All

Programming Notes

All multisampled render target surfaces must have this field set to MSFMT_MSSIF this field is MSFMT_DEPTH_STENCIL, the only sampling engine messages allowed are “ld2dms”, “resinfo”, and “sampleinfo”.

This field is ignored if Number of Multisamples is MULTISAMPLECOUNT_1

Errata	Description	Project
	If the surface’s Number of Multisamples is MULTISAMPLECOUNT_8, Width is >= 8192 (meaning the actual surface width is >= 8193 pixels), this field must be set to MSFMT_MSS.	
	If the surface’s Number of Multisamples is MULTISAMPLECOUNT_8, ((Depth+1) * (Height+1)) is > 4,194,304, OR if the surface’s Number of Multisamples is MULTISAMPLECOUNT_4, ((Depth+1) * (Height+1)) is > 8,388,608, this field must be set to MSFMT_DEPTH_STENCIL.This field must be set to MSFMT_DEPTH_STENCIL if Surface Format is one of the following:	

		I24X8_UNORM, L24X8_UNORM, A24X8_UNORM, or R24_UNORM_X8_TYPELESS.	

5:3 Number of Multisamples

Project:	All
Exists If:	[Surface Type] != SURFTYPE_STRBUF
Format:	U3 enumerated type

This field indicates the number of multisamples on the surface.

Value	Name	Project
0h	MULTISAMPLECOUNT_1	All
1h	Reserved	All
2h	MULTISAMPLECOUNT_4	All
3h	MULTISAMPLECOUNT_8	All
4h-7h	Reserved	All

Programming Notes

If this field is any value other than MULTISAMPLECOUNT_1, the Surface Type must be SURFTYPE_2D

This field must be set to MULTISAMPLECOUNT_1 unless the surface is a Sampling Engine surface or Render Target surface.

This field must be set to MULTISAMPLECOUNT_1 for SINT MSRTs when all RT channels are not written

If this field is any value other than MULTISAMPLECOUNT_1, Surface Min LOD, Mip Count / LOD, and Resource Min LOD must be set to zero

26:0 Minimum Array Element

Project:	All
Exists If:	[Surface Type] == SURFTYPE_STRBUF

This field indicates the minimum array element that can be accessed as part of this surface. This field is added to the delivered array index before it is used to address the surface.

Value	Name
[0,226]	

2:0 Multisample Position Palette Index

Exists If:	[Surface Type] != SURFTYPE_STRBUF
Format:	U3

This field indicates the index into the sample position palette that the multisampled surface is using. This field is only used as a return value for the sampleinfo message, and is otherwise not used by hardware.

Value	Name
[0,7]	

5 31:25 X Offset

Project:	All
Format:	PixelOffset[8:2]

This field specifies the horizontal offset in pixels from the Surface Base Address to the start (origin) of the surface. This field effectively loosens the alignment restrictions on the origin of tiled surfaces. Previously, tiled surface origin was (by definition) located at the base address, and thus needed to satisfy the 4KB base address alignment restriction. Now the origin can be specified at a finer (4-wide x 2-high pixel) resolution.

Value	Name	Description
[0,508]		in multiples of 4 (low 2 bits missing)

Programming Notes

For linear surfaces, this field must be zero.
For surfaces accessed with the Data Port Media Block Read/Write message, the pixel size is assumed to be 32 bits in width.
For Surface Format with other than 8, 16, 32, 64, or 128 bits per pixel, this field must be zero.
If Render Target Rotation is set to other than RTROTATE_0DEG, this field must be zero.
If Surface Type is SURFTYPE_STRBUF, this field must be zero.
This field must be zero if Surface Format is PLANAR*. For all other surfaces, Xoffset must be programmed such that (max X of the draw rectangle)+Xoffset < 16K (max surface width)
For YUV422 surfaces, the pixel offset is in multiples of 2. Pixel offset specified in this case is PixelOffset[7:1]

24 **Reserved**

Project:	All
Format:	MBZ

23:20 **Y Offset**

Project:	All
Format:	RowOffset[4:1]

This field specifies the vertical offset in rows from the Surface Base Address to the start of the surface. (See additional description in the X Offset field)

Value	Name	Description
[0,30]		in multiples of 2 (low bit missing)

Programming Notes

For linear surfaces, this field must be zero. For render targets in which the Render Target Array Index is not zero, this field must be zero. For Surface Format with other than 8, 16, 32, 64, or 128 bits per pixel, this field must be zero. If Render Target Rotation is set to other than RTROTATE_0DEG, this field must be zero. For surfaces accessed in field mode (Vertical Line Stride = 1 or equivalent Media Block Read/Write message override), this field must be set to a multiple of 4. If Surface Type is SURFTYPE_STRBUF, this field must be zero. This field must be zero if Surface Format is PLANAR*. For all other surfaces, Yoffset must be programmed such that (Maximum Yof draw rectangle) + Yoffset < 16K (max surface height)

19:16 **Surface Object Control State**

Project:	All
Format:	MEMORY_OBJECT_CONTROL_STATE

Specifies the memory object control state for this surface.

15:14 **Reserved**

Format:	MBZ

13:8 **Reserved**

Project:	All
Format:	MBZ

7:4 **Surface Min LOD**

Project:	All
Format:	U4 in LOD units

		For Sampling Engine and Typed Surfaces:This field indicates the most detailed LOD that can be accessed as part of this surface. This field is added to the delivered LOD (sample_l, ld, or resinfo message types) before it is used to address the surface.For Other Surfaces:This field is ignored.

Value	Name
[0,14]	

Programming Notes
This field must be zero if the Surface Format is MONO8

	3:0	**MIP Count / LOD**

Project:	All
Format:	Sampling Engine and Typed Surfaces: U4 in (LOD units – 1)Render Target Surfaces: U4 in LOD units

For Sampling Engine and Typed Surfaces:This field indicates the number of MIP levels allowed to be accessed starting at Surface Min LOD, which must be less than or equal to the number of MIP levels actually stored in memory for this surface. Force the mip map access to be between the mipmap specified by the integer bits of the Min LOD and the ceiling of the value specified here.For Render Target Surfaces:This field defines the MIP level that is currently being rendered into. This is the absolute MIP level on the surface and is not relative to the Surface Min LOD field, which is ignored for render target surfaces.For Other Surfaces:This field is reserved : MBZ

Value	Name	Description	Project
[0,14]		Sampling Engine and Typed Surfaces: representing [1,15] MIP levels	
[0,14]		Render Target Surfaces: representing LOD	
0		Other Surfaces	
0h	Disable		All
1h	Enable		All

Programming Notes
The LOD of a render target must be the same as the LOD of the other render target(s) and of the depth buffer (defined in 3DSTATE_DEPTH_BUFFER).
For render targets with YUV surface formats, the LOD must be zero.
It is not legal to have more than one 1x1 mipmap. Software must ensure that MIP Count is set to end on the first 1x1 mipmap (or before).

6 This DW6 appllies only if Surface Format is not PLANAR*	31:30	**Reserved: MBZ**

Project:	All
Exists If:	[Surface Format] == PLANAR
Format:	MBZ

29:16	**X Offset for UV Plane**

Exists If:	[Surface Format] == PLANAR
Format:	U14 Row Offset

This field specifies the horizontal offset in pixels from the Surface Base Address to the start

(origin) of the interleaved UV plane. This field is only used for PLANAR surface formats.

Programming Notes
This field must indicate an even number of pixels.

15:14 Reserved

Project:	All
Exists If:	[Surface Format] == PLANAR
Format:	MBZ

31:12 MCS Base Address

Project:	All
Exists If:	([Surface Format] != 'PLANAR') && ([MCS Enable] == 'Enabled')
Format:	GraphicsAddress[31:12]

Specifies the 4kbyte-aligned base address of the MCS surface associated with the MSS surface specified in other 32 fields.

Programming Notes
The MCS surface must be stored as Tile Y.
The MCS surface shares Height, Width, Depth, Surface Min LOD, MIP Count / LOD, Surface Object Control State, Surface Array Spacing, and Minimum Array Element with the primary surface.

31:6 Append Counter Address

Project:	All
Exists If:	([Surface Format] != 'PLANAR') && ([MCS Enable] == 'Disabled')
Format:	GraphicsAddress[31:6]

Specifies the 64byte-aligned base address of the Append counter associated with this surface specified in other SURFACE_STATE fields.

11:3 MCS Surface Pitch

Project:	All
Exists If:	([Surface Format] != 'PLANAR') && ([MCS Enable] == 'Enabled')
Format:	U9-1 pitch in #Tiles

This field specifies the MCS surface pitch in (#Tiles – 1).

Value	Name	Description
[0,511]		representing [1 tile, 512 tiles]

5:2 Reserved

Project:	All
Exists If:	([Surface Format] != 'PLANAR') && ([MCS Enable] == 'Disabled')
Format:	MBZ

2:1 Reserved

Project:	All
Exists If:	([Surface Format] != 'PLANAR') && ([MCS Enable] == 'Enabled')
Format:	MBZ

1 Append Counter Enable

Project:	All
Exists If:	([Surface Format] != 'PLANAR') && ([MCS Enable] == 'Disabled')
Format:	Enable

Enables the use of the Append Counter with this surface. If disabled, all other Append counter fields are ignored.

	13:0	**Y Offset for UV Plane** Exists If: [Surface Format] == PLANAR Format: 14 Row Offset This field specifies the vertical offset in rows from the Surface Base Address to the start (origin) of the interleaved UV plane. This field is only used for PLANAR surface formats. **Programming Notes** This field must indicate an even number (bit 0 = 0).
	0	**MCS Enable** Project: All Format: Enable Enables the use of the MCS with this surface. If disabled, all other MCS fields are ignored. For Render Target and Sampling Engine Surfaces:If the surface is multisampled (Number of Multisamples any value other than MULTISAMPLECOUNT_1), this field must be enabled. For Other Surfaces:This field and the other MCS fields are ignored. **Programming Notes** When accessing a multisampled surface using the sampling engine, the MCS surface is read in a separate pass and is considered by hardware to be an independent surface. This same bitfield is used when MCS is enabled; also when disabled.

Errata	Description	Project
	If this field is disabled and the sampling engine *ld_mcs* message is issued on this surface, the MCS surface may be accessed. Software must ensure that the surface is defined to avoid GTT errors. This same bitfield is used when MCS is enabled; also when disabled. This field must be set to 0 for all SINT MSRTs when all RT channels are not written	
	When ld2dms from MCS on a subspan that some of the pixel on the pixel are not valid, can lead to data corruption.	

7	31	**Red Clear Color** Format: U1 enumerated type For Sampling Engine Multisampled Surfaces and Render Targets:Specifies the clear value for the red channel.For Other Surfaces:This field is ignored.

Value	Name	Description	Project
0	CC_ZERO	Clear color value is 0.0, correctly interpreted based on surface format.	All
1	CC_ONE	Clear color value is 1.0, correctly interpreted based on surface format.	All

	30	**Green Clear Color** Format: U1 enumerated type For Sampling Engine Multisampled Surfaces and Render Targets:Specifies the clear value for the green channel.For Other Surfaces:This field is ignored.

Value	Name	Description	Project

0	CC_ZERO	Clear color value is 0.0, correctly interpreted based on surface format.	All
1	CC_ZERO	Clear color value is 1.0, correctly interpreted based on surface format.	All

29 **Blue Clear Color**

Format:	U1 enumerated type

For Sampling Engine Multisampled Surfaces and Render Targets:Specifies the clear value for the blue channel.For Other Surfaces:This field is ignored.

Value	Name	Description	Project
0	CC_ZERO	Clear color value is 0.0, correctly interpreted based on surface format.	All
1	CC_ONE	Clear color value is 1.0, correctly interpreted based on surface format.	All

28 **Alpha Clear Color**

Format:	U1 enumerated type

For Sampling Engine Multisampled Surfaces and Render Targets:Specifies the clear value for the alpha channel.For Other Surfaces:This field is ignored.

Value	Name	Description	Project
0	CC_ZERO	Clear color value is 0.0, correctly interpreted based on surface format.	All
1	CC_ONE	Clear color value is 1.0, correctly interpreted based on surface format.	All

15:12 **Reserved**

Project:	All
Format:	MBZ

11:0 **Resource Min LOD**

Project:	All
Format:	U4.8 in LOD units

For Sampling Engine Surfaces:This field indicates the most detailed LOD that is present in the resource underlying the surface. Refer to the “LOD Computation Pseudocode” section for the use of this field.For Other Surfaces:This field is ignored.

Value	Name
[0,14]	

Programming Notes
This field must be zero if the Surface Format is MONO8
This field must be zero if the **ChromaKey Enable** is enabled in the associated sampler.

2.12.2.1.1 Surface Formats

The following table is used ONLY when ASTC_ENABLE is set to 0, which indicates the supported surface formats and the 9-bit encoding for each. Note that some of these formats are used not only by the Sampling Engine, but also by the Data Port and the Vertex Fetch unit. When ASTC_ENABLE is set to 1, please refer to SURFACE_STATE table on the surface format value and description. The name for all the ASTC texture format is ASTC.

SURFACE_FORMAT

Project:	All
Size (in bits):	9

The following table indicates the supported surface formats and the 9-bit encoding for each. Note that some of these formats are used not only by the Sampling Engine, but also by the Data Port and the Vertex Fetch unit.

Value	Name	Bits Per Element (BPE)	Description
000h	R32G32B32A32_FLOAT	128	
001h	R32G32B32A32_SINT	128	
002h	R32G32B32A32_UINT	128	
003h	R32G32B32A32_UNORM	128	
004h	R32G32B32A32_SNORM	128	
005h	R64G64_FLOAT	128	
006h	R32G32B32X32_FLOAT	128	
007h	R32G32B32A32_SSCALED	128	
008h	R32G32B32A32_USCALED	128	
020h	R32G32B32A32_SFIXED	128	
021h	R64G64_PASSTHRU	128	
040h	R32G32B32_FLOAT	96	
041h	R32G32B32_SINT	96	
042h	R32G32B32_UINT	96	
043h	R32G32B32_UNORM	96	
044h	R32G32B32_SNORM	96	
045h	R32G32B32_SSCALED	96	
046h	R32G32B32_USCALED	96	
050h	R32G32B32_SFIXED	96	
080h	R16G16B16A16_UNORM	64	
081h	R16G16B16A16_SNORM	64	
082h	R16G16B16A16_SINT	64	
083h	R16G16B16A16_UINT	64	
084h	R16G16B16A16_FLOAT	64	
085h	R32G32_FLOAT	64	
086h	R32G32_SINT	64	
087h	R32G32_UINT	64	
088h	R32_FLOAT_X8X24_TYPELESS	64	
089h	X32_TYPELESS_G8X24_UINT	64	
08Ah	L32A32_FLOAT	64	
08Bh	R32G32_UNORM	64	
08Ch	R32G32_SNORM	64	
08Dh	R64_FLOAT	64	
08Eh	R16G16B16X16_UNORM	64	
08Fh	R16G16B16X16_FLOAT	64	
090h	A32X32_FLOAT	64	
091h	L32X32_FLOAT	64	
092h	I32X32_FLOAT	64	
093h	R16G16B16A16_SSCALED	64	
094h	R16G16B16A16_USCALED	64	
095h	R32G32_SSCALED	64	

SURFACE_FORMAT			
096h	R32G32_USCALED	64	
0A0h	R32G32_SFIXED	64	
0A1h	R64_PASSTHRU	64	
0C0h	B8G8R8A8_UNORM	32	
0C1h	B8G8R8A8_UNORM_SRGB	32	
0C2h	R10G10B10A2_UNORM	32	
0C3h	R10G10B10A2_UNORM_SRGB	32	
0C4h	R10G10B10A2_UINT	32	
0C5h	R10G10B10_SNORM_A2_UNORM	32	
0C7h	R8G8B8A8_UNORM	32	
0C8h	R8G8B8A8_UNORM_SRGB	32	
0C9h	R8G8B8A8_SNORM	32	
0CAh	R8G8B8A8_SINT	32	
0CBh	R8G8B8A8_UINT	32	
0CCh	R16G16_UNORM	32	
0CDh	R16G16_SNORM	32	
0CEh	R16G16_SINT	32	
0CFh	R16G16_UINT	32	
0D0h	R16G16_FLOAT	32	
0D1h	B10G10R10A2_UNORM	32	
0D2h	B10G10R10A2_UNORM_SRGB	32	
0D3h	R11G11B10_FLOAT	32	
0D6h	R32_SINT	32	
0D7h	R32_UINT	32	
0D8h	R32_FLOAT	32	
0D9h	R24_UNORM_X8_TYPELESS	32	
0DAh	X24_TYPELESS_G8_UINT	32	
0DDh	L32_UNORM	32	
0DEh	A32_UNORM	32	
0DFh	L16A16_UNORM	32	
0E0h	I24X8_UNORM	32	
0E1h	L24X8_UNORM	32	
0E2h	A24X8_UNORM	32	
0E3h	I32_FLOAT	32	
0E4h	L32_FLOAT	32	
0E5h	A32_FLOAT	32	
0E6h	X8B8_UNORM_G8R8_SNORM	32	
0E7h	A8X8_UNORM_G8R8_SNORM	32	
0E8h	B8X8_UNORM_G8R8_SNORM	32	
0E9h	B8G8R8X8_UNORM	32	
0EAh	B8G8R8X8_UNORM_SRGB	32	
0EBh	R8G8B8X8_UNORM	32	
0ECh	R8G8B8X8_UNORM_SRGB	32	
0EDh	R9G9B9E5_SHAREDEXP	32	
0EEh	B10G10R10X2_UNORM	32	
0F0h	L16A16_FLOAT	32	
0F1h	R32_UNORM	32	
0F2h	R32_SNORM	32	
0F3h	R10G10B10X2_USCALED	32	

SURFACE_FORMAT			
0F4h	R8G8B8A8_SSCALED	32	
0F5h	R8G8B8A8_USCALED	32	
0F6h	R16G16_SSCALED	32	
0F7h	R16G16_USCALED	32	
0F8h	R32_SSCALED	32	
0F9h	R32_USCALED	32	
100h	B5G6R5_UNORM	16	
101h	B5G6R5_UNORM_SRGB	16	
102h	B5G5R5A1_UNORM	16	
103h	B5G5R5A1_UNORM_SRGB	16	
104h	B4G4R4A4_UNORM	16	
105h	B4G4R4A4_UNORM_SRGB	16	
106h	R8G8_UNORM	16	
107h	R8G8_SNORM	16	
108h	R8G8_SINT	16	
109h	R8G8_UINT	16	
10Ah	R16_UNORM	16	
10Bh	R16_SNORM	16	
10Ch	R16_SINT	16	
10Dh	R16_UINT	16	
10Eh	R16_FLOAT	16	
10Fh	A8P8_UNORM_PALETTE0	16	
110h	A8P8_UNORM_PALETTE1	16	
111h	I16_UNORM	16	
112h	L16_UNORM	16	
113h	A16_UNORM	16	
114h	L8A8_UNORM	16	
115h	I16_FLOAT	16	
116h	L16_FLOAT	16	
117h	A16_FLOAT	16	
118h	L8A8_UNORM_SRGB	16	
119h	R5G5_SNORM_B6_UNORM	16	
11Ah	B5G5R5X1_UNORM	16	
11Bh	B5G5R5X1_UNORM_SRGB	16	
11Ch	R8G8_SSCALED	16	
11Dh	R8G8_USCALED	16	
11Eh	R16_SSCALED	16	
11Fh	R16_USCALED	16	
122h	P8A8_UNORM_PALETTE0	16	
123h	P8A8_UNORM_PALETTE1	16	
124h	A1B5G5R5_UNORM	16	
125h	A4B4G4R4_UNORM	16	
126h	L8A8_UINT	16	
127h	L8A8_SINT	16	
140h	R8_UNORM	8	
141h	R8_SNORM	8	
142h	R8_SINT	8	
143h	R8_UINT	8	
144h	A8_UNORM	8	

SURFACE_FORMAT			
145h	I8_UNORM	8	
146h	L8_UNORM	8	
147h	P4A4_UNORM_PALETTE0	8	
148h	A4P4_UNORM_PALETTE0	8	
149h	R8_SSCALED	8	
14Ah	R8_USCALED	8	
14Bh	P8_UNORM_PALETTE0	8	
14Ch	L8_UNORM_SRGB	8	
14Dh	P8_UNORM_PALETTE1	8	
14Eh	P4A4_UNORM_PALETTE1	8	
14Fh	A4P4_UNORM_PALETTE1	8	
150h	Y8_UNORM	8	
152h	L8_UINT	8	
153h	L8_SINT	8	
154h	I8_UINT	8	
155h	I8_SINT	8	
180h	DXT1_RGB_SRGB	0	
181h	R1_UNORM	1	
182h	YCRCB_NORMAL	0	
183h	YCRCB_SWAPUVY	0	
184h	P2_UNORM_PALETTE0	2	
185h	P2_UNORM_PALETTE1	2	
186h	BC1_UNORM	0	(DXT1)
187h	BC2_UNORM	0	(DXT2/3)
188h	BC3_UNORM	0	(DXT4/5)
189h	BC4_UNORM	0	
18Ah	BC5_UNORM	0	
18Bh	BC1_UNORM_SRGB	0	(DXT1_SRGB)
18Ch	BC2_UNORM_SRGB	0	(DXT2/3_SRGB)
18Dh	BC3_UNORM_SRGB	0	(DXT4/5_SRGB)
18Eh	MONO8	1	
18Fh	YCRCB_SWAPUV	0	
190h	YCRCB_SWAPY	0	
191h	DXT1_RGB	0	
192h	FXT1	0	
193h	R8G8B8_UNORM	24	
194h	R8G8B8_SNORM	24	
195h	R8G8B8_SSCALED	24	
196h	R8G8B8_USCALED	24	
197h	R64G64B64A64_FLOAT	256	
198h	R64G64B64_FLOAT	192	
199h	BC4_SNORM	0	
19Ah	BC5_SNORM	0	
19Bh	R16G16B16_FLOAT	48	
19Ch	R16G16B16_UNORM	48	
19Dh	R16G16B16_SNORM	48	
19Eh	R16G16B16_SSCALED	48	
19Fh	R16G16B16_USCALED	48	
1A1h	BC6H_SF16	0	

SURFACE_FORMAT			
1A2h	BC7_UNORM	0	
1A3h	BC7_UNORM_SRGB	0	
1A4h	BC6H_UF16	0	
1A5h	PLANAR_420_8	0	
1A8h	R8G8B8_UNORM_SRGB	24	
1A9h	ETC1_RGB8	0	
1AAh	ETC2_RGB8	0	
1ABh	EAC_R11	0	
1ACh	EAC_RG11	0	
1ADh	EAC_SIGNED_R11	0	
1AEh	EAC_SIGNED_RG11	0	
1AFh	ETC2_SRGB8	0	
1B0h	R16G16B16_UINT	48	
1B1h	R16G16B16_SINT	48	
1B2h	R32_SFIXED	32	
1B3h	R10G10B10A2_SNORM	32	
1B4h	R10G10B10A2_USCALED	32	
1B5h	R10G10B10A2_SSCALED	32	
1B6h	R10G10B10A2_SINT	32	
1B7h	B10G10R10A2_SNORM	32	
1B8h	B10G10R10A2_USCALED	32	
1B9h	B10G10R10A2_SSCALED	32	
1BAh	B10G10R10A2_UINT	32	
1BBh	B10G10R10A2_SINT	32	
1BCh	R64G64B64A64_PASSTHRU	256	
1BDh	R64G64B64_PASSTHRU	192	
1C0h	ETC2_RGB8_PTA	0	
1C1h	ETC2_SRGB8_PTA	0	
1C2h	ETC2_EAC_RGBA8	0	
1C3h	ETC2_EAC_SRGB8_A8	0	
1C8h	R8G8B8_UINT	24	
1C9h	R8G8B8_SINT	24	
1FFh	RAW	0	

NOTE: “RAW” is supported only with buffers and structured buffers accessed via the untyped surface read/write and untyped atomic operation messages, which do not have a column in the table.

Errata : RT format of A8_UNORM, R32_UINT, R32_SINT, R32G32_SINT,R32G32_UINT are not supported in MSAA 4x/8x mode.

Errata : BC6H_SF16, BC6H_UF16, and BC7_SRGB are not supported and may result in data corruption if used.

2.12.2.1.2 Sampler Output Channel Mapping

The following table indicates the mapping of the channels from the surface to the channels output from the sampling engine. Formats with all four channels (R/G/B/A) in their name map each surface channel to the corresponding output, thus those formats are not shown in this table.

Some formats are supported only in DX10/OGL **Border Color Mode**. Those formats have only that mode indicated. Formats that behave the same way in both **Border Color Modes** are indicated by that column being blank.

Surface Format Name	Filtering	Shadow Map	Chroma Key	Border Color Mode	R	G	B	A	Border Color Mode	R	G	B	A
R32G32B32A32_FLOAT					R	G	B	A					
R32G32B32A32_SINT					R	G	B	A					
R32G32B32A32_UINT					R	G	B	A					
R32G32B32X32_FLOAT					R	G	B	1.0					
R32G32B32_FLOAT					R	G	B	1.0					
R32G32B32_SINT					R	G	B	1.0					
R32G32B32_UINT				DX10/OGL	R	G	B	1.0					
R16G16B16A16_UNORM					R	G	B	A					
R16G16B16A16_SNORM					R	G	B	A					
R16G16B16A16_SINT					R	G	B	A					
R16G16B16A16_UINT					R	G	B	A					
R16G16B16A16_FLOAT					R	G	B	A					
R32G32_FLOAT				DX10/OGL	R	G	0.0	1.0	DX9	R	G	1.0	1.0
R32G32_SINT				DX10/OGL	R	G	0.0	1.0					
R32G32_UINT				DX10/OGL	R	G	0.0	1.0					
R32_FLOAT_X8X24_TYPELESS				DX10/OGL	R	0.0	0.0	1.0					
X32_TYPELESS_G8X24_UINT				DX10/OGL	0.0	G	0.0	1.0					
L32A32_FLOAT				DX10/OGL	L	L	L	A					
R16G16B16X16_UNORM					R	G	B	1.0					
R16G16B16X16_FLOAT					R	G	B	1.0					
A32X32_FLOAT					0.0	0.0	0.0	A					
L32X32_FLOAT					L	L	L	1.0					
I32X32_FLOAT					I	I	I	I					
B8G8R8A8_UNORM					R	G	B	A					
B8G8R8A8_UNORM_SRGB					R	G	B	A					
R10G10B10A2_UNORM					R	G	B	A					
R10G10B10A2_UNORM_SRGB					R	G	B	A					
R10G10B10A2_UINT					R	G	B	A					
R10G10B10_SNORM_A2_UNORM					R	G	B	A					
R8G8B8A8_UNORM					R	G	B	A					
R8G8B8A8_UNORM_SRGB					R	G	B	A					
R8G8B8A8_SNORM					R	G	B	A					
R8G8B8A8_SINT					R	G	B	A					
R8G8B8A8_UINT					R	G	B	A					
R16G16_UNORM				DX10/OGL	R	G	0.0	1.0	DX9	R	G	1.0	1.0
R16G16_SNORM				DX10/O	R	G	0.0	1.0	DX9	R	G	1.0	1.0

Surface Format Name	Filtering	Shadow Map	Chroma Key	Border Color Mode	R	G	B	A	Border Color Mode	R	G	B	A
				GL									
R16G16_SINT				DX10/OGL	R	G	0.0	1.0					
R16G16_UINT				DX10/OGL	R	G	0.0	1.0					
R16G16_FLOAT				DX10/OGL	R	G	0.0	1.0	DX9	R	G	1.0	1.0
B10G10R10A2_UNORM					R	G	B	A					
B10G10R10A2_UNORM_SRGB					R	G	B	A					
R11G11B10_FLOAT					R	G	B	1.0					
R32_SINT				DX10/OGL	R	0.0	0.0	1.0					
R32_UINT				DX10/OGL	R	0.0	0.0	1.0					
R32_FLOAT				DX10/OGL	R	0.0	0.0	1.0	DX9	R	1.0	1.0	1.0
R24_UNORM_X8_TYPELESS				DX10/OGL	R	0.0	0.0	1.0					
X24_TYPELESS_G8_UINT				DX10/OGL	0.0	G	0.0	1.0					
L16A16_UNORM					L	L	L	A					
I24X8_UNORM					I	I	I	I					
L24X8_UNORM					L	L	L	1.0					
A24X8_UNORM					0.0	0.0	0.0	A					
I32_FLOAT					I	I	I	I					
L32_FLOAT					L	L	L	1.0					
A32_FLOAT					0.0	0.0	0.0	A					
B8G8R8X8_UNORM					R	G	B	1.0					
B8G8R8X8_UNORM_SRGB					R	G	B	1.0					
R8G8B8X8_UNORM					R	G	B	1.0					
R8G8B8X8_UNORM_SRGB					R	G	B	1.0					
R9G9B9E5_SHAREDEXP					R	G	B	1.0					
B10G10R10X2_UNORM					R	G	B	1.0					
L16A16_FLOAT					L	L	L	A					
B5G6R5_UNORM					R	G	B	1.0					
B5G6R5_UNORM_SRGB					R	G	B	1.0					
B5G5R5A1_UNORM					R	G	B	A					
B5G5R5A1_UNORM_SRGB					R	G	B	A					
B4G4R4A4_UNORM					R	G	B	A					
B4G4R4A4_UNORM_SRGB					R	G	B	A					
R8G8_UNORM				DX10/OGL	R	G	0.0	1.0	DX9	R	G	1.0	1.0
R8G8_SNORM				DX10/OGL	R	G	0.0	1.0	DX9	R	G	1.0	1.0
R8G8_SINT				DX10/OGL	R	G	0.0	1.0					
R8G8_UINT				DX10/OGL	R	G	0.0	1.0					

Surface Format Name	Filtering	Shadow Map	Chroma Key	Border Color Mode	R	G	B	A	Border Color Mode	R	G	B	A
R16_UNORM				DX10/OGL	R	0.0	0.0	1.0					
R16_SNORM				DX10/OGL	R	0.0	0.0	1.0					
R16_SINT				DX10/OGL	R	0.0	0.0	1.0					
R16_UINT				DX10/OGL	R	0.0	0.0	1.0					
R16_FLOAT				DX10/OGL	R	0.0	0.0	1.0	DX9	R	1.0	1.0	1.0
A8P8_UNORM_PALETTE0					R	G	B	A					
A8P8_UNORM_PALETTE1					R	G	B	A					
I16_UNORM					I	I	I	I					
L16_UNORM					L	L	L	1.0					
A16_UNORM					0.0	0.0	0.0	A					
L8A8_UNORM					L	L	L	A					
I16_FLOAT					I	I	I	I					
L16_FLOAT					L	L	L	1.0					
A16_FLOAT					0.0	0.0	0.0	A					
L8A8_UNORM_SRGB					L	L	L	A					
R5G5_SNORM_B6_UNORM					R	G	B	1.0					
P8A8_UNORM_PALETTE0					R	G	B	A					
P8A8_UNORM_PALETTE1					R	G	B	A					
R8_UNORM				DX10/OGL	R	0.0	0.0	1.0					
R8_SNORM				DX10/OGL	R	0.0	0.0	1.0					
R8_SINT				DX10/OGL	R	0.0	0.0	1.0					
R8_UINT				DX10/OGL	R	0.0	0.0	1.0					
A8_UNORM					0.0	0.0	0.0	A					
I8_UNORM					I	I	I	I					
L8_UNORM					L	L	L	1.0					
P4A4_UNORM_PALETTE0					R	G	B	A					
A4P4_UNORM_PALETTE0					R	G	B	A					
P8_UNORM_PALETTE0					R	G	B	A					
L8_UNORM_SRGB					L	L	L	1.0					
P8_UNORM_PALETTE1					R	G	B	A					
P4A4_UNORM_PALETTE1					R	G	B	A					
A4P4_UNORM_PALETTE1					R	G	B	A					
DXT1_RGB_SRGB					R	G	B	1.0					
R1_UNORM					R	0.0	0.0	1.0					
YCRCB_NORMAL					Cr	Y	Cb	1.0					
YCRCB_SWAPUVY					Cr	Y	Cb	1.0					
P2_UNORM_PALETTE0					R	G	B	A					
P2_UNORM_PALETTE1					R	G	B	A					
BC1_UNORM					R	G	B	A					
BC2_UNORM					R	G	B	A					

Surface Format Name	Filtering	Shadow Map	Chroma Key	Border Color Mode	R	G	B	A	Border Color Mode	R	G	B	A
BC3_UNORM					R	G	B	A					
BC4_UNORM				DX10/OGL	R	0.0	0.0	1.0					
BC5_UNORM				DX10/OGL	R	G	0.0	1.0					
BC1_UNORM_SRGB					R	G	B	A					
BC2_UNORM_SRGB					R	G	B	A					
BC3_UNORM_SRGB					R	G	B	A					
MONO8					N/A	N/A	N/A	N/A					
YCRCB_SWAPUV					Cr	Y	Cb	1.0					
YCRCB_SWAPY					Cr	Y	Cb	1.0					
DXT1_RGB					R	G	B	1.0					
FXT1					R	G	B	A					
BC4_SNORM				DX10/OGL	R	0.0	0.0	1.0					
BC5_SNORM				DX10/OGL	R	G	0.0	1.0					
R16G16B16_FLOAT					R	G	B	1.0					
BC6H_SF16					R	G	B	1.0					
BC7_UNORM					R	G	B	A					
BC7_UNORM_SRGB					R	G	B	A					
BC6H_UF16					R	G	B	1.0					

2.12.2.2 SURFACE_STATE for deinterlace, sample_8x8, and VME

MEDIA_SURFACE_STATE

Exists If: (MessageType == 'Deinterlace') || (MessageType == 'Sample_8x8')

Default Value: 0x00000000, 0x00000000, 0x00000000, 0x00000000, 0x00000000, 0x00000000

This is the SURFACE_STATE used by only deinterlace, sample_8x8, and VME messages.

DWord	Bit	Description
0	31:0	**Surface Base Address** Format: GraphicsAddress[31:0] Specifies the byte-aligned base address of the surface **Programming Notes** For SURFTYPE_BUFFER render targets, this field specifies the base address of first element of the surface. The surface is interpreted as a simple array of that single element type. The address must be naturally-aligned to the element size (e.g., a buffer containing R32G32B32A32_FLOAT elements must be 16-byte aligned).For SURFTYPE_BUFFER non-rendertarget surfaces, this field specifies the base address of the first element of the surface, computed in software by adding the surface base address to the byte offset of the element in the buffer. Mipmapped, cube and 3D sampling engine surfaces are stored in a “monolithic” (fixed) format, and only require a single address for the base texture.Linear render target surface base addresses must be element-size aligned, for non-YUV surface formats, or a multiple of 2 element-sizes for YUV surface formats. Other linear surfaces have no alignment requirements (byte alignment is sufficient.)Linear depth buffer surface base addresses must be 64-byte

aligned. Note that while render targets (color) can be SURFTYPE_BUFFER, depth buffers cannot.Tiled surface base addresses must be 4KB-aligned. Note that only the offsets from Surface Base Address are tiled, Surface Base Address itself is not transformed using the tiling algorithm.Tiled surface base addresses must be 4KB-aligned. Note that only the offsets from Surface Base Address are tiled, Surface Base Address itself is not transformed using the tiling algorithm.For tiled surfaces, the actual start of the surface can be offset from the Surface Base Address by the X Offset and Y Offset fields.Certain message types used to access surfaces have more stringent alignment requirements. Please refer to the specific message documentation for additional restrictions.

1 — 31:18 Height

Format:	U14

This field specifies the height of the surface in units of pixels. For PLANAR surface formats, this field indicates the height of the Y (luma) plane.

Value	Name	Description
[0,16383]		representing heights [1,16384]

Programming Notes

Height (field value + 1) must be a multiple of 2 for PLANAR_420 surfaces.If Vertical Line Stride is 1, this field indicates the height of the field, not the height of the frame.

17:4 Width

Format:	U14

This field specifies the width of the surface in units of pixels. For PLANAR surface formats, this field indicates the width of the Y (luma) plane.

Value	Name	Description
[0,16383]		representing widths [1,16384]

Programming Notes

- The Width specified by this field multiplied by the pixel size in bytes must be less than or equal to the surface pitch (specified in bytes via the Surface Pitch field).
- Width (field value + 1) must be a multiple of 2 for PLANAR_420, PLANAR_422, and all YCRCB_* and PLANAR_Y16_UNORM surfaces, and must be a multiple of 4 for PLANAR_411 and PLANAR_Y8_UNORM surfaces.
- For deinterlace messages, the Width (field value + 1) must be a multiple of 8.

3:2 Picture Structure

Specifies the encoding of the current picture.

Value	Name
00b	Frame Picture
01b	Top Field Picture
10b	Bottom Field Picture
11b	Invalid, not allowed

1:0 Cr(V)/Cb(U) Pixel Offset V Direction

Format:	U0.2

Specifies the distance to the U/V values with respect to the even numbered Y channels in the V direction

Programming Notes

This field is ignored for all formats except PLANAR_420_8

2 — 31:28 Surface Format

MEDIA_SURFACE_STATE

Specifies the format of the surface. All of the Y and G channels will use table 0 and all of the Cr/Cb/R/B channels will use table 1

Value	Name	Description
0	YCRCB_NORMAL	
1	YCRCB_SWAPUVY	
2	YCRCB_SWAPUV	
3	YCRCB_SWAPY	
4	PLANAR_420_8	
5	PLANAR_411_8	Deinterlace only
6	PLANAR_422_8	Deinterlace only
7	STMM_DN_STATISTICS	Deinterlace only
8	R10G10B10A2_UNORM	Sample_8x8 only
9	R8G8B8A8_UNORM	Sample_8x8 only
10	R8B8_UNORM (CrCb)	Sample_8x8 only
11	R8_UNORM (Cr/Cb)	Sample_8x8 only
12	Y8_UNORM	
15	Reserved	

27 **Interleave Chroma**

Format:	Enable

This field indicates that the chroma fields are interleaved in a single plane rather than stored as two separate planes. This field is only used for PLANAR surface formats.

26 **Reserved**

Format:	MBZ

25:22 **Surface Object Control State (MEMORY_OBJECT_CONTROL_STATE)**

This 4-bit field is used in various state commands and indirect state objects to define LLC cacheability including graphics data type for memory objects.

21 **Reserved**

Format:	MBZ

20:3 **Surface Pitch**

Format:	U18-1 pitch in Bytes

This field specifies the surface pitch in (#Bytes - 1).

Value	Name	Description
[0,2047]		For surfaces of type SURFTYPE_BUFFER: representing [1B, 2048B]
[0,2047]		For surfaces of type SURFTYPE_STRBUF: representing [1B, 2048B]
[0,524287]		For other linear surfaces: representing [1B, 256KB]
[511, 131071]		For X-tiled surface: representing [512B, 256KB] = [1 tile, 512 tiles]
[127, 131071]		For Y-tiled surfaces: representing [128B, 256KB] = [1 tile, 2048 tiles]

Programming Notes

For tiled surfaces, the pitch must be a multiple of the tile widthIf Half Pitch for Chroma is set, this field

must be a multiple of two tile widths for tiled surfaces, or a multiple of 2 bytes for linear surfaces.The Surface Pitches of current picture and reference picture should be declared as the identical type in VDI mode with identical Height, Width and Format.

	2	**Half Pitch for Chroma** Format: Enable This field indicates that the chroma plane(s) will use a pitch equal to half the value specified in the Surface Pitch field. This field is only used for PLANAR surface formats.
	1:0	**Tile Mode** Format: U2 enumerated type This field specifies the type of memory tiling (Linear, WMajor, XMajor, or YMajor) employed to tile this surface. See Memory Interface Functions for details on memory tiling and restrictions.

Value	Name	Description	Project
0h	TILEMODE_LINEAR	Linear mode (no tiling)	All
1h	Reserved	Reserved	All
2h	TILEMODE_XMAJOR	X major tiling	All
3h	TILEMODE_YMAJOR	Y major tiling	All

Programming Notes

- Refer to *Memory Data Formats* for restrictions on TileMode direction for the various buffer types. (Of particular interest is the fact that YMAJOR tiling is not supported for display/overlay buffers).
- The corresponding cache(s) must be invalidated before a previously accessed surface is accessed again with an altered state of this field.
- Linear surfaces can be mapped to Main Memory (uncached) or System Memory (cacheable, snooped). Tiled (X/Y/W) surfaces can only be mapped to Main Memory.

3	31:30	**Reserved** Project: All Format: MBZ
	29:16	**X Offset for U(Cb)** Format: U14 Pixel Offset For Planar surfaces this field specifies the horizontal offset in pixels from the Surface Base Address to the start (origin) of the U(Cb) plane or the interleaved UV plane if Interleave Chroma is enabled. For non planar surfaces this field specifies the horizontal offset in pixels from the Surface Base Address to the start (origin) of the surface

Programming Notes

For PLANAR_420 and PLANAR_422 surface formats, this field must indicate an even number of pixels.

	15	**Reserved** Format: MBZ
	14:0	**Y Offset for U(Cb)** Format: U15 Row Offset

MEDIA_SURFACE_STATE		
		For Planar surfaces this field specifies the vertical offset in rows from the Surface Base Address to the start (origin) of the U(Cb) plane or the interleaved UV plane if Interleave Chroma is enabled. For non planar surfaces this field specifies the vertical offset in pixels from the Surface Base Address to the start (origin) of the surface **Programming Notes** / **Project** This field must indicate an even number (bit 0 = 0).
4	31:30	**Reserved** Project: All Format: MBZ
	29:16	**X Offset for V(Cr)** Format: U14 Pixel Offset **Programming Notes** For PLANAR_420 and PLANAR_422 surface formats, this field must indicate an even number of pixels.
	15	**Reserved** Project: All Format: MBZ
	14:0	**Y Offset for V(Cr)** Format: U15 Row Offset This field specifies the veritical offset in rows from the Surface Base Address to the start (origin) of the V(Cr) plane. This field is only used for PLANAR surface formats with Interleave Chroma disabled. **Value** / **Name** 0,16380 **Programming Notes** / **Project** This field must indicate an even number (bit 0 = 0).
5	31:30	**Reserved** Project: Format: MBZ
	29:7	**Reserved** Format: MBZ
	6:0	**Reserved** Format: MBZ

2.12.3 SAMPLER_STATE

SAMPLER_STATE has different formats, depending on the message type used:

- For , the sample_8x8 and deinterlace messages use a different format of SAMPLER_STATE as detailed in the corresponding sections.

- For The **Min LOD** and **Max LOD** fields need range increased from [0.0,13.0] to [0.0,14.0] and fractional bits increased from 6 to 8. This requires a few fields to be moved as indicated in the text.

2.12.3.1 Sampler_State for Most Messages

SAMPLER_STATE

Exists If: (MessageType != 'Deinterlace') && (MessageType != 'Sample_8x8')

Default Value: 0x00000000, 0x00000000, 0x00000000, 0x00000000

This is the normal sampler state used by all messages that use SAMPLER_STATE except sample_8x8 and deinterlace. The sampler state is stored as an array of up to 16 elements, each of which contains the dwords described here. The start of each element is spaced 4 dwords apart. The first element of the sampler state array is aligned to a 32-byte boundary.

DWord	Bit	Description
0	31	**Sampler Disable** Project: All Format: Disable This field allows the sampler to be disabled. If disabled, all output channels will return 0.
	30	**Reserved** Project: All Format: MBZ
	29	**Texture Border Color Mode** Project: All Format: U1 enumerated type For some surface formats, the 32 bit border color is decoded differently based on the border color mode. In addition, the default value of channels not included in the surface may be affected by this field. Refer to the “Sampler Output Channel Mapping” table for the values of these channels, and for surface formats that may only support one of these modes. Also refer to the definition of SAMPLER_BORDER_COLOR_STATE for more details on the behavior of the two modes defined by this field.

Value	Name	Description	Project
0h	DX10/OGL	DX10/OGL mode for interpreting the border color	All
1h	DX9	DX9 and earlier mode for interpreting the border color	All

Programming Notes

If this bit changes for a given map with surface format R8G8_SNORM or R16_FLOAT, there must be two different SURFACE_STATE pointers for Texture Border Color Mode = 0 and 1. This is done to prevent an aliasing problem in the L1 cache with the default value for the missing channels changing. Alternately there could be a flush every time this changes.

This field must be set to DX9 mode when used with surfaces that have Surface Format P4A4_UNORM or A4P4_UNORM.

This field must be set to DX10/OGL mode when used with surfaces that have Surface Format YCRCB_SWAPUV or YCRCB_SWAPY.

This field must be set to DX10/OGL mode if **Surface Format** for the associated surface is UINT OR SINT.

This field must be set to DX10/OGL mode if REDUCTION_MINIMUM or REDUCTION_MAXIMUM or message type is sample_min or sample_max.

This field must be set to DX10/OGL mode if either **Min** or **Mag Mode Filter** is set to MAPFILTER_FLEXIBLE.

28 **LOD PreClamp Enable**

Format:	U1 enumerated type

When enabled, the computed LOD is clamped to [max,min] mip level before the mag-vs-min determination is performed. This is how the OpenGL API currently performs min/mag determination, and therefore it is expected that an OpenGL driver would need to set this bit.

Value	Name	Description
1h	OGL	OGL Mode (LOD PreClamp enabled)

27 **Reserved**

Format:	MBZ

26:22 **Base Mip Level**

Format:	U4.1

Range: [0.0, 14.0]

Specifies which mip level is considered the “base” level when determining mag-vs-min filter and selecting the “base” mip level.

21:20 **Mip Mode Filter**

Project:	All
Format:	U2 enumerated type

This field determines if and how mip map levels are chosen and/or combined when texture filtering.

Value	Name	Description	Project
0h	MIPFILTER_NONE	Disable mip mapping – force use of the mipmap level corresponding to Min LOD.	All
1h	MIPFILTER_NEAREST	Nearest, Select the nearest mip map	All
2h	Reserved		All
3h	MIPFILTER_LINEAR	Linearly interpolate between nearest mip maps (combined with linear min/mag filters this is analogous to “Trilinear” filtering).	All

Programming Notes

MIPFILTER_LINEAR is not supported for surface formats that do not support “Sampling Engine Filtering” as indicated in the Surface Formats table unless using the sample_c message type or minimum/maximum operation.

19:17 **Mag Mode Filter**

Project:	All
Format:	U2 enumerated type

This field determines how texels are sampled/filtered when a texture is being “magnified” (enlarged). For volume maps, this filter mode selection also applies to the 3rd (inter-layer) dimension.

SAMPLER_STATE

Value	Name	Description	Project
0h	MAPFILTER_NEAREST	Sample the nearest texel	All
1h	MAPFILTER_LINEAR	Bilinearly filter the 4 nearest texels	All
2h	MAPFILTER_ANISOTROPIC	Perform an “anisotropic” filter on the chosen mip level	All
4h-5h	Reserved		All
6h	MAPFILTER_MONO	Perform a monochrome convolution filter	All
7h	Reserved		All

Programming Notes
Only MAPFILTER_NEAREST and MAPFILTER_LINEAR are supported for surfaces of type SURFTYPE_3D.
Only MAPFILTER_NEAREST is supported for surface formats that do not support “Sampling Engine Filtering” as indicated in the Surface Formats table unless using the sample_c message type or minimum/maximum operation.
MAPFILTER_MONO: Only CLAMP_BORDER texture addressing mode is supported. . Both Mag Mode Filter and Min Mode Filter must be programmed to MAPFILTER_MONO. Mip Mode Filter must be MIPFILTER_NONE. Only valid on surfaces with Surface Format MONO8 and with Surface Type SURFTYPE_2D.
MAPFILTER_FLEXIBLE: The Surface Type of the surface being sampled must be SURFTYPE_2D.
MAPFILTER_ANISOTROPIC may cause artifacts at cube edges if enabled for cube maps with the TEXCOORDMODE_CUBE addressing mode.
MAPFILTER_ANISOTROPIC will be overridden to MAPFILTER_LINEAR when using a sample_l or sample_l_c message type or when Force LOD to Zero is set in the message header.

Errata	Description	Project
	MAPFILTER_ANISOTROPIC may have data corruption when sampled from surface with BC6H_UF16 or BC6H_SF16	

16:14 **Min Mode Filter**

Project:	All
Format:	U2 enumerated type

This field determines how texels are sampled/filtered when a texture is being “minified” (shrunk). For volume maps, this filter mode selection also applies to the 3rd (inter-layer) dimension.See Mag Mode Filter

13:1 **Texture LOD Bias**

Project:	All
Format:	S4.8 2’s complement

Range: [-16.0, 16.0)

This field specifies the signed bias value added to the calculated texture map LOD prior to min-vs-mag determination and mip-level clamping. Assuming mipmapping is enabled, a positive LOD bias will result in a somewhat blurrier image (using less-detailed mip levels) and possibly higher performance, while a negative bias will result in a somewhat crisper image (using more-detailed mip levels) and may lower performance.

		Programming Notes There is no requirement or need to offset the LOD Bias in order to produce a correct LOD for texture filtering (as was required for correct bilinear and anisotropic filtering in some legacy devices).
	0	**Anisotropic Algorithm** Project: All Format: U1 enumerated type Controls which algorithm is used for anisotropic filtering. Generally, the EWA approximation algorithm results in higher image quality than the legacy algorithm.

Value	Name	Description	Project
0h	LEGACY	Use the legacy algorithm for anisotropic filtering	All
1h	EWA Approximation	Use the new EWA approximation algorithm for anisotropic filtering	All

1	31:20	**Min LOD** Project: All Format: U4.8 in LOD units Range: [0.0, 14.0], where the upper limit is also bounded by the Max LOD. This field specifies the minimum value used to clamp the computed LOD after LOD bias is applied. Note that the minification-vs.-magnification status is determined after LOD bias and before this maximum (resolution) mip clamping is applied.The integer bits of this field are used to control the “maximum” (highest resolution) mipmap level that may be accessed (where LOD 0 is the highest resolution map). The fractional bits of this value effectively clamp the inter-level trilinear blend factor when trilinear filtering is in use. **Programming Notes** If Min LOD is greater than Max LOD, Min LOD takes precedence, i.e. the resulting LOD will always be Min LOD. This field must be zero if the Min or Mag Mode Filter is set to MAPFILTER_MONO
	19:8	**Max LOD** Project: All Format: U4.8 in LOD units Range: [0.0, 14.0] This field specifies the maximum value used to clamp the computed LOD after LOD bias is applied. Note that the minification-vs.-magnification status is determined after LOD bias and before this minimum (resolution) mip clamping is applied.The integer bits of this field are used to control the “minimum” (lowest resolution) mipmap level that may be accessed.The fractional bits of this value effectively clamp the inter-level trilinear blend factor when trilinear filtering is in use.Force the mip map access to be between the mipmap specified by the integer bits of the Min LOD and the ceiling of the value specified here.

SAMPLER_STATE

	7:4	**Reserved** Format: MBZ
	3:1	**Shadow Function** Project: All Format: U3 enumerated type This field is used for shadow mapping support via the sample_c message type, and specifies the specific comparison operation to be used. The comparison is between the texture sample red channel (except for alpha-only formats which use the alpha channel), and the “ref” value provided in the input message.

Value	Name	Project
0h	PREFILTEROP_ALWAYS	All
1h	PREFILTEROP_NEVER	All
2h	PREFILTEROP_LESS	All
3h	PREFILTEROP_EQUAL	All
4h	PREFILTEROP_LEQUAL	All
5h	PREFILTEROP_GREATER	All
6h	PREFILTEROP_NOTEQUAL	All
7h	PREFILTEROP_GEQUAL	All

	0	**Cube Surface Control Mode** Project: All Format: U1 enumerated type When sampling from a SURFTYPE_CUBE surface, this field controls whether the TC* Address Control Mode fields are interpreted as programmed or overridden to TEXCOORDMODE_CUBE.

Value	Name	Project
0h	CUBECTRLMODE_PROGRAMMED	All
1h	CUBECTRLMODE_OVERRIDE	All

Programming Notes
This field must be set to CUBECTRLMODE_PROGRAMMED

2	31:5	**Border Color Pointer** Format: DynamicStateOffset[31:5]

Description	Project
This field specifies the pointer to SAMPLER_BORDER_COLOR_STATE, which contains the “border” color to be used when accessing texels not contained within the texture map.	
This pointer is relative to the Dynamic State Base Address.	
Field definition if Flexible Filter Mode = FLEX_NONSEP:	

Errata	Description	Project
	In order to ensure correct data sampling, it must be ensured that the bits of this field with three LSBs of zero appended do not match the offset (without base address added) of	

		any instance of BINDING_TABLE_STATE (includes each 32-bit entry) used by the sampling engine between texture cache invalidations.
	4:0	**Reserved** Format: MBZ
3	31:26	**Reserved** Format: MBZ
	25	ChromaKey EnableFormat: Enable This field enables the chroma key function. **Programming Notes** Supported only on a specific subset of surface formats. See section “Surface Formats” for supported formats.This field must be disabled if min or mag filter is MAPFILTER_MONO or MAPFILTER_ANISOTROPIC.This field must be disabled if used with a surface of type SURFTYPE_3D.
	24:23	**ChromaKey Index** Format: U2 This field specifies the index of the ChromaKey Table entry associated with this Sampler. This field is a “don’t care” unless ChromaKey Enable is ENABLED.

Value	Name
[0,3]	

22	**ChromaKey Mode** Format: U1 Enumerated Type This field specifies the behavior of the device in the event of a ChromaKey match. This field is ignored if ChromaKey is disabled.KEYFILTER_KILL_ON_ANY_MATCH:In this mode, if any contributing texel matches the chroma key, the corresponding pixel mask bit for that pixel is cleared. The result of this operation is observable only if the Killed Pixel Mask Return flag is set on the input message. KEYFILTER_REPLACE_BLACK:In this mode, each texel that matches the chroma key is replaced with (0,0,0,0) (black with alpha=0) prior to filtering. For YCrCb surface formats, the black value is A=0, R(Cr)=0x80, G(Y)=0x10, B(Cb)=0x80. This will tend to darken/fade edges of keyed regions. Note that the pixel pipeline must be programmed to use the resulting filtered texel value to gain the intended effect, e.g., handle the case of a totally keyed-out region (filtered texel alpha==0) through use of alpha test, etc.

Value	Name	Project
0h	KEYFILTER_KILL_ON_ANY_MATCH	All
1h	KEYFILTER_REPLACE_BLACK	All

21:19	**Maximum Anisotropy** Project: All Format: U3 enumerated type This field clamps the maximum value of the anisotropy ratio used by the MAPFILTER_ANISOTROPIC filter (Min or Mag Mode Filter).

Value	Name	Description	Project
0h	ANISORATIO_2	At most a 2:1 aspect ratio filter is used	All

1h	ANISORATIO_4	At most a 4:1 aspect ratio filter is used	All
2h	ANISORATIO_6	At most a 6:1 aspect ratio filter is used	All
3h	ANISORATIO_8	At most a 8:1 aspect ratio filter is used	All
4h	ANISORATIO_10	At most a 10:1 aspect ratio filter is used	All
5h	ANISORATIO_12	At most a 12:1 aspect ratio filter is used	All
6h	ANISORATIO_14	At most a 14:1 aspect ratio filter is used	All
7h	ANISORATIO_16	At most a 16:1 aspect ratio filter is used	All

18:13 **Address Rounding Enable**

Project:	All
Format:	6-bit mask of enables

Controls whether the U/V/R texture address is rounded or truncated before being used to select texels to sample. Each bit provides independent control of rounding on one texture address dimension (U/V/R) in either mag or min filter mode.

Value	Name
1xxxxxb	U address mag filter
x1xxxxb	U address min filter
xx1xxxb	V address mag filter
xxx1xxb	V address min filter
xxxx1xb	R address mag filter
xxxxx1b	R address min filter

Programming Notes	Project
Hardware will force rounding enable to 0 when message is **gather4**, **gather4_po**, **gather4_c**, or **gather4_po_c**.	

12:11 **Trilinear Filter Quality**

Project:	All
Format:	U2 enumerated type

Selects the quality level for the trilinear filter.

Value	Name	Description	Project
0	TRIQUAL_FULL	Full Quality. Both mip maps are sampled under all circumstances.	All
2	TRIQUAL_MED	Medium Quality. If the contribution of one mip map is less than 25%, only the other mip map contributes.	All
3	TRIQUAL_LOW	Low Quality. If the contribution of one mip map is less than 37.5%, only the other mip map contributes.	All

10 **Non-normalized Coordinate Enable**

Project:	All
Format:	Enable

This field, if enabled, specifies that the input coordinates (U/V/R) are in non-normalized space, where each integer increment is one texel on LOD 0. If disabled, coordinates are normalized, where the range 0 to 1 spans the entire surface.

Programming Notes

The following state must be set as indicated if this field is *enabled*:

- TCX/Y/Z Address Control Mode must be TEXCOORDMODE_CLAMP, TEXCOORDMODE_HALF_BORDER, or TEXCOORDMODE_CLAMP_BORDER.

- Surface Type must be SURFTYPE_2D or SURFTYPE_3D.
- Mag Mode Filter must be MAPFILTER_NEAREST or MAPFILTER_LINEAR.
- Min Mode Filter must be MAPFILTER_NEAREST or MAPFILTER_LINEAR.
- Mip Mode Filter must be MIPFILTER_NONE.
- Min LOD must be 0.
- Max LOD must be 0.
- MIP Count must be 0.
- Surface Min LOD must be 0.
- Texture LOD Bias must be 0.

9 **Reserved**

Format:	MBZ

8:6 **TCX Address Control Mode**

Project:	All
Format:	U3 enumerated type

Controls how the 1st (TCX, aka U) component of input texture coordinates are mapped to texture map addresses – specifically, how coordinates "outside" the texture are handled (wrap/clamp/mirror). The setting of this field is subject to being overridden by the Cube Surface Control Mode field when sampling from a SURFTYPE_CUBE surface.

Value	Name	Description	Project
0h	TEXCOORDMODE_WRAP	Map is repeated in the U direction	All
1h	TEXCOORDMODE_MIRROR	Map is mirrored in the U direction	All
2h	TEXCOORDMODE_CLAMP	Map is clamped to the edges of the accessed map	All
3h	TEXCOORDMODE_CUBE	For cube-mapping, filtering in edges access adjacent map faces	All
4h	TEXCOORDMODE_CLAMP_BORDER	Map is infinitely extended with the border color	All
5h	TEXCOORDMODE_MIRROR_ONCE	Map is mirrored once about origin, then clamped	All
7h	Reserved		All

Programming Notes	Project
When using cube map texture coordinates, only TEXCOORDMODE_CLAMP and TEXCOORDMODE_CUBE settings are valid, and each TC component must have the same Address Control mode.	
When TEXCOORDMODE_CUBE is not used accessing a cube map, the map's Cube Face Enable field must be programmed to 111111b (all faces enabled).	
MAPFILTER_MONO: Texture addressing modes must all be set to TEXCOORDMODE_CLAMP_BORDER. The **Border Color** is ignored in this mode, a constant value of 0 is used for border color. Software must pad the border texels within the map itself with 0.	

5:3 **TCY Address Control Mode**

Project:	All
Format:	U3 enumerated type

SAMPLER_STATE		
		Controls how the 2nd (TCY, aka V) component of input texture coordinates are mapped to texture map addresses – specifically, how coordinates “outside” the texture are handled (wrap/clamp/mirror). See Address TCX Control Mode above for details **Programming Notes** If this field is set to TEXCOORDMODE_CLAMP_BORDER or TEXCOORDMODE_HALF_BORDER and a 1D surface is sampled, incorrect blending with the border color in the vertical direction may occur.
	2:0	**TCZ Address Control Mode** Project: All Format: U3 enumerated type **Description** / **Project** Controls how the 3rd (TCZ) component of input texture coordinates are mapped to texture map addresses – specifically, how coordinates “outside” the texture are handled (wrap/clamp/mirror).See Address TCX Control Mode above for details If this field is set to TEXCOORDMODE_CLAMP_BORDER for 3D maps on formats without an alpha channel, samples straddling the map in the Z direction may have their alpha channels off by 1.

2.12.3.2 SAMPLER_STATE for Sample_8x8 Message

SAMPLER_STATE for Sample_8x8 Message		
Default Value: 0x00000000, 0x00000000, 0x0D090801, 0x721A03C6		
. This state definition is used only by the sample_8x8 message. This state is stored as an array of up to 16 elements, each of which contains the dwords described here. The start of each element is spaced 4 dwords apart. The first element of the array is aligned to a 32-byte boundary. The index with range 0-15 that selects which element is being used to determine the **Sampler Index** in the message descriptor.		
Programming Notes **IEF Filter Type** was dropped and is assumed to be Detailed filter **IEF Filter Size** was dropped and assumed to be 5x5. **IEF bypass** must always be forced to 1, if Y/G-channel is masked.		
DWord	**Bit**	**Description**
0	31:30	**Reserved** Format: MBZ
	29	**IEF Bypass** Format: MBZ

SAMPLER_STATE for Sample_8x8 Message

		Causes IEF function to be bypassed, VSA will output neutral values.
	28:19	**Reserved** Format: MBZ
	18	**ChromaKey Enable** Format: Enable This field enables chroma keying when accessing this particular texture map. **Programming Notes** For sample_8x8 instructions KEYFILTER_REPLACE_BLACK is assumed if chromakey is enabled. For 10 bit formats only the 8 MSBs will be compared.
	17:16	**ChromaKey Index** Format: U2 This field specifies the index of the ChromaKey Table entry associated with this Sampler. This field is a “don’t care” unless ChromaKey Enable is ENABLED. **Value** / **Name** [0,3]
	15:8	**Reserved** Format: MBZ
	7:0	**Global Noise Estimation** Format: U8 Global noise estimation of previous frame.
1	31:5	**Sampler 8x8 State Pointer** Format: DynamicStateOffset[31:5] This field specifies the pointer to the SAMPLER_8x8_STATE structure. This pointer is relative to the Dynamic State Base Address. **Programming Notes** This field must be set to the same value in all sample_8x8 type SAMPLER_STATE instances applied to a given primitive. PIPE_CONTROL with State/Instruction Cache Invalidate set and the CS Stall field set is required between primitives that use different values of this field.
	4:0	**Reserved** Format: MBZ
2	31	**Reserved** Format: MBZ
	30:26	**R5c Coefficient** Default Value: 3 Format: U0.5 IEF smoothing coefficient, see IEF map.

SAMPLER_STATE for Sample_8x8 Message

	25:21	**R5x Coefficient** Default Value: 8 Format: U0.5 IEF smoothing coefficient, see IEF map.
	20:16	**R5c Coefficient** Default Value: 9 Format: U0.5 IEF smoothing coefficient, see IEF map.
	15:14	**Reserved** Format: MBZ
	13:8	**Strong Edge Threshold** Default Value: 8 Format: U6 If EM > Strong Edge Threshold, the basic VSA detects a strong edge.
	7:6	**Reserved** Format: MBZ
	5:0	**Weak Edge Threshold** Default Value: 1 Format: U6 If Strong Edge Threshold > EM > Weak Edge Threshold, the basic VSA detects a weak edge.
3	31	**IEF4Smooth Enable** **Value** / **Name** / **Description** 1 / / IEF is operating as a content adaptive smooth filter based on 3x3 region 0 / **[Default]** / IEF is operating as a content adaptive detail filter based on 5x5 region
	30:28	**Strong Edge Weight** Default Value: 7 Format: U3 Sharpening strength when a strong edge is found in basic VSA..
	27	**Reserved** Format: MBZ
	26:24	**Regular Weight** Default Value: 2 Format: U3 Sharpening strength when a weak edge is found in basic VSA.
	23	**Reserved** Format: MBZ
	22:20	**Non Edge Weight** Default Value: 1 Format: U3 Sharpening strength when no edge is found in basic VSA.

SAMPLER_STATE for Sample_8x8 Message		
	19:14	**Gain Factor** Default Value: 40 Format: U6 User control sharpening strength.
	13:11	**Reserved** Format: MBZ
	10:6	**R3c Coefficient** Default Value: 15 Format: 0.5 IEF smoothing coefficient, see IEF map.
	5	**Reserved** Format: MBZ
	4:0	**R3x Coefficient** Default Value: 6 Format: U6 IEF smoothing coefficient, see IEF map.

2.12.3.3 For Deinterlace Message

DEINTERLACE_SAMPLER_STATE		
Exists If:	MessageType == 'Deinterlace'	
Default Value:	0x00000800, 0x00000000, 0x04950100, 0x407D0000, 0x00000000, 0x00000000, 0x00000000, 0x005064A5	
This state definition is used only by the *deinterlace* message. This state is stored as an array of up to 8 elements, each of which contains the dwords described here. The start of each element is spaced 8 dwords apart. The first element of the array is aligned to a 32-byte boundary. The index with range 0-7 that selects which element is being used is multiplied by 2 to determine the **Sampler Index** in the message descriptor.		
DWord	**Bit**	**Description**
0	31:24	**Denoise STAD Threshold** Threshold for denoise sum of temporal absolute differences.
	23:16	**Denoise Maximum History** Maximum allowed value for denoise history. **Value** / **Name** / **Description** 128-240
	15	**Reserved** Format: MBZ
	14	**VDI Walker Frame Sharing Enable** Format: U1 enumerated type For a GT2 system with 2 half-slices, this field controls how the frame is shared by the two deinterlacer

walkers.

Value	Name
0	There is only a single deinterlacer which must walk the entire frame. VDI Walker Y Stride is ignored.
1	The screen is shared by the two deinterlacers as controlled by the VDI Walker Y Stride

13:12 **VDI Walker Y Stride**

Format:	U2 enumerated type

This field controls if the VDI walker skips pixels as it goes down the screen. This is used when a pair of VDI'S are splitting the frame between them. The stride also implies the offset used by the 2nd half-slice.

Value	Name
0	Stride of 1 block (where a block is 4x4 when DI is enabled and 4x8 when DN only), offset for the 2nd half-slice is the surface height.
1	Stride of 2 blocks (every other row of blocks calculated by this VDI), offset for the 2nd half-slice is 1 block.
2	Stride of 4 blocks (2 vertical blocks calclated by this VDI, then skip 2), offset for the 2nd half-slice is 2 blocks.
3	Stride of 8 blocks (4 vertical blocks calculated by this VDI, then skip 4), offset for the 2nd half-slice is 4 blocks.

11:8 **Denoise History Delta**

Default Value:	8

Amount that denoise_history is increased.

7:0 **Denoise ASD Threshold**

Threshold for denoise absolute sum of differences.

Value	Name	Description
0-63		

1 31:30 **Reserved**

Format:	MBZ

29:24 **Temporal Difference Threshold**

Programming Notes
The difference between **Temporal Difference Threshold** and **Low Temporal Difference Threshold** must be larger than 0 and less than or equal to 16, except when both thresholds are set to 0.

23:22 **Reserved**

Format:	MBZ

21:16 **Low Temporal Difference Threshold**

Programming Notes
The difference between **Temporal Difference Threshold** and **Low Temporal Difference Threshold** must be larger than 0 and less than or equal to 16, except when both thresholds are set to 0.

15:13 **STMM C2**

Bias for divisor in STMM equation. The range represents values [1,8]

Value	Name	Description
0-7		

12:8 **Denoise Moving Pixel Threshold**

Threshold for number of moving pixels to declare a block to be moving.

Value	Name	Description
0-16		

DEINTERLACE_SAMPLER_STATE

	7:0	**Denoise Threshold for Sum of Complexity Measure**
2	31:30	**Reserved** Format: MBZ
	29:24	**Good Neighbor Threshold** Maximum difference from current pixel for neighboring pixels to be considered a good neighbor. **Value** / **Name** / **Description** 4 / **[Default]** / depending on GNE of previous frame
	23:20	**CAT Slope** Format: U4-1 Determines the slope of the Content Adaptive Threshold. +1 added internally to get CAT_slope. **Value** / **Name** / **Description** 9 / **[Default]** / CAT_slope value = 10
	19:16	**SAD Tight Threshold** Default Value: 5 Format: U4
	15:14	**Smooth MV Threshold** Format: U2
	13:12	**Reserved** Format: MBZ
	11:8	**BNE Edge Threshold** Default Value: 1 Format: U4 Threshold for detecting an edge in block noise estimate.
	7:0	**Block Noise Estimate Noise Threshold** Format: U8 Threshold for noise maximum/minimum. **Value** / **Name** / **Description** 0-31
3	31	**STMM Blending Constant Select** Format: U1 **Value** / **Name** 0 / Use Minimum STMM for stmm_md_th 1 / Use Maximum STMM for stmm_md_th
	30:24	**Blending constant across time for large values of STMM** Default Value: 64 Format: U7
	23:16	**Blending constant across time for small values of STMM** Default Value: 125 Format: U8
	15:14	**Reserved** Format: MBZ
	13:8	**Multiplier for VECM** Format: U6 Determines the strength of the vertical edge complexity measure.

DEINTERLACE_SAMPLER_STATE

	7:0	**Maximum STMM** Format: U8 Largest allowed STMM in blending equations
4	31:24	**Minimum STMM** Format: U8 Smallest allowed STMM in blending equations
	23:22	**STMM Shift Down** Format: U2 Amount to shift STMM down (quantize to fewer bits)

Value	Name
0	Shift by 4
1	Shift by 5
2	Shift by 6
3	Reserved

	21:20	**STMM Shift Up** Format: U2 Amount to shift STMM up (set range).

Value	Name
0	Shift by 6
1	Shift by 7
2	Shift by 8
3	Reserved

	19:16	**STMM Output Shift** Format: U4 Amount to shift output of STMM blend equation

Value	Name	Description
0-16		

Programming Notes
The value of this field must satisfy the following equation: stmm_max – stmm_min = 2 ^ stmm_output_shift

	15:8	**SDI Threshold** Format: U8 Threshold for angle detection in SDI algorithm.
	7:0	**SDI Delta** Format: U8 Delta value for angle detection in SDI algorithm.
5	31:24	**SDI Fallback Mode 1 T1 Constant** Format: U8
	23:16	**SDI Fallback Mode 1 T2 Constant** Format: U8
	15:8	**SDI Fallback Mode 2 Constant (Angle2x1)**

DEINTERLACE_SAMPLER_STATE

		Format: U8
	7:0	**FMD Temporal Difference Threshold** Format: U8
6	31:24	**FMD #1 Vertical Difference Threshold** Format: U8
	23:16	**FMD #2 Vertical Difference Threshold** Format: U8
	15:14	**CAT Threshold 1** Default Value: 0 Format: U2
	13:8	**FMD Tear Threshold** Format: U6
	7	**MCDI Enable** Use Motion Compensated Deinterlace algorithm. Ignored if DI Enable is off.
	6	**Progressive DN** Format: Enable Indicates that the denoise algorithm should assume progressive input when filtering neighboring pixels. **DI Enable** must be disabled when this field is enabled

Value	Name
0	DN assumes interlaced video and filters alternate lines together
1	DN assumes progressive video and filters neighboring lines together

	5	**DN/DI First Frame** Format: Enable Indicates that this is the first frame of the stream, so previous clean is not available

Value	Name
0	Not first field; previous clean surface state is valid
1	First field; previous clean surface state is invalid

	4	**DN/DI Stream ID** Format: U1 Distinguishes between the two simultaneous streams that are supported. Used to update the GNE and FMD counters for that stream.
	3	**DN/DI Top First** Format: Enable Indicates the top field is first in sequence, otherwise bottom is first

Value	Name
0	Bottom field occurs first in sequence
1	Top field occurs first in sequence

	2	**DI Partial** Format: Enable If **DI Enable** and **DI Partial** are both enabled, the deinterlacer will output the partial VDI writeback message.

Value	Name
0	Output normal VDI writeback message (only if DI Enable is enabled also)
1	Output partial VDI writeback message (only if DI Enable is enabled also)

	1	**DI Enable**

DEINTERLACE_SAMPLER_STATE

		Format: Enable

Deinterlacer is bypassed if this is disabled: the output is the same as the input (same as a 2:2 cadence). FMD and STMM are not calculated and the values in the response message are 0.

Value	Name
0	Do not calculate DI
1	Calculate DI

Programming Notes
DI Enable and **DN Enable** cannot both be disabled

	0	**DN Enable**
		Format: Enable

Denoise is bypassed if this is low BNE is still calculated and output, but the denoised fields are not. VDI does not read in the denoised previous frame but uses the pointer for the original previous frame.

Value	Name
0	Do not denoise frame
1	Denoise frame

Programming Notes
DI Enable and **DN Enable** cannot both be disabled

7	31:23	**Column Width Minus 1**
		Format: U9

This field specifies the (column width-1) / stride in units of blocks (Each blocks has width 16 pixels). A column width * 16 that equals the width of the frame means the walker will walk to the end of the frame.
The value of this field is interpreted as binary value + 1, so the range represents column widths of [1,512].

Value	Name	Description
0-511		

	22:19	**Neighbor Pixel Threshold**
		Default Value: 10
		Format: U4
	18	**VDI Walker Enable**
		Format: U1

Value	Name
0	Walker Disabled. Use XY generated by Driver.
1	Walker Enabled. Use XY generated by VDIunit.

Programming Notes
When enabled frame size should be aligned to 16x8 in DN only mode and 16x4 in DI enabled mode
When walker is enabled in a GT2 system, the MEDIA_OBJECT commands dispatching work to the VDI must use the Half-Slice Destination Select field to split the work between the two half-slices; the Half-Slice Destination Select must never be set to 00 (either half-slice).

	17:16	**FMD for 2nd field of previous frame**
		Format: U2

DEINTERLACE_SAMPLER_STATE		

Value	Name
0	Deinterlace (not progressive output)
1	Put together with previous field in sequence (1st field of previous frame)
2	Put together with next field in sequence (1st field of current frame)

15:10	**MC Pixel Consistency Threshold**	
	Default Value:	25
	Format:	U6
9:8	**FMD for 1st field of current frame**	
	Format:	U2

Value	Name
0	Deinterlace (not progressive output)
1	Put together with previous field in sequence (2nd field of previous frame)
2	Put together with next field in sequence (2nd field of current frame)

7:4	**SAD Threshold B**	
	Default Value:	10
	Format:	U4
3:0	**SAD Threshold A**	
	Default Value:	5
	Format:	U4

This state definition is used only by the *deinterlace* message. This state is stored as an array of up to 8 elements, each of which contains the dwords described here. The start of each element is spaced 8 dwords apart. The first element of the array is aligned to a 32-byte boundary. The index with range 0-7 that selects which element is being used is multiplied by 2 to determine the **Sampler Index** in the message descriptor.

2.12.3.3.1 Restrictions

1. VDIWalker can be enabled only when frame is aligned to block size of 16x4 if DI is enabled (interlaced) and 16x8 if DN only (Progressive).
2. When VDIWalker Frame Sharing is enabled driver should dispatch same number of Media Objects to both half slice by explicitly programming half slice destination select as 01 and 10 alternately (Note: Dispatch of threads should be in ping pong fashion to have load balance between both Halfslice and better L3 utilization).
3. For VDIWalker disabled mode (when frame size is not aligned to 16x4 or 16x8) it is recommended to have a simplified SW walker. Using Half Slice Destination Select 00 will affect performance significantly.

2.12.3.3.2 Dispatch of Media Object Commands for VDIWalker Enabled

1. Frame Sharing is Disabled:
   a. Program all MO commands to have Half Slice destination select as either “01” or “10”
   b. Y_stride programmed in Sampler State will be ignored

2. Frame Sharing Enabled:
   a. if Frame_height (in blocks) % 2 = 0 (where block height = 4 when DI enabled, 8 when DN only) dispatch MO in ping pong fashion
   b. Y_Stride of 0,1,2,3 is valid and VDIwalker will divide frame into multiple slices based on stride value
   c. if Frame_height (in blocks) % 2 > 0, then dispatch MO in ping pong fashion and all threads for blocks from residual row to be sent to Half Slice0

2.12.3.3.3 Psuedo Code for Media Object Dispatch

```
//Variables
Frame Height in pixels => frame_height
Frame Width in pixels => frame_width
Frame Height in Blocks => fh
Frame Width in Blocks => fw
Block Height in Pixels => block_height = Interlaced? 4 : 8

//Code
fw = frame_width / 16;
fh = frame_height / block_height;
```

2.12.3.3.4 Calculate Residual Blocks

```
If ( fh % (2**stride) ) ≠ 0 {
 Y_Blocks_Remainder = (fh % (2**stride))
 If (Y_Blocks_Remainder > (2**stride) / 2) {

 Y_Blocks_Remainder_HS1 = (2**stride) / 2
 Y_Blocks_Remainder_HS2 = Y_Blocks_Remainder - (2**stride) / 2
 }
 Else {
 Y_Blocks_Remainder_HS1 = Y_Blocks_Remainder
 Y_Blocks_Remainder_HS2 = 0
 }
}
Else {
 Y_Blocks_Remainder_HS1 = 0
 Y_Blocks_Remainder_HS2 = 0
}
```

2.12.3.3.5 Dispatch Media Object

```
total_media_obj_cnt = fw * fh;
reminder_media_obj_cnt_HS1 = fw * Y_Blocks_Remainder_HS1;
reminder_media_obj_cnt_HS2 = fw * Y_Blocks_Remainder_HS2;

ping_pong_media_obj_cnt =
total_media_obj_cnt – (reminder_media_obj_cnt_HS1 + reminder_media_obj_cnt_HS1);
```

```
for ( i = 0; i < ping_pong_media_obj_cnt; i++) {
 if ( i % 2 == 0) {
 dispatch_media_object_hs1;
 }
 else {
 dispatch_media_object_hs2;
 }
}

for ( i = 0; i < reminder_media_obj_cnt_HS1; i++) {
 dispatch_media_object_hs1;
}

for ( i = 0; i < reminder_media_obj_cnt_HS2; i++) {
 dispatch_media_object_hs2;
}
```

2.12.4 SAMPLER_8x8_STATE

SAMPLER_8x8_STATE	
Exists If:	MessageType == 'Sample_8x8'
Default Value:	0x00000000, 0x00000000

The 8x8 coefficients and other state used by the sample_8x8 message are stored as indirect state, pointed to by a field in SAMPLER_STATE. There are four different tables loaded using this structure (0X, 0Y, 1X, and 1Y). Each table is stored as an array of 17 elements, each with either 4 or 8 coefficients.

DWord	Bit	Description	
0	31:24	**Table 0X Filter Coefficient[0,3]**	
		Format:	S1.6 In 2's complement format

SAMPLER_8x8_STATE

Description	Project
Range: [-2.0, +2.0)	

23:16 Table 0X Filter Coefficient[0,2]

Format:	S1.6 In 2's complement format

Range: [-1, +1)

15:8 Table 0X Filter Coefficient[0,1]

Format:	S1.6 In 2's complement format

Range = [-2^{-1}, $+2^{-1}$)

Programming Notes
Must be zero if the format is R10G10B10A2_UNORM or R8G8B8A8_UNORM.

7:0 Table 0X Filter Coefficient[0,0]

Format:	S1.6 In 2's complement format

Range = [-2^{-2}, $+2^{-2}$)

Programming Notes
Must be zero if the format is R10G10B10A2_UNORM or R8G8B8A8_UNORM

1 31:24 Table 0X Filter Coefficient[0,7]

Format:	S1.6 In 2's complement format

Range = [-2^{-2}, $+2^{-2}$)

23:16 Table 0X Filter Coefficient[0,6]

Format:	S1.6 In 2's complement format

Range = [-2^{-1}, $+2^{-1}$)

15:8 Table 0X Filter Coefficient[0,5]

Format:	S1.6 In 2's complement format

Range: [-1, +1)

7:0 Table 0X Filter Coefficient[0,4]

Format:	S1.6 In 2's complement format

SAMPLER_8x8_STATE

		Description / **Project** Range: [-2.0, +2.0)
2..3	31:24	**Table 0Y Filter Coefficient[0,7]** Format: S1.6 In 2's complement format Range = [-2^{-2}, $+2^{-2}$)
	23:16	**Table 0Y Filter Coefficient[0,6]** Format: S1.6 In 2's complement format Range = [-2^{-1}, $+2^{-1}$)
	15:8	**Table 0Y Filter Coefficient[0,5]** Format: S1.6 In 2's complement format Range: [-1, +1)
	7:0	**Table 0Y Filter Coefficient[0,4]** Format: S1.6 In 2's complement format **Description** / **Project** Range: [-2.0, +2.0)
4	31:24	**Table 1X Filter Coefficient[0,3]** Format: S1.6 In 2's complement format Range: [0.0, +2.0)
	23:16	**Table 1X Filter Coefficient[0,2]** Format: S1.6 In 2's complement format Range: [-1, +1)
	15	**Adaptive Filter for all channels** Only to be enabled if 8-tap Adaptive filter mode is on. Else it should be disabled. **Value** / **Name** 1 – Enable adaptive filter on UV/RB channels 0 – Disable adaptive filter on UV/RB channels
	14	**Enable RGB Adaptive for RGB input only :** This should be always set to 0 for YUV input and can be enabled/disabled for RGB input. This should be enabled only if we enable 8-tap adaptive filter for RGB input **Value** / **Name** 1 – Enable the RGB Adaptive filter using the equation (Y=(R+2G+B)>>2)

SAMPLER_8x8_STATE

		0 Disable the RGB Adaptive equation and use G-Ch directly for adaptive filter
	13:0	**Reserved** Format: MBZ
5	31:16	**Reserved** Format: MBZ
	15:8	**Table 1X Filter Coefficient[0,5]** Format: S1.6 In 2's complement format Range: [-1, +1)
	7:0	**Table 1X Filter Coefficient[0,4]** Format: S1.6 In 2's complement format Range: [0.0, +2.0)
6..7	31:16	**Reserved** Format: MBZ
	15:8	**Table 1Y Filter Coefficient[0,5]** Format: S1.6 In 2's complement format Range: [-1, +1)
	7:0	**Table 1Y Filter Coefficient[0,4]** Format: S1.6 In 2's complement format Range: [0.0, +2.0)
8..15	31:0	**Filter Coefficient[1,7:0]**
16..23	31:0	**Filter Coefficient[2,7:0]**
24..31	31:0	**Filter Coefficient[3,7:0]**
32..39	31:0	**Filter Coefficient[4,7:0]**
40..47	31:0	**Filter Coefficient[5,7:0]**
48..55	31:0	**Filter Coefficient[6,7:0]**
56..63	31:0	**Filter Coefficient[7,7:0]**
64..71	31:0	**Filter Coefficient[8,7:0]**
72..79	31:0	**Filter Coefficient[9,7:0]**
80..87	31:0	**Filter Coefficient[10,7:0]**
88..95	31:0	**Filter Coefficient[11,7:0]**
96..103	31:0	**Filter Coefficient[12,7:0]**
104..111	31:0	**Filter Coefficient[13,7:0]**
112..119	31:0	**Filter Coefficient[14,7:0]**
120..127	31:0	**Filter Coefficient[15,7:0]**
128..135	31:0	**Filter Coefficient[16,7:0]**
136	31:24	**Default Sharpness Level** Format: U8

SAMPLER_8x8_STATE

		When adaptive scaling is off, determines the balance between sharp and smooth scalers. Value: 0 — Contribute 1 from the smooth scalar Value: 255 — Contribute 1 from the sharp scalar
	23:16	**Max Derivative 4 Pixels** Format: U8 Used in adaptive filtering to specify the lower boundary of the smooth 8 pixel area.
	15:8	**Max Derivative 8 Pixels** Format: U8 Used in adaptive filtering to specify the lower boundary of the smooth 8 pixel area.
	7	**Reserved** Format: MBZ
	6:4	**Transition Area with 4 Pixels** Format: U3 Used in adaptive filtering to specify the width of the transition area for the 4 pixel calculation.
	3	**Reserved** Format: MBZ
	2:0	**Transition Area with 8 Pixels** Format: U3 Used in adaptive filtering to specify the width of the transition area for the 8 pixel calculation.
137	31:23	**Reserved** Format: MBZ
	22	**Bypass X Adaptive Filtering** Format: Disable When disabled, the X direction will use **Default Sharpness Level** to blend between the smooth and sharp filters rather than the calculated value. Value: 1 — Disable X adaptive filtering Value: 0 — Enable X adaptive filtering
	21	**Bypass Y Adaptive Filtering** Format: Disable When disabled, the Y direction will use **Default Sharpness Level** to blend between the smooth and sharp filters rather than the calculated value. Value: 1 — Disable X adaptive filtering Value: 0 — Enable X adaptive filtering
	20:2	**Reserved** Format: MBZ
	1	**Reserved** Project: Format: MBZ
	0	**Reserved** Project: Format: MBZ

2.12.5 SAMPLER_BORDER_COLOR_STATE

SAMPLER_BORDER_COLOR_STATE

Default Value: 0x00000000, 0x00000000, 0x00000000, 0x00000000

This structure is pointed to by a field in SAMPLER_STATE. The interpretation of the border color depends on the Texture Border Color Mode field in SAMPLER_STATE as follows:In DX9 mode, the border color is 8-bit UNORM format, regardless of the surface format chosen. For surface formats with one or more channels missing (i.e. R5G6R5_UNORM is missing the alpha channel), the value from the border color, if selected, will be used even for the missing channels.In DX10/OGL mode, the format of the border color is R32G32B32A32_FLOAT, regardless of the surface format chosen. For surface formats with one or more channels missing, the value from the border color is not used for the missing channels, resulting in these channels resulting in the overall default value (0 for colors and 1 for alpha) regardless of whether border color is chosen. The surface formats with “L” and “I” have special behavior with respect to the border color. The border color value used for the replicated channels (RGB for “L” formats and RGBA for “I” formats) comes from the red channel of border color. In these cases, the green and blue channels, and also alpha for “I”, of the border color are ignored.The format of this state depends on the Texture Border Color Mode field.

Programming Notes

- DX9 mode is not supported for surfaces with more than 16 bits in any channel, other than 32-bit float formats which are supported.
- The conditions under which this color is used depend on the **Surface Type** – 1D/2D/3D surfaces use the border color when the coordinates extend beyond the surface extent; cube surfaces use the border color for “empty” (disabled) faces.
- The border color itself is accessed through the texture cache hierarchy rather than the state cache hierarchy. Thus, if the border color is changed in memory, the texture cache must be invalidated and the state cache does not need to be invalidated.
- MAPFILTER_MONO: The border color is ignored. Border color is fixed at a value of 0 by hardware.

DWord	Bit	Description
0	31:24	**Border Color Alpha** Format: UNORM8 Texture Border Color Mode = DX9
	23:16	**Border Color Blue** Format: UNORM8 Texture Border Color Mode = DX9
	15:8	**Border Color Green** Format: UNORM8 Texture Border Color Mode = DX9
	31:0	**Border Color Red** Format: IEEE_FP Texture Border Color Mode = DX10/OGL
	7:0	**Border Color Red** Format: UNORM8

SAMPLER_BORDER_COLOR_STATE		
		Texture Border Color Mode = DX9
1	31:0	**Border Color Green** Format: IEEE_FP Texture Border Color Mode = DX10/OGL
2	31:0	**Border Color Blue** Format: IEEE_FP Texture Border Color Mode = DX10/OGL
3	31:0	**Border Color Alpha** Format: IEEE_FP Texture Border Color Mode = DX10/OGL

2.12.5.1 SAMPLER_BORDER_COLOR_STATE

For, if border color is used, all formats must be provided. Hardware will choose the appropriate format based on **Surface Format** and **Texture Border Color Mode**. The values represented by each format should be the same (other than being subject to range-based clamping and precision) to avoid unexpected behavior.

DWord	Bit	Description
0	31:24	**Border Color Alpha** Format = UNORM8
	23:16	**Border Color Blue** Format = UNORM8
	15:8	**Border Color Green** Format = UNORM8
	7:0	**Border Color Red** Format = UNORM8
1	31:0	**Border Color Red** Format = IEEE_FP
2	31:0	**Border Color Green** Format = IEEE_FP
3	31:0	**Border Color Blue** Format = IEEE_FP
4	31:0	**Border Color Alpha**

DWord	Bit	Description
		Format = IEEE_FP
5	31:16	**Border Color Green** Format = FLOAT16
	15:0	**Border Color Red** Format = FLOAT16
6	31:16	**Border Color Alpha** Format = FLOAT16
	15:0	**Border Color Blue** Format = FLOAT16
7	31:16	**Border Color Green** Format = UNORM16
	15:0	**Border Color Red** Format = UNORM16
8	31:16	**Border Color Alpha** Format = UNORM16
	15:0	**Border Color Blue** Format = UNORM16
9	31:16	**Border Color Green** Format = SNORM16
	15:0	**Border Color Red** Format = SNORM16
10	31:16	**Border Color Alpha** Format = SNORM16
	15:0	**Border Color Blue** Format = SNORM16
11	31:24	**Border Color Alpha** Format = SNORM8
	23:16	**Border Color Blue** Format = SNORM8
	15:8	**Border Color Green**

DWord	Bit	Description
		Format = SNORM8
	7:0	**Border Color Red** Format = SNORM8

2.12.6 3DSTATE_CHROMA_KEY

3DSTATE_CHROMA_KEY

Project:	All
Source:	RenderCS
Length Bias:	2

The 3DSTATE_CHROMA_KEY instruction is used to program texture color/chroma-key key values. A table containing four set of values is supported. The ChromaKey Index sampler state variable is used to select which table entry is associated with the map. Texture chromakey functions are enabled and controlled via use of the ChromaKey Enable texture sampler state variable.Texture Color Key (keying on a paletted texture index) is not supported.

DWord	Bit	Description
0	31:29	**Command Type** Default Value: 3h GFXPIPE Format: Opcode
	28:27	**Command SubType** Default Value: 3h GFXPIPE_3D Format: Opcode
	26:24	**3D Command Opcode** Default Value: 1h 3DSTATE Format: Opcode
	23:16	**3D Command Sub Opcode** Default Value: 04h 3DSTATE_CHROMA_KEY Format: Opcode
	15:8	**Reserved** Project: All Format: MBZ
	7:0	**DWord Length** Default Value: 2h Excludes DWord (0,1) Format: =n Total Length - 2
1	31:30	**ChromaKey Table Index** Project: All Format: U2 index Selects which entry in the ChromaKey table is to be loaded
		Value: [0,3] — Name:
	29:0	**Reserved** Project: All Format: MBZ

3DSTATE_CHROMA_KEY		
2	31:0	**ChromaKey Low Value** This field specifies the “low” (minimum) value of the chroma key range. Texel samples are considered “matching the key” if each component of the texel falls within the (inclusive) chroma range.See ChromaKey High Value for further format, programming info.
3	31:0	**ChromaKey High Value** This field specifies the “high” (maximum) value of the chroma key range. Texel samples are considered “matching the key” if each component of the texel falls within the (inclusive) chroma range.

Programming Notes

ChromaKey values are specified using 8-bit channels. When using surface formats with less than 8 bits per channel, the device will expand channels by replicating the required number of MSBs into the LSBs of each channel. Software must account for this conversion when it programs Chromakey Low/High Values (e.g., by performing the same replication).

For channels that do not exist in the actual surface (e.g., Alpha channel for non-ARGB maps), software must explicitly program full range high/low values (High=FFh, Low=0h for formats using unsigned chroma key values, High=7Fh, Low=FFh for formats using sign magnitude chroma key values) in order to effectively remove the comparison of that field from the ChromaKey function.

For channels in SNORM format in the surface format, the value in the high/low value for that channel is interpreted in sign magnitude format. Negative zero value is not supported (use positive zero instead). For channels with mixed UNORM/SNORM formats (i.e. R5G5_SNORM_B6_UNORM), the ChromaKey is programmed as if all channels are SNORM.

YUV ChromaKey will use an interpolated chrominance value from the map for comparison to the chroma key values for those texels without chrominance due to downsampling. The chrominance value used is the average of values to the left and right of the texel in question.

It is UNDEFINED to program any component of the ChromaKey High Value to be less than the corresponding component of ChromaKey Low Value.

Format = interpreted according to associated texel format “class”:

Only the surface formats listed as supported for chroma key in the surface formats table can be used with this feature. Use of any other surface format with chroma key enabled is UNDEFINED.

Surface Format	31:24	23:15	16:8	7:0
ARGB and BC (DXT) formats	A	R	G	B
YCrCb formats	A	Cr	Y	Cb

2.12.7 3DSTATE_SAMPLER_PALETTE_LOAD0

3DSTATE_SAMPLER_PALETTE_LOAD0	
Project:	All
Source:	RenderCS
Length Bias:	2

Description	Project
The 3DSTATE_SAMPLER_PALETTE_LOAD0 instruction is used to load 32-bit values into the first texture palette. The texture palette is used whenever a texture with a paletted format (containing “Px [palette0]”) is referenced by the sampler.	
This instruction is used to load all or a subset of the 256 entries of the first palette. Partial loads always start from the first (index 0) entry.	

DWord	Bit	Description	
0	31:29	**Command Type**	
		Default Value:	3h GFXPIPE
		Format:	Opcode

3DSTATE_SAMPLER_PALETTE_LOAD0

	28:27	**Command SubType** Default Value: 3h GFXPIPE_3D Format: Opcode
	26:24	**3D Command Opcode** Default Value: 1h 3DSTATE Format: Opcode
	23:16	**3D Command Sub Opcode** Default Value: 02h 3DSTATE_SAMPLER_PALETTE_LOAD0 Format: Opcode
	15:8	**Reserved** Project: All Format: MBZ
	7:0	**DWord Length** Default Value: 0h Excludes DWord (0,1) Format: =n Total Length - 2
1..n	31:24	**Palette Alpha[0:N-1]** Project: All Format: U8 Alpha channel loaded into the Nth entry of the texture color palette.
	23:16	**Palette Red[0:N-1]** Project: All Format: U8 Alpha channel loaded into the Nth entry of the texture color palette.
	15:8	**Palette Green[0:N-1]** Project: All Format: U8 Alpha channel loaded into the Nth entry of the texture color palette.
	7:0	**Palette Blue[0:N-1]** Project: All Format: U8 Alpha channel loaded into the Nth entry of the texture color palette.

2.12.8 3DSTATE_MONOFILTER_SIZE

3DSTATE_MONOFILTER_SIZE

Source: RenderCS

Length Bias: 2

This state specifies the size of the filter which is used when filtering in MAPFILTER_MONO mode.

DWord	Bit	Description
0	31:29	**Command Type** Default Value: 3h GFXPIPE Format: OpCode
	28:27	**Command SubType** Default Value: 3h GFXPIPE_3D Format: OpCode
	26:24	**3D Command Opcode** Default Value: 1h 3DSTATE_NONPIPELINED Format: OpCode
	23:16	**3D Command Sub Opcode** Default Value: 11h 3DSTATE_MONOFILTER_SIZE Format: OpCode
	15:8	**Reserved** Project: All Format: MBZ
	7:0	**DWord Length** Default Value: 0h Excludes DWord (0,1) Project: All Format: =n Total Length - 2
1	31:6	**Reserved** Project: All Format: MBZ
	5:3	**Monochrome Filter Width** Project: All Format: U3 This field specifies the width of the monochrome filter. It is ignored if the monochrome filter is not enabled. Value: [1,7] — Name:
	2:0	**Monochrome Filter Height** Project: All Format: U3 This field specifies the height of the monochrome filter. It is ignored if the monochrome filter is not enabled. Value: [1,7] — Name:

2.13 Messages

Restrictions:

- Use of any message to the Sampling Engine function with the **End of Thread** bit set in the message descriptor is not allowed.

2.13.1 Initiating Message

Execution Mask

SIMD16. The 16-bit execution mask forms the valid pixel signals. This determines which pixels are sampled and results returned to the GRF registers. Samples for invalid pixels are not overwritten in the GRF. However, if LOD needs to be computed for a subspan based on the message type and MIP filter mode and at least one pixel in the subspan being valid, the sampling engine assumes that the parameters for the upper left, upper right, and lower left pixels in the subspan are valid regardless of the execution mask, as these are needed for the LOD computation.

SIMD8. The lower 8 bits of the execution mask forms the valid pixel signals. If LOD needs to be computed based on MIP filter mode and at least one pixel in the subspan being valid, the sampling engine assumes that the parameters for the upper left, upper right, and lower left pixels in the subspan are valid regardless of the execution mask, since these are needed for the LOD computation.

SIMD4x2. The lower 8 bits of the execution mask is interpreted in groups of four. If any of the high 4 bits are asserted, that sample is valid. If any of the low 4 bits are asserted, that sample is valid. The **Write Channel Mask** rather than the execution mask determines which channels are written back to the GRF.

SIMD32. The execution mask is ignored, all pixels are considered valid and all channels are returned regardless of the execution mask.

2.13.1.1 Message Descriptor

Bit	Description
19	**Header Present:** Specifies whether the message includes a header phase. If the header is not present (this field is zero), all of the fields normally contained in the header are assumed to be 0. Format = Enable
18:17	**SIMD Mode:** Specifies the SIMD mode of the message being sent. Format = U2 0 = SIMD4x2 1 = SIMD8 2 = SIMD16 3 = SIMD32/64
16:12	**Message Type:** Specifies the type of message being sent. Format = U5 Refer to the table in section *Payload Parameter Definition* for encoding details.
11:8	**Sampler Index:** Specifies the index into the sampler state table. Ignored for “ld”, “resinfo”, “sampleinfo” and

Bit	Description
	"cache_flush"type messages. Format = U4 Range = [0,15] **Programming Notes:** • for the deinterlace message, this field must be a multiple of 2 (even) • for the sample_8x8 message, this field must be a multiple of 4
7:0	**Binding Table Index:** Specifies the index into the binding table. Ignored for "cache_flush" type messages. Format = U8 Range = [0,255]

2.13.1.2 Message Header

The message header for the sampling engine is the same regardless of the message type. If the header is not present (**only**), behavior is as if the message was sent with all fields in the header set to zero (write channel masks are all enabled and offsets are zero). When Response length is 0 for sample_8x8 message then the data from sampler is directly written out to memory using media write message.

DWord	Bit	Description
M0.7	31:0	
M0.6	31:0	
M0.5	31:5	Reserved
	4:0	Reserved
M0.4	31:0	Reserved
M0.3	31:5	**Sampler State Pointer:** Specifies the 32-byte aligned pointer to the sampler state table. This field is ignored for "ld" and "resinfo" message types. This pointer is relative to the **General State Base Address**. Format = GeneralStateOffset[31:5] Ignored
	4:0	Ignored
M0.2	31:24	**Ignored**
	23	Reserved
	21:20	Ignore
	19:18	**SIMD32/64 Output Format Control** The contents of this field are ignored. The "16 bit Full" mode is always selected.
	17	
	17:16	**Gather4 Source Channel Select:** Selects the source channel to be sampled in the gather4* messages. Ignored for other message types. 0: Red channel 1: Green channel 2: Blue channel 3: Alpha channel

DWord	Bit	Description
		Programming Note: • For gather4*_c messages, this field must be set to 0 (Red channel).
	16	**Force LOD to Zero:** If this bit is enabled, the calculated LOD is replaced with zero. The LOD is replaced just before entering the pseudocode in section *LOD Computation Pseudocode*, therefore the LOD is still subject to bias, overriding by sample_l delivered LOD, and clamping. Format = Enable Ignored
	15	**Alpha Write Channel Mask:** Enables the alpha channel to be written back to the originating thread. 0: Alpha channel will be written back 1: Alpha channel will not be written back **Programming Notes:** • a message with all four channels masked is not allowed. • this field is ignored for the sample_unorm*. The write channel mask is generated from the message type itself. • this field is ignored for the deinterlace message. • this field must be set to zero for sample_8x8 in VSA mode.
	14	**Blue Write Channel Mask:** See Alpha Write Channel Mask
	13	**Green Write Channel Mask:** See Alpha Write Channel Mask
	12	**Red Write Channel Mask:** See Alpha Write Channel Mask
	11:8	**U Offset:** the u offset from the _aoffimmi modifier on the “sample” or “ld” instruction in DX10. Must be zero if the **Surface Type** is SURFTYPE_CUBE or SURFTYPE_BUFFER. Must be set to zero if _aoffimmi is not specified. Format is S3 2’s complement. **Programming Note:** • this field is ignored for the sample_unorm*, sample_8x8, and deinterlace messages • this field is ignored if the “offu” parameter is included in the gather4* messages
	7:4	**V Offset:** the v offset from the _aoffimmi modifier on the “sample” or “ld” instruction in DX10. Must be zero if the **Surface Type** is SURFTYPE_CUBE or SURFTYPE_BUFFER. Must be set to zero if _aoffimmi is not specified. Format is S3 2’s complement. **Programming Note:** • this field is ignored for the sample_unorm*, sample_8x8, and deinterlace messages • this field is ignored if the “offu” parameter is included in the gather4* messages
	3:0	**R Offset:** the r offset from the _aoffimmi modifier on the “sample” or “ld” instruction in DX10. Must be zero if the **Surface Type** is SURFTYPE_CUBE or SURFTYPE_BUFFER. Must be set to zero if _aoffimmi is not specified. Format is S3 2’s complement. **Programming Note:** • this field is ignored for the sample_unorm*, sample_8x8, and deinterlace messages
M0.1	31:0	Ignored

DWord	Bit	Description
M0.0	31:0	Ignored

2.13.1.3 Payload Parameter Definition

The table below shows all of the messages supported by the sampling engine. The message type field in the message descriptor in combination with the message length determines which message is being sent. The table defines all of the *parameters* sent for each message type. The position of the parameters in the payload is given in the section following corresponding to the *SIMD mode* given in the table. The instruction column indicates the DX10 shader instructions expected to be translated to each message type.

All parameters are of type IEEE_Float, except those in the ld and resinfo instruction message types, which are of type S31. Any parameter indicated with a blank entry in the table is unused. A message register containing only unused parameters not included as part of the message. The response lengths given below assume all channels are unmasked. SIMD16 messages with masked channels will have reduced response length.

2.13.1.3.1 Payload Parameter Definition

The table below shows all of the message types supported by the sampling engine. The **Message Type** field in the message descriptor determines which message is being sent. The **SIMD Mode** field determines the number of instances (i.e. pixels) and the formatting of the initiating and writeback messages. The **Header Present** field determines whether a header is delivered as the first phase of the message or the default header from R0 of the thread's dispatch is used. The **Message Length** field is used to vary the number of parameters sent with each message. Higher-numbered parameters are optional, and default to a value of 0 if not sent but needed for the surface being sampled.

The message lengths are computed as follows, where "N" is the number of parameters ("N" is rounded up to the next multiple of 4 for SIMD4x2), and "H" is 1 if the header is present, 0 otherwise. The maximum message length allowed to the sampler is 11.

SIMD Mode	Message Length
SIMD4x2	H + (N/4)
SIMD8*	H + N
SIMD16	H + (2*N)

The response lengths are computed as follows:

SIMD Mode		Response Length
SIMD4x2		1
SIMD8	sample+killpix	5
	all other message types	4
SIMD16		8 *

* For SIMD16, phases in the response length are reduced by 2 for each channel that is masked.

SIMD16 messages with six or more parameters exceed the maximum message length allowed, in which case they are not supported. This includes some forms of sample_b_c, sample_l_c, and gather4_po_c message types. Note that even for these messages, if 5 or fewer parameters are included in the message, the SIMD16 form of the message is allowed. SIMD16 forms of sample_d and sample_d_c are not allowed, regardless of the number of parameters sent.

SIMD8 and SIMD16 Messages:

Message Type	mnemonic	parameters										
		0	1	2	3	4	5	6	7	8	9	10
00000	sample	u	v	r	ai	mlod*						
00001	sample_b	bias	u	v	r	ai	mlod*					
00010	sample_l	lod	u	v	r	ai						
00011	sample_c	ref	u	v	r	ai	mlod*					
00100	sample_d	u	dudx	dudy	v	dvdx	dvdy	r	drdx	drdy	ai	mlod*
00101	sample_b_c	ref	bias	u	v	r	ai					
00110	sample_l_c	ref	lod	u	v	r	ai					
00111	ld	u	lod	v	r							
00111	ld †	u	v	lod	r							
01000	gather4	u	v	r	ai							
01001	LOD	u	v	r	ai							
01010	resinfo	lod										
01011	sampleinfo											
01011	sampleinfo †	x										
01100	sample+killpix	u	v	r								
10000	gather4_c	ref	u	v	r	ai						
10001	gather4_po	u	v	offu	offv	r						
10010	gather4_po_c	ref	u	v	offu	offv	r					
10100	sample_d_c	ref	u	dudx	dudy	v	dvdx	dvdy	r	drdx	drdy	ai
11100	ld2dms_w	si	mcsl	mcsh	u	v	r	lod *				
10110	sample_min	u	v									
10111	sample_max	u	v									
11101	ld_mcs	u	v	r	lod *							
11110	ld2dms	si	mcs	u	v	r	lod *					
11111	ld2dss	ssi	u	v	r	lod *						
11000	sample_lz	u	v	r	ai							
11001	sample_c_lz	ref	u	v	r	ai						
11010	ld_lz	u	v	r								

SIMD4x2 Messages:

Message Type	mnemonic	parameters										
		0	1	2	3	4	5	6	7	8	9	10
00010	sample_l	u	v	r	ai	lod						
00100	sample_d	u	v	r	ai	dudx	dudy	dvdx	dvdy	drdx	drdy	mlod*
00110	sample_l_c	u	v	r	ai	ref	lod					
00111	ld	u	v	r	lod							
01000	gather4	u	v	r	ai							
01010	resinfo	lod										
01011	sampleinfo											
10000	gather4_c	u	v	r	ai	ref						
10001	gather4_po	u	v	r	ai	offu	offv					
10010	gather4_po_c	u	v	r	ref	offu	offv					
10100	sample_d_c	u	v	r	ai	dudx	dudy	dvdx	dvdy	drdx	drdy	ref

11100	ld2dms_w	u	v	r	lod *	si	mcsl	mcsh				
11101	ld_mcs	u	v	r	lod *							
11110	ld2dms	u	v	r	lod *	si	mcs					

SIMD32/SIMD64 Messages:

Message Type	mnemonic	Payload Layout	Message Length	Response Length
00000	sample_unorm	Pixel Shader	H + 1	8 **
00010	sample_unorm+killpix	Pixel Shader	H + 1	9 **
00011	sample_8x8	Pixel Shader	H + 1	16 *
01000	deinterlace	Pixel Shader	H + 1	†
01100	sample_unorm	Media	H + 1	8 **
01010	sample_unorm+killpix	Media	H + 1	9 **
01011	sample_8x8	Media	H + 1	16 *
11111	cache_flush	no payload	1	1

* For sample_8x8, phases in the response length are reduced by 4 for each channel that is masked.

** For sample_unorm, phases in the response length are reduced by 2 for each channel that is masked.

† For deinterlace, response length depending on certain state fields. Refer to writeback message definition for details.

2.13.1.4 Message Types

The behavior of each message type is as follows:

Message Type	Description
sample	The surface is sampled using the indicated sampler state. LOD is computed using gradients between adjacent pixels. One, two, or three parameters may be specified depending on how many coordinate dimensions the indicated surface type uses. Extra parameters specified are ignored. Missing parameters are defaulted to 0. **Programming Notes:** • The Surface Type of the associated surface must be SURFTYPE1D, SURFTYPE_2D, SURFTYPE_3D, or SURFTYPE_CUBE. • The Surface Format of the associated surface cannot be MONO8. • If the Surface Format of the associated surface is UINT or SINT, the Surface Type cannot be SURFTYPE_3D or SURFTYPE_CUBE and Address Control Mode cannot be CLAMP_BORDER or HALF_BORDER. • sample is not supported in SIMD4x2 mode. • **:Number of Multisamples** on the associated surface must be MULTISAMPLECOUNT_1.
sample+killpix	The surface is sampled as in the sample message type. An additional register is returned after the sample results which contains the kill pixel mask. This message type is required to allow the result of a chroma key enabled sampler in KEYFILTER_KILL_ON_ANY_MATCH mode to affect the final pixel mask. **Programming Notes:** • The Surface Type of the associated surface must be SURFTYPE1D, SURFTYPE_2D, SURFTYPE_3D, or SURFTYPE_CUBE. • The Surface Format of the associated surface cannot be MONO8. • If the Surface Format of the associated surface is UINT or SINT, the Surface Type cannot be SURFTYPE_3D or SURFTYPE_CUBE and Address Control Mode cannot be

Message Type	Description
	CLAMP_BORDER or HALF_BORDER. • sample+killpix is supported only in SIMD8 mode. • **Number of Multisamples** on the associated surface must be MULTISAMPLECOUNT_1.
sample_b	The surface is sampled using the indicated sampler state. LOD is computed using gradients between adjacent pixels, then the value in the parameter is added to the LOD for each pixel. The LOD bias delivered in the **bias** parameter is restricted to a range of [-16.0, +16.0). Values outside this range produce undefined results. ***Programming Notes:*** • The *Surface Type* of the associated surface must be SURFTYPE1D, SURFTYPE_2D, SURFTYPE_3D, or SURFTYPE_CUBE. • The **Surface Format** of the associated surface cannot be MONO8 • If the Surface Format of the associated surface is UINT or SINT, the Surface Type cannot be SURFTYPE_3D or SURFTYPE_CUBE and Address Control Mode cannot be CLAMP_BORDER or HALF_BORDER • **sample_b** is not supported in SIMD4x2 mode. • **Number of Multisamples** on the associated surface must be MULTISAMPLECOUNT_1.
sample_l sample_lz	The surface is sampled using the indicated sampler state. LOD is not computed, but instead is taken from the **lod** parameter. ***Programming Notes:*** • The *Surface Type* of the associated surface must be SURFTYPE1D, SURFTYPE_2D, SURFTYPE_3D, or SURFTYPE_CUBE. • If the Surface Format of the associated surface is UINT or SINT, the Surface Type cannot be SURFTYPE_3D or SURFTYPE_CUBE and Address Control Mode cannot be CLAMP_BORDER or HALF_BORDER. • **Number of Multisamples** on the associated surface must be MULTISAMPLECOUNT_1.
sample_c sample_c_lz	The surface is sampled using the indicated sampler state. All four coordinates must be specified, however v and r may not be used depending on the indicated surface type. The ai parameter indicates the array index for a cube surface.The ref parameter specifies the reference value that is compared against the red channel of the sampled surface, and the texel is replaced with either white or black depending on the result of the comparison. The WGF sample_c_lz instruction is implemented by issuing the sample_c message with Force LOD to Zero enabled in the message header or by issuing the sample_l_c message with the LOD parameter set to zero. **Programming Notes:** • The **Surface Type** of the associated surface must be SURFTYPE1D, SURFTYPE_2D, or SURFTYPE_CUBE. • The **Surface Format** of the associated surface must be indicated as supporting shadow mapping as indicated in the surface format table. • With *sample_c*, MIPFILTER_LINEAR, MAPFILTER_LINEAR, MAPFILTER_ANISOTROPIC are allowed even for surface formats that are listed as not supporting filtering in the surface formats table. • Use of the SIMD4x2 form of *sample_c* without **Force LOD to Zero** enabled in the message header is not allowed, as it is not possible for the hardware to compute LOD for

<table>
<tr><th>Message Type</th><th>Description</th></tr>
<tr><td></td><td>SIMD4x2 messages. For, sample_c is not supported in SIMD4x2 mode.
• Use of sample_c with DX9 Texture Border Color Mode and either of the following is undefined:
  • any applicable Address Control Mode (depending on Surface Type) is set to TEXCOORDMODE_CLAMP_BORDER or TEXCOORDMODE_HALF_BORDER
  • Surface Type is SURFTYPE_CUBE and any Cube Face Enable is disabled
• Use of sample_c with SURFTYPE_CUBE surfaces is undefined with the following surface formats: I24X8_UNORM, L24X8_UNORM, A24X8_UNORM, I32_FLOAT, L32_FLOAT, A32_FLOAT.
• Number of Multisamples on the associated surface must be MULTISAMPLECOUNT_1.</td></tr>
<tr><td>sample_b_c</td><td>This is a combination of sample_b and sample_c. Both the LOD bias and reference values are delivered. All restrictions applying to both sample_b and sample_c must be honored.</td></tr>
<tr><td>sample_l_c</td><td>This is a combination of sample_l and sample_c. Both the LOD and reference values are delivered. All restrictions applying to both sample_l and sample_c must be honored. However, unlike sample_c, sample_l_c is allowed as a SIMD4x2 message.

Programming Notes:</td></tr>
<tr><td>sample_g
sample_d</td><td>The surface is sampled using the indicated sampler state. LOD is computed using the gradients present in the message. The r coordinate and its gradients are required only for surface types that use the third coordinate. Usage of this message type on cube surfaces assumes that the u, v, and gradients have already been transformed onto the appropriate face, but still in [-1,+1] range. The r coordinate contains the faceid, and the r gradients are ignored by hardware.
Programming Notes:
• The Surface Type of the associated surface must be SURFTYPE1D, SURFTYPE_2D, SURFTYPE_3D, or SURFTYPE_CUBE.
• The Surface Format of the associated surface cannot be MONO8.
• If the Surface Format of the associated surface is UINT or SINT, the Surface Type cannot be SURFTYPE_3D or SURFTYPE_CUBE and Address Control Mode cannot be CLAMP_BORDER or HALF_BORDER.
• Number of Multisamples on the associated surface must be MULTISAMPLECOUNT_1.</td></tr>
<tr><td>sample_g_c
sample_d_c</td><td>This is a combination of sample_g and sample_c. Both the gradients for calculating LOD and reference values are delivered. All restrictions applying to both sample_g and sample_c must be honored. However, unlike sample_c, sample_g_c is allowed as a SIMD4x2 message.</td></tr>
<tr><td>resinfo</td><td>The surface indicated in the surface state is not sampled. Instead, the width, height, depth, and MIP count of the surface are returned as indicated in the table below. The format of the returned data is UINT32 for. The width, height, and depth may be shifted right, per pixel, by the LOD value provided in the lod parameter to give the dimensions of the specified mip level. The lod parameter is an unsigned 32-bit integer in this mode (note that sending a signed 32-bit integer always has the same effect, as negative values are out-of-range when interpreted as unsigned integers). The Sampler State Pointer and Sampler Index are ignored.
<table>
<tr><td>surface type</td><td>red</td><td>green</td><td>blue</td><td>alpha</td></tr>
</table></td></tr>
</table>

Message Type	Description				
	SURFTYPE1D	(Width+1)>>LOD	Surface Array? Depth+1 : 0	0	MIPCount
	SURFTYPE_2D	(Width+1)>>LOD	(Height+1)>>LOD	: Surface Array? Depth+1 : 0	MIPCount
	SURFTYPE_3D	(Width+1)>>LOD	(Height+1)>>LOD	(Depth+1)>>LOD	MIPCount
	SURFTYPE_CUBE	(Width+1)>>LOD	(Height+1)>>LOD	Depth==0 ? 0 : Depth+1 : Surface Array ? Depth+1 : 0	MIPCount
	SURFTYPE_BUFFER SURFTYPE_STRBUF	undefined Buffer size (from combined Depth/Height/Width)	undefined	undefined	undefined
	SURFTYPE_NULL	0	0	0	0
ld ld2dms ld2dms_w ld_mcs ld2dss ld_lz	The surface is sampled using a default sampler state, indicated below. The *lod* parameter contains the LOD of the mip map to be sampled. If the message doesn't include an lod parameter, the message samples from LOD 0. The parameter *si* contains the sample index, which is clamped to the number of samples on the surface (supported by some messages on only). The v and r channel may be ignored depending on the indicated surface type. All incoming values are unsigned 32-bit integers in this mode. The u, v, and r parameters contain integer texel addresses on the LOD indicated in the parameter. The Sampler State Pointer and Sampler Index are ignored. For these message types, the sampler state is defaulted as follows: ***•min, mag, and mip filter modes are "nearest"*** ***•all address control modes are "zero" (a special mode in which any texel off the map or outside the MIP range of the surface has a value of zero in all channels, except for surface formats without an alpha channel, which will return a value of one in the alpha channel)*** Errata:Address offset needs to be zero for ld2dms/ld2dss messages The mcs parameter in the ld2dms message defines the multisample control data and is used only to sample from a multisampled surface. The ld_mcs message uses the MCS Base Address and MCS Surface Pitch fields in SURFACE_STATE to determine the base address and pitch of the surface. Surface Format is overridden to R8_UINT if Number of Multisamples is 4, or R32_UINT if Number of Multisamples is 8. This message cannot be used on a non-multisampled surface. Otherwise, ld_mcs behaves like the ld message. If ld_mcs is issued on a surface with MCS disabled, this message returns zeros in all channels. The ssi parameter in the ld2dss message defines the sample slice that will be sampled from. Refer to the multisample storage format in the GPU Overview volume for more details.				

<table>
<tr><th>Message Type</th><th>Description</th></tr>
<tr><td></td><td>Programming Notes:
The Surface Type of the associated surface must be SURFTYPE1D, SURFTYPE_2D, SURFTYPE_3D, or SURFTYPE_BUFFER for the ld message.
The Surface Type of the associated surface must be SURFTYPE_2D for the ld_mcs , ld2dms , and ld2dss messages.
The Surface Format of the associated surface cannot be MONO8.
Number of Multisamples on the associated surface must be MULTISAMPLECOUNT_1 for the ld message type.
Errata: Surface formats R32G32B32X32_FLOAT, X32_TYPELESS_G8X24_UINT, R16G16B16X16_UNORM, R16G16B16X16_FLOAT, X24_TYPELESS_G8_UINT, L24X8_UNORM, L32_FLOAT, B8G8R8X8_UNORM, B8G8R8X8_UNORM_SRGB, R8G8B8X8_UNORM, R8G8B8X8_UNORM_SRGB, B10G10R10X2_UNORM, B5G6R5_UNORM, B5G6R5_UNORM_SRGB, L16_UNORM, R5G5_SNORM_B6_UNORM, L8_UNORM, L8_UNORM_SRGB, R1_UNORM, BC4_UNORM (DXT4/5) will return zero in the alpha channel, for out of bound case.</td></tr>
<tr><td>sampleinfo</td><td>only: The surface indicated in the surface state is not sampled. Instead, the number of samples (UINT32) and the sample position palette index (UINT32) for the surface are returned in the red and alpha channels respectively as UINT32 values. The sample position palette index returned in alpha is incremented by one from its value in the surface state. The Sampler State Pointer and Sampler Index are ignored.
Programming Notes:
The Surface Type of the associated surface must be SURFTYPE_2D or SURFTYPE_NULL</td></tr>
<tr><td>LOD</td><td>only: The surface indicated in the surface state is not sampled. Instead, LOD is computed as if the surface will be sampled, using the indicated sampler state, and the clamped and unclamped LOD values are returned in the red and green channels, respectively, in FLOAT32 format. The blue and alpha channels are undefined, and can be masked to avoid returning them. LOD is computed using gradients between adjacent pixels. Three parameters are always specified, with extra parameters not needed for the surface being ignored.

Programming Notes:
The Surface Type of the associated surface must be SURFTYPE1D, SURFTYPE_2D, SURFTYPE_3D, or SURFTYPE_CUBE.
The Surface Format of the associated surface cannot be MONO8
The Surface Format of the associated surface cannot be any UINT or SINT format.
LOD is not supported in SIMD4x2 mode.
Number of Multisamples on the associated surface must be MULTISAMPLECOUNT_1.</td></tr>
<tr><td>gather4
gather4_po
(load4)</td><td>The surface is sampled using bilinear filtering, regardless of the filtering mode specified in the sampler state. For SURFTYPE_2D LOD is forced to zero before sampling. The samples are not filtered, but instead the four samples are returned directly in the sample’s corresponding four channels as follows:
<table><tr><td>upper left sample = alpha channel</td><td>upper right sample = blue channel</td></tr><tr><td>lower left sample = red channel</td><td>lower right sample = green channel</td></tr></table>Two or three parameters may be specified depending on how many coordinate dimensions the</td></tr>
</table>

Message Type	Description

	indicated surface type uses. Extra parameters specified are ignored. Missing parameters default to 0.
	
	The gather4_po message has offu and offv parameters, which contain texel-space offsets that override the U/V Offset fields in the message header. Unlike the message header fields however, these offsets have a wider range [-32,+31], and can differ per pixel or sample. The format of the data is 32-bit 2's complement signed integer, but hardware only interprets the least significant 6 bits of each value, treating it as a 6-bit 2's complement signed integer.
	
	Programming Notes:
	• The Surface Type of the associated surface must be SURFTYPE_2D or SURFTYPE_CUBE. If the message type is gather4_po, only SURFTYPE_2D is allowed.
	• The Surface Format of the associated surface cannot be MONO8
	• The **Surface Format** of the associated surface cannot be any UINT or SINT format.
	• The channel selected is determined by the Gather4 Source Channel Select field in the message header.
	• Mip Mode Filter must be set to MIPFILTER_NONE
	• **Number of Multisamples** on the associated surface must be MULTISAMPLECOUNT_1.
	• Use of gather4 or gather4_po with DX9 Border Color Mode and either of the following is underfined:
	  ○ any applicable Address Control Mode (depending on Surface Type) is set to TEXCOORDMODE_CLAMP_BORDER or TEXCOORDMODE_HALF_BORDER
	  ○ Surface Type is SURFTYPE_CUBE and any Cube Face Enable is disabled

gather4_c
gather4_po_c
	only: The surface is sampled using bilinear filtering, regardless of the filtering mode specified in the sampler state. For SURFTYPE_2D LOD is forced to zero before sampling. The samples are not filtered, but instead the four samples are returned, after being compared with the ref paramater as in the sample_c message. Each texel is replaced with either white or block depending on the result of the comparison. The four samples are returned in the sample's corresponding four channels in the same mapping as the gather4 message. The offu and offv parameters in the gather4_po_c message cause offset override behavior as described in the gather4 message.
	Programming Notes:
	• The Surface Type of the associated surface must be SURFTYPE_2D or SURFTYPE_CUBE. If the message type is gather4_po_c, only SURFTYPE_2D is allowed.
	• The Surface Format of the associated surface must be one of the following: R32_FLOAT_X8X24_TYPELESS, R32_FLOAT, R24_UNORM_X8_TYPELESS, R16_UNORM.
	• The channel selected is determined by the Gather4 Source Channel Select field in the message header.
	• Mip Mode Filter must be set to MIPFILTER_NONE
	• Use of gather4_c or gather4_po_c with DX9 Border Color Mode and either of the following is underfined:
	  ○ Surface Type is SURFTYPE_CUBE and any Cube Face Enable is disabled
	  ○ any applicable Address Control Mode (depending on Surface Type) is set to

Message Type	Description
	TEXCOORDMODE_CLAMP_BORDER or TEXCOORDMODE_HALF_BORDER • **Number of Multisamples** on the associated surface must be MULTISAMPLECOUNT_1.

sample_unorm	only: The surface is sampled using the indicated sampler state. 32 contiguous pixels in a 8-wide by 4-high arrangement are sampled. The U and V addresses for the upper left pixel is delivered in this message along with a Delta U and Delta V parameter. Given a pixel at (x,y) relative to the upper left pixel (where (0,0) is the upper left pixel), the U and V for that pixel are computed as follows:
	U(x,y) = U(0,0) + DeltaU * x
	V(x,y) = V(0,0) + DeltaV * y
	Programming Notes:
	• The Surface Type of the associated surface must be SURFTYPE_2D
	• The Surface Format of the associated surface must be UNORM with <= 8 bits per channel
	• The MIP Count, Depth, Surface Min LOD, and Min Array Element of the associated surface must be 0
	• The Min and Mag Mode Filter must be MAPFILTER_NEAREST or MAPFILTER_LINEAR
	• The Mip Mode Filter must be MIPFILTER_NONE
	• The TCX and TCY Address Control Mode cannot be
	`TEXCOORDMODE_CLAMP_BORDER`
	`TEXCOORDMODE_HALF_BORDER`
	`TEXCOORDMODE_MIRROR`
	`TEXCOORDMODE_MIRROR_ONCE`
	`TEXCOORDMODE_WRAP`
	• DeltaU * Width of the associated surface must be less than or equal to 3.0
	• DeltaV * Height of the associated surface must be less than or equal to 3.0
	• **Number of Multisamples** on the associated surface must be MULTISAMPLECOUNT_1.

sample_unorm_RG +killpix	
sample_unorm +killpix	only: This message is identical to the sample_unorm message except it returns a kill pixel mask in addition to the selected channels in the writeback message. This message type is required to allow the result of a chroma key enabled sampler in KEYFILTER_KILL_ON_ANY_MATCH mode to affect the final pixel mask. All restrictions of the sample_unorm message apply to this message also.
sample_8x8	*only:*The surface is sampled using an optional 8x8 filter followed by an optional image enhancement filter, using state defined in SAMPLER_STATE and SAMPLER_8x8_STATE. The input consists of 64 contiguous pixels in an 16-wide by 4-high arrangement. The address control mode behaves as clamp mode. The U and V addresses for the upper left pixel are delivered in this message along with a Delta U and Delta V parameter. Given a pixel at (x,y) relative to the upper left pixel (where (0,0) is the upper left pixel), the U and V for that pixel are

Message Type	Description
	computed as follows: U(x,y) = U(0,0) + DeltaU * x + U_2nd_Derivative * x * (x - 1)/2 V(x,y) = V(0,0) + DeltaV * y + V_2nd_Derivative * y * (y - 1)/2 **Programming Notes:** • The *Surface Type* of the associated surface must be SURFTYPE_2D • The *Surface Format* of the associated surface must be UNORM with <= 10 bits per channel • *DeltaV * Height* of the associated surface must be less than 16.0 • *Map Width* must be >= 4 • :Errata IEF_OFF YUV |U_2nd_Derivative| < 0.02/MapWidth • :Errata IEF_OFF RGB |U_2nd_Derivative| < 0.05/MapWidth • :Errata for IEF_OFF set DeltaU = DeltaU + 2x U_2nd_Derivative
deinterlace	to The surface is deinterlaced and/or denoised, using state defined in SAMPLER_STATE. The U and V addresses for the upper left pixel are delivered in this message. **Programming Notes:**

Programming Notes:

- For surfaces of type SURFTYPE_CUBE, the sampling engine requires u, v, and r parameters that have already been divided by the absolute value of the parameter (u, v, or r) with the largest absolute value.

2.13.1.5 Parameter Types

sample*, LOD, and gather4 messages

For all of the sample*, LOD, and gather4 message types, all parameters are 32-bit floating point, except the 'mcs', 'offu', and 'offv' parameters. Usage of the u, v, and r parameters is as follows based on **Surface Type**. Normalized values range from [0,1] across the surface, with values outside the surface behaving as specified by the **Address Control Mode** in that dimension. Unnormalized values range from [0,n-1] across the surface, where n is the size of the surface in that dimension, with values outside the surface being clamped to the surface.

Surface Type	*u*	*v*	*r*	*ai*
SURFTYPE1D	***normalized 'x' coordinate***	***unnormalized array index***	***ignored***	***ignored***
SURFTYPE_2D	***normalized 'x' coordinate***	***normalized 'y' coordinate***	***unnormalized array index***	***ignored***
SURFTYPE_3D	***normalized 'x' coordinate***	***normalized 'y' coordinate***	***normalized 'z' coordinate***	***ignored***
SURFTYPE_CUBE	***normalized 'x' coordinate***	***normalized 'y' coordinate***	***normalized 'z' coordinate***	***unnormalized array index***

mcs parameter

The ‘mcs’ parameter delivers the multisample control data. The format of this parameter is always a 32-bit unsigned integer. Refer to the section titled “Multisampled Surface Behavior” for details on this parameter.

Ld* messages

For the ld message types, all parameters are 32-bit signed integers, except the ‘mcs’ parameter. Usage of the u, v, and r parameters is as follows based on **Surface Type**. Unnormalized values range from [0,n-1] across the surface, where n is the size of the surface in that dimension. Input of any value outside of the range returns zero.

Surface Type	*u*	*v*	*r*
SURFTYPE1D	***unnormalized ‘x’ coordinate***	***unnormalized array index***	***ignored***
SURFTYPE_2D	***unnormalized ‘x’ coordinate***	***unnormalized ‘y’ coordinate***	***unnormalized array index***
SURFTYPE_3D	***unnormalized ‘x’ coordinate***	***unnormalized ‘y’ coordinate***	***unnormalized ‘z’ coordinate***
SURFTYPE_BUFFER	***unnormalized ‘x’ coordinate***	***ignored***	***ignored***

2.13.1.6 SIMD16 Payload

The payload of a SIMD16 message provides addresses for the sampling engine to process 16 entities (examples of an entity are vertex and pixel). The number of parameters required to sample the surface depends on the state that the sampler/surface is in. Each parameter takes two message registers, with 8 entities, each a 32-bit floating point value, being placed in each register. Each parameter always takes a consistent position in the input payload. The length field can be used to send a shorter message, but intermediate parameters cannot be skipped as there is no way to signal this. For example, a 2D map using “sample_b” needs only u, v, and bias, but must send the r parameter as well.

DWord	Bit	Description
M1.7	31:0	**Subspan 1, Pixel 3 (lower right) Parameter 0** Specifies the value of the pixel’s parameter 0. The actual parameter that maps to parameter 0 is given in the table in section *Payload Parameter Definition*. Format = IEEE Float for all sample* message types, U32 for ld and resinfo message types.
M1.6	31:0	**Subspan 1, Pixel 2 (lower left) Parameter 0**
M1.5	31:0	**Subspan 1, Pixel 1 (upper right) Parameter 0**
M1.4	31:0	**Subspan 1, Pixel 0 (upper left) Parameter 0**
M1.3	31:0	**Subspan 0, Pixel 3 (lower right) Parameter 0**
M1.2	31:0	**Subspan 0, Pixel 2 (lower left) Parameter 0**
M1.1	31:0	**Subspan 0, Pixel 1 (upper right) Parameter 0**
M1.0	31:0	**Subspan 0, Pixel 0 (upper left) Parameter 0**

DWord	Bit	Description
M2.7	31:0	**Subspan 3, Pixel 3 (lower right) Parameter 0**
M2.6	31:0	**Subspan 3, Pixel 2 (lower left) Parameter 0**
M2.5	31:0	**Subspan 3, Pixel 1 (upper right) Parameter 0**
M2.4	31:0	**Subspan 3, Pixel 0 (upper left) Parameter 0**
M2.3	31:0	**Subspan 2, Pixel 3 (lower right) Parameter 0**
M2.2	31:0	**Subspan 2, Pixel 2 (lower left) Parameter 0**
M2.1	31:0	**Subspan 2, Pixel 1 (upper right) Parameter 0**
M2.0	31:0	**Subspan 2, Pixel 0 (upper left) Parameter 0**
M3 – Mn		Repeat packets 1 and 2 to cover all required parameters

2.13.1.7 SIMD8 Payload

This message is intended to be used in a SIMD8 thread, or in pairs from a SIMD16 thread. Each message contains sample requests for just 8 pixels.

DWord	Bit	Description
M1.7	31:0	**Subspan 1, Pixel 3 (lower right) Parameter 0** Specifies the value of the pixel's parameter 0. The actual parameter that maps to parameter 0 is given in the table in section *Payload Parameter Definition*. Format = IEEE Float for all sample* message types, U32 for ld and resinfo message types.
M1.6	31:0	**Subspan 1, Pixel 2 (lower left) Parameter 0**
M1.5	31:0	**Subspan 1, Pixel 1 (upper right) Parameter 0**
M1.4	31:0	**Subspan 1, Pixel 0 (upper left) Parameter 0**
M1.3	31:0	**Subspan 0, Pixel 3 (lower right) Parameter 0**
M1.2	31:0	**Subspan 0, Pixel 2 (lower left) Parameter 0**
M1.1	31:0	**Subspan 0, Pixel 1 (upper right) Parameter 0**
M1.0	31:0	**Subspan 0, Pixel 0 (upper left) Parameter 0**
M2 – Mn		Repeat packet 1 to cover all required parameters

2.13.1.8 SIMD8D Payload

This message is intended to be used in a SIMD8 thread, or in pairs from a SIMD16 thread. Each message contains sample requests for just 8 pixels. The "u" and "v" parameters are delivered in double precision floating point, and thus it takes two message phases to deliver 8 values. These are labeled in

the Payload Parameter Definition table as “u0”, “u1”, “v0”, and “v1”. *The number after the u/v indicate which subspan is contained in that parameter.*

Parameters “u0”, “u1”, “v0”, or “v1”:

DWord	Bit	Description
Mn.7	31:0	**Pixel 3 (lower right) Parameter n – upper 32 bits** Specifies the value of the pixel’s parameter n. The actual parameter that maps to parameter n is given in the table in section Payload Parameter Definition. Format = Double precision IEEE Float, upper 32 bits
Mn.6	31:0	**Pixel 3 (lower right) Parameter n – lower 32 bits** Format = Double precision IEEE Float, lower 32 bits
Mn.5	31:0	**Pixel 2 (lower left) Parameter n – upper 32 bits**
Mn.4	31:0	**Pixel 2 (lower left) Parameter n – lower 32 bits**
Mn.3	31:0	**Pixel 1 (upper right) Parameter n – upper 32 bits**
Mn.2	31:0	**Pixel 1 (upper right) Parameter n – lower 32 bits**
Mn.1	31:0	**Pixel 0 (upper left) Parameter n – upper 32 bits**
Mn.0	31:0	**Pixel 0 (upper left) Parameter n – lower 32 bits**

All other parameters:

DWord	Bit	Description
Mn.7	31:0	**Subspan 1, Pixel 3 (lower right) Parameter n** Specifies the value of the pixel’s parameter n. The actual parameter that maps to parameter n is given in the table in section Payload Parameter Definition. Format = IEEE Float
Mn.6	31:0	**Subspan 1, Pixel 2 (lower left) Parameter n**
Mn.5	31:0	**Subspan 1, Pixel 1 (upper right) Parameter n**
Mn.4	31:0	**Subspan 1, Pixel 0 (upper left) Parameter n**
Mn.3	31:0	**Subspan 0, Pixel 3 (lower right) Parameter n**
Mn.2	31:0	**Subspan 0, Pixel 2 (lower left) Parameter n**
Mn.1	31:0	**Subspan 0, Pixel 1 (upper right) Parameter n**
Mn.0	31:0	**Subspan 0, Pixel 0 (upper left) Parameter n**

2.13.1.9 SIMD4x2 Payload

DWord	Bit	Description
M1.7	31:0	**Sample 1 Parameter 3** Specifies the value of the pixel’s parameter 3. The actual parameter that maps to parameter 3 is given in the table in section *Payload Parameter Definition*. Format = IEEE Float for all sample* message types, U32 for ld and resinfo message types.
M1.6	31:0	**Sample 1 Parameter 2**
M1.5	31:0	**Sample 1 Parameter 1**

DWord	Bit	Description
M1.4	31:0	**Sample 1 Parameter 0**
M1.3	31:0	**Sample 0 Parameter 3**
M1.2	31:0	**Sample 0 Parameter 2**
M1.1	31:0	**Sample 0 Parameter 1**
M1.0	31:0	**Sample 0 Parameter 0**
M2		Parameters 4-7 if present
M3		Parameters 8-11 if present

2.13.1.10 SIMD32/64 Payload

2.13.1.10.1 Pixel Shader

This position of **Delta U/V** in the pixel shader payload layout is to allow the register delivered in the pixel shader dispatch containing the coefficients for the texture coordinates to be left in their original position (Delta U = Cxs, Delta V = Cyt). The values for U and V are computed in the pixel shader into the unused positions in this register.

DWord	Bit	Description
M1.7	31:0	Ignored
M1.6	31:0	**Pixel 0 V Address** Format: sample_unorm* and sample_8x8: IEEE_Float in normalized space deinterlace: U32 (Range: [0,2046])
M1.5	31:0	**Delta V**: defines the difference in V for adjacent pixels in the Y direction. **Programming Notes**: • **Delta V** multiplied by **Height** in SURFACE_STATE must be less than or equal to 3 for sample_unorm* message types. • **Delta V** multiplied by **Height** in SURFACE_STATE must be less than 16 for the sample_8x8 message type. • This field is ignored for the deinterlace message type. Format = IEEE_Float in normalized space
M1.4	31:0	Ignored
M1.3	31:0	Ignored
M1.2	31:0	**Pixel 0 U Address** Format: sample_unorm* and sample_8x8: IEEE_Float in normalized space deinterlace: U32 (Range: [0,4095])
M1.1	31:0	**U 2nd Derivative**

DWord	Bit	Description
		Defines the change in the delta U for adjacent pixels in the X direction. **Programming Notes**: • This field is ignored for messages other than sample_8x8. Format = IEEE_Float in normalized space Ignored
M1.0	31:0	**Delta U**: defines the difference in U for adjacent pixels in the X direction. **Programming Notes**: • **Delta U** multiplied by **Width** in SURFACE_STATE must be less than or equal to 3 for sample_unorm* message types. • This field is ignored for the deinterlace message type. Format = IEEE_Float in normalized space

2.13.1.10.2 Media

DWord	Bits	Description
M1.7	31:0	**Group ID Number** Used to group messages for reorder for sample_8x8 messages. All messages with the same Group ID must have the following in common: SURFACE_STATE, SAMPLER_STATE, destination register on *send* instruction, M0, and M1 except for **Horizontal** and **Vertical Block Number**.
M1.6	31:0	**U 2nd Derivative** Defines the change in the delta U for adjacent pixels in the X direction. **Programming Notes:** • This field is ignored for messages other than sample_8x8. • *(64 – (2*du))/35 >= ddu >= -du/18* Format = IEEE_Float in normalized space.
M1.5	31:0	**Delta V**: defines the difference in V for adjacent pixels in the Y direction. **Programming Notes:** • **Delta V** multiplied by **Height** in SURFACE_STATE must be less than or equal to 3 for sample_unorm* message types. • **Delta V** multiplied by **Height** in SURFACE_STATE must be less than 16 for the sample_8x8 message type. • This field is ignored for the deinterlace message type. • Negative **Delta V** are not supported and should be clamped to 0. Format = IEEE_Float in normalized space.
M1.4	31:0	**Delta U:** defines the difference in U for adjacent pixels in the X direction. **Programming Notes:** • **Delta U** multiplied by **Width** in SURFACE_STATE must be less than or equal to 3 for sample_unorm* message types.

DWord	Bits	Description
		• **Delta U** multiplied by **Width** in SURFACE_STATE must be less than 16 for the sample_8x8 message type. • This field is ignored for the deinterlace message type. • Negative **Delta U** are not supported and should be clamped to 0. Format = IEEE_Float in normalized space.
M1.3	31:0	**Pixel 0 V Address** Format: sample_unorm* and sample_8x8: IEEE_Float in normalized space. Deinterlace: U32 (Range: [0,2046]) Specifies the address for the pixel at the top left of the group and not the top of the message block sent in.

M1.2	31:0	**Pixel 0 U Address**
		Format: sample_unorm* and sample_8x8: IEEE_Float in normalized space.
		Deinterlace: U32 (Range: [0,4095])
		Specifies the address for the pixel at the top left of the group and not the top of the message block sent in.
		WA for Sample_8x8 messages only:
		//Only for YUV packed surfaces, NV12 and Y-channel only for Planar surfaces
		if (((int)(u_left*width + 5.0/256) > (int)(uleft*width))
		{
		modified_u_coord = u_coord – 5.0/(256*width); //floating point
		}
		else if(((int)(u_left*width + 255.0/256) == (int)(u_left*width))
		{
		modified_u_coord = u_coord + 1.0/(256*width); //floating point
		}
		Else{
		modified_u_coord = u_coord;
		}
		Where u_left = u – 2*du + 3*ddu for IEF On
		And u_left = u for IEF Off case
		u_coord is the intended Pixel 0 U address and Modified_u_coord is what is sent in this field.

M1.1	31:0	**Vertical Block Number** Specifies the vertical block offset for the 8x8 block being sent for the sample_8x8 message. This will be equal to the vertical pixel offset from the given address pixel 0 V address divided by 8. Format: U9
M1.0	31:0	Ignored

2.13.2 Writeback Message

Corresponding to the four input message definitions are four writeback messages. Each input message generates a corresponding writeback message of the same type (SIMD16, SIMD8, SIMD4x2, or SIMD32/64).

2.13.2.1 SIMD16

A SIMD16 writeback message consists of up to 8 destination registers. Which registers are returned is determined by the write channel mask received in the corresponding input message. Each asserted write channel mask results in both destination registers of the corresponding channel being skipped in the writeback message, and all channels with higher numbered registers being dropped down to fill in the space occupied by the masked channel. For example, if only red and alpha are enabled, red is sent to regid+0 and regid+1, and alpha to regid+2 and regid+3. The pixels written within each destination register is determined by the execution mask on the "send" instruction.

DWord	Bit	Description
W0.7	31:0	**Subspan 1, Pixel 3 (lower right) Red:** Specifies the value of the pixel's red channel. Format = IEEE Float, S31 signed 2's comp integer, or U32 unsigned integer. Format depends on the **Data Return Format** programmed for the surface being sampled.
W0.6	31:0	**Subspan 1, Pixel 2 (lower left) Red**
W0.5	31:0	**Subspan 1, Pixel 1 (upper right) Red**
W0.4	31:0	**Supspan 1, Pixel 0 (upper left) Red**
W0.3	31:0	**Subspan 0, Pixel 3 (lower right) Red**
W0.2	31:0	**Subspan 0, Pixel 2 (lower left) Red**
W0.1	31:0	**Subspan 0, Pixel 1 (upper right) Red**
W0.0	31:0	**Supspan 0, Pixel 0 (upper left) Red**
W1.7	31:0	**Subspan 3, Pixel 3 (lower right) Red**
W1.6	31:0	**Subspan 3, Pixel 2 (lower left) Red**
W1.5	31:0	**Subspan 3, Pixel 1 (upper right) Red**
W1.4	31:0	**Supspan 3, Pixel 0 (upper left) Red**
W1.3	31:0	**Subspan 2, Pixel 3 (lower right) Red**
W1.2	31:0	**Subspan 2, Pixel 2 (lower left) Red**
W1.1	31:0	**Subspan 2, Pixel 1 (upper right) Red**
W1.0	31:0	**Supspan 2, Pixel 0 (upper left) Red**

DWord	Bit	Description
W2		**Subspans 1 and 0 of Green:** See W0 definition for pixel locations
W3		**Subspans 3 and 2 of Green:** See W1 definition for pixel locations
W4		**Subspans 1 and 0 of Blue:** See W0 definition for pixel locations
W5		**Subspans 3 and 2 of Blue:** See W1 definition for pixel locations
W6		**Subspans 1 and 0 of Alpha:** See W0 definition for pixel locations
W7		**Subspans 3 and 2 of Alpha:** See W1 definition for pixel locations
W8.7:1		Reserved (not written): W8 is only delivered when **Pixel Fault Mask Enable** is enabled.
W8.0	31:16	Reserved: always written as 0xffff
	15:0	**Pixel Fault Mask:** This field has the bit for all pixels set to 1 except those pixels in which a page fault has occurred.

2.13.2.2 SIMD8/SIMD8D

This writeback message consists of four registers, or five in the case of sample+killpix. As opposed to the SIMD16 writeback message, channels that are masked in the write channel mask are not skipped, all four channels are always returned. The masked channels, however, are not overwritten in the destination register.

For the sample+killpix message types, an additional register (W4) is included after the last channel register.

DWord	Bits	Description
W0.7	31:0	**Subspan 1, Pixel 3 (lower right) Red:** Specifies the value of the pixel's red channel. Format = IEEE Float, S31 signed 2's comp integer, or U32 unsigned integer. Format depends on the **Data Return Format** programmed for the surface being sampled.
W0.6	31:0	**Subspan 1, Pixel 2 (lower left) Red**
W0.5	31:0	**Subspan 1, Pixel 1 (upper right) Red**
W0.4	31:0	**Supspan 1, Pixel 0 (upper left) Red**
W0.3	31:0	**Subspan 0, Pixel 3 (lower right) Red**
W0.2	31:0	**Subspan 0, Pixel 2 (lower left) Red**
W0.1	31:0	**Subspan 0, Pixel 1 (upper right) Red**
W0.0	31:0	**Supspan 0, Pixel 0 (upper left) Red**
W1		**Subspans 1 and 0 of Green:** See W0 definition for pixel locations
W2		**Subspans 1 and 0 of Blue:** See W0 definition for pixel locations
W3		**Subspans 1 and 0 of Alpha:** See W0 definition for pixel locations

DWord	Bits	Description
W4.7:1		Reserved (not written) **:** This W4 is only delivered for the sample+killpix message type
W4.0	31:16	**Dispatch Pixel Mask:** This field is always 0xffff to allow dword-based ANDing with the R0 header in the pixel shader thread.
	15:0	**Active Pixel Mask:** This field has the bit for all pixels set to 1 except those pixels that have been killed as a result of chroma key with kill pixel mode. Since the SIMD8 message applies to only 8 pixels, only the low 8 bits within this field are used. The high 8 bits are always set to 1. Errata: Active Pixel Mask needs to be ORed with the inverse of the EMask before it is ANDed with the DMask. Also if the sample instruction is within a conditional then the active pixel mask will be overwritten with the partial mask on each different sample instruction so this will have to be done for each instance of the sample instruction not just as the end.
W4.7:1		Reserved (not written): This W4 is only delivered when **Pixel Fault Mask Enable** is enabled.
W4.0	31:8	Reserved: always written as 0xffffff
	7:0	**Pixel Fault Mask:** This field has the bit for all pixels set to 1 except those pixels in which a page fault has occurred.

2.13.2.3 SIMD4x2

A SIMD4x2 writeback message always consists of a single message register containing all four channels of each of the two “pixels” (called “samples” here, as they are not really pixels) of data. The write channel mask bits as well as the execution mask on the “send” instruction are used to determine which of the channels in the destination register are overwritten. If any of the four execution mask bits for a sample is asserted, that sample is considered to be active. The active channels in the write channel mask will be written in the destination register for that sample. If the sample is inactive (all four execution mask bits deasserted), none of the channels for that sample will be written in the destination register.

DWord	Bit	Description
W0.7	31:0	**Sample 1 Alpha:** Specifies the value of the pixel’s alpha channel. Format = IEEE Float, S31 signed 2’s comp integer, or U32 unsigned integer. Format depends on the **Data Return Format** programmed for the surface being sampled.
W0.6	31:0	**Sample 1 Blue**
W0.5	31:0	**Sample 1 Green**
W0.4	31:0	**Sample 1 Red**
W0.3	31:0	**Sample 0 Alpha**
W0.2	31:0	**Sample 0 Blue**
W0.1	31:0	**Sample 0 Green**
W0.0	31:0	**Sample 0 Red**
W1.7:1		Reserved (not written) : W4 is only delivered when **Pixel Fault Mask Enable** is enabled.
W1.0	31:2	Reserved: always written as 0x3fffffff
	1:0	**Pixel Fault Mask:** This field has the bit for all samples set to 1 except those pixels in which a page fault has occurred.

2.13.2.4 SIMD32/64

2.13.2.4.1 Sample_unorm*

Pixels are numbered as follows:

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

Which registers are returned is determined by the write channel mask received in the corresponding input message. Each asserted write channel mask results in both destination registers of the corresponding channel being skipped in the writeback message, and all channels with higher numbered registers being dropped down to fill in the space occupied by the masked channel. For example, if only red and alpha are enabled, red is sent to regid+0 and regid+1, and alpha to regid+2 and regid+3 (using 16 bit Full mode as an example).

“16 bit Full” Output Format Control Mode

DWord	Bit	Description
W0.7	31:16	**Pixel 15 Red** Format = 16-bit UNORM with an 8-bit range (the value FF00h maps to a real value of 1.0) Range = [0000h:FF00h]
	15:0	**Pixel 14 Red**
W0.6		**Pixel 13 & 12 Red**
W0.5		**Pixel 11 & 10 Red**
W0.4		**Pixel 9 & 8 Red**
W0.3		**Pixel 7 & 6 Red**
W0.2		**Pixel 5 & 4 Red**
W0.1		**Pixel 3 & 2 Red**
W0.0		**Pixel 1 & 0 Red**
W1.7		**Pixel 31 & 30 Red**
W1.6		**Pixel 29 & 28 Red**
W1.5		**Pixel 27 & 26 Red**
W1.4		**Pixel 25 & 24 Red**
W1.3		**Pixel 23 & 22 Red**
W1.2		**Pixel 21 & 20 Red**

DWord	Bit	Description
W1.1		**Pixel 19 & 18 Red**
W1.0		**Pixel 17 & 16 Red**
W2		**Pixels 15:0 Green**
W3		**Pixels 31:16 Green**
W4		**Pixels 15:0 Blue**
W5		**Pixels 31:16 Blue**
W6		**Pixels 15:0 Alpha**
W7		**Pixels 31:16 Alpha**

"16 Bit Chrominance Downsampled" Output Format Control Mode

In this mode the odd pixel red & blue channels are not included.

DWord	Bit	Description
W0.7	31:16	**Pixel 30 Red** Format = 16-bit UNORM with an 8-bit range (the value FF00h maps to a real value of 1.0) Range = [0000h:FF00h]
	15:0	**Pixel 28 Red**
W0.6		**Pixel 26 & 24 Red**
W0.5		**Pixel 22 & 20 Red**
W0.4		**Pixel 18 & 16 Red**
W0.3		**Pixel 14 & 12 Red**
W0.2		**Pixel 10 & 8 Red**
W0.1		**Pixel 6 & 4 Red**
W0.0		**Pixel 2 & 0 Red**
W1.7	31:16	**Pixel 15 Green**
	15:0	**Pixel 14 Green**
W1.6		**Pixel 13 & 12 Green**
W1.5		**Pixel 11 & 10 Green**
W1.4		**Pixel 9 & 8 Green**

DWord	Bit	Description
W1.3		**Pixel 7 & 6 Green**
W1.2		**Pixel 5 & 4 Green**
W1.1		**Pixel 3 & 2 Green**
W1.0		**Pixel 1 & 0 Green**
W2.7		**Pixel 31 & 30 Green**
W2.6		**Pixel 29 & 28 Green**
W2.5		**Pixel 27 & 26 Green**
W2.4		**Pixel 25 & 24 Green**
W2.3		**Pixel 23 & 22 Green**
W2.2		**Pixel 21 & 20 Green**
W2.1		**Pixel 19 & 18 Green**
W2.0		**Pixel 17 & 16 Green**
W3.7	31:16	**Pixel 30 Blue**
	15:0	**Pixel 28 Blue**
W3.6		**Pixel 26 & 24 Blue**
W3.5		**Pixel 22 & 20 Blue**
W3.4		**Pixel 18 & 16 Blue**
W3.3		**Pixel 14 & 12 Blue**
W3.2		**Pixel 10 & 8 Blue**
W3.1		**Pixel 6 & 4 Blue**
W3.0		**Pixel 2 & 0 Blue**
W4.7	31:16	**Pixel 15 Alpha**
	15:0	**Pixel 14 Alpha**
W4.6		**Pixel 13 & 12 Alpha**
W4.5		**Pixel 11 & 10 Alpha**
W4.4		**Pixel 9 & 8 Alpha**

DWord	Bit	Description
W4.3		**Pixel 7 & 6 Alpha**
W4.2		**Pixel 5 & 4 Alpha**
W4.1		**Pixel 3 & 2 Alpha**
W4.0		**Pixel 1 & 0 Alpha**
W5.7		**Pixel 31 & 30 Alpha**
W5.6		**Pixel 29 & 28 Alpha**
W5.5		**Pixel 27 & 26 Alpha**
W5.4		**Pixel 25 & 24 Alpha**
W5.3		**Pixel 23 & 22 Alpha**
W5.2		**Pixel 21 & 20 Alpha**
W5.1		**Pixel 19 & 18 Alpha**
W5.0		**Pixel 17 & 16 Alpha**

"8 Bit Full" Output Format Control Mode

DWord	Bit	Description
W0.7	31:24	**Pixel 31 Red** Format = 8-bit UNORM Range = [00h:FFh]
	23:16	**Pixel 30 Red**
	15:8	**Pixel 29 Red**
	7:0	**Pixel 28 Red**
W0.6		**Pixel 27:24 Red**
W0.5		**Pixel 23:20 Red**
W0.4		**Pixel 19:16 Red**
W0.3		**Pixel 15:12 Red**
W0.2		**Pixel 11:8 Red**
W0.1		**Pixel 7:4 Red**
W0.0		**Pixel 3:0 Red**

DWord	Bit	Description
W1		**Pixels 31:0 Green**
W2		**Pixels 31:0 Blue**
W3		**Pixels 31:0 Alpha**

“8 Bit Chrominance Downsampled” Output Format Control Mode

If either red or blue channel (but not both) are masked, the W0 register is included in the payload but the masked channel is not written to the GRF. If both are masked, W0 is not included in the payload (reducing the response length by one).

DWord	Bit	Description
W0.7	31:24	**Pixel 30 Red** Format = 16-bit UNORM with an 8-bit range (the value FF00h maps to a real value of 1.0) Range = [0000h:FF00h]
	23:16	**Pixel 28 Red**
	15:8	**Pixel 26 Red**
	7:0	**Pixel 24 Red**
W0.6		**Pixel 22, 20, 18, 16 Red**
W0.5		**Pixel 14, 12, 10, 8 Red**
W0.4		**Pixel 6, 4, 2, 0 Red**
W0.3		**Pixel 30, 28, 26, 24 Blue**
W0.2		**Pixel 22, 20, 18, 16 Blue**
W0.1		**Pixel 14, 12, 10, 8 Blue**
W0.0		**Pixel 6, 4, 2, 0 Blue**
W1.7	31:24	**Pixel 31 Green**
	23:16	**Pixel 30 Green**
	15:8	**Pixel 29 Green**
	7:0	**Pixel 28 Green**
W1.6		**Pixel 27:24 Green**
W1.5		**Pixel 23:20 Green**
W1.4		**Pixel 19:16 Green**

DWord	Bit	Description
W1.3		**Pixel 15:12 Green**
W1.2		**Pixel 11:8 Green**
W1.1		**Pixel 7:4 Green**
W1.0		**Pixel 3:0 Green**
W2.7		**Pixel 31:28 Alpha**
W2.6		**Pixel 27:24 Alpha**
W2.5		**Pixel 23:20 Alpha**
W2.4		**Pixel 19:16 Alpha**
W2.3		**Pixel 15:12 Alpha**
W2.2		**Pixel 11:8 Alpha**
W2.1		**Pixel 7:4 Alpha**
W2.0		**Pixel 3:0 Alpha**

Additional Writeback Phase for sample_unorm+killpix

For the sample_unorm+killpix messages, an additional writeback phase is returned. The value of “n” depends on which channels are enabled for return and the **Output Format Control Mode**, this register will immediately follow the first part of the writeback message.

<table>
<tr><th>DWord</th><th>Bit</th><th>Description</th></tr>
<tr><td>Wn.7:1</td><td></td><td>Reserved (not written)</td></tr>
<tr><td>Wn.0</td><td>31:0</td><td>Active Pixel Mask: This field has the bit for all pixels set to 1 except those pixels that have been killed as a result of chroma key with kill pixel mode.

The bits in this mask correspond to the pixels as follows and they are listed from upper left (MSB) lower right LSB:
<table><tr><td>31</td><td>30</td><td>29</td><td>28</td><td>27</td><td>26</td><td>25</td><td>24</td></tr><tr><td>23</td><td>22</td><td>21</td><td>20</td><td>19</td><td>18</td><td>17</td><td>16</td></tr><tr><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td></tr><tr><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr></table></td></tr>
</table>

2.13.2.5 Sample_8x8 Writeback Messages

2.13.2.5.1 Sample_8x8 Writeback Messages

The writeback message for sample_8x8 consists of up to 16 destination registers. Which registers are returned is determined by the write channel mask received in the corresponding input message. Each asserted write channel mask results in all four destination registers of the corresponding channel being

skipped in the writeback message, and all channels with higher numbered registers being dropped down to fill in the space occupied by the masked channel.

Pixels are numbered as follows:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63

"16 bit Full" Output Format Control Mode

DWord	Bit	Description
W0.7	31:16	**Pixel 15 Red** Format = 16-bit UNORM with an 8-bit range (the value FF00h maps to a real value of 1.0) Range = [0000h:FF00h]
	15:0	**Pixel 14 Red**
W0.6		**Pixel 13 & 12 Red**
W0.5		**Pixel 11 & 10 Red**
W0.4		**Pixel 9 & 8 Red**
W0.3		**Pixel 7 & 6 Red**
W0.2		**Pixel 5 & 4 Red**
W0.1		**Pixel 3 & 2 Red**
W0.0		**Pixel 1 & 0 Red**
W1		**Pixel 31:16 Red**
W2		**Pixels 47:32 Red**
W3		**Pixels 63:33 Red**
W4		**Pixels 15:0 Green**
W5		**Pixels 31:16 Green**
W6		**Pixels 47:32 Green**
W7		**Pixels 63:33 Green**
W8		**Pixels 15:0 Blue**
W9		**Pixels 31:16 Blue**

W10		**Pixels 47:32 Blue**
W11		**Pixels 63:33 Blue**
W12		**Pixels 15:0 Alpha**
W13		**Pixels 31:16 Alpha**
W14		**Pixels 47:32 Alpha**
W15		**Pixels 63:33 Alpha**

“16 Bit Chrominance Downsampled” Output Format Control Mode

In this mode the odd pixel red & blue channels are not included.

DWord	Bit	Description
W0.7	31:16	**Pixel 30 Red** Format = 16-bit UNORM with an 8-bit range (the value FF00h maps to a real value of 1.0) Range = [0000h:FF00h]
	15:0	**Pixel 28 Red**
W0.6		**Pixel 26 & 24 Red**
W0.5		**Pixel 22 & 20 Red**
W0.4		**Pixel 18 & 16 Red**
W0.3		**Pixel 14 & 12 Red**
W0.2		**Pixel 10 & 8 Red**
W0.1		**Pixel 6 & 4 Red**
W0.0		**Pixel 2 & 0 Red**
W1.7		**Pixel 62 & 60 Red**
W1.6		**Pixel 58 & 56 Red**
W1.5		**Pixel 54 & 52 Red**
W1.4		**Pixel 50 & 48 Red**
W1.3		**Pixel 46 & 44 Red**
W1.2		**Pixel 42 & 40 Red**
W1.1		**Pixel 38 & 36 Red**

W1.0		**Pixel 34 & 32 Red**
W2.7	31:16	**Pixel 15 Green**
	15:0	**Pixel 14 Green**
W2.6		**Pixel 13 & 12 Green**
W2.5		**Pixel 11 & 10 Green**
W2.4		**Pixel 9 & 8 Green**
W2.3		**Pixel 7 & 6 Green**
W2.2		**Pixel 5 & 4 Green**
W2.1		**Pixel 3 & 2 Green**
W2.0		**Pixel 1 & 0 Green**
W3		**Pixel 31:16 Green**
W4		**Pixel 47:32 Green**
W5		**Pixel 63:48 Green**
W6.7	31:16	**Pixel 30 Blue**
	15:0	**Pixel 28 Blue**
W6.6		**Pixel 26 & 24 Blue**
W6.5		**Pixel 22 & 20 Blue**
W6.4		**Pixel 18 & 16 Blue**
W6.3		**Pixel 14 & 12 Blue**
W6.2		**Pixel 10 & 8 Blue**
W6.1		**Pixel 6 & 4 Blue**
W6.0		**Pixel 2 & 0 Blue**
W7.7		**Pixel 62 & 60 Blue**
W7.6		**Pixel 58 & 56 Blue**
W7.5		**Pixel 54 & 52 Blue**
W7.4		**Pixel 50 & 48 Blue**

W7.3		**Pixel 46 & 44 Blue**
W7.2		**Pixel 42 & 40 Blue**
W7.1		**Pixel 38 & 36 Blue**
W7.0		**Pixel 34 & 32 Blue**
W8.7	31:16	**Pixel 15 Alpha**
	15:0	**Pixel 14 Alpha**
W8.6		**Pixel 13 & 12 Alpha**
W8.5		**Pixel 11 & 10 Alpha**
W8.4		**Pixel 9 & 8 Alpha**
W8.3		**Pixel 7 & 6 Alpha**
W8.2		**Pixel 5 & 4 Alpha**
W8.1		**Pixel 3 & 2 Alpha**
W8.0		**Pixel 1 & 0 Alpha**
W9		**Pixel 31:16 Alpha**
W10		**Pixel 47:32 Alpha**
W11		**Pixel 63:48 Alpha**

“8 Bit Full” Output Format Control Mode

DWord	Bit	Description
W0.7	31:24	**Pixel 31 Red** Format = 8-bit UNORM Range = [00h:FFh]
	23:16	**Pixel 30 Red**
	15:8	**Pixel 29 Red**
	7:0	**Pixel 28 Red**
W0.6		**Pixel 27:24 Red**
W0.5		**Pixel 23:20 Red**
W0.4		**Pixel 19:16 Red**

W0.3		**Pixel 15:12 Red**
W0.2		**Pixel 11:8 Red**
W0.1		**Pixel 7:4 Red**
W0.0		**Pixel 3:0 Red**
W1.7		**Pixel 63:60 Red**
W1.6		**Pixel 59:56 Red**
W1.5		**Pixel 55:52 Red**
W1.4		**Pixel 51:48 Red**
W1.3		**Pixel 47:44 Red**
W1.2		**Pixel 43:40 Red**
W1.1		**Pixel 39:36 Red**
W1.0		**Pixel 35:52 Red**
W2		**Pixels 31:0 Green**
W3		**Pixels 63:32 Green**
W4		**Pixels 31:0 Blue**
W5		**Pixels 63:32 Blue**
W6		**Pixels 31:0 Alpha**
W7		**Pixels 63:32 Alpha**

“8 Bit Chrominance Downsampled” Output Format Control Mode

DWord	Bit	Description
W0.7	31:24	**Pixel 62 Red** Format = 8-bit UNORM Range = [00h:FFh]
	23:16	**Pixel 60 Red**
	15:8	**Pixel 58 Red**
	7:0	**Pixel 56 Red**
W0.6		**Pixel 54, 52, 50, 48 Red**

W0.5		**Pixel 46, 44, 42, 40 Red**
W0.4		**Pixel 38, 36, 34, 32 Red**
W0.3		**Pixel 30, 28, 26, 24 Red**
W0.2		**Pixel 22, 20, 18, 16 Red**
W0.1		**Pixel 14, 12, 10, 8 Red**
W0.0		**Pixel 6, 4, 2, 0 Red**
W1.7	31:24	**Pixel 31 Green**
	23:16	**Pixel 30 Green**
	15:8	**Pixel 29 Green**
	7:0	**Pixel 28 Green**
W1.6		**Pixel 27:24 Green**
W1.5		**Pixel 23:20 Green**
W1.4		**Pixel 19:16 Green**
W1.3		**Pixel 15:12 Green**
W1.2		**Pixel 11:8 Green**
W1.1		**Pixel 7:4 Green**
W1.0		**Pixel 3:0 Green**
W2		**Pixel 63:32 Green**
W3.7	31:24	**Pixel 62 Blue**
	23:16	**Pixel 60 Blue**
	15:8	**Pixel 58 Blue**
	7:0	**Pixel 56 Blue**
W3.6		**Pixel 54, 52, 50, 48 Blue**
W3.5		**Pixel 46, 44, 42, 40 Blue**
W3.4		**Pixel 38, 36, 34, 32 Blue**
W3.3		**Pixel 30, 28, 26, 24 Blue**

W3.2		**Pixel 22, 20, 18, 16 Blue**
W3.1		**Pixel 14, 12, 10, 8 Blue**
W3.0		**Pixel 6, 4, 2, 0 Blue**
W4.7	31:24	**Pixel 31 Alpha**
	23:16	**Pixel 30 Alpha**
	15:8	**Pixel 29 Alpha**
	7:0	**Pixel 28 Alpha**
W4.6		**Pixel 27:24 Alpha**
W4.5		**Pixel 23:20 Alpha**
W4.4		**Pixel 19:16 Alpha**
W4.3		**Pixel 15:12 Alpha**
W4.2		**Pixel 11:8 Alpha**
W4.1		**Pixel 7:4 Alpha**
W4.0		**Pixel 3:0 Alpha**
W5		**Pixel 63:32 Alpha**

2.13.2.6 Deinterlace

The deinterlace message has three different writeback messages, depending on the **DI Enable** and **DI Partial** fields of SAMPLER_STATE.

Pixels are indicated by an (X, Y) pair. The following tables indicate the format of common **Luma, Chroma, STMM,** and **Block Noise Estimate/Denoise History** blocks defined as portions of the specific writeback messages defined in the following sections. Each block defines one register.

Luma block definition:

DWord	Bit	Description
Wn.7	31:24	**Luma (15,1)** Format = U8
	23:16	**Luma (14,1)**
	15:8	**Luma (13,1)**
	7:0	**Luma (12,1)**
Wn.6	31:0	**Luma (11:8,1)**

DWord	Bit	Description
Wn.5	31:0	**Luma (7:4,1)**
Wn.4	31:0	**Luma (3:0,1)**
Wn.3	31:0	**Luma (15:12,0)**
Wn.2	31:0	**Luma (11:8,0)**
Wn.1	31:0	**Luma (7:4,0)**
Wn.0	31:0	**Luma (3:0,0)**

Chroma block definition:

DWord	Bit	Description
Wp.7	31:24	**Cb (14,1)** Format = U8
	23:16	**Cr (14,1)** Format = U8
	15:8	**Cb (12,1)**
	7:0	**Cr (12,1)**
Wp.6	31:0	**Cr & Cb (10:8,1)**
Wp.5	31:0	**Cr & Cb (6:4,1)**
Wp.4	31:0	**Cr & Cb (2:0,1)**
Wp.3	31:0	**Cr & Cb (14:12,0)**
Wp.2	31:0	**Cr & Cb (10:8,0)**
Wp.1	31:0	**Cr & Cb (6:4,0)**
Wp.0	31:0	**Cr & Cb (2:0,0)**

STMM block definition:

DWord	Bit	Description
Wr.7	31:24	**STMM (14,3)** Format = U8
	23:16	**STMM (12,3)**
	15:8	**STMM (10,3)**
	7:0	**STMM (8,3)**

DWord	Bit	Description
Wr.6	31:0	**STMM (6:0,3)**
Wr.5	31:0	**STMM (14:8,2)**
Wr.4	31:0	**STMM (6:0,2)**
Wr.3	31:0	**STMM (14:8,1)**
Wr.2	31:0	**STMM (6:0,1)**
Wr.1	31:0	**STMM (14:8,0)**
Wr.0	31:0	**STMM (6:0,0)**

Block Noise Estimate/Denoise History block definition: [prior to Gen6]

DWord	Bit	Description
Wq.7	31:0	Reserved : MBZ
Wq.6	31:0	Reserved : MBZ
Wq.5	31:0	Reserved : MBZ
Wq.4	31:0	Reserved : MBZ
Wq.3	31:0	Reserved : MBZ
Wq.2	31:0	Reserved : MBZ
Wq.1	31:8	Reserved : MBZ
Wq.1	7:0	**Block Noise Estimate** Format = U8
Wq.0	31:24	**Denoise History** for 4x4 at Y = 15 to 12, X = 3 to 0 Format = U8
Wq.0	23:16	**Denoise History** for 4x4 at Y = 11 to 8, X = 3 to 0
Wq.0	15:8	**Denoise History** for 4x4 at Y = 7 to 4, X = 3 to 0
Wq.0	7:0	**Denoise History** for 4x4 at Y = 3 to 0, X = 3 to 0

Block Noise Estimate/Denoise History block definition: [Gen6]

Wq.7	31:16	**Y[15:0] - For a 1080 screen & 4 high blocks we need 9-bits**
Wq.7	15:0	**X[15:0] – for a 2048 screen & 16 wide block we need 7-bits**
Wq.6	31:24	**STAD0** – Sum in time of absolute differences for 16x8 (DN only) or 16x4 (DN/DI) Format = U8
Wq.6	23:16	**STAD1**
Wq.6	15:8	**STAD2**

Wq.6	7:0	**STAD3**
Wq.5	31:24	**SHCM0** – Sum horizontaly of absolute differences
Wq.5	23:16	**SHCM1**
Wq.5	15:8	**SHCM2**
Wq.5	7:0	**SHCM3**
Wq.4	31:24	**SVCM0** – Sum vertically of absolute differences.
Wq.4	23:16	**SVCH1**
Wq.4	15:8	**SVCH2**
Wq.4	7:0	**SVCH3**
Wq.3	31:16	**FMD Variance[0]** - Diff_cTpT – difference in top fields of current and previous frame Format = U16
Wq.3	15:0	**FMD Variance[1]** – Diff_cBpB – difference in bottom field of current and previous frame
Wq.2	31:16	**FMD Variance[2]** – Diff_cTcB – difference between top and bottom field in current frame.
Wq.2	15:0	**FMD Variance[3]** – Diff_cTpB – difference between current top and previous bottom
Wq.1	31:16	**FMD Variance[4] –** Diff_cBpT – difference between current bottom and previous top**.**
Wq.1	15:8	**FMD Variance[7]** – sum of pixels that are moving (different above a threshold) Format = U8
Wq.1		
Wq.0		
Wq.0		
Wq.0		
Wq.0		

Block Noise Estimate/Denoise History block definition: [Gen6 DI enabled]

DWord	Bit	Description
Wq.7	31:16	Y[15:0] – Location of 16x4
Wq.7	15:0	X[15:0] - Location of 16x4
Wq.6	31:24	STAD0 - Sum in time of absolute differences for 4x4 Format = U8 [STAD values are 0 if DN is disabled]
Wq.6	23:16	STAD1
Wq.6	15:8	STAD2
Wq.6	7:0	STAD3 (Ignore when both DN & DI are enabled)
Wq.5	31:24	SHCM0 - Sum horizontally of absolute differences for 4x4

		Format = U8 [SHCM values are 0 if DN is disabled]
Wq.5	23:16	SHCM1
Wq.5	15:8	SHCM2
Wq.5	7:0	SHCM3 (Ignore when both DN & DI are enabled)
Wq.4	31:24	SVCM0 Sum Vertically of absolute differences for 4x4 Format = U8 [SVCM values are 0 if DN is disabled]
Wq.4	23:16	SVCM1
Wq.4	15:8	SVCM2
Wq.4	7:0	SVCM3 (Ignore when both DN & DI are enabled)
Wq.3	31:16	Diff_cTpT - difference in top fields of current and previous frame Format = U16
Wq.3	15:0	Diff_cBpB - difference in bottom field of current and previous frame
Wq.2	31:16	Diff_cTcB - difference between top and bottom field in current frame.
Wq.2	15:0	Diff_cTpB - difference between current top and previous bottom
Wq.1	31:16	Diff_cBpT - difference between current bottom and previous top.
Wq.1	15:8	Motion_Count - number of pixels that are moving (different above a threshold) Format = U8
Wq.1	7:0	Block Noise Estimate for 16x4 (Valid only if DN is enabled)
Wq.0	31:24	**Denoise History** for 4x4 at Y = 15 to 12, X = 3 to 0 Format = U8
Wq.0	23:16	**Denoise History** for 4x4 at Y = 11 to 8, X = 3 to 0
Wq.0	15:8	**Denoise History** for 4x4 at Y = 7 to 4, X = 3 to 0
Wq.0	7:0	**Denoise History** for 4x4 at Y = 3 to 0, X = 3 to 0

Block Noise Estimate/Denoise History block definition: [Gen6 DI disabled]

DWord	Bit	Description
Wq.7	31:16	Y[15:0] – Location of 16x4
Wq.7	15:0	X[15:0] - Location of 16x4
Wq.6	31:24	STAD0 - Sum in time of absolute differences for 4x8 Format = U8
Wq.6	23:16	STAD1
Wq.6	15:8	STAD2
Wq.6	7:0	STAD3
Wq.5	31:24	SHCM0 - Sum horizontally of absolute difference for 4x8
Wq.5	23:16	SHCM1
Wq.5	15:8	SHCM2
Wq.5	7:0	SHCM3
Wq.4	31:24	SVCM0 Sum Vertically of absolute difference for 4x8
Wq.4	23:16	SVCM1
Wq.4	15:8	SVCM2
Wq.4	7:0	SVCM3

Wq.3	31:16	Reserved
Wq.3	15:0	Reserved
Wq.2	31:8	Reserved
Wq.2	7:0	Block Noise Estimate for 16x8
Wq.1	31:24	**Denoise History** for 4x4 at X = 15 to 12, Y = 7 to 4 Format = U8
Wq.1	23:16	**Denoise History** for 4x4 at X = 11 to 8, Y = 7 to 4
Wq.1	15:8	**Denoise History** for 4x4 at X = 7 to 4, Y = 7 to 4
Wq.1	7:0	**Denoise History** for 4x4 at X = 15 to 12, Y = 3 to 0
Wq.0	31:24	**Denoise History** for 4x4 at Y = 15 to 12, X = 3 to 0 Format = U8
Wq.0	23:16	**Denoise History** for 4x4 at Y = 11 to 8, X = 3 to 0
Wq.0	15:8	**Denoise History** for 4x4 at Y = 7 to 4, X = 3 to 0
Wq.0	7:0	**Denoise History** for 4x4 at Y = 3 to 0, X = 3 to 0

Block Noise Estimate/Denoise History block definition: [Gen7 +] DI Enabled

DWord	Bit	Description
Wq.7	31:16	Y[15:0]
Wq.7	15:0	X[15:0]
Wq.6	31:16	STAD - Sum in time of absolute differences for 16x4 – value is 0 if DN disabled. Format = U16
Wq.6	15:0	SHCM - Sum horizontaly of absolute differences – value is 0 if DN is disabled. Format = U16
Wq.5	31:16	SVCM - Sum vertically of absolute differences – value is 0 if DN is disabled.. Format = U16
Wq.5	15:0	Diff_cTpT - sum of differences in top fields of current and previous frame Format = U16
Wq.4	31:16	Diff_cBpB - sum of differences in bottom field of current and previous frame Format = U16
Wq.4	15:0	Diff_cTcB -sum of differences between top and bottom field in current frame. Format = U16
Wq.3	31:16	Diff_cTpB - sum of differences between current top and previous bottom Format = U16
Wq.3	15:0	Diff_cBpT - sum of differences between current bottom and previous top.

DWord	Bit	Description
		Format = U16
Wq.2	31:0	Reserved
Wq.1	31:24	Tearing_Count - number of pixels that have (diff_cTcB > diff_cTcT + diff_cBcB) Format = U8
Wq.1	23:16	Fitting_Count - number of pixels that have (diff_cTcB<=diff_cTcT + diff_cBcB) Format = U8
Wq.1	15:8	Motion_Count - number of pixels that are moving (different above a threshold) Format = U8
Wq.1	7:0	Block Noise Estimate Format = U8
Wq.0	31:24	**Denoise History** for 4x4 at Y = 15 to 12, X = 3 to 0 Format = U8
Wq.0	23:16	**Denoise History** for 4x4 at Y = 11 to 8, X = 3 to 0
Wq.0	15:8	**Denoise History** for 4x4 at Y = 7 to 4, X = 3 to 0
Wq.0	7:0	**Denoise History** for 4x4 at Y = 3 to 0, X = 3 to 0

Block Noise Estimate/Denoise History block definition: [Gen7+] DI Disabled:

DWord	Bit	Description
Wq.7	31:16	Y[15:0]
Wq.7	15:0	X[15:0]
Wq.6	31:16	STAD - Sum in time of absolute differences for top 16x4 Format = U16
Wq.6	15:0	SHCM - Sum horizontaly of absolute differences for top 16x4 Format = U16
Wq.5	31:16	SVCM - Sum vertically of absolute differences for top 16x4 Format = U16
Wq.5	15:0	STAD - Sum in time of absolute differences for bottom 16x4 Format = U16
Wq.4	31:16	SHCM - Sum horizontaly of absolute differences for bottom 16x4 Format = U16
Wq.4	15:0	SVCM - Sum vertically of absolute differences for bottom 16x4 Format = U16
Wq.3	31:0	Reserved
Wq.2	31:8	Reserved
Wq.2	7:0	Block Noise Estimate Format = U8

DWord	Bit	Description
Wq.1	31:24	Denoise History for 4x4 at X = 15 to 12, Y = 7 to 4 Format = U8
Wq.1	23:16	Denoise History for 4x4 at X = 11 to 8, Y = 7 to 4
Wq.1	15:8	Denoise History for 4x4 at X = 7 to 4, Y = 7 to 4
Wq.1	7:0	Denoise History for 4x4 at X = 3 to 0, Y = 7 to 4
Wq.0	31:24	Denoise History for 4x4 at X = 15 to 12, Y = 3 to 0 Format = U8
Wq.0	23:16	Denoise History for 4x4 at X = 11 to 8, Y = 3 to 0
Wq.0	15:8	Denoise History for 4x4 at X = 7 to 4, Y = 3 to 0
Wq.0	7:0	Denoise History for 4x4 at X = 3 to 0, Y = 3 to 0

DI Enabled (Only)

This writeback message is returned when the DI Enable field in SAMPLER_STATE is enabled. The response length possibilities are:

- DN Enabled & surface_format == 4:2:2 packed: 12
- DN Enabled & surface_format != 4:2:2 packed: 11
- DN Disabled: 10

DWord	Bit	Description
W0		**Previous 2nd Field Deinterlaced Luma for Y=0,1** Refer to Luma block above for definition.
W1		**Previous 2nd Field Deinterlaced Luma for Y=2,3**
W2		**Previous 2nd Field Deinterlaced Chroma for Y=0,1** Refer to Chroma block above for definition.
W3		**Previous 2nd Field Deinterlaced Chroma for Y=2,3**
W4		**Current 1st Field Deinterlaced Luma for Y=0,1**
W5		**Current 1st Field Deinterlaced Luma for Y=2,3**
W6		**Current 1st Field Deinterlaced Chroma for Y=0,1**
W7		**Current 1st Field Deinterlaced Chroma for Y=2,3**
W8		**STMM** Refer to STMM block above for definition.
W9		**Block Noise Estimate/Denoise History** Refer to Block Noise Estimate/Denoise History block above for definition.

DWord	Bit	Description
W10		**Current 2nd Field Luma for 16x2** This register is only included if **DN Enable** is enabled.
W11		**Current 2nd Field Chroma** This register is only included if **DN Enable** is enabled. Only valid if input surface format is 4:2:2

The denoised luma for both the current 1st and 2nd field needs to be written out, but only the 2nd field has a dedicated location. This is because the denoised data for the 1st field is in the deinterlaced output for the 1st field – Y=0 and Y=2 are the denoised data, while Y=1 and Y=3 either the deinterlaced lines or copied from the previous or current frame if progressive.

DI Disabled

This writeback message is returned when the **DI Enable** field in SAMPLER_STATE is disabled. The DN with DI disabled responses with a 16x8 block rather than a 16x4 with a response length of 9 for a 4:2:2 input format, or 5 for other formats. Two denoised luma and chroma fields are combined into an interleaved top/bottom format.

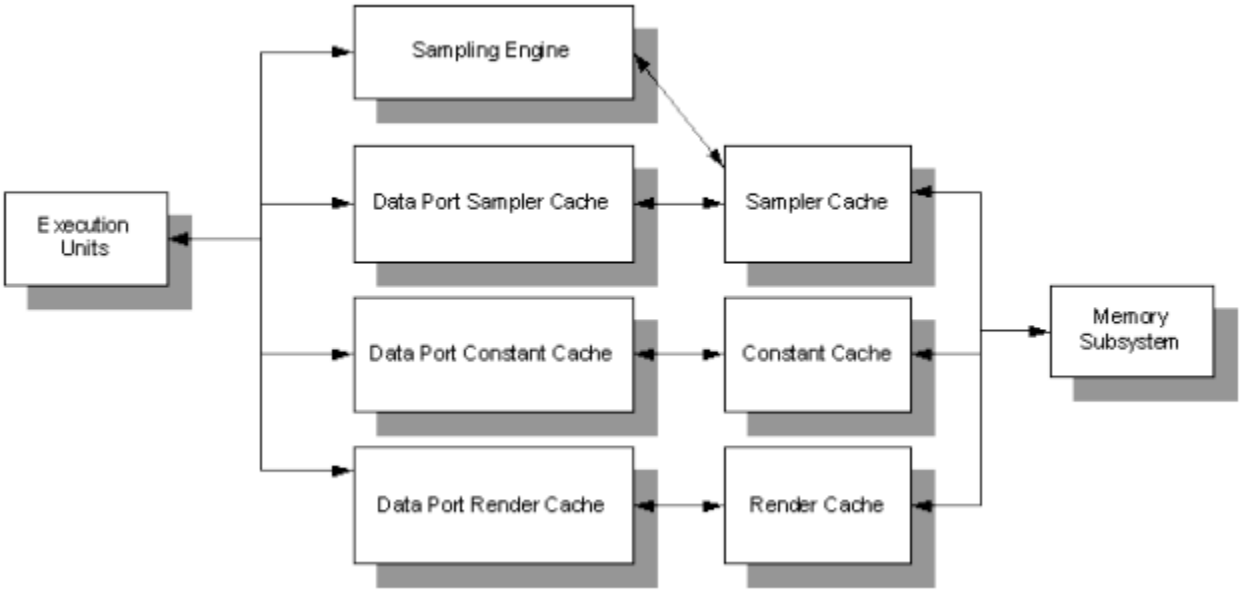
		Description
W0		**Luma for Y=0 & 1** Refer to Luma block above for definition.
W1		**Luma for Y=2 & 3** Refer to Luma block above for definition, but add 2 to Y to get location
W2		**Luma for Y=4 & 5**
W3		**Luma for Y=6 & 7**
W4.7	31:16	**Y[15:0]** Y co-ordinate of the current block within the frame
W4.7	15:0	**X[15:0]** X co-ordinate of the current block within the frame
W4.6	31:24	**STAD0** – Sum in time of absolute differences for the 1st 4x8 Format = U8
W4.6	23:16	**STAD1**– Sum in time of absolute differences for the 2nd 4x8
W4.6	15:8	**STAD2 –** Sum in time of absolute differences for the 3rd 4x8
W4.6	7:0	**STAD3** – Sum in time of aboslute differences for the 4th 4x8
W4.5	31:24	**SHCM0** – Sum horizontaly of absolute differences
W4.5	23:16	**SHCM1**

		Description
W4.5	15:8	**SHCM2**
W4.5	7:0	**SHCM3**
W4.4	31:24	**SVCM0** – Sum vertically of absolute differences.
W4.4	23:16	**SVCH1**
W4.4	15:8	**SVCH2**
W4.4	7:0	**SVCH3**
W4.3	31:0	Reserved : MBZ
W4.2	31:8	Reserved : MBZ
	7:0	**Block Noise Estimate** Format = U8
W4.1	31:24	**Denoise History for 4x4 at X = 15 to 12, Y = 7 to 4**
	23:16	**Denoise History for 4x4 at X = 11 to 8, Y = 7 to 4**
	15:8	**Denoise History for 4x4 at X = 7 to 4, Y = 7 to 4**
	7:0	**Denoise History for 4x4 at X = 3 to 0, Y = 7 to 4**
W4.0	31:24	**Denoise History for 4x4 at X = 15 to 12, Y = 3 to 0**
	23:16	**Denoise History for 4x4 at X = 11 to 8, Y = 3 to 0**
	15:8	**Denoise History for 4x4 at X = 7 to 4, Y = 3 to 0**
	7:0	**Denoise History for 4x4 at X = 3 to 0, Y = 3 to 0**
W5		**Chroma for Y=0 & 1** Refer to Chroma block above for definition. Only delivered if input surface format is 4:2:2
W6		**Chroma for Y=2 & 3** Refer to Chroma block above for definition, but add 2 to Y to get location. Only delivered if input surface format is 4:2:2
W7		**Chroma for Y=4 & 5** Only valid if input surface format is 4:2:2
W8		**Chroma for Y=6 & 7** Only sent if input surface format is 4:2:2

3. Shared Functions – Data Port

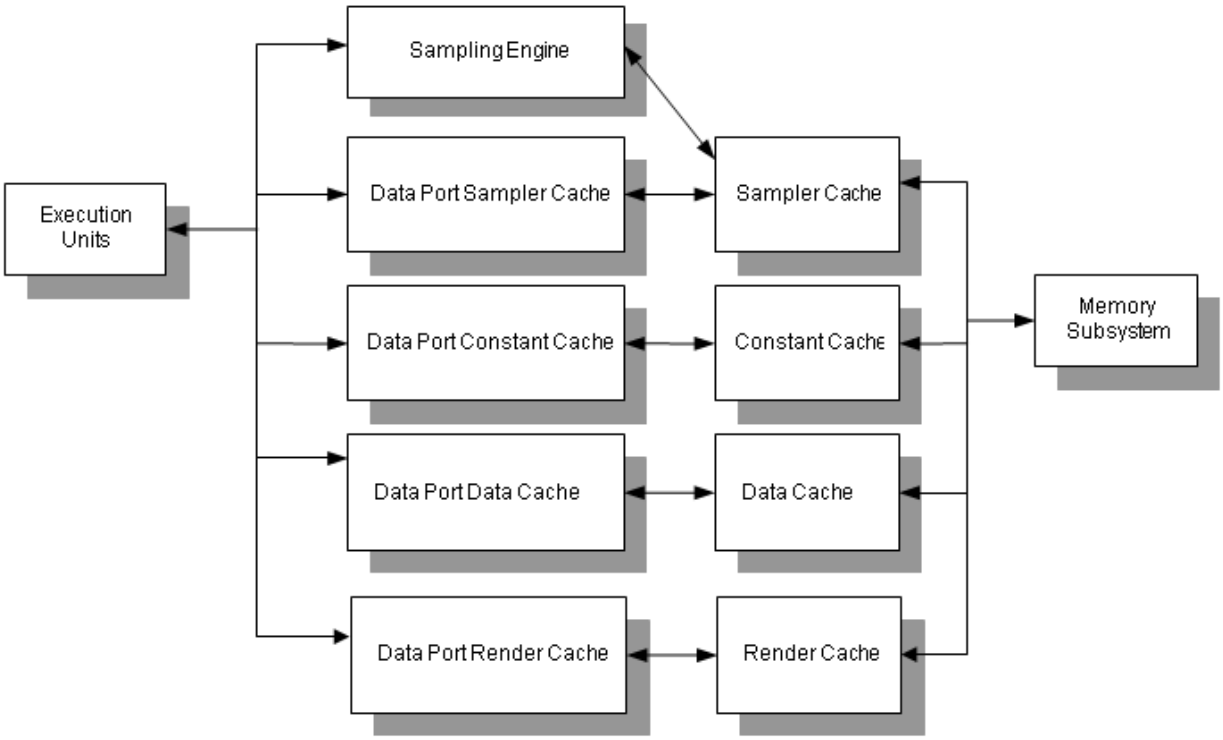
The Data Port provides all memory accesses for the subsystem other than those provided by the sampling engine. These include render target writes, constant buffer reads, scratch space reads/writes, and media surface accesses.

The diagram below shows the three parts of the Data Port (Sampler Cache, Constant Cache, and Render Cache) and how they connect with the caches and memory subsystem. The execution units and sampling engine are shown for clarity.



The kernel programs running in the execution units communicate with the data port via messages, the same as for the other shared function units. The three data ports are considered to be separate shared functions, each with its own shared function identifier.

The diagram below shows the four parts of the Data Port (Sampler Cache, Constant Cache, Data Cache and Render Cache) and how they connect with the caches and memory subsystem. The execution units and sampling engine are shown for clarity.



The kernel programs running in the execution units communicate with the data port via messages, the same as for the other shared function units. The four data ports are considered to be separate shared functions, each with its own shared function identifier.

3.1 Cache Agents

The data port allows access to memory via various caches. The choice of which cache to use for a given application is dictated by its restrictions, coherency issues, and how heavily that cache is used for other purposes.

The cache to use is selected by the shared function accessed.

3.1.1 Render Cache

The render cache is intended to be used for the following surfaces:

- 3D render target surfaces
- destination surfaces for media applications
- intermediate working surfaces for media applications
- scratch space buffers
- streamed vertex buffers

The render cache is a read/write cache that supports 3D render target surfaces, media read/write surfaces, and typed read/write surfaces.

3.1.2 Data Cache

The data cache is a read/write cache that is coherent across the physical instances of this cache. It is intended to be used for the following surfaces:

- constant buffers
- destination surfaces for media applications
- intermediate working surfaces for media applications
- scratch space buffers
- general read/write access of surfaces
- atomic operations
- shared memory for GPGPU thread groups

The data cache can be accessed via the *Data Cache Data Port* shared function, and via the load and store EU messages. Ordering from a single thread is maintained when accessing the data cache using only one of these mechanisms, but is not maintained when using both of these mechanisms from the same thread. In these instances, software must ensure ordering by utilizing write commits and/or waiting for read data to be returned.

3.1.3 Sampler Cache

The sampler cache is a read-only cache that supports both linear and tiled memory. In addition to being used by the sampling engine (via the sampling engine messages), the sampler cache is intended to be used for source surfaces in media applications via the data port. The same application may use the sampler cache via the sampling engine and data port without flushing the pipeline between accesses.

3.2 Surfaces

The data elements accessed by the data port are called “surfaces”. There are two models used by the data port to access these surfaces: surface state model and stateless model.

3.2.1 Surface State Model

The data port uses the binding table to bind indices to surface state, using the same mechanism used by the sampling engine. The surface state model is used when a **Binding Table Index** (specified in the message descriptor) of less than 255 is specified. In this model, the **Binding Table Index** is used to index into the binding table, and the binding table entry contains a pointer to the SURFACE_STATE. SURFACE_STATE contains the parameters defining the surface to be accessed, including its location, format, and size.

This model is intended to be used for constant buffers, render target surfaces, and media surfaces.

3.2.2 Stateless Model

The stateless model is used when a **Binding Table Index** (specified in the message descriptor) of 255 is specified. In this model, the binding table is not accessed, and the parameters that define the surface state are overloaded as follows:

- Surface Type = SURFTYPE_BUFFER
- Surface Format = R32G32B32A32_FLOAT

- Vertical Line Stride = 0
- Surface Base Address = General State Base Address + Immediate Base Address
- Buffer Size = checked only against General State Access Upper Bound
- Surface Pitch = 16 bytes
- Utilize Fence = false
- Tiled = false

This model is primarily intended to be used for scratch space buffers.

3.2.3 Shared Local Memory (SLM)

The shared local memory (SLM) is a high bandwidth memory that is not backed up by system memory. It is enabled by configuring the L3 cache to use a portion of its space for the SLM. One SLM is present in each half slice, and its contents are shared between all of the active threads in that half slice. Its contents are uninitialized after creation, and its contents disappear when deallocated.

The SLM is accessed when a **Binding Table Index** (specified in the message descriptor) of 254 is specified. The binding table is not accessed, and the parameters that define the surface state are overloaded as follows:

- Surface Type = SURFTYPE_BUFFER
- Surface Format = RAW
- Surface Base Address = points to the start of the internal SLM (no memory address is applicable)
- Surface Pitch = 1 byte

Due to the predefined surface state attributes for the SLM, only a subset of the data port messages can be used. This includes the Byte Scattered Read/Write, Untyped Surface Read/Write, and Untyped Atomic Operation messages. In addition, only the data cache data port is supported, the other data ports treat Binding Table Index 254 as a normal surface state access.

Programming Note: Accesses to SLM don't have any bounds checking. Addresses beyond the size (64KB) of the SLM will wrap around.

3.3 Write Commit

For write messages, an optional write commit writeback message can be requested via the Send Write Commit Message bit in the message descriptor. This bit causes a return message to the thread indicating when the write has been committed to the in-order cache pipeline and it is safe to issue another access to the same data with the assurance that it will happen after the first write. A read issued after the write commit ensures that the read will get the newly written data, and another write issued after the write commit will be the last to modify the data. "Committed" does not guarantee that the data has been actually written to the memory subsystem, but only that the write has been scheduled and cannot be passed by another read or write issued subsequently.

If **Send Write Commit Message** is used on a Flush Render Cache message, the write commit is sent only when the render cache has completed its flush to memory. A read issued to another cache after the write commit is received will be guaranteed to retrieve the "new" data that was written before the Flush Render Cache message was issued.

The write commit does not modify the destination register, but merely clears the dependency associated with the destination register. Thus, a simple “mov” instruction using the register as a source is sufficient to wait for the write commit to occur. The following code sequence indicates this:

send r12 m1 DPWRITE; issue write to render cache

mov m1 r3; assemble read message

mov r12 r12; block on write commit

send r13 m1 DPREAD; read same location as write

Prior to End of Thread with a URB_WRITE, the kernel must ensure all writes are complete by sending the final write as a committed write for all non-pixel shaders.

3.4 Read/Write Ordering

Reads and writes issued from the same thread *are* guaranteed to be processed in the same order as they are issued. Software mechanisms must still ensure ordering of accesses issued from different threads.

3.5 Accessing Buffers

There are four data port messages used to access buffers. Three of these are used for both constant buffers and scratch space buffers, the fourth is used by the geometry shader kernel to write to streamed vertex buffers. All of these messages support only buffers, and can use the surface state model as well as the stateless model.

The following table indicates the intended applications of each of the buffer messages.

Message	Applications
OWord Block Read/Write	• constant buffer reads of a single constant or multiple contiguous constants • scratch space reads/writes where the index for each pixel/vertex is the same • block constant reads, scratch memory reads/writes for media
OWord Dual Block Read/Write	• SIMD4x2 constant buffer reads where the indices of each vertex/pixel are different (if there are two indices and they are the same, hardware will optimize the cache accesses and do only one cache access) • SIMD4x2 scratch space reads/writes where the indices are different.
DWord Scattered Read/Write	• SIMD8/16 constant buffer reads where the indices of each pixel are different (read one channel per message) • SIMD8/16 scratch space reads/writes where the indices are different (read/write one channel per message) • general purpose DWord scatter/gathering, used by media
Streamed Vertex Buffer Write	• geometry shader streaming vertex data out

These messages generally ignore the surface format field of the state and perform no format conversion. The exception is the Streamed Vertex Buffer Write, which uses the surface format field to determine only how many channels are to be written. The data contained in each channel is still not converted in any way.

3.6 Accessing Media Surfaces

The Media Block Read/Write message is intended to be used to access 2D media surfaces. The message specifies an X/Y coordinate into the 2D surface as input. Since this message only supports 2D surfaces, the stateless model cannot be used with this message.

3.6.1 Color Processing

The image enhancement color processing pipe, known as IECP or shortly CP. The pipe contains a couple of functions:

- Packer with 422 to 444 converter.
- Skin Tone detection & Enhancement (STDE).
- Color Gamut Compression (CGC) (added for
- TCCE - Automatic Contrast Enhancement (ACE) & Total Color Control (TCC).
- Procamp.
- Color Space Converter (CSC).
- repacker with 444 to 422 converter

Since these functions are performed on per-pixel basis, IECP is integrated in Render Cache Pixel Backend (RCPB). The operation of each functionality could be on/off through the enable bit of each function.

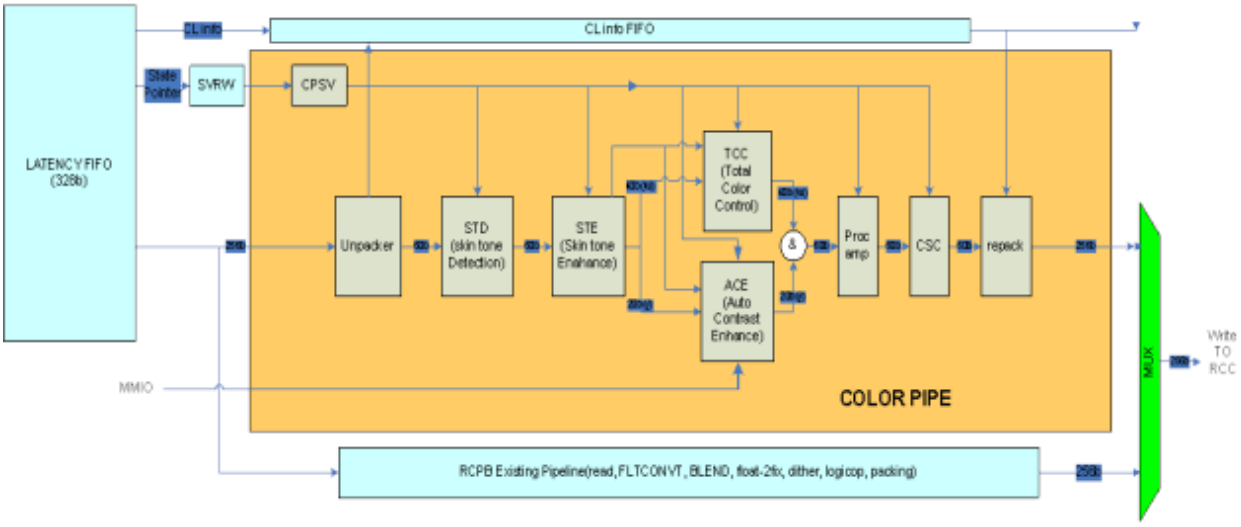
Surface Format Name
R16G16B16A16_UNORM
B8G8R8A8_UNORM
R10G10B10A2_UNORM
R10G10B10A2_UNORM_SRGB
R8G8B8A8_UNORM
R8G8B8A8_UNORM_SRGB
B10G10R10A2_UNORM
B10G10R10A2_UNORM_SRGB
B8G8R8X8_UNORM
R16_UNORM
YCRCB_NORMAL
YCRCB_SWAPUVY
YCRCB_SWAPUV
YCRCB_SWAPY

3.6.1.1 Overview of color processing pipeline

The input message to IECP is 256 bits data from RCPB (contains 2 lines X 2 pixels per clock); along with 256 bits color enhancement state from ISC (State Arbitator for). This **unpacker** converts 256b into two pixels per clock, 36 bits each. In case of 422 inputs the UV are the same for the two pixels in the pair (422 to 444 conversion).

The **Re-packer** (the CSC) delivers 2 pixels in parallel, 36 bits each. The 2x2 message pixels are packed again to 256b and sent with the outgoing message. The 256 bits are organized according to the data type (422/444, 8/16 bits). In case of 422 output, the UV is the average of two adjacent pixels. Also the pipe itself is 12bis/pixel component, in the output message it will be either 8 bit/pixel component (while taking only the 8 MSB) or 16 bits/pixel component (while adding 0000 at the LSB).

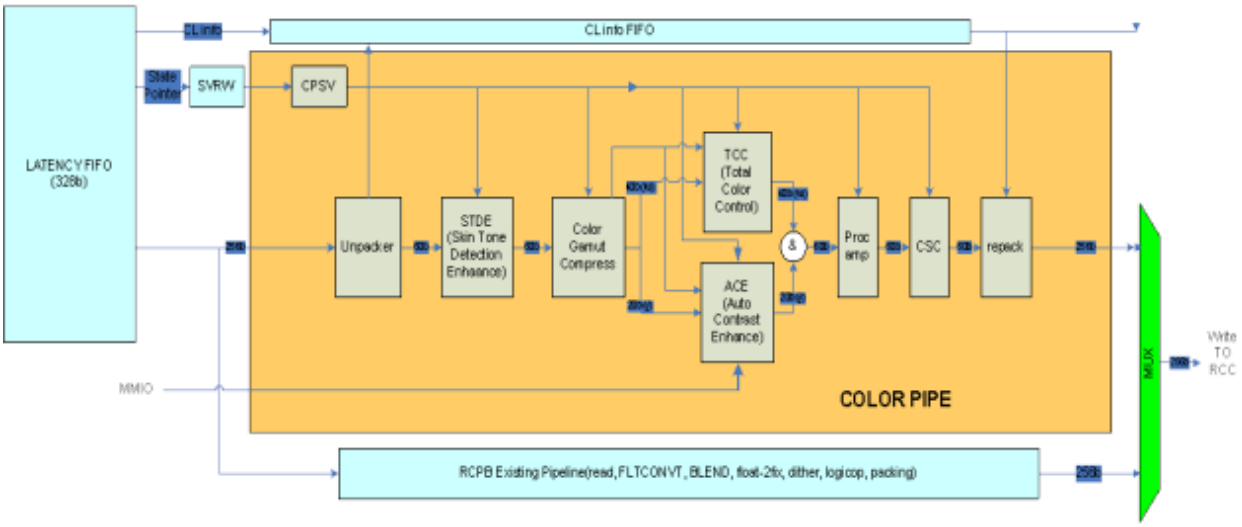
There is statistic information from ACE block (10 bit histogram, 1 bit aoi and 1bit skin pixel) to be sent to VSC (Video Statistic Counter). VSC will process on these data and output the maximum and minimum value of the luma values (Ymax and Ymin) and the number of total skin pixels through MMIO. The Software development can access these data through MMIO and performs the SW part of the color processing algorithms.



The color-processing enables the user to customize visual quality of video playback on the PC platform. The seven functions main goals can be summarized as:

- 422 to 444 converter and the 444 to 422 converter functions enable us some flexibility in the data format input and output.
- Skin Tone Detection/Enhancement function detects skin like color and attempts to change the tone based on user specified parameters to make it more palatable to the user.
- Automatic Contrast Enhancement increases details in dark and bright areas by changing the contrast function in relation to frames luma histogram.
- Total color control allows the user to increase or decrease the color saturation of the six basic colors (Red, Green, Blue, Magenta, Cyan, Yellow).
- Procamp enables the user to control the Brightness, Contrast, Saturation and the Hue.
- Color Space Converter enables the user to convert color space from YUV format to RGB.

The module of color gamut compression is added among STDE and TCC/ACE in the below diagram.



The performance of IVB IECP pipe is improved to 4 pixels per clock for input video of 4:2:2 format and maintained at 2 pixels per clock for input video of 4:4:4 format.

With this performance improvement for 4:2:2 input, there are a couple of addition/modification of IVB pipe to the existing IECP pipe of GT.

The following changes are being done:

Unpacker :

The input format 4:2:2 and 4:4:4 are supported for unpacker. Input video of 4:2:2 format by default is operating at 4 pixels per clock. Input video of 4:4:4 format by default is dependent of an auto UV detection logic to determine if it is operating at 2 pixels per clock, or it could be operating at 4 pixels per clock if detection logic identifies the “true” 4:2:2 format is contained.

A by default auto UV detection logic for the input video of 4:4:4 format tests the MSB 8-bits of the U, V channel value. With the pixel layout below:

Pix_0	Pix_1
Pix_2	Pix_3

The 4:2:2 format is detected if the below conditions are observed

U_pixel_0 = U_pixel_1

U_pixel_2 = U_pixel_3

V_pixel_0 = V_pixel_1

V_pixel_2 = V_pixel_3

When the 4:2:2 format is detected from the input video of 4:4:4 format, the performance improved mode at 4 pixels per clock is applicable. Thus, the average of the U, V values in horizontal direction is used as the U, V output to the remaining pipe. If UV detection logic cannot detect the input as 4:2:2 format, it will operate as 4:4:4 mode.

There is a flag bit as the state parameters to force the operation in 4:2:2 mode (Φορχεδ422_φορ_444) when the input video is of 4:4:4 format. In this case, the horizontal average of U, V pixel values is taken as the output values to the remaining pipe. A flag bit of state parameter, Φορχεδ444_φορ_444, is provided to ensure the 4:4:4 operation for the input of 4:4:4 format. There is also a flag bit of state parameter to force the operation in 4:4:4 mode (Φορχεδ444_φορ_422) when the input video is of 4:2:2 format. In this case, U, V pixel values are horizontally replicated as the output values to the remaining pipe. It is 2

pixel/clk when Φορχεδ444_φορ_444 or Φορχεδ444_φορ_422 is enabled and is 4 pixel/clk when Φορχεδ422 is enabled..

In 4:2:2 mode, all the four Y-channels and two U, V channels in the 422 format as is to STD. No sequencing. An example of the values Y_p0, Y_p1, Y_p2, Y_p3, U_p01, V_p01, U_p23, V_p23 are the pixel values being sent to the pipe.

In 4:4:4 mode, 2 pixels (two Y, U, V, A channels) are forwarded per clock to STD. Sequencing is done across two clocks. To keep the pipe with minimum changes the corresponding channels are sent like below example:

Y_p0, X, Y_p1, X, U_p0, V_p0, U_p1, V_p1 are the pixel values which are sent to the pipe. Where “X” is don’t care.

Skin tone detection /enhancement

For both 4:2:2/4:4:4 the detection in UV space is done for the two UV values independently. i.e. both the rectangle and diamond std factor are calculated in similar way for both 4:2:2/4:4:4 format. Satnew and HueNew are also calculated based on the two UV values based out of the PWL.

For the Y-factor calculation is done for 4 pixels. In case of 4:2:2 all four pixels are valid, but for 4:4:4 only two pixels are valid. The factor calculated on UV space in step 1 is replicated for the horizontal pixels and we get the effective four STD factors out of this step.

In VY-factor calculation, the V is replicated for the horizontal pixels i.e. pixel0 and pixel1 will use the same V pixel01 value and pixel2 and pixel3 will use the same V pixel23 value. We will get four STD factors out of this stage. For 4:2:2 mode, two resultant STD factors can be produced for skin tone enhancement or later stage based on the horizontal minimum, maximum, or average of the STD factors, which can specified via the state parameters ΣΤΔΜαξ, ΣΤΔΜιν, ΣΤΔΑϖε

In skin tone enhancement module, based on Y-channel, MVdark and MVbright are calculated for 4 pixels. Then for 4:2:2 mode, the horizontal minimum, maximum, or average of MVdark values, which could be specified via the state parameters ΜςΔαρκΜαξ, ΜςΔαρκΜιν, ΜςΔαρκΑϖε, can be used to effect the new Satnew and Huenew values. In 4:4:4, we use the corresponding MVdark/bright and discard the other i.e. no averaging.

Delta U/V : For 4:2:2 mode, the two resultant STD factors froom step iii are used to derive Delta U/V. For 4:4:4 mode, the corresponding pixel STD factor is used. Out of this stage we always get two Delta U/V values.

Finally only two updated U/V values using the above delta U/V values come out of the STDE pipe.

Color Gamut Compression:

For 4:2:2 mode, only two U/V values are received, and thus two Hue index are calculated and only two vertex point (Lv, Cv) lookups is checked.

For both 4:2:2/4:4:4, there are 4pixel Y channels so 4 parallel out pixel detection occurs based on the vertex points(Lv, Cv), U, V values. Four SF (scaling factor) are produced.

In 4:4:4 mode, only two of the above are valid and the rest two are discarded. Only the corresponding SF is used to calculate the new U/V values.

In 4:2:2 mode, two SF values are produced based on the horizontal minimum, maximum, or average value of SF, which could be specified via the state parameters ΣΦΜαξ, ΣΦΜιν, ΣΦΑϖε. 4 updated Y values and 2 U/V values are produced out of Gamut compression

TCC works on 2 UV values received from gamut compression and is 2wide for 4:2:2 mode.

ACE works on 4 Y pixel received from Gamut compression and is 4 wide

Procamp optimizes and has 4pixel Y and 2pixel UV values and output the same in 4:2:2 mode

CSC optimized the UV portion and then uses the same result out for the 2 horizontal pixels and adds the result to the individual Ypixels. From CSC we will get 4pixel out for 4:2:2 mode.

Repacker pack sends the data 4 pixels for 4:2:2 mode as is to RCPB, but in the case of 4:4:4 it will combine the 2pixels received from CSC across 2 clocks before outputting the 4pixels to RCPB.

3.6.1.2 Skin Tone Detection/Enhancement (STD/E)

The STD/E unit, composed of the Skin Tone Detection (STD) and Skin Tone Enhancement (STE) units, is part of color processing pipe located at the Render Cache Pixel Backend (RCBP).

The main goal of the STD/E is to reproduce the skin colors in a way that is more palatable to the observer, and by that to increase the sensed image quality. It may also pass indication of skin tones to the TCC and ACE.

The STD unit detects the skin like colors and passes a grade of skin tone color to the STE. The STE modify the saturation and Hue of the pixel. Both the STD and STE are per-pixel basis. The input pixels are required to be on the YUV space.

The skin tone detected factor will be recorded as a 5-bit number and it will be passed to the module of ACE and TCC to indicate the strength of skin tone likelihood.

3.6.1.2.1 STD

The STD operates on digital images in the YUV color space. In these space the skin-tone region is represented by the ellipse in the (U,V) subspace (chroma components), by a trapeze membership function in the Y direction (luma component) and by a piece-wise linear classifier in the (V,Y) subspace.

U,V data is transformed into Hue and Saturation space through shifting and rotation

Step 1: shift rectangle

U_center = U – Y_μιδ_

V_center = V - ς_μιδ_

Step 2: rotate rectangle

Sat = -(U_center * Χοσ - V_center * Σιν)

Hue = -(U_center * Σιν + V_center * Χοσ)

Where: Sin = Σιν(□□, and Cos = Χοσ(□).

Rectangle skin-tone measure determination

Skin-tone detection is described by a continue score on the [0,1] range, where a level 0 means not a skin (SkinToneFactor = 0) , and a level 1 (SkinToneFactor = 1) means a full skin. In between, (0,1) region, we have partial skin-tone detection. This partial skin-tone detection is controlled by a margin parameter, which will be denoted by "ΗΣ_μαργιν". The SkinToneFactor is expressed by 5 bits, and thus have values in the [0,31] range.

```
if( abs(Sat) <= SatMax && abs(Hue) <= HueMax)
{
  if(HS_margin >= 5)
     {
       Sat_Factor = (Sat_max-abs(Sat)) / 2(HS_margin - 5);
```

```
       Hue_Factor = (Hue_max-abs(Hue)) / 2(HS_margin - 5);
     }
  else
     {
 Sat_Factor = (Sat_max-abs(Sat)) * 2(HS_margin - 5);
       Hue_Factor = (Hue_max-abs(Hue)) * 2(HS_margin - 5);
     } //end of if(HS_margin >= 5)
 }
else
{
       Sat_Factor = 0;
       Hue_Factor = 0;
} //end of if( abs(Sat) <= SatMax && abs(Hue) <= HueMax)
Sat_Factor = min(Sat_Factor,31);
Hue_Factor = min(Hue_Factor,31);
Rectagle_SkinToneFactor = min(Sat_Factor, Hue_Factor);
```

Rhombus skin tone detection determination

Similar to the rectangle skin-tone measure, a rhombus-margin (Διαμονδ_μαργιν) is introduced. This introduces a new rhombus region, inner to the original rhombus, in a similar happened with the rectangle. There are two regions such that: outside the original rhombus a SkinToneFactor = 0 (not a skin); inside the inner rhombus SkinToneFactor = 1 (full skin); in between 0 < SkinToneFactor < 1 indicating a partial skin-tone detection. As in the rectangle case, the SkinToneFactor is expressed by 5 bits, and thus have values in the [0,31] range.

A Diamond SkinToneFactor calculations algorithm is:

```
Dist = abs(Sat – Diamond_du) + Diamond_alpha(1/tan(□)) * abs(Hue –
Diamond_dv);
//outside the diamond
if(Dist >= Diamond_TH)
{
 D_Factor = 0; //the point is out of the large rhombus
}
else if(Dist < (Diamond_TH - Diamond_margin))
 {
   D_Factor = 31; //the point is inside the inner rhombus
 }
  else //the point is inbetween the outer and the inner rhombuses
  {
   if(Diamond_margin >= 5)
    {
     D_Factor = (Diamond_TH - Dist) / 2(Diamond_margin - 5);
    }
    else
    {
     D_Factor = (Diamond_TH - Dist) * 2(Diamond_margin - 5);
    } // end of if(Diamond_margin >= 5)
```

```
} // if(D < (Diamond_TH - Diamond_margin))
          Diamond_SkinToneFactor = D_factor;
```

Finally the level of the skin-tone detection in the (U,V) subspace is given by:

UV_SkinToneFactor = min(Rectangle_SkinToneFactor, Diamond_SkinToneFactor);

Detection in Y direction

The detection based on the Y-values, is given by a piece-wise linear membership function, which is defined with 4 points (Ψ_ποιντ_ξ) (x=1, 2, 3, and 4).

```
if(Y >= Y_Point_0 && in_Y < Point_1)
  Y_Factor = (Y - Y_Point_0) * Y_Slope_1;
else if(Y >= Point_1 && Y < Point_2)
       Y_Factor = 31;
    else if(Y >= Point_2 && Y < Point_3)
            Y_Factor = (Point_3 - Y) * Y_Slope_2
         else
            Y_Factor = 0;
```

At the end of the process a double (min,max) clipping is applied:

Y_Factor = min(31,max(Y_Factor,0));

The final Skin-Tone detection is is given by:

SkinToneFactor = min(UV_SkinToneFactor, Y_factor);

Detection in the VY plane (3D-like DTD)

The operation of the detection in VY plane is particularly enabled by ςΨ_ΣΤΔ_Εναβλε bit

It is known that the application of a three-dimensional (3D) classifier in the (Y,U,V) space, instead of a two dimensional (2D) skin-tone detector in the (U,V) plane, is resulted in a better detection. Implementation complexity of the full 3D classifier is too high, and forces us to approximate the classifier by more simple, but useful methods. Skin-tone data distribution implies (it is almost convex, and has a predominate directions) that the 3D classifier could be approximated by the intersection of the three 2D classifiers in (U,V), (U,Y), and (V,Y) subspaces. The (U,V) subspace is the most important one it is already approximated by the ellipse, as was described previously. Our study implies that the (V,Y) subspace is the next most important one. Although the (U,Y) space carries the STD information, it is heavily redundant and has the reduced importance.

Thus the approximation of 3D classifier is an intersection of (U,V) and (V,Y) two-dimensional classifiers. The (V,Y) classifier is given by two piece-wise linear functions (PWLF), Each PWLF is composed of four straight segments. Each segment is described by the three parameters (Point, Slope and bias). Thus a single PWLF (lower or upper) is described by 12 parameters (4 points, 4 biases, 4 slopes).

The parameters of lower part are: 4 point PxL (x=0, 1, 2, 3), 4 bias BxL (x=0, 1, 2, 3) and 4 slope SxL (x=0, 1, 2, 3).

The parameters of upper part are: 4 point PxU (x=0, 1, 2, 3), 4 bias BxU (x=0, 1, 2, 3) and 4 slope SxU (x=0, 1, 2, 3).

There is Programming Restrictions to specify the parameters

The points must be in the non-decreasing order: P0 <= P1 <= P2 <= P3.

The parts must be continues on they ends. Thus the user:

(a). must set: $P0_L = P0_U$ (continuity at the leftmost points).

(b). must care for continuity at the rightmost points.

Margin for the detection in the VY plane (3D-like DTD)

Vertical margins of each part were introduced to provide a “soft” continuous detection over the classifier boundaries. There are two parameters defined

$\mathrm{Marginς\Psi\Lambda}$ - the margin of the lower (blue) part.

$\mathrm{Marginς\Psi Y}$ - the margin of the upper (red) part.

Consider a pixel with coordinates (Y,V) = ($P2_L$,V_1),. This pixel has a Y coordinate exactly as of the point $P2_,$ and a V coordinate equal V1. For this pixel the detection relative to the Lower Part will be:

detL = Min (Max ((V1 – B2L) / MarginVYL, 0), 1)

The identical calculations are made for the Upper Line as well:

detU = Min (Max ((VU – V1) / MarginVYU, 0) , 1)

Where:

det_L - is a detection relative to the Lower Part

det_U - is a detection relative to the Upper Part

V_U - is a V value of the Upper PWLF correspond to the $Y=P2_L$

B_U - is a V value of the Lower PWLF correspond to the $Y=P2_L$

The inverse operation of (1/ MarginVYL), and (1/ MarginVYU) is specified by the parameters $\mathrm{INV\_MAP\Gamma IN\_\Psi\Lambda}$ and $\mathrm{IN\varsigma\_Marginv\_\Psi Y}$.

Both detections (det_L, det_U) are reduced to 5 bit representations, and the overal detection in the (V,Y)-plane is given by:

det_VY = min(det_L, det_U)

The final Skin-Tone Detection is given by the minimum of the previously calculated STD in the (U,V)-plane (9), and the current one:

SkinToneFactor = min(SkinToneFactor, det_VY)

This value is represented with 5 bits, and has a [0,31] range.

3.6.1.2.2 STE

The enhancement step is performed on the pixels which were detected as the skin-tone pixels only by the previous (STD) step. This step is divided into two sub-steps: saturation correction enhancemen and hue correction enhancement

STE – Saturation Correction Enhancement

The enhancement is performed by the transformation $Sat_{New} = F_{Sat}(Sat_{Old})$, which is realized by the piece-wise linear function (PWLF) with a 4 straight segments.

The parameters of this PWLF are:

- Points:

SATP0 = -SatMax

ΣΑΤΠξ (x=1,2,3) – defined by the user

SATP4 = SatMax

- Biases:

SATB0 = -SatMax

ΣΑΤΒξ (x=1,2,3) – defined by the user

SATB4 = SatMax

- Slopes:

ΣΑΤΣξ (x=0,1,2,3) – defined by the user

There is Programming Restrictions to specify the parameters

The point Sat = $-Sat_{Max}$ maps to itself: $(-Sat_{Max})$ □ $(-Sat_{Max})$.

The point Sat = Sat_{Max} maps to itself: (Sat_{Max}) □ (Sat_{Max}).

The correction function is continuous.

The correction function is non-decreasing.

Sat_{Old}

Sat_{New}

$(-Sat_{Max}, -Sat_{Max})$

(Sat_{Max}, Sat_{Max})

Identity

transformation

Fig.. General form of the Saturation correction PWLF.

Correction Function

STE – Hue Correction Enhancement

The enhancement is performed by the transformation $Hue_{New} = F_{Sat}(Hue_{Old})$, which is realized by the piece-wise linear function (PWLF) with a 4 straight segments.

The parameters of this PWLF are:

- Points:

HUEP0 = -HueMax

ΗΥΕΠξ (x=1,2,3) – defined by the user

HUEP4 = HueMax

- Biases:

HUEB0 = -HueMax

ΗΥΕΒξ (x=1,2,3) – defined by the user

HUEB4 = HueMax

- Slopes:

HYEΣξ (x=0,1,2,3) – defined by the user

There are Programming Restrictions to specify the parameters

The point Hue = -HueUE$_{Max}$ maps to itself: (-Hue$_{Max}$) □ (-Hue$_{Max}$).

The point Hue = Hue$_{Max}$ maps to itself: (Hue$_{Max}$) □ (Hue$_{Max}$).

The correction function is continuous.

The correction function is non-decreasing.

Hue$_{Old}$

Hue$_{New}$

(-Hue$_{Max}$,-Hue$_{Max}$)

(Hue$_{Max}$,Hue$_{Max}$)

Identity

transformation

Fig. General form of the Hue correction PWLF.

Correction Function

STE – Skin Type Correction Enhancement

The operation of this mode is enabled by the control parameter Σκιν_τψπεσ_εναβλε.

The Saturation and Hue enhancement processes are basic STE procedure. The advanced mode to adjust the enhacement based on the skin type define the second set of the Sat and the Hue enhancement parameters, which has an identical structure as the previous one (Points, Biases, Slopes) but having different values. We will refer one set of parameters to the Bright skin (Bs), and the other to the Dark skin (Ds). Each pixel is referred as belongs to the Bright, the Dark, or to the both skin types with a different membership values. The Dark/Bright skin classifier is defined by the two parameters: Σκιν_τψπεσ_τηεση, and Σκιν_τψπεσ_μαργιν. It works on the luma (Y) values.

The parameters related are

Points:

HYEΠξ_ΔAPK (x=1,2,3) – defined by the user

ΣATΠξ_ΔAPK (x=1,2,3) – defined by the user

Biases:

HYEBξ_ΔAPK (x=1,2,3) – defined by the user

ΣATBξ_ΔAPK (x=1,2,3) – defined by the user

Slopes:

HYEΣξ_ΔAPK (x=0,1,2,3) – defined by the user

ΣATΣξ_ΔAPK (x=0,1,2,3) – defined by the user

For the luma value Y, we define

Y_A = skinTypesThesh - skinTypesMargin

Y_B = skinTypesThesh + skinTypesMargin

MV_{Dark} = 1 , if $Y < Y_A$

= 0, if $Y > Y_B$

= $(Y_B - Y) / (2*$ skinTypesMargin), if $Y_A <= Y <= Y_B$

$MV_{Bright} = 1 - mV_{Dark}$

Where MV_{Dark} and MV_{Bright} are the membership value of the Dark and Bright skin (belongnes). *(Note: the membership values represent the "belongness" of the skin pixel to the different skin types).* Also, we mark that the inversee operation of 1/(2* Skin_types_margin) wil be specified by the parameter INς_σκιν_τψπε_μαργιν.

In previous sections the procedure for the calculation of the Sat_{New} and Hue_{New} values was described. We calculate these values for the two skin types and thus get $Sat_{New\ B}$, $Hue_{New\ B}$, and $Sat_{New\ D}$, $Hue_{New\ D}$ values , where and subscribes "B" and "D" stands for the Bright and the Dark skin types, respectively. (In this case, the parameters with "_DARK" extension are used to work out $Sat_{New\ D}$ and $Hue_{New\ D}$, and the other set of the parameter could be reloaded with the parameters to work out $Sat_{New\ D}$, $Hue_{New\ D.}$)The final values of the enhanced pixel will be given by:

$Sat_{New} = MV_{Dark} * Sat_{New\ D} + MV_{Bright} * Sat_{New\ B}$

$Hue_{New} = MV_{Dark} * Hue_{New\ D} + MV_{Bright} * Hue_{New\ B}$

STE – (Sat, Hue) to (U, V) transformation

In prior session,, the (U,V) □ (Sat,Hue) transformation was proceeded by the two steps: σηιφτ, and ροτατιον. Thus the backward transformation should be done in the inverse order: a ροτατιον, and then a σηιφτ.

// Rotate back:

U_Center_New = (Sat_New * Cos) + (Hue_New * Sin)

V_Center_New = -(Sat_New * Sin) + (Hue_New * Cos)

// Shift:

U_New = U_Center_New + U_mid

V_New = V_Center_New + V_mid

The (U_new, V_new) are the (Sat_{New}, Hue_{New}) values in transformed to the original (U,V) coordinates.

Let denote the original (U,V) values of the pixel by (U_in,V_in). Thus the difference between the corrected and the original values are:

DU = U_new – U_in

DV = V_new – V_in

The final correction must be depended by the *SkinToneFactor* value, and therefore DU, DV are corrected by:

DU = DU * STD_ Likelihood_Factor

DV = DV * STD_ Likelihood_Factor

Where:

STD_ Likelihood_Factor = (SkinToneFactor / 32)

(Remember that the 0 <= SkinToneFactor <= 31).

After the DU and DV were corrected by the STD likelihood factor, the final (U,V) will be calculated by:

U = U_in + DU

V = V_in + DV

3.6.1.2.3 STD Score Output

This mode outputs the STD score, which is controlled by the state bit “Output STD Decisions” instead of the pixel values. In this mode, the STD should be enabled and other functions in the IECP after STDE in the pipe should be disabled. Only ACE can be enabled to collect the histogram of the STD score values.

The output when “Output STD Decision” is enabled should be as follows:

Y = 0x7FF + + (STD_Score <<6)

U = 0x7FF

V = 0x7FF

In this mode, a histogram of skin tone distribution can be obtained in ACE module, and a special ACE PWLF curve (step function) can be configurated to produce a bi-level picture to illustrate the pixels based on the level of skin tone detection.

3.6.1.3 Adaptive Contrast Enhancement (ACE)

The Automatic Contrast Enhancement (ACE) is a part of the color processing pipe, which located at the render cache in the RCPB block.

The main goals of the ACE is to improve the overall contrast of the image, and emphasizing details when relevant (such as in dark areas).

The ACE algorithm analyzes the image, and consequently changes contrast of the image according to its characteristics. It works in YCbCr color space, where analysis and changes are performed over the Y component. The result of ACE is a 1d (1 dimension) look up table (1D LUT) operating on Y. The ACE follows the skin tone enhancement module in the pipe.

The ACE is receiving skin information from the STD block. When the frame includes skin the affect of the ACE is reduced in the skin area.

The ACE operation is divided into three stages:

- Collecting information on Y and building the picture histogram. (Hardware)
- Analysis on the collected data. (Software/Kernel)
- Modification of the Y component. (Hardware)

The major steps of ACE can be divided into the following steps and depict in the below diagram.

1. Histogram calculation of the Y values.
2. Limiting extremely large histogram’s bins.
3. Calculate the Image’s gray level mean value (Ymean).
4. Calculate the Image’s “Dark Factor” by the Ymean and external transfer function.
5. Find the PWLF anchor input and output points according to the “Portion Values” and the “Destination Points” of the Bright and the Dark images.
6. Find the PWLF anchor Input points by the blending of the Dark and Bright anchor input points, according to the Dark Factor calculated previously.
7. Find the PWLF anchor Output points by the blending of the Dark and Bright anchor output points, according to the Dark Factor calculated previously.
8. Limit Slopes between the anchor points. This stage’s output is the current’s image ACE PWLF.

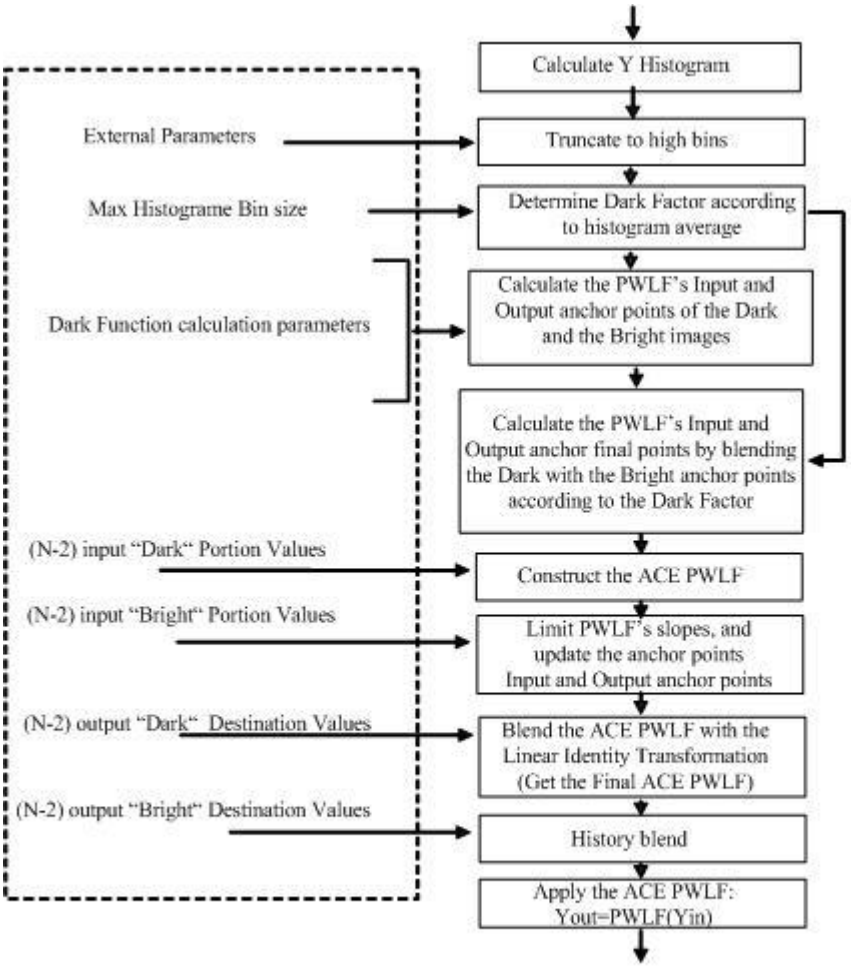
9. “Soften” the ACE PWLF by blending I with the Identity Transformation.
10. Blend the current PWLF with the PWLF of the previous image (History blend).
11. Apply the final PWLF, and get the Yout values.

Note: Step 1 & step 11 are done in HW and steps 2-10 are done in software.

The main ACE goals are overall contrast improvement, and details emphasizing. ACE algorithm generates a Piece-wise Linear Function (PWLF), and the final gray values, Yout, are calculated by Yout = PWLF(Yin).

The HW compares the input pixels to the σκιν_τηρεσηολδ to determine if the target pixel is a skin pixel or not. It operates on all of the input pixels if the Φυλλ_ιμαγε_ηιστογραμ flag is defined. (to ignore the AOI flag). HW output the histogram of luma pixel value to VSC, and at VSC, the maximum and minimum value of luma pixels (Ymax, Ymin) ans the number of skin pixels is determined to be made available to the software development via MMIO register.

An eleven-segment (12 points) was established to implement PWLF via the state parameters (Points: Ψμιν, Ψ1–Ψ10, Ψμαξ, Bias: B1 □ B10, Slope: Σ0–Σ10).



3.6.1.4 Total Color Control (TCC)

The TCC allows users to choose different grades of saturation for each of the six basic colors (Red, Green, Blue, Magenta, Yellow and Cyan) in order to custom the color scheme. The TCC algorithm operates on the UV-color components in the YUV color space. It operates in the pixel-wise mode, without considering any neighborhood information.

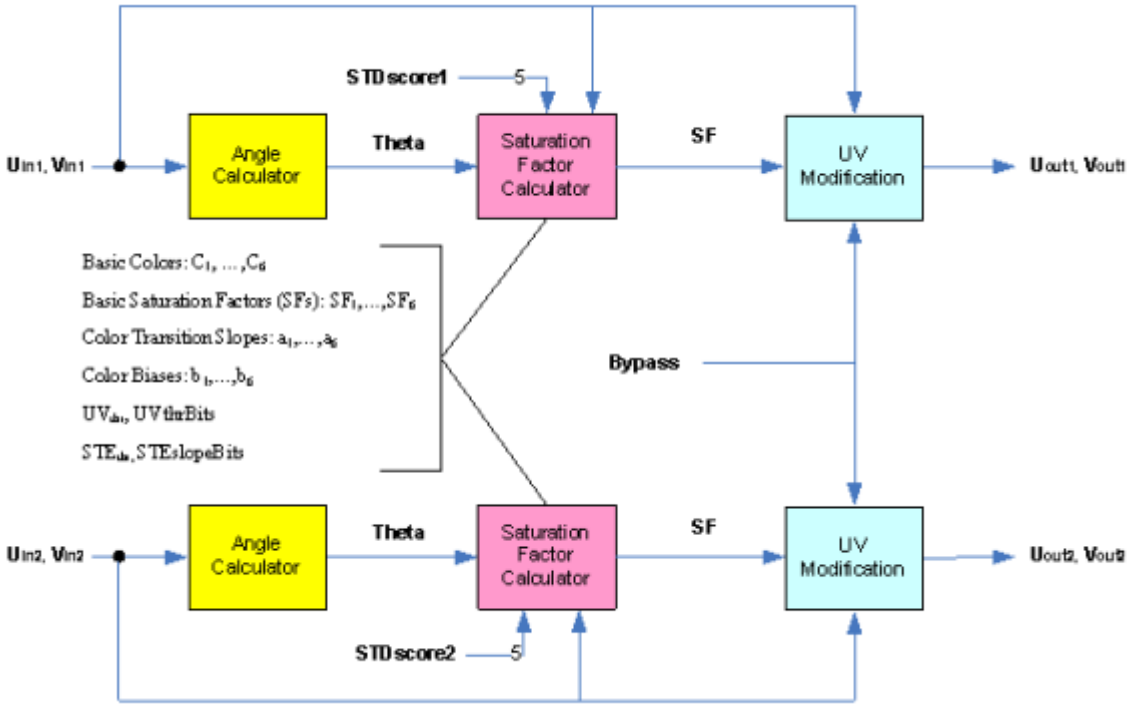
Its input is:

- U,V color components (10 bit)
- Skin-tone detection value (5 bit)
- External control parameters

Its output is the new U, V values (10 bit).

The motivation to implement this block in HW is to reduce the power of the system and therefore the battery life.

The pixel TPT (throughput) is two pixels per clock. The pipeline works in YUV formats only – 10bit pixels. The TCC block is control by state only and does not require any memory access. The TCC block runs at the same frequency of the existing RCPBunit.



There are two paths in parallel to support the requirement of two pixels per clock. Valid out is a signal which high when the pixels are valid.

The TCC block includes three sub blocks.

Angle_calculator

This block receive pixel U and V and perform division of abs|v| by abs|u| using Divider ROM with pipeline.

The division result is used to calculated arctan of the V/U. This result is used to calculate the angle called θ, by using approximation equation. This angle is defined as a 10bit.

To simplify this calculation the “arctangent” function is approximated in the [0,45]° region by the second order polynomial:

□□= arctan(x) = -0.2880x² + 1.0797x - 0.005; (0 <= x <= 1)

The resulted □ is given in radians with the maximal error of 0.005 rad. (0.286 deg.) This approximation is calculated by the minimizing the mean squared error (mse) between the actual “arctan” function, and its polynomial approximation, and thus represents the optimal mse-approximation in the [0,□/4] region. The □ for the all regions is calculated by:

□0.25□; for region I, (0 <= x <= 1),

□/2 – □0.25□; for region II, (1 < (V/U) < infinity)

□/2 + □0.25□; for region III, (-infinity < (V/U) < -1)

□ = □ - □0.25□; for region IV, (-1 <= (V/U) < 0)

□ + □0.25□; for region V, (0 <= (V/U) <= 1)

3□□□ - □0.25□; for region VI, (1 < (V/U) < infinity)

3□□□ + □0.25□; for region VII, (-infinity < (V/U) < -1)

2□□- □0.25□; for region VIII, (-1 <= (V/U) < 0)

Whereas x = (V/U), and the □0.25□ is given by the above equation.

Saturation_Factor_Calculator

This block is using the angle θ, locate where it is in the color wheel, find the appropriate base colors and calculate the proportional distance from the adjacent base color. The result called α. Alpha (α) represent the distance from the two relevant base color.

Calculate the saturation by using the appropriate user parameters. The result is the Saturation factor. This block considering also the threshold and the maximum UV values, and considering also correction for gray colors to minimize the possible noise. In addition the saturation skipping doing saturation when the color is skin and doing alpha blending according the skin factor called STDscore.

This block requires several external parameters such:

ΒασεΧολορ1,□, ΒασεΧολορ6 – Six basic user defined colors.

ΣατΦαχτορ1,□, ΣατΦαχτορ6 – Six basic saturation change user defined factors.

ΧολορΤρανσιτΣλοπε12, □..ΧολορΤρανσιτ61 – Six calculation result of 1/(BaseColorX – BaseColorY)

ColorBias1,…, ColorBias6 – Six color bias.

STDscore – Skin-tone Detection score (from STD/E).

The result of SF is a number of 8bits.

There are four major steps to derive the saturation factor.

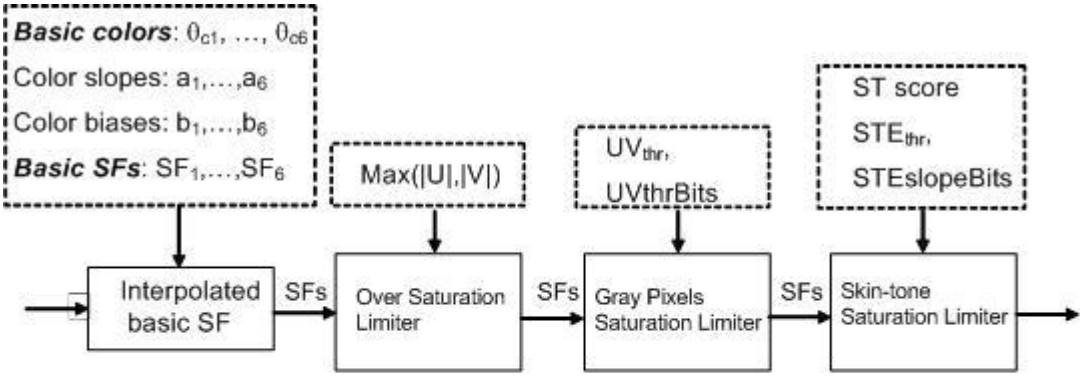


Figure. Calculation of the Saturation Factor (SF).

θ – current pixel's color as calculated by the Eq. (3)

Lined boxes show additional data used by each block.

SFs_i – SF after the step "i".

SFs_4 is the SF_{final}.

The Interpolated Basic SFs_1

With the calculated angle □, which lies in the [□Ci, □Ci+1] interval, the Interpolated Basic SFs_1 will be:

$SFs_1 = (1-□□)ΣατΦαχτορ_i + □ΣατΦαχτορ_{i+1}$

Whereas □ is calculated by:

□ = Min{Max[(□□□□ΒασεΧολορ_i)*ΧολορΤρανσιτΣλοπε_i – ΧολορΒιασ_i, 0], 1}

Over Saturation Limiter SFs2

Over Saturation Limiter block is used to avoid saturation boosting of the already high saturated pixels. The SFs_2 is calculated by:

SFs_1 , *for (SF_1 <= 1)*

$SFs_2 = 1 + (SFs_1 – 1)(MaxColor – UV_{max})/MaxColor$, *for (1 < SF_1 <= 2) AND (UV_{max} <= ΥςΜαξΧολορ)*

1, *for (UV_{max} > ΥςΜαξΧολορ)*

Where the UV_{max} = max(|U|,|V|), and ΥςΜαξΧολορ is an external parameter which in the case of YUV color space is equal to 448 in 10bit representation. Τηε Ινϖ_ΥςΜαξΧολορ was used for the inverse calculation of 1/UVMaxColor.

Note: The last condition (UV_{max} > **UVMaxColor**) is associated with the illegal colors, and usually hasn't to appea (Can this be OK for wide gamut mapping?).

GrayPixels Saturation LimiterSFs3

This block limits the saturation of the almost gray pixels. The reason for this limiter is to prevent the noise amplification by the Saturation increase process. The result of this block is:

$SFs_3 = 1 + dSF * CLF$

Where:

$dSF = SFs_2 – 1$;

And the CLF is called Color Limiting Factor and ranges from 0 to 1. The calculation of the CLF is given by:

= 1; *for (SFs_2 <= 1) AND (any UV_{max})*

CLF = 0; *for (UV_{max} <= Υς_Τηρεσηολδ)*

= (UV_{max} – UV_Threshold) / $2^{UV\_Threshold\_Bits}$; for (UV_Threshold < UV_{max} < (UV_Threshold+$2^{UV\_Threshold\_Bits}$))

Skin-tone Saturation LimiterSFs4

The last block effects TCC strength operation of the Skin-tone pixels. Uncontrolled enhancement of the skin pixels could lead to appearing of artifacts and to undesired results. The final SFs_4 is calculated by a linear blending:

$SFs_4 = (128*STE_{factor} + (256 - STE_{factor}) SFs_3) / 256$

Where the STE_{factor} is called Skin Tone factor and is calculated by:

$diff = (STD_{score} – ΣTE\_Τηρεσηολδ) * 2^3$

Note: the STD_{score} (from STD) and the **STE_Threshold** are presented with 5 bits. The multiplication by 2^3 is in order to raise the "diff" to 8 bits.

STE_{factor} = Min {Max [(diff * $2^{STE\_SlopeBits}$), 0], 255}

The STD_{score} is a result of the Skin-tone Detection module. It is represented with 5 bits, where the values 0 and 31 mean no skin-tone, and full skin-tone detection, respectively. The STE_{factor} is given by 8 bits, where the value 256 represents the number 1.

It is evident that for the high values of STE_{factor} the resulted SFs_4 is close to 1, which means a weak TCC action of this pixel (SFs_4 = 1 actually means TCC is off).

Υς Μοδιφιχατιον – The input pixels are multiple by the saturation factor. The results are the output pixels.

SF_{final} is the final saturation factor which actually resulted from the forth SFcalculation block:

$SF_{final} = SFs_4$

The calculation of the U_{new}, and V_{new} output values. They are calculated below:

$U_{new} = U * SF_{final}$

$V_{new} = V * SF_{final}$

Whereas (U,V) are the original input color components,

Because these pixels are represented in the unbiased form, which is the result of substraction of the value 512 from the original [U,V] values, the final [U_{out}, V_{out}] values are given by:

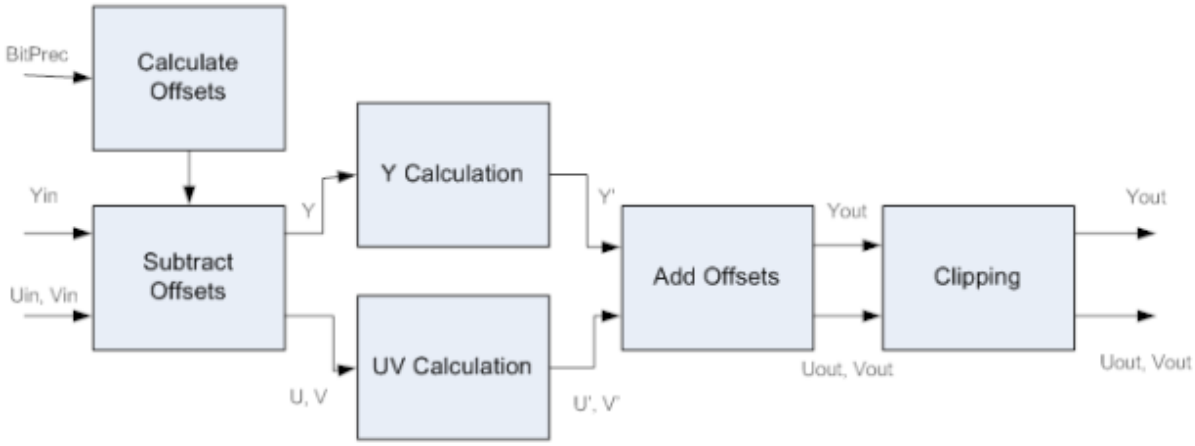
$U_{out} = U_{new} + 512$

$V_{out} = V_{new} + 512$

This is the final TCC output represented with 10 bits.

3.6.1.5 ProcAmp

The PROCAMP block modifies the brightness, contrast, hue and saturation of an image in YCbCr color space (or similar).



The algorithm itself uses 8-16 bits per color.

Y Processing: 256 is subtracted from the Y values to position the black level at zero. This removes the DC offset so that adjusting the contrast does not vary the black level. Since Y values may be less than 256, negative Y values should be supported at this point. Contrast is adjusted by multiplying the YUV pixel values by a constant. If U and V are adjusted, a color shift will result whenever the contrast is changed. The brightness property value is added (or subtracted) from the contrast adjusted Y values; this is done to avoid introducing a DC offset due to adjusting the contrast. Finally the value 64 is added to reposition the black level at 256. The equation for processing of Y values is:

Y' = ((Y-256) x C) + B + 256,

where C is the Contrast value and B is the Brightness value.

UV Processing: 2048 is first subtracted from both U and V values to position the range around zero. The hue property is implemented by mixing the U and V values together:

U' = (U-2048) x Cos(H) + (V-2048) x Sin(H)

V' = (V-2048) x Cos(H) – (U-2048) x Sin(H)

Where H represents the desired Hue angle; Saturation is adjusted by multiplying both U and V by a constant.

Finally, the value 2048 is added to both U and V. The combined processing of Hue and Saturation on the UV data is:

U' = (((U-2048) x Cos(H) + (V-2048) x Sin(H)) x C x S) + 2048

V' = (((V-2048) x Cos(H) - (U-2048) x Sin(H)) x C x S) + 2048

Where C is the contrast, H is Hue angle and S is the Saturation and the combination of Cos(H)*C*S and Sin(H)*C*S is specified by parameters Cos_c_s and Sin_c_s.

3.6.1.6 Color Space Conversion

The CSC block enables linear conversion between color spaces using vector shift, matrix multiplication, and additional shift.

The CSC algorithm is a linear coordinate transformation, comprising of the following stages:

Shifting the input color coordinate.

Multiply by 3*3 matrix

Shifting the output color coordinate

Formula representation of last 3 steps:

$$\begin{pmatrix} vout\_1 \\ vout\_2 \\ vout\_3 \end{pmatrix} = \begin{pmatrix} a11 & a12 & a13 \\ a21 & a22 & a23 \\ a31 & a32 & a33 \end{pmatrix} * \begin{pmatrix} vin\_1+v0\_1 \\ vin\_2+v0\_2 \\ vin\_3+v0\_3 \end{pmatrix} + \begin{pmatrix} u0\_1 \\ u0\_2 \\ u0\_3 \end{pmatrix}$$

Where is

aij are the matrix elements, i.e., the transform coefficients: X0, X1, X2, X3, X4, X5, X6, X7, X8.

vin_i is the input pixel color components

v0_i is the input offset vector, i.e., Οφφσετ_ιν_1, Οφφσετ_ιν_2, Οφφσετ_ιν_3.

u0_1_i is the output offset vector. i.e., Οφφσετ_ουτ_1, Οφφσετ_ουτ_2, Οφφσετ_ουτ_3.

Clipping the output to ensure each component is in allowed range.

The parameters ΨΥς_IN is used to set input to be RGB format and ΨΥς_OYT is uased to set output to be RGB format

Notes about Repacker:

There are two states to be used in the repacker: Αλπηα φρομ Στατε Σελεχτ and χολορ πιπε αλπηα. The last module in the IECP pipeline.

If Alpha from State Select is set, the Y, U ,V is packed with the information from color pipe alpha, and then the data is sent out to RCPB.

Otherwise, “0” is inserted in the 4LSB (alpha) and the packed data is sent out to RCPB.

3.6.1.7 Color Gamut Compression

3.6.1.7.1 Background of Color Gamut Compression

While most photography today complies with the sRGB standard color space, which covers around 72% of color perceived by human being, this 72% content looks incorrect/unnatural on wide gamut displays, which can extend more than 100%. Therefore, a gamut mapping (GM) algorithm is required to adjust when the input gamut range is different to the output gamut range such as the input sRGB color space to be displayed onto the WG display, or to adjust the wide gamut content to be displayed onto the traditional lower gamut display.

The easiest compression method applied to displaying wider gamut content on lower gamut displays is to clip the out of range primary values to the valid range (i.e., 0-1). Although this simple clipping procedure leads to acceptable visual appearance in most cases, loss of color depth can be observed in the video containing out-of-range pixels. The reason behind this effect should be the uniform quantization process applied to out-of-range values (e.g., two distinct out-of-range red colors are mapped to the same boundary red color). Moreover, the simple clipping method treats each color channel independently. This may lead to unexpected perceptual loss since the composite ratios of three primaries have been changed. An approach which takes these two factors into account while scaling down the out of range values can possibly maintain the detail information of the image.

3.6.1.7.2 Usage Models

There are two usage models depending on the set up of ΦυλλΡανγεΜαππινγΕναβλε bit:

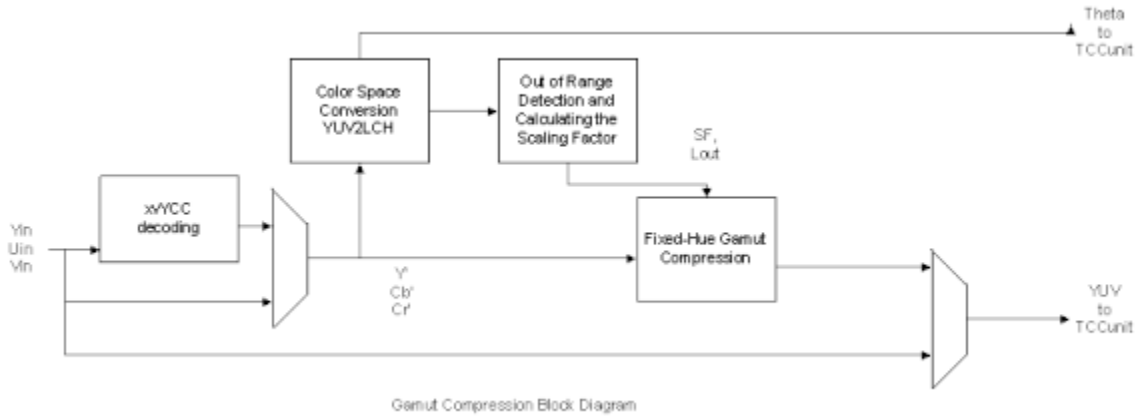
Basic mode: fixed-hue color gamut clipping mode

Advanced mode: fixed-hue full range mapping mode

The application of basic mode of the fixed-hue color gamut clipping is preferred when the content having the smaller percentage of out-of-range pixels in the scene. The advanced mode of fixed-hue full range mapping mode may also change the in-range pixels and is thus preferred when the percentage of out-of-range pixel is large. The outcome of the in/out range pixel percentage is derived from the out-of range color gamut detection module to provide an indicator to operate among basic mode and advanced mode.

3.6.1.7.3 Gamut compression module overview

The main goal of color gamut compression module algorithm is to compress out-of-range pixel values while keeping their hue values same as it is before compression. A block diagram to color gamut compress the xv Color video into sRGB format is shown below.



Gamut Compression Block Diagram

AT the pipeline level, the input into Gamut compression unit is from STDE unit and the output from the Gamut compression goes to TCCE unit. The Gamut compression comprises of the following stages:

xvYCC decoding

YUV2LCH color space conversion

Out of range Gamut pixel detection

Scaling factor calculation

Find out the Euclidean distance for the out of range pixel for advance mode

Fixed-hue Gamut compression

Bring the out of range pixel to the boundary for basic mode

Bring the out of range pixel depending on the distance and apply uniform quantization process in advance mode

xvYCC encoding

3.6.1.7.4 xvYCC decoding

The non-linear YCbCr values (i.e.,Y'Cb'Cr', or Y'UV) is decoded from an example of 8-bit/channel below:

$$Y' = (Y_{xvYCC} - 16)/219$$

$$U = Cb' = (Cb_{xvYCC} - 128)/224$$

$$V = Cr' = (Cr_{xvYCC} - 128)/224$$

.

For 12-bit/channel the above equation can be re-written as follows:

$$Y' = (Y_{xvYCC} - 256) * 4096/3504 = (Y_{xvYCC} - 256) * 4788 >> 12$$

$$U = Cb' = (Cb_{xvYCC} - 2048) * 4096/3584 = (Cb_{xvYCC} - 2048) * 4681 >> 12$$

$$V = Cr' = (Cr_{xvYCC} - 2048) * 4096/3584 = (Cr_{xvYCC} - 2048) * 4681 >> 12$$

3.6.1.7.5 YUV2LCH

The parameters for scaling the out-of-range pixel values are determined in *LCHuv* space, which is the cylindrical version of LUV space. For every input pixel, (*Y_in = Y', U_in = Cb', V_in = Cr'*) we find its chrominance value (i.e., C) and hue value (i.e., H)

$$L = Y_{in},$$

$$C = \sqrt{u_{in}^2 + v_{in}^2},$$

$$H = \tan^{-1}\frac{v_{in}}{u_{in}}$$

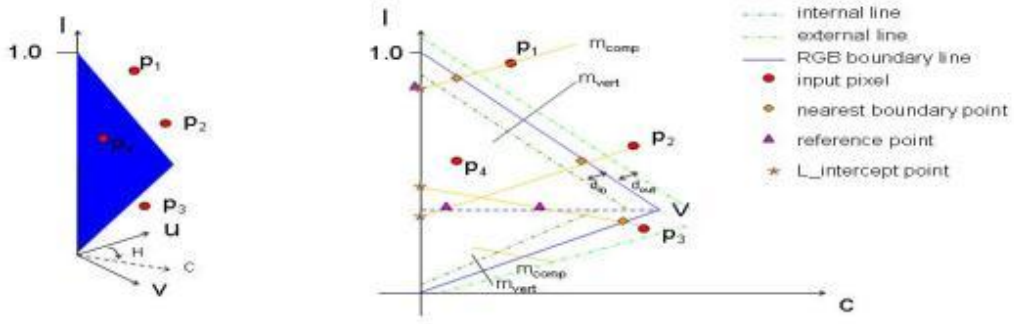
The approximation of hue angle calculation is described in the TCC session.

3.6.1.7.6 Out-of-range gamut pixel detection

An input pixel is denoted as $P_i = (c_{p_i}, l_{p_i})$ in the *LCH* color space. If $c_{p_i} = 0$, the pixel can be outputted without pixel value change.

Every input pixel is associated with a hue value in *LCH* space. From the calculated hue value, we can read a corresponding vertex point from the pre-calculated table which contained the 512-entry vertex points of the below color triangle in *LCH* space with the designated hue value derived off-line. Note that with the symmetric property of the lightness vertex, only vertices in $[0 \sim \pi)$ need to be stored in the pre-calculated LUT. Therefore, the 512x2 components in the LUT correspond to the information of 512 equally-distributed hue angles in the range of $[0 \sim \pi)$. Moreover, the vertex value of a certain hue angle is dependent of the color space (e.g., BT. 709 or BT. 601). Here, the vertex point *V* for a hue angle is denoted as

$$V = (c_V, l_V)$$.(3)



Left: The RGB boundary corresponding to hue value H. Right: the RGB boundary in left projected on the luminance-chrominance plane.

Vertex point look up table:

Utilizing the hue angle calculation module in TCC, the equally-spaced, discrete hue angle ranging from 0 to 2π is represented with an integer (i.e., *angle index*) in [0~1023]. Since the LUT in color compression only stores vertices in [0~π), a mapping procedure is required to remap angles in $[\pi \sim 2\pi)$ to $[0 \sim \pi)$ before indexing the LUT:

if((*angle_index* < 512) && (*angle_index* > =0))

angle_index = angle_index;

else

angle_index = angle_index – 512;

With the properly remapped *angle index* for accessing the LUT, the information of the vertex point can be obtained as below.

Sat_Vertex **=** m_SatVertex[*angle_index*]; // Sat Vertex form the LUT

Luma1 **=** m_LumaVertex[*angle_index*]; // lightness vertex in [0~pi)

Luma2 **=** denorm-Luma1; // symmetric lightness vertex; denorm = 4096 in 12 bit representation

Note that the lightness vertex is symmetric in uv/ CbCr-plane (i.e., Luma1 + Luma2 = denom), and the lightness vertex stored in the LUT correspond to those in $[0 \sim \pi)$. Thus, one has to check the hue side (i.e., whether this pixel is originally in $[0 \sim \pi)$ or in $[\pi \sim 2\pi)$ of the current pixel to acquire the correct lightness vertex for this pixel:

if(srcV == 0)// 0 or 180 degree

{

if (srcU > 0)// srcU == 0 Cp = 0

Luma_Vertex = Luma1;

else

Luma_Vertex = Luma2;

}

else

{

if(srcV > 0)

Luma_Vertex = Luma1;

else

Luma_Vertex = Luma2;

}

An input pixel $P_i = (c_{p_i}, l_{p_i})$ is detected as an out-of-range pixel if the below condition is true.

$$if\{(l_{p_i} >= l_V)\ \&\ \&(sign[l_{p_i} - (\frac{l_V - 1}{c_V})c_{p_i} - 1] > 0))\ OR\{(l_{p_i} < l_V)\ \&\ \&(sign[l_{p_i} - \frac{l_V}{c_V}c_{p_i}] < 0))\} \quad (4)$$

A statistics parameter, νυμβερ_οφ_ουτ–οφ–ρανγε_πιξελ, is incremented if the above equation is true. The νυμβερ_οφ_ουτ–οφ–ρανγε_πιξελ will be collected at picture level through VSC unit to assess the property of a picture to determine the strategy of ways to do gamut compression.

Note: If P_i is an in-range pixel, the pixel will be outputted according to equation (13).

3.6.1.7.7 Scaling factor – Basic mode

The slope of a compression line is defined from the vertex point table.

$m_{comp} = m_{vert} >> (compression\_line\_shift)$, with χομπρεσσιον_λινε_σηιφτ default to be 3.(5)

m_{comp} in the above equation is the slope of the compression line while m_{vert} represents the slope of the line perpendicular to the RGB boundary line:

$m_{vert} = -\frac{1}{m_{boundary}}$, and(6)

$m_{boundary} = \frac{(l_V - e_V)}{c_V}$ where $e_V = \begin{cases} 1, & \text{if } l_{p_i} > l_V \\ 0, & \text{else} \end{cases}$.(7)

The intersection between the compression line for pixel p_i and the *L*-axis is denoted as I_{P_i}

$I_{p_i} = (c_{I_{p_i}}, l_{I_{p_i}})$, then

$c_{Ip_i} = 0$, and(8)

$l_{I_{p_i}} = l_{p_i} - c_{p_i} \times m_{comp}$

The point nearest to the input pixel p_i on the RGB boundary along the compression direction (i.e., intersection between the compression line and the RGB boundary) be B_{p_i}, then

$B_{p_i} = (c_{B_{p_i}}, l_{B_{p_i}})$, with

$$c_{B_{Pi}} = \frac{(l_{I_{Pi}} - e_V)}{(m_{boundary} - m_{comp})}$$, and (9)

$l_{B_{p_i}} = c_{B_{p_i}} \times m_{boundary} + e_V$.

Scaling factor is denoted as

$$sf_{p_i} = \frac{c_{p_i,output}}{c_{p_i}}$$.(10)

For the usage of Basic mode - fixed-hue color gamut clipping mode, all out-of-range pixels will be clipped to the boundary, which means

$c_{p_i,output} = c_{B_{p_i}}$ (11)

And the luma is mapped at along the compression line to hit the boundary line at

$l_{p_i,output} = l_{I_{p_i}} + c_{p_i,output} \times m_{comp}$ (12)

3.6.1.7.8 Fixed-hue compression

The output of fixed compression is based on the scaling factor and the property of pixel.

$$\begin{cases} u_{p_i,out} = u_{p_i,in} \\ v_{p_i,out} = v_{p_i,in} \\ y_{p_i,out} = y_{p_i,in} \end{cases}$$, if $c_{p_i} = 0$ or $P_i \in in\_range\_pixel$, else

$$\begin{cases} u_{p_i,out} = u_{p_i,in} \times sf_{p_i} \\ v_{p_i,out} = v_{p_i,in} \times sf_{p_i} \\ y_{p_i,out} = l_{p_i,output} \end{cases}$$. (13)

3.6.1.7.9 Scaling factor – Advanced mode

The out-of-range pixel values can be mapped inwards according to how far they are from the boundary from the following equation:

$$c_{p_i,output} = c_{R_{p_i}} + (c_{p_i} - c_{R_{p_i}}) \times \frac{d_{p_i,final}}{d(R_{p_i}, p_i)}$$.(14)

$$l_{p_i,output} = l_{I_{p_i}} + c_{p_i,output} \times m_{comp}$$.

Where $c_{R_{p_i}}$ is coming from the reference point as the origin of the linear transformation for compressing pixel p_i as

Denote the reference point

$$R_{p_i} = (c_{R_{p_i}}, l_{R_{p_i}})$$

$$l_{R_{p_i}} = \begin{cases} \max(l_{I_{p_i}}, l_V), & \text{if } l_{p_i} > l_V \\ \min(l_{I_{p_i}}, l_V), & \text{otherwise} \end{cases}$$ (15)

$$c_{R_{pi}} = (l_{R_{pi}} - l_{I_{pi}}) \times \frac{1}{m_{comp}}$$

3.6.1.7.10 xvYCC encoding

The output of Y, Cb and Cr values are scaled back through xvYCC encoding process, and the example for the 12bit format is provided below:

$$Y_{\_out} = [(Y_{p_i,out} * 3504) >> 12] + 256$$

$$U_{\_out} = [(U_{p_i,out} * 3584) >> 12] + 2048$$

$$V_{\_out} = [(V_{p_i,out} * 3584) >> 12] + 2048$$

$$Y_{\_out} = Clamp(0, 4095, Y_{\_out})$$

$$U_{\_out} = Clamp(0, 4095, U_{\_out})$$

$$V_{\_out} = Clamp(0, 4095, V_{\_out})$$

3.6.2 Boundary Behavior

The table below summarizes the behavior of the **Media Boundary Pixel Mode** field (SURFACE_STATE) in combination with the **Vertical Line Stride** and **Vertical Line Stride Offset** fields (both of which are subject to being overridden by the Data Port message descriptor fields). The Behavior column illustrates behavior for a surface with four rows numbered 0 to 3. The bold indicators are off-surface behavior and the non-bold indicators are on-surface behavior. Input row addresses range from -3 to +7 going left to right.

Media Boundary Pixel Mode	Vertical Line Stride	Vertical Line Stride Offset	Usage Model	Behavior
0	0	X	normal frame	000001233333
0	1	0	normal field even	000002222222
0	1	1	normal field odd	111113333333
2	0	X	frame / progressive	000001233333
2	1	0	field even / progressive	000002333333
2	1	1	field odd / progressive	000013333333
3	0	X	frame / interlaced	010101232323
3	1	0	field even / interlaced	000002222222
3	1	1	field odd / interlaced	111113333333

3.7 Accessing Render Targets

Render targets are the surfaces that the final results of pixel shaders are written to. The render targets support a large set of surface formats (refer to surface formats table in *Sampling Engine* for details) with

hardware conversion from the format delivered by the thread. The render target message also causes numerous side effects, including potentially alpha test, depth test, stencil test, alpha blend (which normally causes a read of the render target), and other functions. These functions are covered in the *Windower* chapter as some of them (depth/stencil test) are also partially done in the Windower.

The render target write messages are specifically for the use of pixel shader threads that are spawned by the windower, and may not be used by any other threads. This is due to the pixel scoreboard side-effects that sending of this message entails. The pixel scoreboard ensures that incorrect ordering of reads and writes to the same pixel does not occur.

3.7.1 Single Source

The “normal” render target messages are single source. There are two forms, SIMD16 and SIMD8, intended for the equivalent-sized pixel shader threads. A single color (4 channels) is delivered for each of the 16 or 8 pixels in the message payload. Optional depth, stencil, and antialias alpha information can also be delivered with these messages.

The pixel scoreboard bits corresponding to the dispatched pixel mask (or half of the mask in the case of SIMD8 messages) are cleared only if the **Last Render Target Select** bit is set in the message descriptor.

The single source message will not cause a write to the render target if **Dual Source Blend Enable** in 3DSTATE_WM is *enabled*. However, if **Last Render Target Select** is set, the message will still cause pixel scoreboard clear and depth/stencil buffer updates if enabled.

3.7.2 Dual Source

The dual source render target messages only have SIMD8 forms due to maximum message length limitations. SIMD16 pixel shaders must send two of these messages to cover all of the pixels. Each message contains two colors (4 channels each) for each pixel in the message payload. In addition to the first source, the second source can be selected as a blend factor (BLENDFACTOR_*_SRC1_* options in the blend factor fields of COLOR_CALC_STATE or BLEND_STATE). Optional depth, stencil, and antialias alpha information can also be delivered with these messages.

Each dual source message delivered will clear the corresponding pixel scoreboard bits if the **Last Render Target Select** bit in the message descriptor is set.

The dual source message will revert to a single source message using source 0 if **Dual Source Blend Enable** in 3DSTATE_WM is disabled.

3.7.3 Replicate Data

The replicate data render target message is used for “fast clear” functionality in cases where the color data for each pixel is identical. This message performs better than the other messages due to its smaller message length. This message does not support depth, stencil, or antialias alpha data being sent with it. This message must target only tiled memory. Access of linear memory using this message type is UNDEFINED. The depth buffer can be cleared through the “early depth” function in conjunction with a pixel shader using this message. Refer to the *Windower* chapter for more details on the early depth function.

The pixel scoreboard bits corresponding to the dispatched pixel mask are cleared only if the **Last Render Target Select** bit is set in the message descriptor.

3.7.4 Multiple Render Targets (MRT)

Multiple render targets are supported with the single source and replicate data messages. Each render target is accessed with a separate Render Target Write message, each with a different surface indicated (different binding table index). The depth buffer is written only by the message(s) to the last render target, indicated by the **Last Render Target Select** bit set to clear the pixel scoreboard bits.

MRT is not supported when one or more RTs have this surface formats: YCRCB_SWAPUVY, YCRCB_SWAPUV, YCRCB_SWAPY, YCRCB_NORMAL

3.8 State

3.8.1 BINDING_TABLE_STATE

The data port uses the binding table to retrieve surface state. Refer to *State*in the Sampling Engine section for the definition of this state.

3.8.2 SURFACE_STATE

The data port uses the surface state for constant buffers, render targets, and media surfaces. Refer to *SURFACE_STATE* in the Sampling Engine section for the definition of this state.

3.8.3 COLOR_PROCESSING_STATE

The following state structures contain different states used by the color processing function.

COLOR_PROCESSING_STATE - STD/STE State

Default Value: 0x9A6E39F0, 0x400C0000, 0x00001180, 0xFE2F2E00, 0x000000FF, 0x00140000, 0xD82E0000, 0x8285ECEC, 0x00008282, 0x00000000, 0x02117000, 0xA38FEC96, 0x00008CC8, 0x00000000, 0x01478000, 0x0007C300, 0x00000000, 0x00000000, 0x1C180000, 0x00000000, 0x00000000, 0x00000000, 0x0007CF80, 0x00000000, 0x00000000, 0x1C080000, 0x00000000, 0x00000000, 0x00000000

This state structure contains the STD/STE state used by the color processing function.

DWord	Bit	Description	
0	31:24	**V_Mid**	
		Default Value:	154
		Format:	U8
		Rectangle middle-point V coordinate	
	23:16	**U_Mid**	
		Default Value:	110
		Format:	U8
		Rectangle middle-point U coordinate	
	15:10	**Hue Max**	
		Default Value:	14
		Format:	U6
		Rectangle half width	

COLOR_PROCESSING_STATE - STD/STE State

	9:4	**Sat Max** Default Value: 31 Format: U6 Rectangle half length.
	3	**Reserved** Format: MBZ
	2	**Output Control** Value: 0 — Name: Output Pixels **[Default]** Value: 1 — Name: Output STD Decisions
	1	**STE Enable** Format: Enable
	0	**STD Enable** Format: Enable
1	31	**Reserved** Format: MBZ
	30:28	**Diamond Margin** Default Value: 4 Format: U3
	27:21	**Diamond du** Default Value: 0 Format: S7 2's complement Rhombus center shift in the sat-direction, relative to the rectangle center.
	20:18	**HS Margin** Default Value: 3 Format: U3
	17:10	**Cos(α)** Format: S0.7 2's Compliment The default is 79/128
	9:8	**Reserved** Format: MBZ
	7:0	**Sin(α)** Format: S0.7 2's Compliment The default is 101/128
2	31:21	**Reserved** Format: MBZ

COLOR_PROCESSING_STATE - STD/STE State

	20:13	**Diamond Alpha** Format: U2.6 1 / tan(β) The default is 100/64
	12:7	**Diamond Th** Default Value: 35 Format: U6 Half length of the rhombus axis in the sat-direction.
	6:0	**Diamond dv** Default Value: 0 Format: S6 2's complement
3	31:24	**Y_point_3** Default Value: 254 Format: U8 Third point of the Y piecewise linear membership function.
	23:16	**Y_point_2** Default Value: 47 Format: U8 Second point of the Y piecewise linear membership function.
	15:8	**Y_point_1** Default Value: 46 Format: U8 First point of the Y piecewise linear membership function.
	7	**VY_STD_Enable** Format: Enable Enables STD in the VY subspace.
	6:0	**Reserved** Format: MBZ
4	31:18	**Reserved** Format: MBZ
	17:13	**Y_Slope_2** Format: U2.3 Slope between points Y3 and Y4. The default is 31/8.
	12:8	**Y_Slope_1** Format: U2.3 Slope between points Y1 and Y2. The default is 31/8.

COLOR_PROCESSING_STATE - STD/STE State

	7:0	**Y_point_4** Default Value: 255 Format: U8 Fourth point of the Y piecewise linear membership function
5	31:16	**INV_skin_types_margin** Format: U0.16 1/(2* Skin_types_margin) Value: 20 [Default] — Name: Skin_Type_margin — Description
	15:0	**Inverse Margin VYL** Format: U0.16 1 / Margin_VYL The default is 3300/65536
6	31:24	**P1L** Default Value: 216 Format: U8 Y Point 1 of the lower part of the detection PWLF.
	23:16	**P0L** Default Value: 46 Format: U8 Y Point 0 of the lower part of the detection PWLF.
	15:0	**Inverse Margin VYU** Format: U0.16 1 / Margin_VYU The default is 1600/65536.
7	31:24	**B1L** Default Value: 130 Format: U8 V Bias 1 of the lower part of the detection PWLF.
	23:16	**B0L** Default Value: 133 Format: U8 V Bias 0 of the lower part of the detection PWLF.
	15:8	**P3L** Default Value: 236 Format: U8 Y Point 3 of the lower part of the detection PWLF.
	7:0	**P2L** Default Value: 236 Format: U8 Y point 2 of the lower part of the detection PWLF.

COLOR_PROCESSING_STATE - STD/STE State

8	31:27	**Reserved** Format: MBZ
	26:16	**S0L** Format: S2.8 2's complement Slope 0 of the lower part of the detection PWLF. The default is -5/256.
	15:8	**B3L** Default Value: 130 Format: U8 V Bias 3 of the lower part of the detection PWLF.
	7:0	**B2L** Default Value: 130 Format: U8 V Bias 2 of the lower part of the detection PWLF.
9	31:22	**Reserved** Format: MBZ
	21:11	**S2L** Format: S2.8 2's complement Slope 2 of the lower part of the detection PWLF. The default is 0/256.
	10:0	**S1L** Format: S2.8 2's complement Slope 1 of the lower part of the detection PWLF. The default is 0/256.
10	31:27	**Reserved** Format: MBZ
	26:19	**P1U** Default Value: 66 Format: U8 Y Point 1 of the upper part of the detection PWLF.
	18:11	**P0U** Default Value: 46 Format: U8 Y Point 0 of the upper part of the detection PWLF.
	10:0	**S3L**

COLOR_PROCESSING_STATE - STD/STE State

		Format: S2.8 2's complement Slope 3 of the lower part of the detection PWLF. The default is 0/256.
11	31:24	**B1U** Default Value: 163 Format: U8 V Bias 1 of the upper part of the detection PWLF.
	23:16	**B0U** Default Value: 143 Format: U8 V Bias 0 of the upper part of the detection PWLF.
	15:8	**P3U** Default Value: 236 Format: U8 Y Point 3 of the upper part of the detection PWLF.
	7:0	**P2U** Default Value: 150 Format: U8 Y Point 2 of the upper part of the detection PWLF.
12	31:27	**Reserved** Format: MBZ
	26:16	**S0U** Format: S2.8 2's complement Slope 0 of the upper part of the detection PWLF. The default is 256/256.
	15:8	**B3U** Default Value: 140 Format: U8 V Bias 3 of the upper part of the detection PWLF.
	7:0	**B2U** Default Value: 200 Format: U8 V Bias 2 of the upper part of the detection PWLF.
13	31:22	**Reserved** Format: MBZ
	21:11	**S2U** Format: S2.8 2's complement

COLOR_PROCESSING_STATE - STD/STE State

		Slope 2 of the upper part of the detection PWLF. The default is -179/256.
	10:0	**S1U** Format: S2.8 2's complement Slope 1 of the upper part of the detection PWLF. The default is -113/256.
14	31:28	**Reserved** Format: MBZ
	27:20	**Skin Types Margin** Default Value: 20 Format: U8 Skin types Y margin.
	19:12	**Skin Types Thresh** Default Value: 120 Format: U8 Skin types Y threshold.
	11	**Skin Type Enable** Format: Enable Treat differently bright and dark skin types. Value / Name / Description 0 / **[Default]** / Disable
	10:0	**S3U** Format: S2.8 2's complement Slope 3 of the upper part of the detection PWLF. The default is 0/256.
15	31	**Reserved** Format: MBZ
	30:21	**SATB1** Format: S7.2 2's complement First bias for the saturation PWLF (bright skin). The default is -8/4.
	20:14	**SATP3** Default Value: 31 Format: S6 2's complement

COLOR_PROCESSING_STATE - STD/STE State

		Third point for the saturation PWLF (bright skin).
	13:7	**SATP2** Default Value: 6 Format: S6 2's complement Second point for the saturation PWLF (bright skin).
	6:0	**SATP1** Format: S6 2's complement First point for the saturation PWLF (bright skin). The default is -6.
16	31	**Reserved** Format: MBZ
	30:20	**SATS0** Format: U3.8 Zeroth slope for the saturation PWLF (bright skin). The default is 297/256.
	19:10	**SATB3** Format: S7.2 2's complement Third bias for the saturation PWLF (bright skin). The default is 124/4.
	9:0	**SATB2** Format: S7.2 2's complement Second bias for the saturation PWLF (bright skin). The default is 8/4.
17	31:22	**Reserved** Format: MBZ
	21:11	**SATS2** Format: U3.8 Second slope for the saturation PWLF (bright skin). The default is 297/256.
	10:0	**SATS1** Format: U3.8 First slope for the saturation PWLF (bright skin). The default is 85/256.

COLOR_PROCESSING_STATE - STD/STE State

18	31:25	**HUEP3** Default Value: 14 Format: S6 2's complement Third point for the hue PWLF (bright skin)
	24:18	**HUEP2** Default Value: 6 Format: S6 2's complement Second point for the hue PWLF (bright skin)
	17:11	**HUEP1** Format: S6 2's complement First point for the hue PWLF (bright skin). The default is -6.
	10:0	**SATS3** Format: U3.8 Thrid slope for the saturation PWLF (bright skin). The default is 256/256.
19	31:30	**Reserved** Format: MBZ
	29:20	**HUEB3** Format: S7.2 2's complement Third bias for the hue PWLF (bright skin). The default is 56/4.
	19:10	**HUEB2** Format: S7.2 2's complement Second bias for the hue PWLF (bright skin). The default is 8/4.
	9:0	**HUEB1** Format: S7.2 2's complement First bias for the hue PWLF (bright skin). The default is -8/4.

COLOR_PROCESSING_STATE - STD/STE State

20	31:22	**Reserved** Format: MBZ
	21:11	**HUES1** Format: U3.8 First slope for the hue PWLF (bright skin) The default is 85/256.
	10:0	**HUES0** Format: U3.8 Zeroth slope for the hue PWLF (bright skin) The default is 384/256.
21	31:22	**Reserved** Format: MBZ
	21:11	**HUES3** Format: U3.8 Third slope for the hue PWLF (bright skin) The default is 256/256.
	10:0	**HUES2** Format: U3.8 Second slope for the hue PWLF (bright skin) The default is 384/256.
22	31	**Reserved**
	30:21	**SATB1_DARK** Format: S7.2 2's complement First bias for the saturation PWLF (dark skin) The default is 0/4.
	20:14	**SATP3_DARK** Default Value: 31 Format: S6 2's complement Third point for the saturation PWLF (dark skin)
	13:7	**SATP2_DARK** Default Value: 31 Format: S6 2's complement Second point for the saturation PWLF (dark skin)
	6:0	**SATP1_DARK** Format: S6 2's complement

COLOR_PROCESSING_STATE - STD/STE State

		First point for the saturation PWLF (dark skin). The default is -11.
23	31	**Reserved** Format: MBZ
	30:20	**SATS0_DARK** Format: U3.8 Zeroth slope for the saturation PWLF (dark skin). The default is 397/256.
	19:10	**SATB3_DARK** Format: S7.2 2's complement Third bias for the saturation PWLF (dark skin). The default is 124/4.
	9:0	**SATB2_DARK** Format: S7.2 2's complement Second bias for the saturation PWLF (dark skin). The default is 124/4.
24	31:22	**Reserved** Format: U3.8
	21:11	**SATS2_DARK** Format: U3.8 Second slope for the saturation PWLF (dark skin). The default is 256/256.
	10:0	**SATS1_DARK** Format: U3.8 First slope for the saturation PWLF (dark skin). The default is 189/256.
25	31:25	**HUEP3_DARK** Default Value: 14 Format: S6 2's complement Third point for the hue PWLF (dark skin).
	24:18	**HUEP2_DARK** Default Value: 2 Format: S6 2's complement

COLOR_PROCESSING_STATE - STD/STE State

		Third point for the hue PWLF (dark skin).
	17:11	**HUEP1_DARK** Default Value: 0 Format: S6 2's complement Third point for the hue PWLF (dark skin).
	10:0	**SATS3_DARK** Format: U3.8 Third slope for the saturation PWLF (dark skin). The default is 256/256.
26	31:30	**Reserved** Format: MBZ
	29:20	**HUEB3_DARK** Format: S7.2 2's complement Third bias for the hue PWLF (dark skin). The default is 56/4.
	19:10	**HUEB2_DARK** Format: S7.2 2's complement Second bias for the hue PWLF (dark skin). The default is 0/4.
	9:0	**HUEB1_DARK** Format: S7.2 2's complement First bias for the hue PWLF (dark skin). The default is 0/4.
27	31:22	**Reserved** Format: MBZ
	21:11	**HUES1_DARK** Format: U3.8 First slope for the hue PWLF (dark skin). The default is 0/256.
	10:0	**HUES0_DARK** Format: U3.8 Zeroth slope for the hue PWLF (dark skin). The default is 256/256.

COLOR_PROCESSING_STATE - STD/STE State

28	31:22	**Reserved** Format: MBZ
	21:11	**HUES3_DARK** Format: U3.8 Third slope for the hue PWLF (dark skin). The default is 256/256.
	10:0	**HUES2_DARK** Format: U3.8 Second slope for the hue PWLF (dark skin). The default is 299/256.

COLOR_PROCESSING_STATE - ACE State

Default Value: 0x00000068, 0x4C382410, 0x9C887460, 0xEBD8C4B0, 0x604C3824, 0xB09C8874, 0x0000D8C4, 0x00000000, 0x00000000, 0x00000000, 0x00000000, 0x00000000, 0x00000000

This state structure contains the ACE state used by the color processing function.

DWord	Bit	Description
29	31:7	**Reserved** Format: MBZ
	6:2	**Skin Threshold** Format: U5 Used for Y analysis (min/max) for pixels which are higher than skin threshold. Value: 1-31 Value: 26 **[Default]**
	1	**Full Image Histogram** Default Value: 0 Format: Enable Used to ignore the area of interest for full image histogram.
	0	**ACE Enable** Format: Enable
30	31:24	**Y3** Default Value: 76 Format: U8 The value of the y_pixel for point 3 in PWL.
	23:16	**Y2** Default Value: 56 Format: U8 The value of the y_pixel for point 2 in PWL.
	15:8	**Y1** Default Value: 36

COLOR_PROCESSING_STATE - ACE State

		Format: U8 The value of the y_pixel for point 1 in PWL.
	7:0	**Ymin** Default Value: 16 Format: U8 The value of the y_pixel for point 0 in PWL.
31	31:24	**Y7** Default Value: 156 Format: U8 The value of the y_pixel for point 7 in PWL.
	23:16	**Y6** Default Value: 136 Format: U8 The value of the y_pixel for point 6 in PWL.
	15:8	**Y5** Default Value: 116 Format: U8 The value of the y_pixel for point 5 in PWL.
	7:0	**Y4** Default Value: 96 Format: U8 The value of the y_pixel for point 4 in PWL.
32	31:24	**Ymax** Default Value: 235 Format: U8 The value of the y_pixel for point 11 in PWL.
	23:16	**Y10** Default Value: 216 Format: U8 The value of the y_pixel for point 10 in PWL.
	15:8	**Y9** Default Value: 196 Format: U8 The value of the y_pixel for point 9 in PWL.
	7:0	**Y8** Default Value: 176 Format: U8 The value of the y_pixel for point 8 in PWL.
33	31:24	**B4**

COLOR_PROCESSING_STATE - ACE State

		Default Value: 96 Format: U8 The value of the bias for point 4 in PWL.
	23:16	**B3** Default Value: 76 Format: U8 The value of the bias for point 3 in PWL.
	15:8	**B2** Default Value: 56 Format: U8 The value of the bias for point 2 in PWL.
	7:0	**B1** Default Value: 36 Format: U8 The value of the bias for point 1 in PWL.
34	31:24	**B8** Default Value: 176 Format: U8 The value of the bias for point 8 in PWL.
	23:16	**B7** Default Value: 156 Format: U8 The value of the bias for point 7 in PWL.
	15:8	**B6** Default Value: 136 Format: U8 The value of the bias for point 6 in PWL.
	7:0	**B5** Default Value: 116 Format: U8 The value of the bias for point 5 in PWL.
35	31:16	**Reserved** Format: MBZ
	15:8	**B10** Default Value: 216 Format: U8 The value of the bias for point 10 in PWL.
	7:0	**B9** Default Value: 196 Format: U8

COLOR_PROCESSING_STATE - ACE State		
		The value of the bias for point 9 in PWL.
36	31:27	**Reserved** Format: MBZ
	26:16	**S1** Format: U1.10 The value of the slope for point 1 in PWL. The default is 1024/1024.
	15:11	**Reserved** Format: MBZ
	10:0	**S0** Format: U1.10 The value of the slope for point 0 in PWL. The default is 1024/1024.
37	31:27	**Reserved** Format: MBZ
	26:16	**S3** Format: U1.10 The value of the slope for point 3 in PWL. The default is 1024/1024.
	15:11	**Reserved** Format: MBZ
	10:0	**S2** Format: U1.10 The value of the slope for point 2 in PWL. The default is 1024/1024.
38	31:27	**Reserved** Format: MBZ
	26:16	**S5** Format: U1.10 The value of the slope for point 5 in PWL. The default is 1024/1024.
	15:11	**Reserved** Format: MBZ
	10:0	**S4** Format: U1.10 The value of the slope for point 4 in PWL. The default is 1024/1024.
39	31:27	**Reserved** Format: MBZ
	26:16	**S7** Format: U1.10

COLOR_PROCESSING_STATE - ACE State

		The value of the slope for point 7 in PWL. The default is 1024/1024.
	15:11	**Reserved** Format: MBZ
	10:0	**S6** Format: U1.10 The value of the slope for point 6 in PWL. The default is 1024/1024.
40	31:27	**Reserved** Format: MBZ
	26:16	**S9** Format: U1.10 The value of the slope for point 9 in PWL. The default is 1024/1024.
	15:11	**Reserved** Format: MBZ
	10:0	**S8** Format: U1.10 The value of the slope for point 8 in PWL. The default is 1024/1024.
41	31:11	**Reserved** Format: MBZ
	10:0	**S10** Format: U1.10 The value of the slope for point 10 in PWL. The default is 1024/1024.

COLOR_PROCESSING_STATE - TCC State

Default Value: 0xDCDCDC00, 0xDCDCDC00, 0x1E34CC91, 0x3E3CCE91, 0x02E80195, 0x0197046B, 0x01790174, 0x00096000, 0x00000000, 0x03030000, 0x009201C0

This state structure contains the TCC state used by the color processing function.

DWord	Bit	Description
42	31:24	**SatFactor3** Default Value: 220 Format: U1.7 The saturation factor for yellow.
	23:16	**SatFactor2** Default Value: 220

COLOR_PROCESSING_STATE - TCC State

		Format: U1.7 The saturation factor for red.
	15:8	**SatFactor1** Default Value: 220 Format: U1.7 The saturation factor for magenta.
	7	**TCC Enable** Format: Enable
	6:0	**Reserved** Format: MBZ
43	31:24	**SatFactor6** Default Value: 220 Format: U1.7 The saturation factor for blue.
	23:16	**SatFactor5** Default Value: 220 Format: U1.7 The saturation factor for cyan.
	15:8	**SatFactor4** Default Value: 220 Format: U1.7 The saturation factor for green.
	7:0	**Reserved** Format: MBZ
44	31:30	**Reserved** Format: MBZ
	29:20	**Base Color 3** Default Value: 483 Format: U10
	19:10	**Base Color 2** Default Value: 307 Format: U10
	9:0	**Base Color 1** Default Value: 145 Format: U10
45	31:30	**Reserved** Format: MBZ
	29:20	**Base Color 6** Default Value: 995 Format: U10
	19:10	**Base Color 5** Default Value: 819

COLOR_PROCESSING_STATE - TCC State

		Format:	U10
	9:0	**Base Color 4**	
		Default Value:	657
		Format:	U10
46	31:16	**Color Transit Slope 23**	
		Default Value:	744
		Format:	U0.16
		The calculation result of 1 / (BC3 – BC2) [1/62]	
	15:0	**Color Transit Slope 12**	
		Default Value:	405
		Format:	U0.16
		The calculation result of 1 / (BC2 – BC1) [1/57]	
47	31:16	**Color Transit Slope 45**	
		Default Value:	407
		Format:	U0.16
		The calculation result of 1 / (BC5 – BC4) [1/57]	
	15:0	**Color Transit Slope 34**	
		Default Value:	1131
		Format:	U0.16
		The calculation result of 1 / (BC4 – BC3) [1/61]	
48	31:16	**Color Transit Slope 61**	
		Default Value:	377
		Format:	U0.16
		The calculation result of 1 / (BC1 – BC6) [1/62]	
	15:0	**Color Transit Slope 56**	
		Default Value:	372
		Format:	U0.16
		The calculation result of 1 / (BC6 – BC5) [1/62]	
49	31:22	**Color Bias 3**	
		Default Value:	0
		Format:	U2.8
		Color bias for BaseColor3.	
	21:12	**Color Bias 2**	
		Default Value:	150
		Format:	U2.8
		Color bias for BaseColor2.	
	11:2	**Color Bias 1**	
		Default Value:	0
		Format:	U2.8
		Color bias for BaseColor1.	

COLOR_PROCESSING_STATE - TCC State				
	1:0	**Reserved**		
		Format:	MBZ	
50	31:22	**Color Bias 6**		
		Default Value:	0	
		Format:	U2.8	
		Color bias for BaseColor6.		
	21:12	**Color Bias 5**		
		Default Value:	0	
		Format:	U2.8	
		Color bias for BaseColor5.		
	11:2	**ColorBias4**		
		Default Value:	0	
		Format:	U2.8	
		Color bias for BaseColor4.		
	1:0	**Reserved**		
		Format:	MBZ	
51	31	**Reserved**		
		Format:	MBZ	
	30:24	**UV Threshold**		
		Default Value:	3	
		Format:	U7	
		Low UV threshold.		
	23:19	**Reserved**		
		Format:	MBZ	
	18:16	**UV Threshold Bits**		
		Default Value:	3	
		Format:	U3	
		Low UV transition width bits.		
	15:13	**Reserved**		
		Format:	MBZ	
	12:8	**STE Threshold**		
		Default Value:	0	
		Format:	U5	
		Skin tone pixels enhancement threshold.		
	7:3	**Reserved**		
		Format:	MBZ	
	2:0	**STE Slope Bits**		
		Default Value:	0	
		Format:	U3	
		Skin tone pixels enhancement slope bits.		
52	31:16	**Inverse UVMax Color**		

COLOR_PROCESSING_STATE - TCC State

		Default Value: 146
		Format: U0.16
		1 / UVMaxColor. Used for the SFs2 calculation.
	15:9	**Reserved**
		Format: MBZ
	8:0	**UVMax Color**
		Default Value: 448
		Format: U9
		The maximum absolute value of the legal UV pixels. Used for the SFs2 calculation.

COLOR_PROCESSING_STATE - PROCAMP State

Default Value: 0x00020001, 0x01000000

This state structure contains the PROCAMP state used by the color processing function.

DWord	Bit	Description
53	31:28	**Reserved** Format: MBZ
	27:17	**Contrast** Default Value: 1 Format: U4.7 Contrast magnitude.
	16:13	**Reserved** Format: MBZ
	12:1	**Brightness** Default Value: 0 Format: S7.4 2's complement Brightness magnitude.
	0	**PROCAMP Enable** Default Value: 1 Format: Enable
54	31:16	**Cos_c_s** Default Value: 256 Format: S7.8 2's complement UV multiplication cosine factor.
	15:0	**Sin_c_s** Default Value: 0 Format: S7.8 2's complement

COLOR_PROCESSING_STATE - PROCAMP State

		UV multiplication sine factor.	

COLOR_PROCESSING_STATE - CSC State

Default Value: 0x00002000, 0x00000000, 0x00000400, 0x00000000, 0x000004B4, 0x00000000, 0x00000000, 0x00000000, 0x00000000

This state structure contains the CSC state used by the color processing function.

DWord	Bit	Description
55	31:29	**Reserved** Format: MBZ
	28:16	**C1** Default Value: 0 Format: S2.10 2's complement Transform coefficient
	15:3	**C0** Default Value: 1024 Format: S2.10 2's complement Transform coefficient
	2	**YUV_IN** Default Value: 0 Format: YUV CSC input offset enable.
	1	**YUV_OUT** Default Value: 0 Format: RGB CSC output offset enable.
	0	**Transform Enable** Format: Enable
56	31:26	**Reserved** Format: MBZ
	25:13	**C3** Default Value: 0 Format: S2.10 2's complement

COLOR_PROCESSING_STATE - CSC State

		Transform coefficient.
	12:0	**C2** Default Value: 0 Format: S2.10 2's complement Transform coefficient.
57	31:26	**Reserved** Format: MBZ
	25:13	**C5** Default Value: 0 Format: S2.10 2's complement Transform coefficient.
	12:0	**C4** Default Value: 1024 Format: S2.10 2's complement Transform coefficient.
58	31:26	**Reserved** Format: MBZ
	25:13	**C7** Default Value: 0 Format: S2.10 2's complement Transform coefficient.
	12:0	**C6** Default Value: 0 Format: S2.10 2's complement Transform coefficient.
59	31:13	**Reserved** Format: MBZ
	12:0	**C8** Default Value: 1204 Format: S2.10 2's complement Transform coefficient.

COLOR_PROCESSING_STATE - CSC State

60	31:20	**Reserved** Format: MBZ
	19:10	**Offset out 1** Default Value: 0 Format: S9 2's complement Offset Out for Y/R.
	9:0	**Offset In 1** Default Value: 0 Format: S9 2's complement Offset in for Y/R.
61	31:20	**Reserved** Format: MBZ
	19:10	**Offset out 2** Default Value: 0 Format: S9 2's complement Offset out for U/G.
	9:0	**Offset in 2** Default Value: 0 Format: S9 2's complement Offset in for U/G.
62	31:20	**Reserved** Format: MBZ
	19:10	**Offset out 3** Default Value: 0 Format: S9 2's complement Offset out for V/B.
	9:0	**Offset in 3** Default Value: 0 Format: S9 2's complement Offset in for V/B.

COLOR_PROCESSING_STATE - CSC State

63	31:17	**Reserved** Format: MBZ
	16	**Alpha from State Select** Format: U1 Enumerated Type

Value	Name	Description
0		Alpha is taken from message
1		Alpha is taken from state

	15:0	**Color Pipe Alpha** Format: U16

COLOR_PROCESSING_STATE - CGC State

Default Value: 0x0CD2911F, 0x30000334, 0x8A800000

This state structure contains the CGC state used by the color processing function.

DWord	Bit	Description
64	31	**Color Gamut Compression Enable**
	30	**Full Range Mapping Enable** Value: 0 — Name: Basic Mode **[Default]** Value: 1 — Name: Advanced Mode
	29:20	**d(in,default)** Default Value: 205 Format: U10 $d_{in.default}$ **InnerTriangleMappingLength**
	19:10	**d(out,default)** Default Value: 164 Format: U10 $d_{out.default}$ **OuterTriangleMappingLength**
	9:0	**d1(out)** Default Value: 287 Format: U10 d^{1}_{out} **OuterTriangleMappingLengthBelow**
65	31	**Reserved** Format: MBZ
	30:28	**Compression Line Shift** Value: 0-4 — Name: Value: 3 — Name: **[Default]**

COLOR_PROCESSING_STATE - CGC State

	27:10	**Reserved** Format: MBZ
	9:0	**d1(in)** Default Value: 820 Format: U10 d^{1}_{in}**InnerTriangleMappingLengthBelow**
66	31	**xvYcc Decode Encode Enable** **Value** / **Name** / **Description** 1 / **[Default]** / Both xvYcc decode and xvYcc encode are enabled 0 / / Disable both xvYcc decode and xvYcc encode **Programming Notes** This bit is valid only when **ColorGamutCompressionnEnable** is on.
	30	**Forced 444 for 444** Default Value: 0 Force the 4:4:4 operation when input video of 4:4:4 format
	29	**Forced 422 for 444** Default Value: 0 Force the 4:2:2 operation when input video of 4:4:4 format
	28	**Forced 444 for 422** Default Value: 0 Force the 4:4:4 operation when input video of 4:2:2 format
	27:26	**STD Factor Mode** **Value** / **Name** / **Description** 00b / STDMin / Select the minimum value of the STD factors 01b / STDMax / Select the maximum value of the STD factors 10b / STDAve **[Default]** / Select the average value of the STD factors 11b / Reserved / **Programming Notes** This field is only valid for input of 4:2:2 (**Forced444_for 422** is disabled), or when (**Forced422_for444** is enabled).
	25:24	**MV Dark Factor Mode** **Value** / **Name** / **Description** 00b / MVDarkMin / Select the minimum value of the MVDark factors 01b / MVDarkMax / Select the maximum value of the MVDark factors 10b / MVDarkAve **[Default]** / Select the average value of the MVDark factors 11b / Reserved / **Programming Notes** This field is only valid for input of 4:2:2 (**Forced444_for 422** is disabled), or when (**Forced422_for444** is enabled).

<table>
<tr><th colspan="4">COLOR_PROCESSING_STATE - CGC State</th></tr>
<tr><td></td><td>23:22</td><td colspan="2">Scaling Factor Mode
This mode is for color gamut compression module
<table><tr><th>Value</th><th>Name</th><th>Description</th></tr><tr><td>00b</td><td>SFMin</td><td>Select the minimum value of the Scaling Factors</td></tr><tr><td>01b</td><td>SFMax</td><td>Select the maximum value of the Scaling Factors</td></tr><tr><td>10b</td><td>SFAve [Default]</td><td>Select the average value of the Scaling Factors</td></tr><tr><td>11b</td><td>Reserved</td><td></td></tr></table>
<table><tr><th>Programming Notes</th></tr><tr><td>This field is only valid for input of 4:2:2 (Forced444_for 422 is disabled), or when (Forced422_for444 is enabled).</td></tr></table></td></tr>
<tr><td></td><td>21:5</td><td colspan="2">Reserved
<table><tr><td>Format:</td><td>Reserved</td></tr></table></td></tr>
<tr><td></td><td>4</td><td colspan="2">Override Saturation Equal Zero
<table><tr><td>Format:</td><td>MBZ</td></tr></table>
<table><tr><th>Programming Notes</th></tr><tr><td>This bit should always be 0.</td></tr></table></td></tr>
<tr><td></td><td>3:0</td><td colspan="2">Display Color Space Mode
<table><tr><th>Value</th><th>Name</th></tr><tr><td>0</td><td>BT709</td></tr><tr><td>1</td><td>BT601</td></tr><tr><td>2-15</td><td>Reserved</td></tr></table></td></tr>
</table>

3.9 Messages

3.9.1 Global Definitions

For data port messages, part of the message descriptor is used to determine the message type. This field is documented here. The remainder of the message descriptor is defined differently depending on the message type, and is documented in the section for the corresponding message.

The Data Port is actually three separate targets, **DataPort Sampler Cache, DataPort Constant Cache,** and **Data Port Render Cache**, each with its own target unit ID. Each target has its own set of message type encodings as shown below.

Restrictions:

Data port messages may not have the **End of Thread** bit set in the message descriptor other than the following exeptions:

The Render Target Write message may have **End of Thread** set for pixel shader threads dispatched by the windower in non-contiguous dispatch mode.

The Render Target UNORM Write message may have **End of Thread** set for pixel shader threads dispatched by the windower in contiguous dispatch mode.

The Media Block Write message may have **End of Thread** set for pixel shader threads dispatched by the windower in contiguous dispatch mode.

3.9.2 Data Port Messages

Most of the messages have an existing definition that is not expected to change. There are several new messages that are documented here.

Data Cache Data Port Message Summary

Message Type	Header Required	Shared Local Memory Support	Stateless Support	Address Modes	Vector Width
OWord Block Read	yes	no	yes	global	1
OWord Block Write	yes	no	yes	global	1
Unaligned OWord Block Read	yes	no	yes	global	1
OWord Dual Block Read	no for stated yes for stateless	no	yes	global + offset	2
OWord Dual Block Write	no for stated yes for stateless	no	yes	global + offset	2
DWord Scattered Read	no for stated yes for stateless	no	yes	global + offset	8, 16
DWord Scattered Write	no for stated yes for stateless	no	yes	global + offset	8, 16
Byte Scattered Read	no for stated yes for stateless	yes		global + offset	8, 16
Byte Scattered Write	no for stated yes for stateless	yes		global + offset	8, 16
Untyped Surface Read	no for stated yes for stateless	yes		1D or 2D	2, 8, 16
Untyped Surface Write	no for stated yes for stateless	yes		1D or 2D	2, 8, 16

Message Type	Header Required	Shared Local Memory Support	Stateless Support	Address Modes	Vector Width
Untyped Atomic Operation	no for stated yes for stateless	yes		1D or 2D	8, 16
Scratch Block Read	yes	no	yes (only)	Imm_Buf + offset	
Scratch Block Write	yes	no	yes (only)	Imm_Buf + offset	
Memory Fence	yes	N/A	N/A	N/A	N/A

“global” is the **Global Offset** in the message header (if header is not present, Global Offset is zero).
“imm_buf” is the Immediate Buffer Base Address provided in message header register M0.5.
“offset” is in the message payload, and is per-slot.
“handle” is the handle address in the message header.
“URBoffset” is the **Global Offset** field in the URB message descriptor.
“1D” and “2D” are the address payload.

Render Cache Data Port Message Summary

Message Type	Header Required	Address Modes	Vector Width
Media Block Read	yes	2D	1
Media Block Write	yes	2D	1
Render Target Write	No[1]	2D + RTAI	8, 16
Typed Surface Read	yes	1D, 2D, 3D, 4D	8
Typed Surface Write	yes	1D, 2D, 3D, 4D	8
Typed Atomic Operation	yes	1D, 2D, 3D, 4D	8
Memory Fence	yes	N/A	N/A

“4D” address refers to U/V/R/LOD for mip-mapped surfaces
“2D + RTAI” address refers to a basic 2D address with render target array index for the third dimension

3.9.2.1 Message Descriptor

The following message descriptor applies to.

DATA PORT SAMPLER CACHE		DATA PORT CONSTANT CACHE		DATA PORT RENDER CACHE	
Bit	Description	Bit	Description	Bit	Description
19	Header Present. If set, indicates that the message includes the header. Refer to Render Target Write message section for more details on this field. Programming Notes: The header must be present unless the message type is Render Target Write Format = Enable				
18	Ignored				
17:16	Ignored	17:16	Ignored	17	Send Write Commit Message. Indicates that a write commit message will be sent back to the thread when the write has been committed. See section *Write*

DATA PORT SAMPLER CACHE		DATA PORT CONSTANT CACHE		DATA PORT RENDER CACHE	
					Commit for more details. This field is ignored on read message types. Format = Enable
15:13	Message Type 000: OWord Block Read 010: OWord Dual Block Read 100: Media Block Read 101: Unaligned OWord Block Read 110: DWord Scattered Read All other encodings are reserved.	15:13	Message Type 000: OWord Block Read 010: OWord Dual Block Read 110: DWord Scattered Read All other encodings are reserved.	16:13	Message Type 0000: OWord Block Read 0001: Render Target UNORM Read 0010: OWord Dual Block Read 0100: Media Block Read 0101: Unaligned OWord Block Read 0110: DWord Scattered Read 0111: DWord Atomic write message 1000: OWord Block Write 1001: OWord Dual Block Write 1010: Media Block Write 1011: DWord Scattered Write 1100: Render Target Write 1101: Streamed Vertex Buffer Write 1110: Render Target UNORM Write All other encodings are reserved.
12:8	**Message Specific Control.** Refer to the specific message section for the definition of these bits.				
7:0	**Binding Table Index.** Specifies the index into the binding table for the specified surface. A binding table index of 255 indicates that a stateless model is to be used. The stateless model is allowed only with the render cache data port. Refer to section 2.2.2 for details on the stateless model. Format = U8 Range = [0,255]				

3.9.2.1.1 Message Descriptor

The following message descriptor applies to.

SAMPLER CACHE DATA PORT		RENDER CACHE DATA PORT	
Bit	**Description**	**Bit**	**Description**
19	**Header Present.** If set, indicates that the message includes the header. **Programming Notes:** For the Render Cache Data Port, the header must be present for the following message types: Typed Surface Read/Write Typed Surface Atomic Operation Memory Fence For the Sampler Cache Data Port, the header must be present for the following message types: Unaligned OWord Block Read Media Block Read. Format = Enable		
18	Ignored	18	Ignored
17:14	Message Type 0001: Unaligned OWord Block Read 0100: Media Block Read All other encodings are reserved.	17:14	Message Type 0100: Media Block Read 0101: Typed Surface Read 0110: Typed Atomic Operation 0111: Memory Fence 1010: Media Block Write 1100: Render Target Write 1101: Typed Surface Write All other encodings are reserved.
13:8	**Message Specific Control.** Refer to the specific message section for the definition of these bits.		
7:0	Binding Table Index. Specifies the index into the binding table for the specified surface. Format = U8 Range = [0,255]		

CONSTANT CACHE DATA PORT		DATA CACHE DATA PORT	
Bit	**Description**	**Bit**	**Description**
19	Header Present. If set, indicates that the message includes the header. Programming Notes:		

CONSTANT CACHE DATA PORT		DATA CACHE DATA PORT	
Bit	**Description**	**Bit**	**Description**
	For the Data Cache Data Port, the header must be present for the following message types: OWord Block Read/Write Unaligned OWord Block Read Memory Fence Scratch read/write For the Constant Cache Data Port, the header must be present for the following message types: OWord Block Read Unaligned OWord Block Read. Format = Enable		
18	Ignored	18	Category 0: Legacy DAP-DC messages 1: Scratch Block Read/Write messages
17:14	Message Type 0000: OWord Block Read 0001: Unaligned OWord Block Read 0010: OWord Dual Block Read 0011: DWord Scattered Read All other encodings are reserved.	17:14	Category=0 (legacy dataport) Message Type 0000: OWord Block Read 0001: Unaligned OWord Block Read 0010: OWord Dual Block Read 0011: DWord Scattered Read 0100: Byte Scattered Read 0101: Untyped Surface Read 0110: Untyped Atomic Operation 0111: Memory Fence 1000: OWord Block Write 1010: OWord Dual Block Write 1011: DWord Scattered Write 1100: Byte Scattered Write 1101: Untyped Surface Write All other encodings are reserved. Category=1 (scratch) [17]: 0=Read; 1=write [16]:Type; 0=Oword, 1= Dword

CONSTANT CACHE DATA PORT		DATA CACHE DATA PORT	
Bit	**Description**	**Bit**	**Description**
			[15]:Invalidate after read; [14]:<Reserved, mbz> [13:12]: Block Size 11: 4 registers 10: <reserved> 01: 2 registers 00: 1 register [11:0]: Addr offset (Hword based)
13:8	**Message Specific Control.** Refer to the specific message section for the definition of these bits.		
7:0	Binding Table Index. Specifies the index into the binding table for the specified surface. For the data cache data port, two binding table indexes are used to select special surfaces: 254: A binding table index of 254 indicates that the shared local memory (SLM) is to be used. The SLM is only supported with the Byte Scattered Read/Write, Untyped Surface Read/Write, and Untyped Atomic Operation messages. Refer to the “Shared Local Memory” section earlier in this chapter for further details on its behavior. 255: A binding table index of 255 indicates that a stateless model is to be used. Stateless model is only supported with the OWord Block Read/Write, Unaligned OWord Block Read, Dual OWord Block Read/Write and DWord Scattered Read/Write messages. Refer to section “Stateless Model” section for details on the stateless model. Format = U8 Range = [0,255]		

3.9.2.2 Message Header

This header applies to the following data port messages:

- OWord Block Read/Write
- Unaligned OWord Block Read
- OWord Dual Block Read/Write
- DWord Scattered Read/Write
- Byte Scattered Read/Write
- Scratch Block Read/Write

The header definitions for the other data port messages is in the section for each message.

DWord	Bit	Description
M0.7	31:0	
M0.6	31:0	
M0.5	31:10	**Immediate Buffer Base Address.** Specifies the surface base address for messages in which the Binding Table Index is 255 (stateless model), otherwise this field is ignored.

DWord	Bit	Description
		This pointer is relative to the **General State Base Address**. Format = GeneralStateOffset[31:10]
	9:8	Ignored
	7:0	**Dispatch ID.** This ID is assigned by the fixed function unit and is a unique identifier for the thread. It is used to free up resources used by the thread upon thread completion.
M0.4	31:0	Ignored (reserved for hardware delivery of binding table pointer)
M0.3	31:4	Ignored
	3:0	Programming Notes: This amount is available to the kernel for information only. It will be passed verbatim (if not altered by the kernel) to the Data Port in any scratch space access messages. The data port will use this to bounds check scratch space messages. Writes out of bounds will be ignored. Reads out of bounds will return 0. Format = U4 Range = [0,11] indicating [1k bytes, 2M bytes] in powers of two
M0.2	31:0	Global Offset. : Specifies the global element offset into the buffer. For the Unaligned OWord messages, this offset is in units of Bytes but must be DWord aligned (bits 1:0 MBZ) For the other OWord messages, this offset is in units of OWords For the DWord messages, this offset is in units of DWords For the Byte messages, this offset is in units of Bytes Format = U32 Range = [0,FFFFFFFCh] for Unaliged OWord messages Range = [0,0FFFFFFFh] for other OWord messages Range = [0,3FFFFFFFh] for DWord messages Range = [0,FFFFFFFFh] for Byte messages
M0.1	31:0	Ignored
M0.0	31:0	Ignored

3.9.2.3 Write Commit Writeback Message

The writeback message is only sent on Data Port Write messages if the **Send Write Commit Message** bit in the message descriptor is set. The destination register is not modified. Write messages without the **Send Write Commit Message** bit set will not return anything to the thread (response length is 0 and destination register is null).

DWord	Bit	Description
W0.7:0		Reserved

3.9.3 OWord Block Read/Write

This message takes one offset (Global Offset), and reads or writes 1, 2, 4, or 8 contiguous OWords starting at that offset.

Restrictions:

1. the only surface type allowed is SURFTYPE_BUFFER.
2. the surface format is ignored, data is returned from the constant buffer to the GRF without format conversion.
3. the surface pitch is ignored, the surface is treated as a 1-dimensional surface. An element size (pitch) of 16 bytes is used to determine the size of the buffer for out-of-bounds checking if using the surface state model.
4. the surface cannot be tiled
5. the surface base address must be OWord aligned
6. the **Render Cache Read Write Mode** field in SURFACE_STATE must be set to read/write mode when using this message with the render cache in the surface state model
7. the **Stateless Render Cache Read-Write Mode** field in the SVG_WORK_CTL register must be set to read/write mode when using this message with the render cache in the stateless model

Applications:

Constant buffer reads of a single constant or multiple contiguous constants.

Scratch space reads/writes where the index for each pixel/vertex is the same.

Block constant reads, scratch memory reads/writes for media.

Execution Mask. The low 8 bits of the execution mask are used to enable the 8 channels in the first and third GRF registers returned (W0, W2) for read, or the first and third write registers sent (M1, M3). The high 8 bits are used similarly for the second and fourth (W1, W3 or M2, M4). For reads, any mask bit asserted within a group of four will cause the entire OWord to be read and returned to the destination GRF register. For writes, each mask bit is considered for its corresponding DWord written to the destination surface.

For the 1-OWord messages, only the low 8 bits of the execution mask are used. Either the low 4 bits or the high 4 bits, depending on the position of the OWord to be read or written, is used as the single group of four with behavior following that in the preceding paragraph.

The above behavior enables a SIMD16 thread to use the 8-OWord form of this message to access two channels (red and green) of a single scratch register across 16 pixels. A second message would access the other two channels (blue and alpha). The execution mask is used to ensure that data associated with inactive pixels are not overwritten.

Out-of-Bounds Accesses. Reads to areas outside of the surface return 0. Writes to areas outside of the surface are dropped and will not modify memory contents.

3.9.3.1 Message Descriptor

Bit	Description
13	Invalidate After Read Enable This field, if enabled, causes all lines in the L3 cache accessed by the message to be invalidated after the read occurs, regardless of whether the line contains modified data. It is intended as a performance hint indicating that the data will no longer be used to avoid writing

Bit	Description
	back data to memory. This field is ignored for write messages. Enabling this field is intended for scratch and spill/fill, where the memory is used only by a single thread and thus does not need to be maintained after the thread completes. Format = Enable
12	Ignored
11	Ignored
10:8	**Block Size.** Specifies the number of contiguous OWords to be read or written 000: 1 OWord, read into or written from the low 128 bits of the destination register 001: 1 OWord, read into or written from the high 128 bits of the destination register 010: 2 OWords 011: 4 OWords 100: 8 OWords all other encodings are reserved. **Programming Notes**: The 6 OWord block size is valid only with **Data Port Constant Cache**.

3.9.3.2 Message Payload (Write)

For the write operation, the message payload consists of one, two, or four registers (not including the header) depending on the **Block Size** specified in the message. For the one-constant case, data is taken from either the high or low half of the payload register depending on the half selected in **Block Size**. In this case, the other half of the payload register is ignored.

The **Offset** referred to below is the **Global Offset** and is in units of OWords. The **OWord** array index is also in units of OWords.

DWord	Bit	Description
M1.7:4	127:0	**OWord[Offset + 1].** If the block size is 1 OWord to be written from the high 128 bits of the destination, OWord[Offset] will appear in this location
M1.3:0	127:0	OWord[Offset]
M2.7:4	127:0	OWord[Offset+3]
M2.3:0	127:0	OWord[Offset+2]
M3.7:4	127:0	OWord[Offset+5]
M3.3:0	127:0	OWord[Offset+4]
M4.7:4	127:0	OWord[Offset+7]
M4.3:0	127:0	OWord[Offset+6]

3.9.3.3 Writeback Message (Read)

For the read operation, the writeback message consists of one, two, three, or four registers depending on the **Block Size** specified in the message. For the one-constant case, data is placed in either the high or

low half of the returned register depending on the half selected in **Block Size**. In this case, the other half of the register is not changed.

The **Offset** referred to below is the **Global Offset** and is in units of OWords. The **OWord** array index is also in units of OWords.

DWord	Bit	Description
W0.7:4	127:0	**OWord[Offset + 1].** If the block size is 1 OWord to be loaded into the high 128 bits of the destination, OWord[Offset] will appear in this location
W0.3:0	127:0	OWord[Offset]
W1.7:4	127:0	OWord[Offset+3]
W1.3:0	127:0	OWord[Offset+2]
W2.7:4	127:0	OWord[Offset+5]
W2.3:0	127:0	OWord[Offset+4]
W3.7:4	127:0	OWord[Offset+7]
W3.3:0	127:0	OWord[Offset+6]

3.9.4 Unaligned OWord Block Read

This message takes one DWord aligned offset (**Global Offset**), and reads 1, 2, 4, or 8 contiguous OWords starting at that offset. This message is identical to the OWord Block Read message except the offset alignment. For read/write cache, only the read path supports this unaligned OWord Block access.

Restrictions:

1. the only surface type allowed is SURFTYPE_BUFFER.
2. the surface format is ignored, data is returned from the constant buffer to the GRF without format conversion.
3. the surface pitch is ignored, the surface is treated as a 1-dimensional surface. An element size (pitch) of 16 bytes is used to determine the size of the buffer for out-of-bounds checking if using the surface state model.
4. the surface cannot be tiled
5. the surface base address must be **OWord** aligned
6. the **Render Cache Read Write Mode** field in SURFACE_STATE must be set to read/write mode when using this message with the render cache in the surface state model
7. the **Stateless Render Cache Read-Write Mode** field in the SVG_WORK_CTL register must be set to read/write mode when using this message with the render cache in the stateless model

Applications:

Reads with offset that is not aligned with data size, such as row store usage in media

Execution Mask. The execution mask is ignored by this message.

Out-of-Bounds Accesses. Reads to areas outside of the surface return 0.

3.9.4.1 Message Descriptor

Bit	Description
13	Ignored
12:11	Ignored
10:8	**Block Size.** Specifies the number of contiguous OWords to be read 000: 1 OWord, read into the low 128 bits of the destination register 001: 1 OWord, read into the high 128 bits of the destination register 010: 2 OWords 011: 4 OWords 100: 8 OWords all other encodings are reserved.

3.9.4.2 Writeback Message (Read)

For the read operation, the writeback message consists of one, two, or four registers depending on the **Block Size** specified in the message. For the one-constant case, data is placed in either the high or low half of the returned register depending on the half selected in **Block Size**. In this case, the other half of the register is not changed.

The **Global Offset** is in units of **Bytes**, aligned to **DWord** (two LSBs set to zero). The **OWordX** array in units of OWord starts at Global Offset.

DWord	Bit	Description
W0.7:4	127:0	**OWord1 = *(&OWord0 + 1).** If the block size is 1 OWord to be loaded into the high 128 bits of the destination, OWord0 will appear in this location
W0.3:0	127:0	OWord0 = Buffer[Global Offset]
W1.7:4	127:0	OWord3 = *(&OWord2 + 1)
W1.3:0	127:0	OWord2 = *(&OWord1 + 1)
W2.7:4	127:0	OWord5= *(&OWord4 + 1)
W2.3:0	127:0	OWord4 = *(&OWord3 + 1)
W3.7:4	127:0	OWord7 = *(&OWord6 + 1)
W3.3:0	127:0	OWord6 = *(&OWord5 + 1)

3.9.5 OWord Dual Block Read/Write

This message takes two offsets, and reads or writes 1 or 4 contiguous OWords starting at each offset. The Global Offset is added to each of the specific offsets.

The message header is no longer required for the *OWord Dual Block Read/Write* messages if sent to the data cache data port. If header is not sent, the **Global Offset** field is assumed to be zero. The header is required, however, if the binding table index is 255 (stateless model), as the **Immediate Buffer Base Address** field is required.

Programming Restrictions: Writes to overlapping addresses will have undefined write ordering.

Restrictions:

1. The only surface type allowed is SURFTYPE_BUFFER.
2. The surface format is ignored, data is returned from the constant buffer to the GRF without format conversion.
3. The surface pitch is ignored, the surface is treated as a 1-dimensional surface. An element size (pitch) of 16 bytes is used to determine the size of the buffer for out-of-bounds checking if using the surface state model.
4. The surface cannot be tiled
5. The surface base address must be OWord aligned
6. The **Render Cache Read Write Mode** field in SURFACE_STATE must be set to read/write mode when using this message with the render cache in the surface state model
7. the **Stateless Render Cache Read-Write Mode** field in the SVG_WORK_CTL register must be set to read/write mode when using this message with the render cache in the stateless model

Applications:

SIMD4x2 constant buffer reads where the indices of each vertex/pixel are different (if there are two indices and they are the same, hardware will optimize the cache accesses and do only one cache access)

SIMD4x2 scratch space reads/writes where the indices are different

Execution Mask. The low 8 bits of the execution mask are used to enable the 8 channels in the GRF registers returned for read, or each of the write registers sent. For reads, any mask bit asserted within a group of four will cause the entire OWord to be read and returned to the destination GRF register. For writes, each mask bit is considered for its corresponding DWord written to the destination surface.

Out-of-Bounds Accesses. Reads to areas outside of the surface return 0. Writes to areas outside of the surface are dropped and will not modify memory contents.

3.9.5.1 Message Descriptor

Bit	Description
13	Invalidate After Read Enable This field, if enabled, causes all lines in the L3 cache accessed by the message to be invalidated after the read occurs, regardless of whether the line contains modified data. It is intended as a performance hint indicating that the data will no longer be used to avoid writing back data to memory. This field is ignored for write messages. Enabling this field is intended for scratch and spill/fill, where the memory is used only by a single thread and thus does not need to be maintained after the thread completes. Format = Enable
12	Ignored
11:10	Ignored
9:8	**Block Size:** Specifies the number of OWords in each block to be read or written 00: 1 OWord

Bit	Description
	10: 4 OWords all other encodings are reserved.

3.9.5.2 Message Payload

DWord	Bit	Description
M1.7	31:0	Ignored
M1.6	31:0	Ignored
M1.5	31:0	Ignored
M1.4	31:0	Block Offset 1. Specifies the OWord offset of OWord Block 1 into the surface. Format = U32 Range = [0,0FFFFFFFh]
M1.3	31:0	Ignored
M1.2	31:0	Ignored
M1.1	31:0	Ignored
M1.0	31:0	Block Offset 0

3.9.5.3 Additional Message Payload (Write)

For the write operation, the message payload consists of one or four registers (not including the header or the first part of the payload) depending on the **Block Size** specified in the message.

The **Offset1/0** referred to below is the **Global Offset** added to the corresponding **Block Offset 1/0** and is in units of OWords . The **OWord** array index is also in units of OWords.

DWord	Bit	Description
M2.7:4	127:0	OWord[Offset1]
M2.3:0	127:0	OWord[Offset0]
M3.7:4	127:0	OWord[Offset1+1]
M3.3:0	127:0	OWord[Offset0+1]
M4.7:4	127:0	OWord[Offset1+2]
M4.3:0	127:0	OWord[Offset0+2]
M4.7:4	127:0	OWord[Offset1+3]
M4.3:0	127:0	OWord[Offset0+3]

3.9.5.4 Writeback Message (Read)

For the read operation, the writeback message consists of one or four registers depending on the **Block Size** specified in the message.

The **Offset1/0** referred to below is the **Global Offset** added to the corresponding **Block Offset 1/0** and is in units of Owords. The **OWord** array index is also in units of OWords.

DWord	Bit	Description
W0.7:4	127:0	OWord[Offset1]
W0.3:0	127:0	OWord[Offset0]
W1.7:4	127:0	OWord[Offset1+1]
W1.3:0	127:0	OWord[Offset0+1]

DWord	Bit	Description
W2.7:4	127:0	OWord[Offset1+2]
W2.3:0	127:0	OWord[Offset0+2]
W3.7:4	127:0	OWord[Offset1+3]
W3.3:0	127:0	OWord[Offset0+3]

3.9.6 Media Block Read/Write

The read form of this message enables a rectangular block of data samples to be read from the source surface and written into the GRF. The write form enables data from the GRF to be written to a rectangular block.

Restrictions:

1. The only surface type allowed is non-arrayed, non-mipmapped SURFTYPE_2D. Because of this, the stateless surface model is not supported with this message.
2. The surface format is used to determine the pixel structure for boundary clamp, the raw data from the surface is returned to the thread without any format conversion nor filtering operation
3. The target cache cannot be the data cache
4. The surface base address must be 32-byte aligned
5. When a surface is XMajor tiled, (**tilewalk** field in the surface state is set to TILEWALK_XMAJOR), a memory area mapped through the Render Cache cannot be read and/or wrote in mixed frame and field modes. For example, if a memory location is first written with a zero Vertical Line Stride (frame mode), and later on (without render cache flush) read back using Vertical Line Stride of one (field mode), the read data stored in GRF are uncertain.
6. The block width and offset should be aligned to the size of pixels stored in the surface. For a surface with 8bpp pixels for example, the block width and offset can be byte aligned. For a surface with 16bpp pixels, it is word aligned.
7. For YUV422 formats, the block width and offset must be pixel pair aligned (i.e. dword aligned).
8. The write form of message has the additional restriction that both **X Offset** and **Block Width** must be DWord aligned.
9. Pitch must be a multiple of 64 bytes when the surface is linear.

Applications:

Block reads/writes for media

Execution Mask. The execution mask on the send instruction for this type of message is ignored. The data that is read or written is determined completely by the block parameters.

Out-of-Bounds Accesses. Reads outside of the surface results in the address being clamped to the nearest edge of the surface and the pixel in the position being returned. Writes outside of the surface are dropped and will not modify memory contents.

Determining the boundary pixel value depends on the surface format. Surface format definitions can be found in the Surface Formats Section of the Sampling Engine Chapter.

For a surface with 8bpp pixels, the boundary byte is replicated. For example, for a boundary dword B0B1B2B3, to replicate the left boundary byte pixel, the out of bound dwords have the format of B0B0B0B0, and that for right boundary is B3B3B3B3.

This rule applies to all surface formats with BPE of 8. As the data port does not perform format conversion, the most likely used surface formats are R8_UINT and R8_SINT.

For any other surfaces with 16bpp pixels, boundary pixel replication is on words. For example, for a boundary dword B0B1B2B3, to replicate the left boundary word pixel, the out of bound dwords have the format of B0B1B0B1, and that for right boundary is B2B3B2B3.

This rule applies to all surface formats with BPE of 16. As the data port does not perform format conversion, only the formats with integer data types may be useful in practice.

For special surfaces with 16bpp pixels YUV422 packed format, there are two basic cases depending on the Y location: YUYV (surface format YCRCB_NORMAL) and UYVY (surface format YCRCB_SWAPY). Boundary handling for YVYU (surface format YCRCB_SWAPUV) is the same as that for YUYV. Similarly, boundary handling for VYUY (surface format YCRCB_SWAPUVY) is the same as that for UYVY. Note that these four surface formats have 16bpp pixels, even though the BPE fields are set to zero according to the table in the Surface Formats Section.

For a boundary dword Y0U0Y1V0, to replicate the left boundary, we get Y0U0**Y0**V0, and to replicate the right boundary, we get **Y1**U0Y1V0.

For a boundary dword U0Y0V0Y1, to replicate the left boundary, we get U0Y0V0**Y0**, and to replicate the right boundary, we get U0**Y1**V0Y1.

For a surface with 32bpp pixels, the boundary dword pixel is replicated.

This rule applies to all surface formats with BPE of 32. As the data port does not perform format conversion, some of the formats may not be useful in practice.

Hardware behavior for any other surface types is undefined.

When Color Processing Enable is set to 1 and the IECP output surface to be written is NV12 format (R16_UNORM surface format 0x10A, should be used if the output surface is NV12 format).

NV12 surface state : The width of the surface should be always multiples of 4pixels. For 16bpp input message (422 8-bit) the width will always need to be in multiples of 8bytes and for 32bpp input message (422 16-bit or 444 8-bit) the width should be in multiples of 16bytes. Height should be in multiples of 2pixel high. (presently the MFX restriction is that width should be in multiples of 2pixels).

y-offset of the media block write from the EU should be always even

x-offset of the media block write from the EU should be in multiples of 4 pixel.

The media block dword write can have only the following combinations (for IECP when NV12 output format is used):

- 8pixel wide for 422 8-bit mode
- 4pixel wide for 422 8-bit mode
- 4pixel wide for 422 16-bit
- 4pixel wide for 444 8-bit.
- 444 16-bit input format cannot be supported when the output format is NV12 (s/w should not use this combination).
- It has to be in multiples of 2pixel high for all above modes.

If 444-format is used then we use only the pixel_0 UV values of the 2x2 pixel and the rest are dropped and in case of 422-format the top UV values are used and the bottom UV values is dropped if the output format is NV12 format.

Assuming IECP messages will always have vertical stride = 0. (since this is only for pre-processing before the encoder).

3.9.6.1 Message Descriptor

<table>
<tr><th>Bit</th><th>Description</th></tr>
<tr><td>13</td><td>Reserved: MBZ</td></tr>
<tr><td>12</td><td>Reserved : MBZ</td></tr>
<tr><td>11</td><td>Reserved : MBZ</td></tr>
<tr><td>10</td><td>Vertical Line Stride Override

Specifies whether the Vertical Line Stride and Vertical Line Stride Offset fields in the surface state should be replaced by bits 9 and 8 below.

If this field is 1, Height in the surface state (see SURFACE_STATE section of Sampling Engine chapter) is modified according the following rules:
<table>
<tr><th>Vertical Line Stride
(in surface state)</th><th>Override Vertical Line Stride</th><th>Derived 1-based surface height
(As a function of the 0-based Height in surface state)</th></tr>
<tr><td>0</td><td>0</td><td>Height + 1
(Normal)</td></tr>
<tr><td>0</td><td>1</td><td>(Height +1) / 2
Restriction: (Height + 1) must be an even number.</td></tr>
<tr><td>1</td><td>0</td><td>(Height + 1) * 2</td></tr>
<tr><td>1</td><td>1</td><td>Height + 1
(Normal)</td></tr>
</table>
For example, for a 720x480 standard resolution video buffer, if Vertical Line Stride in surface state is 0, i.e. a frame, Height (of the frame) should be 479. When accessing the bottom field of this frame video buffer, both Override Vertical Line Stride and Override Vertical Line Stride Offset will be set to 1, then the derived surface height (of the field) will be 240 ((Height + 1) / 2). In contrary, if Vertical Line Stride in surface state is 1 and Vertical Line Stride Offset in surface state is 0, the surface state represents the top field of the video buffer. In this case, Height (of the top field) should be programmed as 239. Accessing the bottom video field will use the same surface height of 240. Accessing the video frame (with Override Vertical Line Stride and Override Vertical Line Stride Offset set to 0) will result in a derived surface height of 480 ((Height + 1) * 2).

0 -- Use parameters in the surface state and ignore bits 9:8

1 -- Use bits 9:8 to provide the Vertical Line Stride and Vertical Line Stride Offset</td></tr>
<tr><td>9</td><td>Override Vertical Line Stride

Specifies number of lines (0 or 1) to skip between logically adjacent lines – provides support of interleaved (field) surfaces as textures.</td></tr>
</table>

Bit	Description
	Format = U1 in lines to skip between logically adjacent lines
8	Override Vertical Line Stride Offset Specifies the offset of the initial line from the beginning of the buffer. Ignored when **Override VerticalLine Stride** is 0. Format = U1 in lines of initial offset (when Vertical Line Stride == 1)

3.9.6.2 Message Header

DWord	Bit	Description
M0.7	31:0	
M0.6	31:0	
M0.5	31:8	Ignored
	7:0	**FFTID.** This ID is assigned by the fixed function unit and is a unique identifier for the thread. It is used to free up resources used by the thread upon thread completion.
M0.4	31:0	Ignored (reserved for hardware delivery of binding table pointer)
M0.3	31:5	**Color Processing State Pointer.** Defines the pointer to COLOR_PROCESSING_STATE. Ignored on read messages and when **Color Processing Enable** is not set. This pointer is relative to the **General State Base Address**. Programming Notes: This pointer is *not* delivered via state variables like most other pointers are delivered. It must be delivered via another software-defined mechanism such as CURBE. Format = GeneralStateOffset[31:5]
	4	Message Mode This field selects the mode of this message as follows: 0: NORMAL. The **Block Height** and **Block Width** fields are set in M0.2. The **Pixel Mask** is not explicitly set but behaves as if it is set to all ones. 1: PIXEL_MASK: The **Pixel Mask** field is set in M0.2. The **Block Height** and **Block Width** are not explicitly set but behave as if they are set to 4 rows and 32 bytes, respectively.
	3:2	**Message Format.** Defines the format of the message if **Color Processing Enable** is set. 0: YUV 4:2:2, 8 bits per channel 1: YUV 4:4:4, 8 bits per channel 2: YUV 4:2:2, 16 bits per channel 3: YUV 4:4:4, 16 bits per channel
	1	**Area of Interest.** This field controls whether the statistic for the luma pixels is collected at VSC for ACE histogram. This field is effective only when the state variable **Full_image_histogram** is disabled.
	0	**Color Processing Enable.** This field controls whether color processing is enabled on a

<table>
<tr><th>DWord</th><th>Bit</th><th>Description</th></tr>
<tr><td></td><td></td><td>media block write message.

Format = Enable

This bit must be set to zero on a Media Block Read to the Render Cache.</td></tr>
<tr><td colspan="3">The following M0.2 definition applies only if the Message Mode field is set to NORMAL:</td></tr>
<tr><td>M0.2</td><td>31:29</td><td>Ignored</td></tr>
<tr><td></td><td>28:24</td><td>Programming Notes:

Sub-Register Offset must be aligned to BasePitch (therefore will be a multiple of DWords as well).

When Register Pitch Control = 0, Sub-Register Offset must align to BasePitch*Block Height and the output fits in a single GRF register.

In general (and specifically when Sub-Register Offset is greater than 0), when the resulting data cross GRF register boundary, the data must be placed symmetrically between GRF registers.

Sub-Register Offset and Register Pitch Control allow software to assembly multiple media block reads directly into a shared GRF register set. For example, if both are set to zero, the read data are written to GRF registers, aligning to the least significant bits of the first register, and the register pitch is equal to the next power-of-2 that is greater than or equal to the Block Width. If Register Pitch Control is non-zero, multiple media block read messages sharing the same Register Pitch Control but with different Sub-Register Offset can fill in the same set of GRF registers with media block data line interleaved.

Format = U5

Range = [0, 28] (Only a multiple of BasePitch, including 0, is valid)

Programming Note: This field must be zero for Render Cache Data Port.</td></tr>
<tr><td></td><td>21:16</td><td>Block Height. Height in rows of block being accessed.

Programming Notes:

The Block Height is restricted to the following maximum values depending on the Block Width:

<table>
<tr><th>Block Width (bytes)</th><th>Maximum Block Height (rows)</th></tr>
<tr><td>1-4</td><td>64</td></tr>
<tr><td>5-8</td><td>32</td></tr>
<tr><td>9-16</td><td>16</td></tr>
<tr><td>17-32</td><td>8</td></tr>
</table>

Format = U6

Range = [0,63] representing 1 to 64 rows</td></tr>
<tr><td></td><td>15:10</td><td>Ignored</td></tr>
<tr><td></td><td>9:8</td><td>Programming Notes:

Register Pitch Control is only allowed to be non-zero, if Block Width is a multiple of</td></tr>
</table>

DWord	Bit	Description
		DWords. The effective register pitch must be less than or equal to 32 bytes (to fit in a single GRF register). Defining **BasePitch** as the next power-of-2 that is greater than or equal to the **Block Width, Register Pitch Control** set the register pitch in term of **BasePitch** as the following. Range = [0,3] representing 1 to 4 **BasePitch** **Programming Note:** This field must be zero for Render Cache Data Port.
	7:5	Ignored
	4:0	**Block Width.** Width in bytes of the block being accessed. Programming Notes: Must be DWord aligned for the write form of the message.
The following M0.2 definition applies only if the Message Mode field is set to PIXEL_MASK:		
MO.2	31:0	**Pixel Mask.** One bit per pixel (each pixel being a DWord) indicating which pixels are to be written. This field is ignored by the read message, all pixels are always returned.. The bits in this mask correspond to the pixels (DWords) as follows: 0 1 4 5 16 17 20 21 2 3 6 7 18 19 22 23 8 9 12 13 24 25 28 29 10 11 14 15 26 27 30 31
M0.1	31:0	**Y offset.** The Y offset of the upper left corner of the block into the surface. Format = S31 Programming Notes: If **Message Mode** is set to PIXEL_MASK, this field must be a multiple of 4
M0.0	31:0	**X offset.** The X offset of the upper left corner of the block into the surface. Must be DWord aligned (Bits 1:0 MBZ) for the write form of the message. The **X offset** field defines the offset in the input message block. This may differ from the offset in the surface if Color Processing is enabled due to format conversion. Programming Notes: If **Message Mode** is set to PIXEL_MASK, this field must be a multiple of 32

Programming Note: The legal combinations of block width, pitch control, sub-register offset and block height are given below:

Block Height for given block width, pitch control, subreg offsets									
		sub-register offsets							
block width	pitch control	0	1	2	3	4	5	6	7
1-4	00	1-64	1	1	1	1	1	1	1
	01	1-64	1-64	illegal	illegal	1-2	1-2	illegal	illegal
	10	illegal	illegal	illegal	illegal	illegal	illegal	illegal	illegal
	11	1-64	1-64	1-64	1-64	illegal	illegal	illegal	illegal
5-8	00	1-32	illegal	1	illegal	1	illegal	1	illegal
	01	1-32	illegal	1-32	illegal	illegal	illegal	illegal	illegal

	10	illegal	illegal	illegal	illegal	illegal	illegal	illegal	illegal
	11	1-32	illegal	1-32	illegal	1-32	illegal	1-32	illegal
9-16	00	1-16	illegal	illegal	illegal	1	illegal	illegal	illegal
	01	1-16	illegal	illegal	illegal	1-16	illegal	illegal	illegal
	10	illegal	illegal	illegal	illegal	illegal	illegal	illegal	illegal
	11	1-16	illegal	illegal	illegal	1-16	illegal	illegal	illegal
7-32	00	1-8	illegal	illegal	illegal	illegal	illegal	illegal	illegal
	01	1-8	illegal	illegal	illegal	illegal	illegal	illegal	illegal
	10	illegal	illegal	illegal	illegal	illegal	illegal	illegal	illegal
	11	1-8	illegal	illegal	illegal	illegal	illegal	illegal	illegal

3.9.6.3 Message Payload (Write)

DWord	Bit	Description
M1:n		**Write Data.** The format of the write data depends on the **Block Height** and **Block Width**. The data is aligned to the least significant bits of the first register, and the register pitch is equal to the next power-of-2 that is greater than or equal to the **Block Width**.

If **Color Processing Enable** is enabled, the write data is divided into pixels according to the **Message Format** field. The fields within each pixel are defined below. For the 4:2:2 modes, each pixel position includes channels for two pixels.

Message Format	31:24	23:16	15:8	7:0
YUV 4:2:2, 8 bits per channel	Cr (V)	right pixel lum (Y1)	Cb (U)	left pixel lum (Y0)
YUV 4:4:4, 8 bits per channel	alpha (A)	luminance (Y)	Cb (U)	Cr (V)
	63:48	47:32	31:16	15:0
YUV 4:2:2, 16 bits per channel	Cr (V)	right pixel lum (Y1)	Cb (U)	left pixel lum (Y0)
YUV 4:4:4, 16 bits per channel	alpha (A)	Cr (V)	luminance (Y)	Cb (U)

3.9.6.4 Writeback Message (Read)

DWord	Bit	Description
W0:n		**Read Data.** The format of the read data depends on the **Block Height** and **Block Width**. The data is aligned to the least significant bits of the first register, and the register pitch is equal to the next power-of-2 that is greater than or equal to the **Block Width**. .

3.9.7 DWord Scattered Read/Write

This message takes a set of offsets, and reads or writes 8 or 16 scattered DWords starting at each offset. The Global Offset is added to each of the specific offsets.

The message header is no longer required for the *OWord DWord Scattered Read/Write* messages if sent to the data cache data port. If header is not sent, the **Global Offset** field is assumed to be zero. The header is required, however, if the binding table index is 255 (stateless model), as the **Immediate Buffer Base Address** field is required.

Programming Restrictions: Writes to overlapping addresses will have undefined write ordering.

For read messages with X/Y offsets that are outside the bounds of the surface, the address is clamped to the nearest edge of the surface. For write messages with X/Y offsets that are outside the bounds of the surface, the behavior is undefined.

Hardware does check for and optimize for cases where offsets are equal or contiguous, however for optimal performance in some these cases a different message may provide higher performance.

Restrictions:

The only surface type allowed is SURFTYPE_BUFFER.

The surface format is ignored, data is returned from the constant buffer to the GRF without format conversion.

The surface pitch is ignored, the surface is treated as a 1-dimensional surface. An element size (pitch) of 16 bytes is used to determine the size of the buffer for out-of-bounds checking if using the surface state model.

The surface cannot be tiled

The surface base address must be DWord aligned

The **Render Cache Read Write Mode** field in SURFACE_STATE must be set to read/write mode when using this message with the render cache in the surface state model

The **Stateless Render Cache Read-Write Mode** field in the SVG_WORK_CTL register must be set to read/write mode when using this message with the render cache in the stateless model

Applications:

SIMD8/16 constant buffer reads where the indices of each pixel are different (read one channel per message)

SIMD8/16 scratch space reads/writes where the indices are different (read/write one channel per message)

general purpose DWord scatter/gathering, used by media

Execution Mask. Depending on the block size, either the low 8 bits or all 16 bits of the execution mask are used to determine which DWords are read into the destination GRF register (for read), or which DWords are written to the surface (for write).

Out-of-Bounds Accesses. Reads to areas outside of the surface return 0. Writes to areas outside of the surface are dropped and will not modify memory contents.

3.9.7.1 Message Descriptor

Bit	Description
13	**Invalidate After Read Enable** This field, if enabled, causes all lines in the L3 cache accessed by the message to be invalidated after the read occurs, regardless of whether the line contains modified data. It is intended as a performance hint indicating that the data will no longer be used to avoid writing back data to memory. This field is ignored for write messages. Enabling this field is intended for scratch and spill/fill, where the memory is used only by a single thread and thus does not need to be maintained after the thread completes. Format = Enable
12:10	Reserved

Bit	Description
9:8	**Block Size.** Specifies the number of DWords to be read or written 10: 8 DWords 11: 16 DWords All other encodings are reserved.

3.9.7.2 Message Payload

DWord	Bit	Description
M1.7	31:0	Offset 7. Specifies the DWord offset of DWord 7 into the surface. Format = U32 Range = [0,3FFFFFFFh]
M1.6	31:0	Offset 6
M1.5	31:0	Offset 5
M1.4	31:0	Offset 4
M1.3	31:0	Offset 3
M1.2	31:0	Offset 2
M1.1	31:0	Offset 1
M1.0	31:0	Offset 0
M2.7	31:0	**Offset 15.** This message register is included only if the block size is 16 DWords.
M2.6	31:0	Offset 14
M2.5	31:0	Offset 13
M2.4	31:0	Offset 12
M2.3	31:0	Offset 11
M2.2	31:0	Offset 10
M2.1	31:0	Offset 9
M2.0	31:0	Offset 8

3.9.7.3 Additional Message Payload (Write)

For the write operation, either one or two additional registers (depending on the block size) of payload contain the data to be written.

The **Offsetn** referred to below is the **Global Offset** added to the corresponding **Offset n** and is in units of DWords . The **DWord** array index is also in units of DWords.

DWord	Bit	Description
M3.7	31:0	DWord[Offset7]
M3.6	31:0	DWord[Offset6]
M3.5	31:0	DWord[Offset5]
M3.4	31:0	DWord[Offset4]
M3.3	31:0	DWord[Offset3]
M3.2	31:0	DWord[Offset2]
M3.1	31:0	DWord[Offset1]
M3.0	31:0	DWord[Offset0]

DWord	Bit	Description
M4.7	31:0	**DWord[Offset15].** This message register is included only if the block size is 16 DWords
M4.6	31:0	DWord[Offset14]
M4.5	31:0	DWord[Offset13]
M4.4	31:0	DWord[Offset12]
M4.3	31:0	DWord[Offset11]
M4.2	31:0	DWord[Offset10]
M4.1	31:0	DWord[Offset9]
M4.0	31:0	DWord[Offset8]

3.9.7.4 Writeback Message (Read)

For the read operation, the writeback message consists of either one or two registers depending on the block size.

The **DWord** array index is also in units of DWords.

DWord	Bit	Description
W0.7	31:0	DWord[Offset7]
W0.6	31:0	DWord[Offset6]
W0.5	31:0	DWord[Offset5]
W0.4	31:0	DWord[Offset4]
W0.3	31:0	DWord[Offset3]
W0.2	31:0	DWord[Offset2]
W0.1	31:0	DWord[Offset1]
W0.0	31:0	DWord[Offset0]
W1.7	31:0	**DWord[Offset15].** This writeback message register is included only if the block size is 16 DWords.
W1.6	31:0	DWord[Offset14]
W1.5	31:0	DWord[Offset13]
W1.4	31:0	DWord[Offset12]
W1.3	31:0	DWord[Offset11]
W1.2	31:0	DWord[Offset10]
W1.1	31:0	DWord[Offset9]
W1.0	31:0	DWord[Offset8]

3.9.8 Byte Scattered Read/Write

These messages are supported on only.

These messages take a set of offsets, and read or write 8 or 16 scattered and possibly misaligned bytes, words, or dwords starting at each offset. The **Global Offset** from the message header is added to each of the specific offsets.

Restrictions:

The only surface type allowed is SURFTYPE_BUFFER.

The surface format is ignored, data is returned from the buffer to the GRF without format conversion.

The surface pitch is ignored, the surface is treated as a 1-dimensional surface. An element size (pitch) of 4 bytes is used to determine the size of the buffer for out-of-bounds checking if using the surface state model.

The surface cannot be tiled

The surface base address must be DWord aligned

The stateless model is not supported.

Applications:

Byte aligned buffer accesses in compute shaders

Execution Mask. Depending on the block size, either the low 8 bits or all 16 bits of the execution mask are used to determine which slots are read into the destination GRF register (for read), or which slots are written to the surface (for write).

Out-of-Bounds Accesses. Reads to areas outside of the surface return 0. Writes to areas outside of the surface are dropped and will not modify memory contents.

Programming Restrictions: Writes to overlapping addresses will have undefined write ordering.

3.9.8.1 Message Descriptor

Bit	Description
13:12	Ignored
11:10	**Data Size.** Specifies the data size for each slot. 0: 1 byte 1: 2 bytes 2: 4 bytes 3: Reserved
9	Ignored
8	**SIMD Mode.** Specifies the SIMD mode of the message (number of slots processed). 0: SIMD8 1: SIMD16

3.9.8.2 Message Payload

DWord	Bit	Description
M1.7	31:0	Offset 7. Specifies the byte offset of DWord 7 into the surface. Format = U32 Range = [0,FFFFFFFFh]
M1.6	31:0	Offset 6
M1.5	31:0	Offset 5
M1.4	31:0	Offset 4
M1.3	31:0	Offset 3
M1.2	31:0	Offset 2
M1.1	31:0	Offset 1
M1.0	31:0	Offset 0
M2.7	31:0	**Offset 15.** This message register is included only if the **SIMD Mode** is SIMD16.

DWord	Bit	Description
M2.6	31:0	Offset 14
M2.5	31:0	Offset 13
M2.4	31:0	Offset 12
M2.3	31:0	Offset 11
M2.2	31:0	Offset 10
M2.1	31:0	Offset 9
M2.0	31:0	Offset 8

3.9.8.3 Additional Message Payload (Write)

For the write operation, either one or two additional registers (depending on the block size) of payload contain the data to be written.

The **Offsetn** referred to below is the **Global Offset** added to the corresponding **Offset n** and is in units of bytes. The length of **Data** written depends on the **Data Size** and is right-justified within the 32-bit field. The upper bits are ignored for 1 byte and 2 byte **Data Size**.

DWord	Bit	Description
M3.7	31:0	Data[Offset7]
M3.6	31:0	Data[Offset6]
M3.5	31:0	Data[Offset5]
M3.4	31:0	Data[Offset4]
M3.3	31:0	Data[Offset3]
M3.2	31:0	Data[Offset2]
M3.1	31:0	Data[Offset1]
M3.0	31:0	Data[Offset0]
M4.7	31:0	**Data[Offset15].** This message register is included only if the **SIMD Mode** is SIMD16.
M4.6	31:0	Data[Offset14]
M4.5	31:0	Data[Offset13]
M4.4	31:0	Data[Offset12]
M4.3	31:0	Data[Offset11]
M4.2	31:0	Data[Offset10]
M4.1	31:0	Data[Offset9]
M4.0	31:0	Data[Offset8]

3.9.8.4 Writeback Message (Read)

For the read operation, the writeback message consists of either one or two registers depending on the block size.

The **Offsetn** referred to below is the **Global Offset** added to the corresponding **Offset n** and is in units of bytes. The length of **Data** written depends on the **Data Size** and is right-justified within the 32-bit field and only the requeted bytes are written to the GRF.

DWord	Bit	Description
W0.7	31:0	Data[Offset7]
W0.6	31:0	Data[Offset6]
W0.5	31:0	Data[Offset5]
W0.4	31:0	Data[Offset4]
W0.3	31:0	Data[Offset3]
W0.2	31:0	Data[Offset2]
W0.1	31:0	Data[Offset1]

DWord	Bit	Description
W0.0	31:0	Data[Offset0]
W1.7	31:0	**Data[Offset15].** This message register is included only if the **SIMD Mode** is SIMD16.
W1.6	31:0	Data[Offset14]
W1.5	31:0	Data[Offset13]
W1.4	31:0	Data[Offset12]
W1.3	31:0	Data[Offset11]
W1.2	31:0	Data[Offset10]
W1.1	31:0	Data[Offset9]
W1.0	31:0	Data[Offset8]

3.9.9 Typed/Untyped Surface Read/Write and Typed/Untyped Atomic Operation

Six data port messages (Typed Surface Read, Typed Surface Write, Typed Atomic Operation, Untyped Surface Read, Untyped Surface Write, and Untyped Atomic Operation) allow direct read/write accesses to surfaces. These messages support three major categories of surfaces:

Typed surfaces. These surfaces are of type SURFTYPE_1D, 2D, 3D, or BUFFER and have a supported surface format other than RAW. Supported via the render cache data port.

Programming Restriction: Vertical stride & Vertical Offset fields of the surface state object is only supported for 2D non-array surfaces.

Raw buffer (untyped). These surfaces are of type SURFTYPE_BUFFER and have a surface format of RAW and a surface pitch of 1 byte. Supported via the data cache data port. All SLM accesses are in this category.

Structured buffer (untyped). These surfaces are of type SURFTYPE_STRBUF and have a surface format of RAW. Supported via the data cache data port.

A typed surface uses U, V, R, and LOD address parameters (number of parameters utilized depends on surface type), and performs conversion of type to/from the selected surface format as follows:

Surface formats with UINT require the message data in U32 format

Surface formats with SINT require the message data in S32 format

All other surface formats require the message data in FLOAT32 format

The untyped surface categories, both of which use the RAW surface format, perform no type conversion. A raw buffer uses just the U address parameter, which specifies the byte offset into the surface, which must be a multiple of 4. A structured buffer uses the U address parameter as an array index and the V address parameter as a byte offset into the array element (which also must be a multiple of 4).

For both raw and structured buffers, up to 4 dwords are accessed beginning at the byte address determined. These 4 dwords correspond to the red, green, blue, and alpha channels in that order with red mapping to the lowest order dword. The atomic operation messages will only access the first dword (corresponding to the red channel for typed messages).

The atomic operation messages causes atomic read-modify-write operations on the “destination” location addressed. In the table below, the new value of the destination (new_dst) is computed as indicated based on the old value of the destination (old_dst) and up to two sources included in the message (src0 and src1). Optionally, a value can be returned by the message (ret).

The atomic operations guarantee that the read and the write are performed atomically, meaning that no read or write to the same memory location from this thread or any other thread can occur between the read and the write.

The following atomic operations are available, along with the specific operation performed for each and the return value:

Atomic Operation	new_dst	ret
AOP_AND	old_dst & src0	old_dst
AOP_OR	old_dst | src0	old_dst
AOP_XOR	old_dst ^ src0	old_dst
AOP_MOV	src0	old_dst
AOP_INC	old_dst + 1	old_dst
AOP_DEC	old_dst – 1	old_dst
AOP_ADD	old_dst + src0	old_dst
AOP_SUB	old_dst – src0	old_dst
AOP_REVSUB	src0 – old_dst	old_dst
AOP_IMAX	imax(old_dst, src0)	old_dst
AOP_IMIN	imin(old_dst, src0)	old_dst
AOP_UMAX	umax(old_dst, src0)	old_dst
AOP_UMIN	umin(old_dst, src0)	old_dst
AOP_CMPWR	(src0 == old_dst) ? src1 : old_dst	old_dst
AOP_PREDEC	old_dst – 1	new_dst
AOP_CMPWR8B	(src08B == old_dst8B) ? src18B : old_dst8B	old_dst8B

Programming Note: src08B is 8 bytes, src18B is 8 Bytes and old_dst8B is 8 bytes in length.

Programming Note: AOP_CMPWR8B is not supported for SLM.

Programming Note: AOP_CMPWR8B addresses must be QWORD aligned.

Note: imax/imin assume operands are signed integers, umax/umin assume operands are unsigned integers. All other operations treat all values as 32-bit unsigned integers. Add and subtract operations will wrap without any special indication.

These messages are supported on only.

Restrictions:

For untyped messages, the **Surface Format** must be RAW and the **Surface Type** must be SURFTYPE_BUFFER or SURFTYPE_STRBUF.

For typed messages, the **Surface Type** must be SURFTYPE_1D, 2D, 3D, or BUFFER.

Surface Format Name
R32_SINT
R32_UINT
R32_FLOAT

The Surface Format for typed surface writes must be

Surface Format Name
R32G32B32A32_FLOAT
R32G32B32A32_SINT
R32G32B32A32_UINT
R16G16B16A16_UNORM
R16G16B16A16_SNORM
R16G16B16A16_SINT

Surface Format Name
R16G16B16A16_UINT
R16G16B16A16_FLOAT
R32G32_FLOAT
R32G32_SINT
R32G32_UINT
B8G8R8A8_UNORM
R10G10B10A2_UNORM
R10G10B10A2_UINT
R8G8B8A8_UNORM
R8G8B8A8_SNORM
R8G8B8A8_SINT
R8G8B8A8_UINT
R16G16_UNORM
R16G16_SNORM
R16G16_SINT
R16G16_UINT
R16G16_FLOAT
B10G10R10A2_UNORM
R11G11B10_FLOAT
R32_SINT
R32_UINT
R32_FLOAT
B5G6R5_UNORM
B5G5R5A1_UNORM
B4G4R4A4_UNORM
R8G8_UNORM
R8G8_SNORM
R8G8_SINT
R8G8_UINT
R16_UNORM
R16_SNORM
R16_SINT
R16_UINT
R16_FLOAT
B5G5R5X1_UNORM
R8_UNORM
R8_SNORM
R8_SINT
R8_UINT
A8_UNORM

The **Surface Format** for typed atomic operations must be R32_UINT or R32_SINT.

For untyped messages accessing SURFTYPE_STRBUF, the V address (byte offset) must be DWord aligned (low 2 bits must be zero).

For untyped messages accessing SURFTYPE_BUFFER, the U address (byte offset) must be DWord aligned (low 2 bits must be zero).

Typed messages only support SIMD8.

The stateless model support is limited to untyped messages.

Execution Mask:

SIMD16: The 16 bits of the execution mask are ANDed with the 16 bits of the **Pixel/Sample Mask** from the message header and the resulting mask is used to determine which slots are read into the destination GRF register (for read), or which slots are written to the surface (for write). If the header is not present, only the execution mask is used.

SIMD8: The low 8 bits of the execution mask are ANDed with 8 bits of the **Pixel/Sample Mask** from the message header. For the typed messages, the **Slot Group** in the message descriptor selects either the low or high 8 bits. For the untyped messages, the low 8 bits are always selected. The resulting mask is used to determine which slots are read into the destination GRF register (for read), or which slots are written to the surface (for write). If the header is not present, only the low 8 bits of the execution mask are used.

SIMD4x2: Each group of 4 bits within the low 8 bits of the execution mask are ORed together to create two bits which are used to determine which slots are read into the destination GRF register.

Out–of–Bounds Accesses: Reads to areas outside of the surface return 0, except for the *Typed Surface Read* message which returns 1 in the alpha channel and 0 in the other channels. Writes to areas outside of the surface are dropped and will not modify memory contents.

Errata: The *Typed Surface Read* returns 0 in all channels for out-of-bounds accesses.

Programming Restrictions: Writes to overlapping addresses will have undefined write ordering.

The following table summarizes the SIMD Mode support for each message type:

	Untyped			Typed		
	Read	Write	Atomic	Read	Write	Atomic
SIMD16	x	x	x			
SIMD8	x	x	x	x	x	x

The following table indicates the hardware interpretation of each input parameter based on surface type. Parameters with blank entries are ignored by hardware if delivered.

Surface Type	“Surface Array” field in SURFACE_STATE	U Address	V Address	R Address	LOD
SURFTYPE_1D	disabled	X pixel address			LOD
	enabled	X pixel address	array index		LOD
SURFTYPE_2D	disabled	X pixel address	Y pixel address		LOD
	enabled	X pixel address	Y pixel address	array index	LOD
SURFTYPE_3D	disabled	X pixel address	Y pixel address	Z pixel address	LOD
SURFTYPE_BUFFER	disabled	buffer index			
SURFTYPE_STRBUF	disabled	buffer index	byte offset		

3.9.9.1 Typed Surface Read/Write Message Descriptor

Bit	Description
13	Slot Group This field controls which 8 bits of Pixel/Sample Mask in the message header are ANDed with the execution mask to determine which slots are accessed. This field is ignored if the header is not present. Format = U1 0: Use low 8 slots 1: Use high 8 slots
12	Ignored
11	Alpha Channel Mask For the read message, indicates that alpha will be included in the writeback message. For the write message, indicates that alpha is included in the message payload, and that alpha will be written to the surface. 0: Alpha channel included 1: Alpha channel not included Programming Notes: At least one of the channels must be unmasked (the 4-bit channel mask cannot be 1111b).
10	Blue Channel Mask
9	Green Channel Mask
8	Red Channel Mask

3.9.9.2 Untyped Surface Read/Write Message Descriptor

Bit	Description
13:12	**SIMD Mode** Format = U2 0: SIMD4x2 (valid for read message only) (valid for read message only) , 1: SIMD16 2: SIMD8 3: Reserved
11	**Alpha Channel Mask** For the read message, indicates that alpha will be included in the writeback message. For the write message, indicates that alpha is included in the message payload, and that alpha will be written to the surface. 0: Alpha channel included 1: Alpha channel not included **Programming Notes:**

Bit	Description
	For the *Untyped Surface Write* message, each channel mask cannot be 0 unless all of the lower mask bits are also zero. This means that the only 4-bit channel mask values allowed are 0000b, 1000b, 1100b, and 1110b. Other messages allow any combination of channel masks. For the *Untyped Surface Read* message, at least one of the channels must be unmasked (the 4-bit channel mask cannot be 1111b).
10	**Blue Channel Mask**
9	**Green Channel Mask**
8	**Red Channel Mask**

3.9.9.3 Typed Atomic Operation Message Descriptor

Bit	Description
13	Return Data Control Specifies whether return data is sent back to the thread. Format = Enable
12	Slot Group This field controls which 8 bits of Pixel/Sample Mask in the message header are ANDed with the execution mask to determine which slots are accessed. Format = U1 0: Use low 8 slots 1: Use high 8 slots
11:8	Atomic Operation Type Specifies the atomic operation to be performed. 0000: Reserved 0001: AOP_AND 0010: AOP_OR 0011: AOP_XOR 0100: AOP_MOV 0101: AOP_INC 0110: AOP_DEC 0111: AOP_ADD 1000: AOP_SUB 1001: AOP_REVSUB 1010: AOP_IMAX 1011: AOP_IMIN 1100: AOP_UMAX

Bit	Description
	1101: AOP_UMIN 1110: AOP_CMPWR 1111: AOP_PREDEC

3.9.9.4 Typed Atomic Operation SIMD4x2 Message Descriptor

Bit	Description
13	Return Data Control Specifies whether return data is sent back to the thread. Format = Enable
12	Reserved
11:8	Atomic Operation Type Specifies the atomic operation to be performed. 0000: reserved 0001: AOP_AND 0010: AOP_OR 0011: AOP_XOR 0100: AOP_MOV 0101: AOP_INC 0110: AOP_DEC 0111: AOP_ADD 1000: AOP_SUB 1001: AOP_REVSUB 1010: AOP_IMAX 1011: AOP_IMIN 1100: AOP_UMAX 1101: AOP_UMIN 1110: AOP_CMPWR 1111: AOP_PREDEC

3.9.9.5 Untyped Atomic Operation Message Descriptor

Bit	Description
13	Return Data Control Specifies whether return data is sent back to the thread. Format = Enable

Bit	Description
12	SIMD Mode Format = U1 0: SIMD16 1: SIMD8
11:8	Atomic Operation Type Specifies the atomic operation to be performed. 0000: 0000: AOP_CMPWR8B 0001: AOP_AND 0010: AOP_OR 0011: AOP_XOR 0100: AOP_MOV 0101: AOP_INC 0110: AOP_DEC 0111: AOP_ADD 1000: AOP_SUB 1001: AOP_REVSUB 1010: AOP_IMAX 1011: AOP_IMIN 1100: AOP_UMAX 1101: AOP_UMIN 1110: AOP_CMPWR 1111: AOP_PREDEC

3.9.9.6 Untyped Atomic Operation SIMD4x2 Message Descriptor

Bit	Description
13	Return Data Control Specifies whether return data is sent back to the thread. Format = Enable
12	Reserved
11:8	Atomic Operation Type Specifies the atomic operation to be performed. 0000: AOP_CMPWR8B 0001: AOP_AND

Bit	Description
	0010: AOP_OR 0011: AOP_XOR 0100: AOP_MOV 0101: AOP_INC 0110: AOP_DEC 0111: AOP_ADD 1000: AOP_SUB 1001: AOP_REVSUB 1010: AOP_IMAX 1011: AOP_IMIN 1100: AOP_UMAX 1101: AOP_UMIN 1110: AOP_CMPWR 1111: AOP_PREDEC

3.9.9.7 Atomic Counter Operation Message Descriptor

Bit	Description
13	Return Data Control Specifies whether return data is sent back to the thread. Format = Enable
12	SIMD Mode Format: U1 0: SIM16 1: SIM8 (low 8 slots)
11:8	Atomic Operation Type Specifies the atomic operation to be performed. 0000: Reserved 0001: AOP_AND 0010: AOP_OR 0011: AOP_XOR 0100: AOP_MOV 0101: AOP_INC 0110: AOP_DEC

Bit	Description
	0111: AOP_ADD 1000: AOP_SUB 1001: AOP_REVSUB 1010: AOP_IMAX 1011: AOP_IMIN 1100: AOP_UMAX 1101: AOP_UMIN 1110: AOP_CMPWR 1111: AOP_PREDEC

For Append Counter Operations there is no address payload as the address is provided by the append counter field in the surface state. The write data payloads are the same as untyped atomic. The write back are the same as untyped atomic.

When accessing a surface with the Append Counter Operation, if the Append Counter enable field of the surface state is not 1, it the access will be treated as out of bounds, w/ the writes being ignored and the reads returning 0.

3.9.9.8 Atomic Counter Operation SIMD4x2 Message Descriptor

Bit	Description
13	Return Data Control Specifies whether return data is sent back to the thread. Format = Enable
12	Reserved
11:8	Atomic Operation Type Specifies the atomic operation to be performed. 0000: Reserved 0001: AOP_AND 0010: AOP_OR 0011: AOP_XOR 0100: AOP_MOV 0101: AOP_INC 0110: AOP_DEC 0111: AOP_ADD 1000: AOP_SUB 1001: AOP_REVSUB 1010: AOP_IMAX

Bit	Description
	1011: AOP_IMIN 1100: AOP_UMAX 1101: AOP_UMIN 1110: AOP_CMPWR 1111: AOP_PREDEC

For Append Counter Operations there is no address payload as the address is provided by the append counter field in the surface state. The write data payloads are the same as untyped atomic 4x2. The write back are the same as untyped atomic 4x2.

When accessing a surface with the Append Counter Operation, if the Append Counter enable field of the surface state is not 1, it the access will be treated as out of bounds, w/ the writes being ignored and the reads returning 0.

3.9.9.9 Message Header

The message header for the untyped messages only needs to be delivered for pixel shader threads, where the execution mask may indicate pixels/samples that are enabled only due to derivative (LOD) calculations, but the corresponding slot on the surface must not be accessed. Typed messages (which go to render cache data port) must include the header.

DWord	Bit	Description
M0.7	31:16	Ignored
	15:0	**Pixel/Sample Mask.** This field contains the 16-bit pixel/sample mask to be used for SIMD16 and SIMD8 messages. All 16 bits are used for SIMD16 messages. For typed SIMD8 messages, **Slot Group** selects with 8 bits of this field are used. For untyped SIMD8 messages, the low 8 bits of this field are used. If the header is not delivered, this field defaults to all ones. The field is ignored for SIMD4x2 messages.
M0.6	31:0	Ignored
M0.5	31:0	Format = GeneralStateOffset[31:10]
M0.4	31:0	Ignored (reserved for hardware delivery of binding table pointer)
M0.3	31:0	Ignored
M0.2	31:0	Ignored
M0.1	31:0	Ignored
M0.0	31:0	Ignored

3.9.9.10 Message Payload

The message payload consists of the following:

- For the read messages, only an address payload is delivered
- For the write messages, an address payload is followed by the write data payload
- For the atomic operation messages, an address payload is followed by the source payload
- For SIMD16 and SIMD8 messages, the message length is used to determine how may address parameters are included in the message. The number of message registers in the write data payload is determined by the number of channel mask bits that are enabled, and the number of message

registers in the source payload is determined by the atomic operation operation. Thus, one or neither of these two values (depending on the message type), plus one for the header, can be subtracted from the message length to determine the number of message registers in the address payload, from which the number of address parameters can be determined.

3.9.9.10.1 SIMD16 Address Payload

The payload of a SIMD16 message provides address parameters to process 16 slots. The possible address parameters are U and V (since SIMD16 is only supported with untyped messages). The number of parameters required depends on the surface type being accessed. Each parameter takes two message registers. Each parameter always takes a consistent position in the input payload. The length field can be used to send a shorter message, but intermediate parameters cannot be skipped as there is no way to signal this.

DWord	Bit	Description
M1.7	31:0	Slot 7 U Address Specifies the U Address for slot 7. Format = U32
M1.6	31:0	Slot 6 U Address
M1.5	31:0	Slot 5 U Address
M1.4	31:0	Slot 4 U Address
M1.3	31:0	Slot 3 U Address
M1.2	31:0	Slot 2 U Address
M1.1	31:0	Slot 1 U Address
M1.0	31:0	Slot 0 U Address
M2.7	31:0	Slot 15 U Address
M2.6	31:0	Slot 14 U Address
M2.5	31:0	Slot 13 U Address
M2.4	31:0	Slot 12 U Address
M2.3	31:0	Slot 11 U Address
M2.2	31:0	Slot 10 U Address
M2.1	31:0	Slot 9 U Address
M2.0	31:0	Slot 8 U Address
M3		Slots 7:0 V Address
M4		Slots 15:8 V Address

3.9.9.10.2 SIMD16 Source Payload (Atomic Operation message only)

The source payload follows the address payload for atomic operation messages. Depending on the atomic operation, zero, one, or two sources are required. If the source is not required, it must not be included. Message registers given here could be a lower number if some of the address parameters are not included.

The following atomic operations require no sources, thus the source payload is not delivered: AOP_INC, AOP_DEC, AOP_PREDEC

The following atomic operations require both Source0 and Source1: AOP_CMPWR

All of the remaining atomic operations require Source0 only.

DWord	Bit	Description
M5.7	31:0	Slot 7 Source0 Specifies Source0 for slot 7. Format = S31 for AOP_IMAX and AOP_IMIN, U32 for all other operations
M5.6	31:0	Slot 6 Source0
M5.5	31:0	Slot 5 Source0
M5.4	31:0	Slot 4 Source0
M5.3	31:0	Slot 3 Source0
M5.2	31:0	Slot 2 Source0
M5.1	31:0	Slot 1 Source0
M5.0	31:0	Slot 0 Source0
M6.7	31:0	Slot 15 Source0
M6.6	31:0	Slot 14 Source0
M6.5	31:0	Slot 13 Source0
M6.4	31:0	Slot 12 Source0
M6.3	31:0	Slot 11 Source0
M6.2	31:0	Slot 10 Source0
M6.1	31:0	Slot 9 Source0
M6.0	31:0	Slot 8 Source0
M7		Slots 7:0 Source1
M8		Slots 15:8 Source1

3.9.9.10.3 SIMD16 Source Payload (AOP_CMPWR8B only)

DWord	Bit	Description
M5.7	31:0	Slot 7 Source0[31:0] Specifies Source0[31:0] for slot 7. Format = U32
M5.6	31:0	Slot 6 Source0[31:0]
M5.5	31:0	Slot 5 Source0[31:0]
M5.4	31:0	Slot 4 Source0[31:0]
M5.3	31:0	Slot 3 Source0[31:0]
M5.2	31:0	Slot 2 Source0[31:0]
M5.1	31:0	Slot 1 Source0[31:0]
M5.0	31:0	Slot 0 Source0[31:0]
M6.7	31:0	Slot 15 Source0[31:0]
M6.6	31:0	Slot 14 Source0[31:0]
M6.5	31:0	Slot 13 Source0[31:0]
M6.4	31:0	Slot 12 Source0[31:0]
M6.3	31:0	Slot 11 Source0[31:0]
M6.2	31:0	Slot 10 Source0[31:0]
M6.1	31:0	Slot 9 Source0[31:0]
M6.0	31:0	Slot 8 Source0[31:0]
M7		Slots 7:0 Source0[63:32]
M8		Slots 15:8 Source0[63:32]
M9		Slots 7:0 Source1[31:0]
M10		Slots 15:8 Source1[31:0]
M11		Slots 7:0 Source1[63:32]

DWord	Bit	Description
M12		Slots 15:8 Source1[63:32]

3.9.9.10.4 SIMD16 Write Data Payload (Write message only)

The write data payload follows the address payload for write messages. Actual position within the message may vary if some of the parameters are not included or if some of the channel mask bits are asserted. Any parameter or write channel not included in the payload is skipped, with message phases below it being renumbered to take up the vacated space.

DWord	Bit	Description
M5.7	31:0	Slot 7 Red Specifies the value of the red channel to be written for slot 7. Format = 32 bits raw data.
M5.6	31:0	Slot 6 Red
M5.5	31:0	Slot 5 Red
M5.4	31:0	Slot 4 Red
M5.3	31:0	Slot 3 Red
M5.2	31:0	Slot 2 Red
M5.1	31:0	Slot 1 Red
M5.0	31:0	Slot 0 Red
M6.7	31:0	Slot 15 Red
M6.6	31:0	Slot 14 Red
M6.5	31:0	Slot 13 Red
M6.4	31:0	Slot 12 Red
M6.3	31:0	Slot 11 Red
M6.2	31:0	Slot 10 Red
M6.1	31:0	Slot 9 Red
M6.0	31:0	Slot 8 Red
M7		Slots 7:0 Green
M8		Slots 15:8 Green
M9		Slots 7:0 Blue
M10		Slots 15:8 Blue
M11		Slots 7:0 Alpha
M12		Slots 15:8 Alpha

3.9.9.10.5 SIMD8 Address Payload

The payload of a SIMD8 message provides address parameters to process 8 slots. The possible address parameters are U, V, R, and LOD. The number of parameters required depends on the surface type being accessed. Each parameter takes one message register. Each parameter always takes a consistent position in the input payload. The length field can be used to send a shorter message, but intermediate parameters cannot be skipped as there is no way to signal this.

DWord	Bit	Description
M1.7	31:0	Slot 7 U Address Specifies the U Address for slot 7. Format = U32
M1.6	31:0	Slot 6 U Address
M1.5	31:0	Slot 5 U Address
M1.4	31:0	Slot 4 U Address
M1.3	31:0	Slot 3 U Address
M1.2	31:0	Slot 2 U Address
M1.1	31:0	Slot 1 U Address
M1.0	31:0	Slot 0 U Address
M2		Slots 7:0 V Address
M3		Slots 7:0 R Address Programming Notes: This register can only be delivered for the *Typed* message types.
M4		Slots 7:0 LOD Programming Notes: This register can only be delivered for the *Typed* message types.

Errata: Overlapping addresses (identical U/V/R/LOD) in the same simd8 message is not supported for typed (non-atomic) writes to tiled surfaces.

3.9.9.10.6 SIMD8 Source Payload (Atomic Operation message only)

The source payload follows the address payload for atomic operation messages. Depending on the atomic operation, zero, one, or two sources are required. If the source is not required, it must not be included. Message registers given here could be a lower number if some of the address parameters are not included.

The following atomic operations require no sources, thus the source payload is not delivered: AOP_INC, AOP_DEC, AOP_PREDEC

The following atomic operations require both Source0 and Source1: AOP_CMPWR

All of the remaining atomic operations require Source0 only.

DWord	Bit	Description
M5.7	31:0	Slot 7 Source0 Specifies Source0 for slot 7. Format = S31 for AOP_IMAX and AOP_IMIN, U32 for all other operations
M5.6	31:0	Slot 6 Source0
M5.5	31:0	Slot 5 Source0
M5.4	31:0	Slot 4 Source0
M5.3	31:0	Slot 3 Source0
M5.2	31:0	Slot 2 Source0

DWord	Bit	Description
M5.1	31:0	Slot 1 Source0
M5.0	31:0	Slot 0 Source0
M6		Slots 7:0 Source1

3.9.9.10.7 SIMD8 Write Data Payload (Write message only)

The write data payload follows the address payload for write messages. Actual position within the message may vary if some of the parameters are not included or if some of the channel mask bits are asserted. Any parameter or write channel not included in the payload is skipped, with message phases below it being renumbered to take up the vacated space.

DWord	Bit	Description
M5.7	31:0	Slot 7 Red Specifies the value of the red channel to be written for slot 7. For *Untyped* messages: Format = 32 bits raw data. For *Typed* messages: Format = IEEE Float, S31, or U32 depending on the **Surface Format** of the surface being accessed. SINT formats use S31, UINT formats use U32, and all other formats use Float.
M5.6	31:0	Slot 6 Red
M5.5	31:0	Slot 5 Red
M5.4	31:0	Slot 4 Red
M5.3	31:0	Slot 3 Red
M5.2	31:0	Slot 2 Red
M5.1	31:0	Slot 1 Red
M5.0	31:0	Slot 0 Red
M6		Slots 7:0 Green
M7		Slots 7:0 Blue
M8		Slots 7:0 Alpha

3.9.9.10.8 SIMD8 Write Data Payload (Tile W Write message only)

The write data payload follows the address payload for write messages. Actual position within the message may vary if some of the parameters are not included.

DWord	Bit	Description
M5.7	31:8	Ignored
	7:0	Slot 7 Red Specifies the value of the red channel to be written for slot 7. For *Typed* messages: Format = U8
M5.6	31:8	Ignored
	7:0	Slot 6 Red
M5.5	31:8	Ignored
	7:0	Slot 5 Red

DWord	Bit	Description
M5.4	31:8	Ignored
	7:0	Slot 4 Red
M5.3	31:8	Ignored
	7:0	Slot 3 Red
M5.2	31:8	Ignored
	7:0	Slot 2 Red
M5.1	31:8	Ignored
	7:0	Slot 1 Red
M5.0	31:8	Ignored
	7:0	Slot 0 Red

3.9.9.10.9 SIMD4x2 Address Payload

The payload of a SIMD4x2 message provides address parameters to process 2 slots.

DWord	Bit	Description
M1.7	31:0	Programming Notes: This register can only be delivered for the *Typed* message types.
M1.6	31:0	Programming Notes: This register can only be delivered for the *Typed* message types.
M1.5	31:0	Slot 1 V Address Format = U32
M1.4	31:0	Slot 1 U Address Format = U32
M1.3	31:0	
M1.2	31:0	
M1.1	31:0	Slot 0 V Address
M1.0	31:0	Slot 0 U Address

3.9.9.10.10 SIMD4x2 Source Payload (Atomic Operation message only)

The source payload follows the address payload for atomic operation messages. Depending on the atomic operation, zero, one, or two sources are required. If the source is not required, it must not be included. Message registers given here could be a lower number if some of the address parameters are not included.

The following atomic operations require no sources, thus the source payload is not delivered: AOP_INC, AOP_DEC, AOP_PREDEC

The following atomic operations require both Source0 and Source1: AOP_CMPWR

All of the remaining atomic operations require Source0 only.

DWord	Bit	Description
M2.7	31:0	Ignored
M2.6	31:0	Ignored
M2.5	31:0	Slot 1 Source1

DWord	Bit	Description
		Specifies Source1 for slot 1. Format = S31 for AOP_IMAX and AOP_IMIN, U32 for all other operations
M2.4	31:0	Slot 1 Source0
M2.3	31:0	Ignored
M2.2	31:0	Ignored
M2.1	31:0	Slot 0 Source1
M2.0	31:0	Slot 0 Source0

3.9.9.10.11 SIMD4x2 Source Payload ((AOP_CMPWR8B only)

DWord	Bit	Description
M2.7	31:0	Slot 1 Source1 [63:32]
M2.6	31:0	Slot 1 Source1 [31:0]
M2.5	31:0	Slot 1 Source0 [63:32]
M2.4	31:0	Slot 1 Source0 [31:0]
M2.3	31:0	Slot 0 Source1 [63:32]
M2.2	31:0	Slot 0 Source1 [31:0]
M2.1	31:0	Slot 0 Source0 [63:32]
M2.0	31:0	Slot 0 Source0 [31:0]

3.9.9.10.12 SIMD4x2 Write Data Payload (Write message only)

The write data payload follows the address payload for write messages.

DWord	Bit	Description
M2.7	31:0	Slot 1 Alpha Specifies the value of the red channel to be written for slot 7. For *Untyped* messages: Format = 32 bits raw data. For *Typed* messages: Format = IEEE Float, S31, or U32 depending on the **Surface Format** of the surface being accessed. SINT formats use S31, UINT formats use U32, and all other formats use Float.
M2.6	31:0	Slot 1 Blue
M2.5	31:0	Slot 1 Green
M2.4	31:0	Slot 1 Red
M2.3	31:0	Slot 0 Alpha
M2.2	31:0	Slot 0 Blue
M2.1	31:0	Slot 0 Green
M2.0	31:0	Slot 0 Red

3.9.9.11 Writeback Message

3.9.9.11.1 SIMD16 Read

A SIMD16 writeback message consists of up to 8 destination registers. Which registers are returned is determined by the channel mask in the message descriptor. Each asserted channel mask results in the

destination register of the corresponding channel being skipped in the writeback message, and all channels with higher numbered registers being dropped down to fill in the space occupied by the masked channel. For example, if only red and alpha are enabled, red is sent to regid+0 and regid+1, and alpha to regid+2 and regid+3. The slots written within each destination register is determined by the execution mask on the “send” instruction.

DWord	Bit	Description
W0.7	31:0	**Slot 7 Red:** Specifies the value of the red channel for slot 7. Format = 32 bits raw data.
W0.6	31:0	Slot 6 Red
W0.5	31:0	Slot 5 Red
W0.4	31:0	Slot 4 Red
W0.3	31:0	Slot 3 Red
W0.2	31:0	Slot 2 Red
W0.1	31:0	Slot 1 Red
W0.0	31:0	Slot 0 Red
W1.7	31:0	Slot 15 Red
W1.6	31:0	Slot 14 Red
W1.5	31:0	Slot 13 Red
W1.4	31:0	Slot 12 Red
W1.3	31:0	Slot 11 Red
W1.2	31:0	Slot 10 Red
W1.1	31:0	Slot 9 Red
W1.0	31:0	Slot 8 Red
W2		Slots 7:0 Green
W3		Slots 15:8 Green
W4		Slots 7:0 Blue
W5		Slots 15:8 Blue
W6		Slots 7:0 Alpha
W7		Slots 15:8 Alpha

3.9.9.11.2 SIMD8 Read

A SIMD8 writeback message consists of up to 4 destination registers. Which registers are returned is determined by the channel mask in the message descriptor. Each asserted channel mask results in the destination register of the corresponding channel being skipped in the writeback message, and all channels with higher numbered registers being dropped down to fill in the space occupied by the masked channel. For example, if only red and alpha are enabled, red is sent to regid+0, and alpha to regid+1. The slots written within each destination register is determined by the execution mask on the “send” instruction.

DWord	Bit	Description
W0.7	31:0	**Slot 7 Red:** Specifies the value of the red channel for slot 7. For *Untyped* messages: Format = 32 bits raw data. For *Typed* messages: Format = IEEE Float, S31, or U32 depending on the **Surface Format** of the surface being accessed. SINT formats use S31, UINT formats use U32, and all other formats use Float.

DWord	Bit	Description
W0.6	31:0	Slot 6 Red
W0.5	31:0	Slot 5 Red
W0.4	31:0	Slot 4 Red
W0.3	31:0	Slot 3 Red
W0.2	31:0	Slot 2 Red
W0.1	31:0	Slot 1 Red
W0.0	31:0	Slot 0 Red
W1		Slots 7:0 Green
W2		Slots 7:0 Blue
W3		Slots 7:0 Alpha

3.9.9.11.3 SIMD8 Read (Tile W)

The slots written within each destination register is determined by the execution mask on the “send” instruction.

DWord	Bit	Description
M5.7	31:8	Reserved (0)
	7:0	Slot 7 Red Specifies the value of the red channel to be written for slot 7. For *Typed* messages: Format = U8
M5.6	31:8	Reserved (0)
	7:0	Slot 6 Red
M5.5	31:8	Reserved (0)
	7:0	Slot 5 Red
M5.4	31:8	Reserved (0)
	7:0	Slot 4 Red
M5.3	31:8	Reserved (0)
	7:0	Slot 3 Red
M5.2	31:8	Reserved (0)
	7:0	Slot 2 Red
M5.1	31:8	Reserved (0)
	7:0	Slot 1 Red
M5.0	31:8	Reserved (0)
	7:0	Slot 0 Red

3.9.9.11.4 SIMD4x2 Read

A SIMD4x2 writeback message always consists of a single message register containing all four color channels of each of the two slots. The channel mask bits as well as the execution mask on the “send” instruction are used to determine which of the channels in the destination register are overwritten. If any of the four execution mask bits for a slot is asserted, that slot is considered to be active. The active channels in the channel mask will be written in the destination register for that slot. If the slot is inactive (all four execution mask bits deasserted), none of the channels for that slot will be written in the destination register.

DWord	Bit	Description
W0.7	31:0	**Slot 1 Alpha:** Specifies the value of the pixel’s alpha channel. Format = 32 bits raw data.
W0.6	31:0	Slot 1 Blue
W0.5	31:0	Slot 1 Green
W0.4	31:0	Slot 1 Red
W0.3	31:0	Slot 0 Alpha
W0.2	31:0	Slot 0 Blue
W0.1	31:0	Slot 0 Green
W0.0	31:0	Slot 0 Red

3.9.9.11.5 SIMD16 Atomic Operation

A writeback message is only returned for an Atomic Operation message if the **Send Return Data** field in the message descriptor is enabled. The execution mask on the “send” instruction indicates which channels in the destination registers are overwritten.

DWord	Bit	Description
W0.7	31:0	**Slot 7 Return Data:** Specifies the value of the return data for slot 7. Format = U32
W0.6	31:0	Slot 6 Return Data
W0.5	31:0	Slot 5 Return Data
W0.4	31:0	Slot 4 Return Data
W0.3	31:0	Slot 3 Return Data
W0.2	31:0	Slot 2 Return Data
W0.1	31:0	Slot 1 Return Data
W0.0	31:0	Slot 0 Return Data
W1.7	31:0	Slot 15 Return Data
W1.6	31:0	Slot 14 Return Data
W1.5	31:0	Slot 13 Return Data
W1.4	31:0	Slot 12 Return Data
W1.3	31:0	Slot 11 Return Data
W1.2	31:0	Slot 10 Return Data
W1.1	31:0	Slot 9 Return Data
W1.0	31:0	Slot 8 Return Data

3.9.9.11.6 SIMD16 Atomic Operation (AOP_CMPWR8B only)

A writeback message is only returned for an Atomic Operation AOP_CMPWR8B message if the **Send Return Data** field in the message descriptor is enabled. The execution mask on the “send” instruction indicates which channels in the destination registers are overwritten.

DWord	Bit	Description
W0.7	31:0	**Slot 7 Return Data[31:0]:** Specifies the value of the return data for slot 7. Format = U32
W0.6	31:0	Slot 6 Return Data[31:0]
W0.5	31:0	Slot 5 Return Data[31:0]
W0.4	31:0	Slot 4 Return Data[31:0]
W0.3	31:0	Slot 3 Return Data[31:0]
W0.2	31:0	Slot 2 Return Data[31:0]
W0.1	31:0	Slot 1 Return Data[31:0]
W0.0	31:0	Slot 0 Return Data[31:0]
W1.7	31:0	Slot 15 Return Data[31:0]
W1.6	31:0	Slot 14 Return Data[31:0]
W1.5	31:0	Slot 13 Return Data[31:0]
W1.4	31:0	Slot 12 Return Data[31:0]
W1.3	31:0	Slot 11 Return Data[31:0]
W1.2	31:0	Slot 10 Return Data[31:0]
W1.1	31:0	Slot 9 Return Data[31:0]
W1.0	31:0	Slot 8 Return Data[31:0]
W2		Slot 7:0 Return Data[63:32]
W3		Slot 15:8 Return Data[63:32]

3.9.9.11.7 SIMD8 Atomic Operation

A writeback message is only returned for an Atomic Operation message if the **Send Return Data** field in the message descriptor is enabled. The execution mask on the “send” instruction indicates which channels in the destination registers are overwritten.

DWord	Bit	Description
W0.7	31:0	**Slot 7 Return Data:** Specifies the value of the return data for slot 7. Format = U32
W0.6	31:0	Slot 6 Return Data
W0.5	31:0	Slot 5 Return Data
W0.4	31:0	Slot 4 Return Data
W0.3	31:0	Slot 3 Return Data
W0.2	31:0	Slot 2 Return Data
W0.1	31:0	Slot 1 Return Data
W0.0	31:0	Slot 0 Return Data

3.9.9.11.8 SIMD8 Atomic Operation (AOP_CMPWR8B only)

A writeback message is only returned for an Atomic Operation AOP_CMPWR8B message if the **Send Return Data** field in the message descriptor is enabled. The execution mask on the “send” instruction indicates which channels in the destination registers are overwritten.

DWord	Bit	Description
W0.7	31:0	**Slot 7 Return Data[31:0]:** Specifies the value of the return data for slot 7. Format = U32
W0.6	31:0	Slot 6 Return Data[31:0]
W0.5	31:0	Slot 5 Return Data[31:0]
W0.4	31:0	Slot 4 Return Data[31:0]
W0.3	31:0	Slot 3 Return Data[31:0]
W0.2	31:0	Slot 2 Return Data[31:0]
W0.1	31:0	Slot 1 Return Data[31:0]
W0.0	31:0	Slot 0 Return Data[31:0]
W1.7	31:0	Slot 7 Return Data[63:32]
W1.6	31:0	Slot 6 Return Data[63:32]
W1.5	31:0	Slot 5 Return Data[63:32]
W1.4	31:0	Slot 4 Return Data[63:32]
W1.3	31:0	Slot 3 Return Data[63:32]
W1.2	31:0	Slot 2 Return Data[63:32]
W1.1	31:0	Slot 1 Return Data[63:32]
W1.0	31:0	Slot 0 Return Data[63:32]

3.9.9.11.9 SIMD4x2 Atomic Operation

A writeback message is only returned for an Atomic Operation message if the **Send Return Data** field in the message descriptor is enabled. The execution mask on the “send” instruction indicates which channels in the destination registers are overwritten.

DWord	Bit	Description
W0.7	31:0	reserved – not written to GRF
W0.6	31:0	reserved – not written to GRF
W0.5	31:0	reserved – not written to GRF
W0.4	31:0	**Slot 1 Return Data:** Specifies the value of the return data for slot 1. Format = U32
W0.3	31:0	reserved – not written to GRF
W0.2	31:0	reserved – not written to GRF
W0.1	31:0	reserved – not written to GRF
W0.0	31:0	Slot 0 Return Data

3.9.9.11.10 SIMD4x2 Atomic Operation (AOP_CMPWR8B only)

A writeback message is only returned for an Atomic Operation AOP_CMPWR8B message if the **Send Return Data** field in the message descriptor is enabled. The execution mask on the “send” instruction indicates which channels in the destination registers are overwritten.

DWord	Bit	Description
W0.7	31:0	reserved – not written to GRF
W0.6	31:0	reserved – not written to GRF

DWord	Bit	Description
W0.5	31:0	Slot 1 Return Data: [63:32]
W0.4	31:0	Slot 1 Return Data: [31:0]
W0.3	31:0	reserved – not written to GRF
W0.2	31:0	reserved – not written to GRF
W0.1	31:0	Slot 0 Return Data: [63:32]
W0.0	31:0	Slot 0 Return Data[31:0]

3.9.10 Scratch Block Read/Write

This message performs a read or write operation of between 1 and 4 simd-8 registers to a Hword aligned offset to scratch memory. The Hword offset into the scratch memory is provided in the message descriptor, allowing a single instruction read|write block operation in a single source instruction. 12b are provided for the Hword offset, allowing addressing of 4K Hword locations (128KB).

Two modes of channel-enable interpretation are provided: Dword, which support a simd-8 or simd-16 dword channel-serial view of a register, and Oword, which supports a simd-4x2 view of a register. For operations under conditions of simd-32 processing, two messages should be used, with one of them indicating ‘H2’ to select the upper 16b of execution mask.

This message type can only be used with stateless model memory access. Thus binding table entry 0xFF is hard-coded into the execution of this message.

Applications:

scratch space reads/writes for register spill/fill operations.

Execution Mask. The low 8 bits of the execution mask are used to enable the 8 channels in the first and third GRF registers returned (W0, W2) for read, or the first and third write registers sent (M1, M3). The high 8 bits are used similarly for the second and fourth (W1, W3 or M2, M4).

For Dword mode, the execution mask delivered with the message dictates dword-based control of read or write operations. For Oword mode, any one or more asserted bits within the Oword’s corresponding execution mask nibble causes read or write operations to occur across all four dwords of the Oword regardless of the setting of any particular dword’s bit.

Out-of-Bounds Accesses. Reads to areas outside of the surface return 0. Writes to areas outside of the surface are dropped and will not modify memory contents.

3.9.10.1 Message Descriptor

Bits	Description
17	Operation Type: 0 = Read, 1 = write
16	Channel Mode: **0: Oword** – Channel enables in effect at the time of ‘send’ are interpreted such if one or more are enabled, the read or write operation occurs on all four dwords. **1: Dword** – Channel enables in effect at the time of the ‘send’ are used as dword enables, causing the read or write operation to occur only on the dwords whose corresponding channel enable is set..
15	**Invalidate after read** – Indicates the cache line should invalidated after the read. 1: Invalidate cache line

Bits	Description
	0: no Invalidate
14	**Reserved - MBZ**
13:12	**Block Size** – indicates the number of simd-8 registers to be read|written. 11: 4 registers 10: <reserved> 01: 2 registers 00: 1 register
11:0	**Offset** – A 12b Hword offset into the memory Immediate Memory buffer as specified by binding table 0xFF.

3.9.10.2 Message Header

DWord	Bit	Description
M0.7	31:16	Ignored
	15:0	Ignored
M0.6	31:0	Ignored
M0.5	31:0	**Immediate Buffer Base Address.** Specifies the surface base address for messages in which the Binding Table Index is 255 (stateless model), otherwise this field is ignored. This pointer is relative to the **General State Base Address**. Format = GeneralStateOffset[31:10]
M0.4	31:0	Ignored
M0.3	31:0	Ignored
M0.2	31:0	Ignored
M0.1	31:0	Ignored
M0.0	31:0	Ignored

3.9.10.3 Message Payload (Write)

The listing below illustrates the write payload for a message of block size = 4;

DWord	Bit	Description
M1.7:0	255:0	HWord[Offset]
M2.7:0	255:0	HWord[Offset+1]
M3.7:0	255:0	HWord[Offset+2]
M3.7:0	255:0	HWord[Offset+3]

3.9.10.4 Message Payload (Read)

Only required a message header.

3.9.10.5 Writeback Message (Read)

The table below illustrates an example where 4 Hwords are read through a scratch block read.

DWord	Bit	Description
W0.7:0	255:0	HWord[Offset]
W1.7:0	255:0	HWord[Offset+1]
W2.7:0	255:0	HWord[Offset+2]
W3.7:0	255:0	HWord[Offset+3]

3.9.11 Render Target Write

This message takes four subspans of pixels for write to a render target. Depending on parameters contained in the message and state, it may also perform a depth and stencil buffer write and/or a render target read for a color blend operation. Additional operations enabled in the Color Calculator state will also be initiated as a result of issuing this message (depth test, alpha test, logic ops, etc.). This message is intended only for use by pixel shader kernels for writing results to render targets.

Restrictions:

All surface types, except SURFTYPE_STRBUF, are allowed.

For SURFTYPE_BUFFER and SURFTYPE_1D surfaces, only the X coordinate is used to index into the surface. The Y coordinate must be zero.

For SURFTYPE_1D, 2D, 3D, and CUBE surfaces, a **Render Target Array Index** is included in the input message to provide an additional coordinate. The **Render Target Array Index** must be zero for SURFTYPE_BUFFER.

The surface format is restricted to the set supported as render target. If source/dest color blend is enabled, the surface format is further restricted to the set supported as alpha blend render target.

The last message sent to the render target by a thread must have the **End Of Thread** bit set in the message descriptor and the dispatch mask set correctly in the message header to enable correct clearing of the pixel scoreboard.

The stateless model cannot be used with this message (**Binding Table Index** cannot be 255).

This message can only be issued from a kernel specified in WM_STATE or 3DSTATE_WM (pixel shader kernel), dispatched in non-contiguous mode. Any other kernel issuing this message will cause undefined behavior.

The dual source message cannot be used if the Render Target Rotation field in SURFACE_STATE is set to anything other than RTROTATE_0DEG.

This message cannot be used on a surface in field mode (**Vertical Line Stride** = 1)

If multiple SIMD8 Dual Source messages are delivered by the pixel shader thread, each SIMD8_DUALSRC_LO message must be issued *before* the SIMD8_DUALSRC_HI message with the same **Slot Group Select** setting.

SIMD8 Image Write: Out of bounds write to SURFTYPE_BUFFER with more than 8K elements is undefined.

Execution Mask. The execution mask for render target messages is ignored. Control of which pixels are active is controlled by the **Pixel/Sample Enables** fields in the message header.

Out-of-Bounds Accesses. Accesses to pixels outside of the surface are dropped and will not modify memory contents. However, if the **Render Target Array Index** is out of bounds, it is set to zero and the surface write is not surpressed.

The following table indicates the surface formats that are supported by this message.

Surface Format Name
R32G32B32A32_FLOAT
R32G32B32A32_SINT
R32G32B32A32_UINT
R16G16B16A16_UNORM
R16G16B16A16_SNORM
R16G16B16A16_SINT
R16G16B16A16_UINT
R16G16B16A16_FLOAT
R32G32_FLOAT
R32G32_SINT
R32G32_UINT
B8G8R8A8_UNORM
B8G8R8A8_UNORM_SRGB
R10G10B10A2_UNORM
R10G10B10A2_UINT
R8G8B8A8_UNORM
R8G8B8A8_UNORM_SRGB
R8G8B8A8_SNORM
R8G8B8A8_SINT
R8G8B8A8_UINT
R16G16_UNORM
R16G16_SNORM
R16G16_SINT
R16G16_UINT
R16G16_FLOAT
B10G10R10A2_UNORM
B10G10R10A2_UNORM_SRGB
R11G11B10_FLOAT
R32_SINT
R32_UINT
R32_FLOAT
B5G6R5_UNORM
B5G6R5_UNORM_SRGB
B5G5R5A1_UNORM
B5G5R5A1_UNORM_SRGB
B4G4R4A4_UNORM
B4G4R4A4_UNORM_SRGB
R8G8_UNORM
R8G8_SNORM
R8G8_SINT
R8G8_UINT
R16_UNORM
R16_SNORM
R16_SINT
R16_UINT
R16_FLOAT
B5G5R5X1_UNORM
B5G5R5X1_UNORM_SRGB
R8_UNORM
R8_SNORM

Surface Format Name
R8_SINT
R8_UINT
A8_UNORM
YCRCB_NORMAL
YCRCB_SWAPUVY
YCRCB_SWAPUV
YCRCB_SWAPY

3.9.11.1 Subspan/Pixel to Slot Mapping

The following table indicates the mapping of subspans, pixels, and samples to slots in the pixel shader dispatch depending on the number of samples and message size. This table applies to all devices, however NumSamples = 4X is supported only on, and NumSamples = 8X is supported only on.

Pixels are numbered as follows within a subspan:

0 = upper left

1 = upper right

2 = lower left

3 = lower right

sspi = Starting Sample Pair Index (from the message header)

Dispatch Size	Num Samples	Slot Mapping (SSPI = Starting Sample Pair Index)
SIMD32	1X	Slot[3:0] = Subspan[0].Pixel[3:0].Sample[0] Slot[7:4] = Subspan[1].Pixel[3:0].Sample[0] Slot[11:8] = Subspan[2].Pixel[3:0].Sample[0] Slot[15:12] = Subspan[3].Pixel[3:0].Sample[0] Slot[19:16] = Subspan[4].Pixel[3:0].Sample[0] Slot[23:20] = Subspan[5].Pixel[3:0].Sample[0] Slot[27:24] = Subspan[6].Pixel[3:0].Sample[0] Slot[31:28] = Subspan[7].Pixel[3:0].Sample[0]
	2X	Slot[3:0] = Subspan[0].Pixel[3:0].Sample[0] Slot[7:4] = Subspan[0].Pixel[3:0].Sample[1] Slot[11:8] = Subspan[1].Pixel[3:0].Sample[0] Slot[15:12] = Subspan[1].Pixel[3:0].Sample[1] Slot[19:16] = Subspan[2].Pixel[3:0].Sample[0] Slot[23:20] = Subspan[2].Pixel[3:0].Sample[1] Slot[27:24] = Subspan[3].Pixel[3:0].Sample[0]

Dispatch Size	Num Samples	Slot Mapping (SSPI = Starting Sample Pair Index)
		Slot[31:28] = Subspan[3].Pixel[3:0].Sample[1]
	4X	Slot[3:0] = Subspan[0].Pixel[3:0].Sample[0] Slot[7:4] = Subspan[0].Pixel[3:0].Sample[1] Slot[11:8] = Subspan[0].Pixel[3:0].Sample[2] Slot[15:12] = Subspan[0].Pixel[3:0].Sample[3] Slot[19:16] = Subspan[1].Pixel[3:0].Sample[0] Slot[23:20] = Subspan[1].Pixel[3:0].Sample[1] Slot[27:24] = Subspan[1].Pixel[3:0].Sample[2] Slot[31:28] = Subspan[1].Pixel[3:0].Sample[3]
SIMD16	8X	Slot[3:0] = Subspan[0].Pixel[3:0].Sample[0] Slot[7:4] = Subspan[0].Pixel[3:0].Sample[1] Slot[11:8] = Subspan[0].Pixel[3:0].Sample[2] Slot[15:12] = Subspan[0].Pixel[3:0].Sample[3] Slot[19:16] = Subspan[0].Pixel[3:0].Sample[4] Slot[23:20] = Subspan[0].Pixel[3:0].Sample[5] Slot[27:24] = Subspan[0].Pixel[3:0].Sample[6] Slot[31:28] = Subspan[0].Pixel[3:0].Sample[7]
	1X	Slot[3:0] = Subspan[0].Pixel[3:0].Sample[0] Slot[7:4] = Subspan[1].Pixel[3:0].Sample[0] Slot[11:8] = Subspan[2].Pixel[3:0].Sample[0] Slot[15:12] = Subspan[3].Pixel[3:0].Sample[0]
	2X	Slot[3:0] = Subspan[0].Pixel[3:0].Sample[0] Slot[7:4] = Subspan[0].Pixel[3:0].Sample[1] Slot[11:8] = Subspan[1].Pixel[3:0].Sample[0] Slot[15:12] = Subspan[1].Pixel[3:0].Sample[1]

Restriction:

When SIMD32 or SIMD16 PS threads send render target writes with multiple SIMD8 and SIMD16 messages, the following must hold:

All the slots (as described above) must have a corresponding render target write irrespective of the slot's validity. A slot is considered valid when at least one sample is enabled. For example, a SIMD16 PS thread must send two SIMD8 render target writes to cover all the slots.

PS thread must send SIMD render target write messages with increasing slot numbers. For example, SIMD16 thread has Slot[15:0] and if two SIMD8 render target writes are used, the first SIMD8 render target write must send Slot[7:0] and the next one must send Slot[15:8].

3.9.11.2 Message Descriptor

Message Descriptor - Render Target Write

Default Value: 0x00000000

DWord	Bit	Description
0	31	**Reserved** Format: MBZ
	29:14	**Reserved** Format: MBZ
	13	**Reserved** Format: MBZ
	12	**Last Render Target Select** This bit must be set on the last render target write message sent for each group of pixels. For single render target pixel shaders, this bit is set on all render target write messages. For multiple render target pixel shaders, this bit is set only on messages sent to the last render target. This bit must be zero for SIMD8 Image Write message. **Programming Notes** In general, when threads are not launched by 3D FF, this bit must be zero.
	11	**Slot Group Select** This field selects whether slots 15:0 or slots 31:16 are used for bypassed data. Bypassed data includes the antialias alpha, multisample coverage mask, and if the header is not present also includes the X/Y addresses and pixel enables. For 8- and 16-pixel dispatches, SLOTGRP_LO must be selected on every message. For 32-pixel dispatches, this field must be set correctly for each message based on which slots are currently being processed. **Value / Name / Description** 0 / SLOTGRP_LO / choose bypassed data for slots 15:0 1 / SLOTGRP_HI / choose bypassed data for slots 31:16 **Programming Notes** For SIMD8 Image Write message thsi field MBZ.
	10:8	**Message Type** This field specifies the type of render target message. For the SIMD8_DUALSRC_xx messages, the low bit indicates which slots to use for the pixel enables, X/Y addresses, and oMask. **Value / Name / Description** 000b / SIMD16 / SIMD16 single source message 001b / SIMD16_REPDATA / SIMD16 single source message with replicated data 010b / SIMD8_DUALSRC_LO / SIMD8 dual source message, use slots 7:0 011b / SIMD8_DUALSRC_HI / SIMD8 dual source message, use slots 15:8 100b / SIMD8_LO / SIMD8 single source message, use slots 7:0 111b / SIMD16_REPDATA / It's only supported when accessing *Tiled Memory*. Using this Message Type to access linear *(Untiled)* memory is UNDEFINED.

Message Descriptor - Render Target Write		
		Programming Notes / Project
		the above slots indicated are within the 16 slots selected by **Slot Group Select**. If SLOTGRP_HI is selected, the SIMD8 message types above reference slots 23:16 or 31:24 instead of 7:0 or 15:8, respectively.
		SIMD16 messages are not supported for 8X MSAA when PS outputs depth.
		SIMD16_REPDATA message must not be used in SIMD8 pixel-shaders.
	7:0	**Reserved**
		Format: MBZ

3.9.11.3 Message Header

The render target write message has a two-register message header.

If the header is not present, behavior is as if the message was sent with most fields set to the same value that was delivered in R0 and R1 on the pixel shader thread dispatch. The following fields, which are not delivered in the pixel shader dispatch, behave as if they are set to zero:

Render Target Index

Source0 Alpha Present to Render Target

DWord	Bits	Description
M0.7	31:0	
M0.6	31:0	
M0.5	31:10	Ignored
	9:8	**Color Code:** This ID is assigned by the Windower unit and is used to track synchronizing events. Format: Reserved for HW Implementation Use.
	7:0	**FFTID.** The Fixed Function Thread ID is assigned by the fixed function unit and is a unique identifier for the thread. It is used to free up resources used by the thread upon thread completion.
M0.4	31:0	Ignored (reserved for hardware delivery of binding table pointer)
M0.3	31:0	Ignored
M0.2	31:3	Ignored
	2:0	**Render Target Index.** Specifies the render target index that will be used to select blend state from BLEND_STATE. Format = U3
M0.1	31:6	**ColorCalculatorState Pointer.** Specifies the 64-byte aligned pointer to the color calculator state. This pointer is relative to the **General State Base Address**. Format = GeneralStateOffset[31:6]
	5:0	Ignored
M0.0	31	Ignored

<table>
<tr><th>DWord</th><th>Bits</th><th>Description</th></tr>
<tr><td></td><td>30:27</td><td>Viewport Index. Specifies the index of the viewport currently being used.
Format = U4
Range = [0,15]</td></tr>
<tr><td></td><td>26:16</td><td>Render Target Array Index. Specifies the array index to be used for the following surface types:
SURFTYPE_1D: specifies the array index. Range = [0,511]
SURFTYPE_2D: specifies the array index. Range = [0,511]
SURFTYPE_3D: specifies the “z” or “r” coordinate. Range = [0,2047]
SURFTYPE_CUBE: specifies the face identifier. Range = [0,5]
SURFTYPE_BUFFER: must be zero.
<table><tr><th>face</th><th>Render Target Array Index</th></tr><tr><td>+x</td><td>0</td></tr><tr><td>-x</td><td>1</td></tr><tr><td>+y</td><td>2</td></tr><tr><td>-y</td><td>3</td></tr><tr><td>+z</td><td>4</td></tr><tr><td>-z</td><td>5</td></tr></table>Format = U11
The Render Target Array Index used by hardware for access to the Render Target is overridden with the Minimum Array Element defined in SURFACE_STATE if it is out of the range between Minimum Array Element and Depth. For cube surfaces, a depth value of 5 is used for this determination.</td></tr>
<tr><td></td><td>15</td><td>Front/Back Facing Polygon. Determines whether the polygon is front or back facing. Used by the render cache to determine which stencil test state to use.
0: Front Facing
1: Back Facing</td></tr>
<tr><td></td><td>14</td><td>Ignored</td></tr>
<tr><td></td><td>14</td><td></td></tr>
<tr><td></td><td>13</td><td>Source Depth Present to Render Target. Indicates that source depth is included in the message.</td></tr>
<tr><td></td><td>12</td><td>oMask to Render Target
This bit indicates that oMask data is present in the message and is to be used to mask off samples.</td></tr>
<tr><td></td><td>11</td><td>Source0 Alpha Present to RenderTarget. This bit indicates that Source0 Alpha (aka o0.a) data is included in RTWrite message. If present, these alpha values are used as inputs to AlphaTest and AlphaToCoverage functions. This is required to meet the API</td></tr>
</table>

DWord	Bits	Description
		rules when writing to multiple render targets (MRTs). Programming Notes: This bit should not be set when writing to RT0, though sending and using redundant alpha will provide the correct results (at lower performance). This bit is not supported on Dual-Source Blend message types, as source0 alpha is already included in those messages. This bit is not supported on replicated data message types.
	10:9	Ignored
	8:6	**Starting Sample Pair Index:** indicates the index of the first sample pair of the dispatch Format = U3 Range = [0,3]
	5:0	Ignored
M1.7	31:16	**Dispatched Pixel/Sample Enables.** One bit per pixel (or sample within pixel) indicating which pixels/samples were originally enabled when the thread was dispatched. This field is only required for the end-of-thread message and on all dual-source messages. The **Dispatched Pixel/Sample Enables***must be unmodified* from the ones sent when the pixel shader thread was initiated. If the **Dispatched Pixel/Sample Enables** are modified, behavior is undefined. Multisample Note: When operating in PERSAMPLE mode these bits correspond to samples, not pixels. Each subspan slot (4 bits) corresponds to a specific sample location for the subspan. Note that in NUMSAMPLES_1 mode, a pixel and sample are synonomous. When operating in PERPIXEL mode, this field is ignored, and instead the SampleEnableMask (obtained via bypass) are used to clear the Depth Scoreboard.
	15:0	**Pixel/Sample Enables.** One bit per pixel/sample indicating which pixels/samples are still lit based on kill instruction activity in the pixel shader. This mask is used to control actual writes to the color buffer. Multisample Note: When operating in PERSAMPLE mode these bits correspond to samples, not pixels, as the PS is run per-sample. Each subspan slot (4 bits) corresponds to a specific sample location for the subspan. When operating in PERPIXEL mode, these bits still correspond to pixels, as the PS is run per-pixel. Each pixel's mask bit is replicated according to **Number of Multisamples** and combined with other masks to control writes to the multisample locations.
M1.6	31:0	Ignored
M1.5	31:16	**Y3.** Y coordinate for upper-left pixel of subspan 3 (slot 12) Format = U16
	15:0	**X3.** X coordinate for upper-left pixel of subspan 3 (slot 12)

DWord	Bits	Description
		Format = U16
M1.4	31:16	Y2
	15:0	X2
M1.3	31:16	Y1
	15:0	X1
M1.2	31:16	Y0
	15:0	X0
M1.1	31:0	Ignored
M1.0	31:0	Ignored

3.9.11.4 Source 0 Alpha Payload

The source 0 alpha registers, if included, appear in M2 and M3, immediately following the header (if present).

For the SIMD8 single source message, only slot 7:0 data is sent (M2). The source 0 alpha phases are not supported for dual source messages.

DWord	Bit	Description
M2.7	31:0	Source 0 Alpha for Slot 7 Format = IEEE_Float This and the next register is only included if **Source 0 Alpha Present** bit is set.
M2.6	31:0	Source 0 Alpha for Slot 6
M2.5	31:0	Source 0 Alpha for Slot 5
M2.4	31:0	Source 0 Alpha for Slot 4
M2.3	31:0	Source 0 Alpha for Slot 3
M2.2	31:0	Source 0 Alpha for Slot 2
M2.1	31:0	Source 0 Alpha for Slot 1
M2.0	31:0	Source 0 Alpha for Slot 0
M3.7	31:0	Source 0 Alpha for Slot 15
M3.6	31:0	Source 0 Alpha for Slot 14
M3.5	31:0	Source 0 Alpha for Slot 13
M3.4	31:0	Source 0 Alpha for Slot 12
M3.3	31:0	Source 0 Alpha for Slot 11
M3.2	31:0	Source 0 Alpha for Slot 10
M3.1	31:0	Source 0 Alpha for Slot 9
M3.0	31:0	Source 0 Alpha for Slot 8

3.9.11.5 oMask Payload ()

The oMask payload, if present, follows source 0 alpha. The value of ‘p’ depends on whether the header and source 0 alpha are present.

Sample “n” for that pixel will be killed (not written to the render target or depth buffer) if bit “n” of the oMask is zero. Bits numbers where “n” is larger than the number of multisamples are ignored.

For the SIMD8 messages, only slots 7:0 data is used, or only slots 15:8 depending on the **Message Type** encoding.

DWord	Bit	Description

DWord	Bit	Description
Mp.7	31:16	oMask for Slot 15 Format = 16-bit mask This register is only included if **oMask Present** bit is set.
	15:0	oMask for Slot 14
Mp.6	31:16	oMask for Slot 13
	15:0	oMask for Slot 12
Mp.5	31:16	oMask for Slot 11
	15:0	oMask for Slot 10
Mp.4	31:16	oMask for Slot 9
	15:0	oMask for Slot 8
Mp.3	31:16	oMask for Slot 7
	15:0	oMask for Slot 6
Mp.2	31:16	oMask for Slot 5
	15:0	oMask for Slot 4
Mp.1	31:16	oMask for Slot 3
	15:0	oMask for Slot 2
Mp.0	31:16	oMask for Slot 1
	15:0	oMask for Slot 0

3.9.11.6 Color Payload: SIMD16 Single Source

3.9.11.6.1 Color Payload

This payload is included if the Message Type is SIMD16 single source. The value of 'm' depends on whether the header, source 0 alpha, and oMask are present.

DWord	Bit	Description
Mm.7	31:0	**Slot 7 Red.** Specifies the value of the slot's red component. Format = IEEE Float, S31, or U32 depending on the **Surface Format** of the surface being accessed. SINT formats use S31, UINT formats use U32, and all other formats use Float.
Mm.6	31:0	Slot 6 Red
Mm.5	31:0	Slot 5 Red
Mm.4	31:0	Slot 4 Red
Mm.3	31:0	Slot 3 Red
Mm.2	31:0	Slot 2 Red
Mm.1	31:0	Slot 1 Red
Mm.0	31:0	Slot 0 Red
M(m+1).7	31:0	Slot 15 Red
M(m+1).6	31:0	Slot 14 Red
M(m+1).5	31:0	Slot 13 Red
M(m+1).4	31:0	Slot 12 Red
M(m+1).3	31:0	Slot 11 Red
M(m+1).2	31:0	Slot 10 Red
M(m+1).1	31:0	Slot 9 Red
M(m+1).0	31:0	Slot 8 Red
M(m+2)		**Slot[7:0] Green.** See Mm definition for slot locations

DWord	Bit	Description
M(m+3)		**Slot[15:8] Green.** See M(m+1) definition for slot locations
M(m+4)		**Slot[7:0] Blue.** See Mm definition for slot locations
M(m+5)		**Slot[15:8] Blue.** See M(m+1) definition for slot locations
M(m+6)		**Slot[7:0] Alpha.** See Mm definition for slot locations
M(m+7)		**Slot[15:8] Alpha.** See M(m+1) definition for slot locations

3.9.11.7 Color Payload: SIMD8 Single Source

This payload is included if the Message Type is SIMD8 single source or SIMD8 Image Write. For , the value of 'm' depends on whether the header, source 0 alpha, and oMask are present.

DWord	Bit	Description
Mm.7	31:0	**Slot 7 Red.** Specifies the value of the slot's red component. Format = IEEE Float, S31, or U32 depending on the **Surface Format** of the surface being accessed. SINT formats use S31, UINT formats use U32, and all other formats use Float.
Mm.6	31:0	Slot 6 Red
Mm.5	31:0	Slot 5 Red
Mm.4	31:0	Slot 4 Red
Mm.3	31:0	Slot 3 Red
Mm.2	31:0	Slot 2 Red
Mm.1	31:0	Slot 1 Red
Mm.0	31:0	Slot 0 Red
M(m+1)		**Slot[7:0] Green.** See Mm definition for slot locations
M(m+2)		**Slot[7:0] Blue.** See Mm definition for slot locations
M(m+3)		**Slot[7:0] Alpha.** See Mm definition for slot locations

3.9.11.8 Color Payload: SIMD16 Replicated Data

This payload is included if the Message Type specifies single source message with replicated data. One set of R/G/B/A data is included in the message, and this data is replicated to all 16 pixels.

This message is legal with color data only (for , oMask is also legal with this message). The registers for depth, stencil, and antialias alpha data cannot be included with this message, and the corresponding bits in the message header must indicate that these registers are not present.

Programming Notes:

This message is allowed only on tiled surfaces

DWord	Bit	Description
Mm.7:4	31:0	Reserved

DWord	Bit	Description
Mm.3	31:0	**Alpha.** Specifies the value of all slots' alpha channel. Format = IEEE Float, S31, or U32 depending on the **Surface Format** of the surface being accessed. SINT formats use S31, UINT formats use U32, and all other formats use Float.
Mm.2	31:0	Blue
Mm.1	31:0	Green
Mm.0	31:0	Red

3.9.11.9 Message Sequencing Summary

This section summarizes the sequencing that occurs for each legal render target write message. All messages have the M0 and M1 header registers if the header is present. If the header is not present, all registers below are renumbered starting with M0 where M2 appears. All cases not shown in this table are illegal.

Key:

s0, s1 = source 0, source 1

1/0 = slots 15:8

3/2 = slots 7:0

sZ = source depth

oM = oMask

Message Type	oMask Present	Source Depth Present	Source 0 Alpha Present	M2	M3	M4	M5	M6	M7	M8	M9	M10	M11	M12	M13	M14
000	0	0	0	1/0R	3/2R	1/0G	3/2G	1/0B	3/2B	1/0A	3/2A					
000	0	0	1	1/0s0A	3/2s0A	1/0R	3/2R	1/0G	3/2G	1/0B	3/2B	1/0A	3/2A			
000	0	1	0	1/0R	3/2R	1/0G	3/2G	1/0B	3/2B	1/0A	3/2A	1/0sZ	3/2sZ			
000	0	1	1	1/0s0A	3/2s0A	1/0R	3/2R	1/0G	3/2G	1/0B	3/2B	1/0A	3/2A	1/0sZ	3/2sZ	
000	1	0	0	oM	1/0R	3/2R	1/0G	3/2G	1/0B	3/2B	1/0A	3/2A				
000	1	0	1	1/0soA	3/2s0A	oM	1/0R	3/2R	1/0G	3/2G	1/0B	3/2B	1/0A	3/2A		
000	1	1	0	oM	1/0R	3/2R	1/0G	3/2G	1/0B	3/2B	1/0A	3/2A	1/0sZ	3/2sZ		
000	1	1	1	1/0s0A	3/2s0A	oM	1/0R	3/2R	1/0G	3/2G	1/0B	3/2B	1/0A	3/2A	1/0sZ	3/2sZ
001	0	0	0	RGBA												
001	1	0	0	oM	RGBA											
010	0	0	0	1/0s0R	1/0s0G	1/0s0B	1/0s0A	1/0s1R	1/0s1G	1/0s1B	1/0s1A					
010	0	1	0	1/0s0R	1/0s0G	1/0s0B	1/0s0A	1/0s1R	1/0s1G	1/0s1B	1/0s1A	1/0sZ				
010	1	0	0	oM	1/0s0R	1/0s0G	1/0s0B	1/0s0A	1/0s1R	1/0s1G	1/0s1B	1/0s1A				

Message Type	oMask Present	Source Depth Present	Source 0 Alpha Present	M2	M3	M4	M5	M6	M7	M8	M9	M10	M11	M12	M13	M14
010	1	1	0	oM	1/0s0 R	1/0s0 G	1/0s0 B	1/0s0 A	1/0s1 R	1/0s1 G	1/0s1 B	1/0s1 A	1/0s Z			
011	0	0	0	3/2s0 R	3/2s0 G	3/2s0 B	3/2s0 A	3/2s1 R	3/2s1 G	3/2s1 B	3/2s1 A					
011	0	1	0	3/2s0 R	3/2s0 G	3/2s0 B	3/2s0 A	3/2s1 R	3/2s1 G	3/2s1 B	3/2s1 A	3/2sZ				
011	1	0	0	oM	3/2s0 R	3/2s0 G	3/2s0 B	3/2s0 A	3/2s1 R	3/2s1 G	3/2s1 B	3/2s1 A				
011	1	1	0	oM	3/2s0 R	3/2s0 G	3/2s0 B	3/2s0 A	3/2s1 R	3/2s1 G	3/2s1 B	3/2s1 A	3/2s Z			
100	0	0	0	R	G	B	A									
100	0	0	1	s0A	R	G	B	A								
100	0	1	0	R	G	B	A	sZ								
100	0	1	1	s0A	R	G	B	A	sZ							
100	1	0	0	oM	R	G	B	A								
100	1	0	1	s0A	oM	R	G	B	A							
100	1	1	0	oM	R	G	B	A	sZ							
100	1	1	1	s0A	oM	R	G	B	A	sZ						

Revision History

Revision Number	Description	Revision Date
1.0	First 2012 OpenSource edition	May 2012

§§