Intel® OpenSource HD Graphics Programmer's Reference Manual (PRM)

Volume 2 Part 2: 3D/Media – Media (SandyBridge)

For the 2011 Intel Core Processor Family

May 2011

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.

Doc Ref #: IHD-OS-V2 Pt2 – 05 11

Contents

1. Media and General Purpose Pipeline

1.1 Introduction

This section covers the programming details for the media (general purpose) fixed function pipeline. The **media pipeline** is positioned in parallel with the 3D fixed function pipeline. It is so named as its initial (and primary) usage is to provide media functionalities and it does have media specific fixed function capability. However, the fixed functions are designed to have the general capability of controlling the shared functions and resources, feeding generic threads to the Execution Units to be executed, and interacting with such generic threads during run time. The media pipeline can be used for non-media applications, and therefore, can also be referred to as the **general purpose pipeline**. *For the rest of this chapter, we will refer this fixed function pipeline as the media pipeline, keeping in mind its general purpose capability.*

Concurrency of the media pipeline and the 3D pipeline is not supported. In other words, only one pipeline can be activated at a given time. Switching between the two pipelines within a single context is supported using the MI_PIPELINE_SELECT command.

The followings are some media application examples that can be mapped onto the media pipeline. All these applications are functional; however, what level of performance can be achieved depends on the hardware configuration and is beyond the scope of this document.

- MPEG-2 decode acceleration with HWMC
- MPEG-2 decode acceleration with IS/IDCT and forward
- MPEG-2 decode acceleration with VLD and forward
- WMV-9 decode acceleration with post filters
- WMV-9 decode acceleration with HWMC and post filters
- WMV-9 decode acceleration with IS/IDCT and forward
- AVC decode acceleration with HWMC and forward including Loop Filter
- VC1 decode acceleration with HWMC and forward including Loop Filter
- Advanced deinterlace filter (motion detected or motion compensated deinterlace filter)
- Video encode acceleration (with various level of hardware assistant)

1.1.1 Hardware Feature Map in Products

The following table lists the hardware features in the media pipe.

Table 1-1. Video Front End Features in Device Hardware

Features/ Device	[DevGT+]
Generic Mode	**Y**
Root Threads	**Y**
Parent/Child Threads	**Y**
SRT (Synchronized Root Threads)	**Y**
PRT (Persistent Root Thread)	**Y**
Interface Descriptor Remapping	N
Interface Descriptor Remapping	N
IS Mode (HW Inverse Scan)	N
VLD Mode (HW MPEG2 VLD)	N
AVC MC Mode	N
AVC IT Mode (HW AVC IT)	N
AVC ILDB Filter (in Data Port)	N
VC1 MC Mode	N
VC1 IT Mode (HW VC1 IT)	N
Stalling HW Scoreboard	**Y**
Non-stalling HW Scoreboard	**Y**
HW Walker	**Y**
HW Timer	**Y**
Pipelined State Flush	**Y**
HW Barrier	**Y**

1.2 Media Pipeline Overview

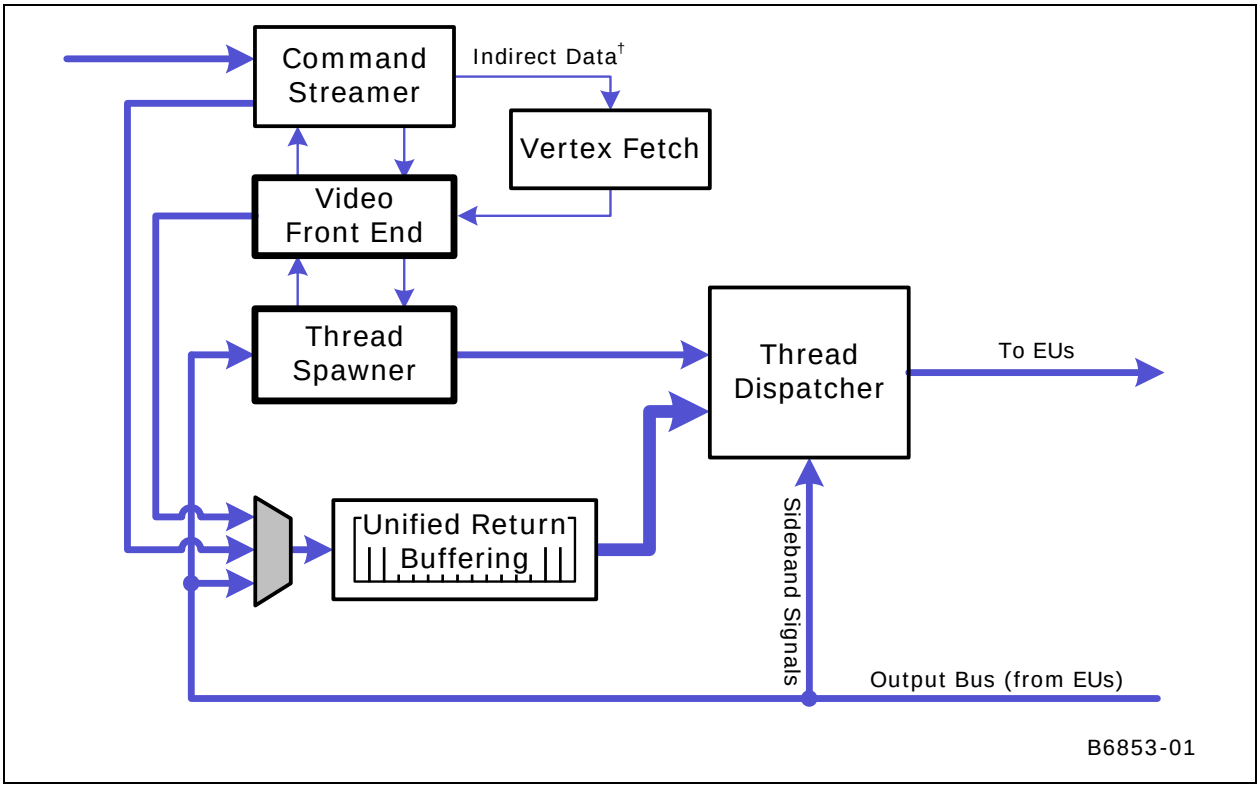
The media (general purpose) pipeline consists of two fixed function units: Video Front End (VFE) unit and Thread Spawner (TS) unit. VFE unit interfaces with the Command Streamer (CS), writes thread payload data into the Unified Return Buffer (URB) and prepares threads to be dispatched through TS unit. VFE unit also contains a hardware Variable Length Decode (VLD) engine for MPEG-2 video decode. TS unit is the only unit of the media pipeline that interfaces to the Thread Dispatcher (TD) unit for new thread generation. It is responsible of spawning root threads (short for the root-node parent threads) originated from VFE unit and spawning child threads (can be either a leaf-node child thread or a branch-node parent thread) originated from the Execution Units (EU) by a parent thread (can be a root-node or a branch-node parent thread).

The fixed functions, VFE and TS, in the media pipeline, in most cases, share the same basic building blocks as the fixed functions in the 3D pipeline. However, there are some unique features in media fixed functions as highlighted by the followings.

- VFE manages URB and only has write access to URB; TS does not interface to URB.

- When URB Constant Buffer is enabled, VFE forwards TS the URB Handler for the URB Constant Buffer received from CS.
- TS interfaces to TD; VFE does not.
- TS can have a message directed to it like other shared functions (and thus TS has a shared function ID), and it does not snoop the Output Bus as some other fixed functions in the 3D pipeline do.
- A root thread generated by the media pipeline can only have up to one URB return handle.
- If a root thread has a URB return handle, VFE creates the URB handle for the payload to initiating the root thread and also passes it alone to the root thread as the return handle. The root thread then uses the same URB handle for child thread generation.
- If URB Constant Buffer is enabled and an interface descriptor indicates that it is also used for the kernel, TS requests TD to load constant data directly to the thread's register space. For root thread, constant data are loaded after R0 and before the data from the other URB handle. For child thread, as the R0 header is provided by the parent thread, Thread Spawner splits the URB handles from the parent thread into two and inserts the constant data after the R0 header.
- A root thread must terminate with a message to TS. A child thread should also terminate with a message to TS.
- High streaming performance of indirect media object load is achieved by utilizing the large vertex cache available in the Vertex Fetch unit (of the 3D pipeline).

Figure 1-1. Top level block diagram of the Media Pipeline



1.3 Programming Media Pipeline

1.3.1 Command Sequence

Media pipeline uses a simple programming model. Unlike the 3D pipeline, it does not support pipelined state changes. Any state change requires an MI_FLUSH or PIPE_CONTROL command. When programming the media pipeline, it should be cautious to not use the pipelining capability of the commands described in the Graphics Processing Engine chapter.

To emphasize the non-pipeline nature of the media pipeline programming model, the programmer should note that if any one command is issued in the “Primitive Command” step, none of the state commands described in the previous steps cannot be issued without preceding with a MI_FLUSH or PIPE_CONTROL command.

Note for [DevSNB+]: With the addition of MEDIA_STATE_FLUSH command, pipelined state changes are allowed on the media pipeline. The MEDIA_STATE_FLUSH serves as a fence for state change by flushing the VFE/TS front ends but not waiting for threads to retire.

The basic steps in programming the media pipeline are listed below. Some of the steps are optional; however, the order must be followed strictly. Some usage restrictions are highlighted for illustration purpose. For details, reader should refer to the respective chapters for these commands.

1.3.1.1 Command Sequence [DevSNB+]

For [DevSNB+], the media pipeline is further simplified with fixed functions like MPEG2 VLD and AVC/VC1 IT removed. The addition includes (1) CURBE command is now unique to the media pipeline and (2) the interface descriptors are delivered directly as a media state command instead of being loaded through indirect state.

The programming model is listed as the following.

- Step1: MI_FLUSH/PIPE_CONTROL
    - This step is mandatory.
    - Multiple such commands in step 1 are allowed, but not recommended for performance reason.
- Step2: State command PIPELINE_ SELECT
    - This step is optional. This command can be omitted if it is known that within the same context media pipeline was selected before Step 1.
    - Multiple such commands in step 2 are allowed, but not recommended for performance reason.
- Step3: State commands configuring pipeline states
    - STATE_BASE_ADDRESS
        - This command is mandatory for this step (i.e. at least one).
        - Multiple such commands in this step are allowed. The last one overwrites previous ones.
        - This command must precede any other state commands below.
        - Particularly, the fields **Indirect Object Base Address** and **Indirect Object Access Upper Bound** are used to control indirect Media object load in VF.
        - The fields **Dynamics Base Address** and **Dynamics Base Access Upper Bound** are used to control indirect Curbe and Interface Descriptor object load in VF.
        - *Note: This command may be inserted before (and after) any commands listed in the previous steps (Step 1 to 3). For example, this command may be placed in the ring buffer while the others are put in a batch buffer.*

- STATE_SIP
  - This command is optional for this step. It is only required when SIP is used by the kernels.
- MEDIA_VFE_STATE
  - This command is mandatory for this step (i.e. at least one).
  - This command cause destruction of all outstanding URB handles in the system. A new set of URB handles will be generated based on state parameters, no. of URB and URB length, programmed in VFE FF state.
  - Multiple such commands in this step are allowed. The last one overwrites previous ones.
- MEDIA_CURBE_LOAD
  - This command is optional.
  - Multiple such commands in this step are allowed. The last one overwrites previous ones.
- MEDIA_INTERFACE_DESCRIPTOR_LOAD
  - This command is mandatory for this step (i.e. at least one).
  - Multiple such commands in this step are allowed. The last one overwrites previous ones.

- Step4: Primitive commands
  - MEDIA_OBJECT
    - This step is optional, but it doesn't make practical sense not issuing media primitive commands after being through previous steps to set up the media pipeline.
    - Multiple such commands in step 4 can be issued to continue processing media primitives.

With the addition of MEDIA_STATE_FLUSH command, pipelined state changes are allowed on the media pipeline. In order to support context switch for barrier groups, watermark and barrier dependencies are added to the MEDIA_STATE_FLUSH command. The usage of barrier group may have strict restriction that all threads belonging to a barrier group must all be present in order to avoid deadlock during context switch. Here are the example programming sequences to allow context switch. Note that the use of MEDIA_OBJECT_PRT is optional. It is required to support fast preemption. Similarly, MI_ARB_ON_OFF is optional and is used to support fast preemption.

- MEDIA_VFE_STATE
- MEDIA_INTERFACE_DESCRIPTOR_LOAD

- MEDIA_CURBE_LOAD (optional)
- MEDIA_GATEWAY_STATE (for example for barrier group 1)
- MEDIA_OBJECT_PRT (with VFE_STATE_FLUSH set and PRT NEEDED set.)
- MEDIA_STATE_FLUSH (with watermark set for group 1)
- MI_ARB_ON_OFF (OFF) // Arbitration must be turned off while sending objects for group 1
- Several MEDIA_OBJECT command (for barrier group 1)
- MI_ARB_ON_OFF (ON) // Arbitration is allowed
- MEDIA_STATE_FLUSH (optional, only if barrier dependency is needed)
- MEDIA_INTERFACE_DESCRIPTOR_LOAD (optional)
- MEDIA_CURBE_LOAD (optional)
- MEDIA_GATEWAY_STATE (for example for barrier group 2)
- MEDIA_STATE_FLUSH (with watermark set for group 1)
- MI_ARB_ON_OFF (OFF) // Arbitration must be turned off while sending objects for group 2
- Several MEDIA_OBJECT command (for barrier group 2)
- MI_ARB_ON_OFF (ON) // Arbitration is allowed
- …
- MI_FLUSH

1.3.2 Interrupt Latency

Command Streamer is capable of context switching between primitive commands.

For all independent threads, it is not much a problem. The interrupt latency is dictated by the longest command that is likely to have the largest number of threads. For VLD mode, such a command may be corresponding to a largest slice in a high definition video frame. This is application dependent, there are not much host software can do. For Generic mode, programmer should consider to constrain the compute workload size of each thread.

In modes with child threads, a root thread may be persist in the system for long period of time – staying until its child threads are all created and terminated. Therefore, the corresponding primitive command may also last for long time. Software designer should partition the workload to restrict the duration of each root thread. For example, this may be achieved by partitioning a video frame and assigning separate primitive commands for different data partitions.

In modes with synchronized root threads, a synchronized root thread is dependent on a previous root or child thread. This means context switch is not allowed between the primitive command for the synchronized root thread and the one for the depending thread. So no command queue arbitration should

be allowed between them. Software designer should also restrict the duration of such non-interruptible primitive command segments.

1.4 Video Front End Unit

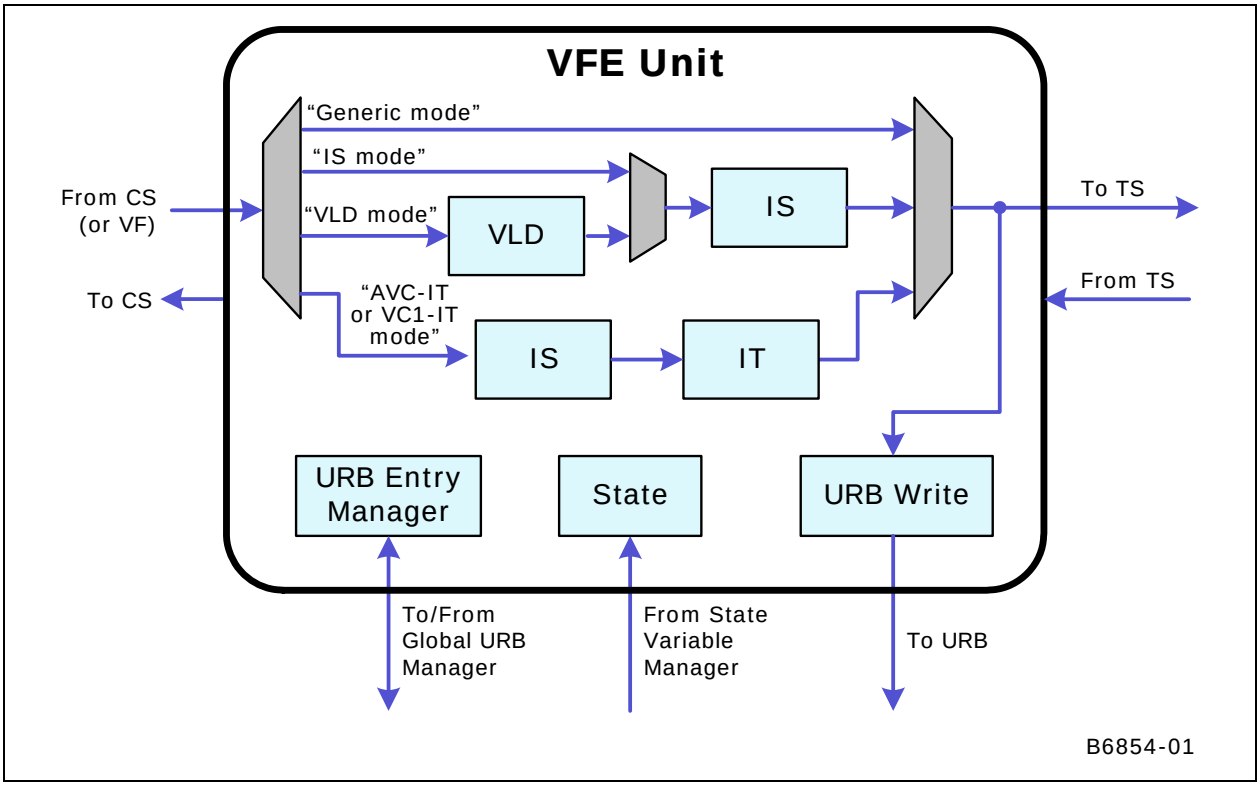
The Video Front End unit is the first fixed function unit in the media pipeline. It processes MEDIA_OBJECT commands to generate root threads by preparing the control (including interface descriptor pointers) and payload (data pushed into the GRF) for the root threads.

VFE supports three modes of operation: Generic mode, Inverse Scan mode and VLD mode.

- **Generic mode**: In the Generic mode, VFE serves as a conduit for general-purpose kernels fully configured by the host software. There is no application specific hardware enabled in this mode.

The following figure illustrates the three modes of operation. The details can be found in the rest of the sections.

Figure 1-2. VFE Functional Blocks and Modes of Operations



MEDIA_STATE_POINTERS command configures VFE in one of the three modes using. Mode switching requires media pipeline state change.

1.4.1 Interfaces

VFE unit acquires its states from Sate Variable Manager, accesses URB handles from the Global URB Manager, receives state and primitive commands from CS unit, writes thread payloads to URB, and sends new thread to TS unit. It does not directly interface to Thread Dispatcher. When VFE is ready for a thread, it sends the interface descriptor pointer for the thread to TS.

1.4.1.1 Interface to Command Streamer

VFE interfaces to CS to acquire the control data, inline data and indirect data of MEDIA_OBJECT commands. The interface supports the throughput of a given mode of operation of VFE. For example, in VLD mode and IS mode, VFE consumes one dword at a time, one dword to the variable length decoder or one dword to the inverse-scan operator. In Generic mode, VFE is capable of a much higher throughput to push indirect data (as thread payload data) into URB. As throughput for indirect data is much higher than that of inline data, when large amount of user data need to be passed through VFE unit, if applicable, it is encouraged to use indirect object load.

1.4.1.2 Interface to Thread Spawner

When a new root thread is fully assembled by VFE, VFE passes to TS the interface descriptor pointer, the URB handle information, etc. In response to this, TS processes the thread information and sends a thread request to TD.

VFE also transmits scratch memory base address received from State Variable Manager to TS, and passes on the Constant URB handle received from CS.

VFE receives URB handle dereference signal from TS.

1.4.1.3 Interface to State Variable Manager

State Variable Manager is responsible of fetching media state structure from memory. VFE only acquires its state variable upon the first primitive command. Therefore, host software is allowed to change media states before issuing primitive commands. As media pipeline does not support pipelined state change, a pipeline flush is required before any state change to make sure that there are no outstanding primitive commands in the pipeline.

1.4.1.4 Interface to Global URB Manager

VFE is responsible for managing URB handles for all root threads. Upon state change, VFE allocates URB handles through the Global URB Manager. VFE manages the URB handles in a circular buffer. URB handle referencing is in a strict order (taking from the head of the circular buffer), even though the handle dereferencing may occur out of order.

When starting a root thread, VFE reference one and only one URB handle, forwarding it to TS. TS then forwards this handle to TD for thread dispatching.

The URB handle for a root thread is used in two ways: (1) serving as buffer space for VFE to assemble thread payload, and (2) serving as the return URB buffer for the root thread to assemble child threads and their payload.

TS sends an indication to VFE when it is safe to dereference the URB handle, and VFE dereferences it. After a URB handle has been dereferenced, VFE can assign it to a new thread.

1.4.1.5 Interface to URB

VFE sends the assembled root thread payload to URB via a wide data bus. In Generic mode, the data comes from the command as inline or indirect data objects. In IS mode, the inline data is directly assembled as URB register wide payloads, and the indirect data are assembled through the Inverse Scan logic. In VLD mode, the data is decoded from the indirect object (i.e. bitstream data).

1.4.2 Mode of Operations

1.4.2.1 Generic Mode

In the Generic mode, VFE serves as a conduit for general-purpose kernels fully configured by the host software. As there is no special fixed function logic used, the Generic mode can also be viewed as a 'pass-through' mode. In this mode, VFE generates a new thread for each MEDIA_OBJECT command. The payload contained in the MEDIA_OBJECT command (inline and/or indirect) is streamed into URB. The interface descriptor pointer is computed by VFE based on the interface descriptor offset value and the interface descriptor base pointer stored in the VFE state. VFE then forwards the interface descriptor pointer and the URB handle to TS to generate a new root thread. Many media processing applications can be supported using the Generic mode: MPEG-2 HWMC, frame rate conversion, advanced deinterface filter, to name a few.

1.4.2.1.1 Interface Descriptor Selection

After populating the URB with the data, VFE notifies TS to initiate the thread. TS needs an interface descriptor pointer to fetch the information for thread initiation. A list of interface descriptors is arranged by the host software as a descriptor array in memory, as shown in the media state model in **Error! Reference source not found.**.

VFE obtains the interface descriptor base pointer from the VFE state structure. The offset into the list of interface descriptors comes from MEDIA_OBJECT command. Each interface descriptor has a fixed size. VFE uses a multiple of the fixed size and the offset to add to the base pointer, and creates the final interface descriptor pointer to be sent to TS.

TS fetches the interface descriptor through the Instruction State Cache (ISC) using the interface descriptor pointer. TS then initializes the thread through the Thread Dispatcher. The interface descriptor pointer is given to TS by VFE for a root thread and by a thread for a child thread. The R0 header is formed by TS for a root thread and is stored in URB by the parent thread for a child thread.

1.4.2.1.2 Scratch Space Allocation

TS handles the allocation of scratch space. Since TS does not have a normal state interface, VFE receives the scratch space configuration with the VFE state, then forwards the configuration to TS with the interface descriptor pointer.

1.4.3 Parameterized Media Walker [DevSNB+]

The Parameterized Media Walker is a hardware thread generation mechanism that creates threads associated with units in a generalized 2-dimensional space, for example, blocks in a 2D image. With a small number of unit step vectors, the walker can implement a large number of walking patterns as described hereafter. This command may provide functions that are normally handled by the host software, thus, may be used to simplify the host software and GPU interface.

The walker described herein is doubly nested, where essentially a “local” walker can perform a variety of 2-dimensional walking patterns and a “global” walker can perform similar 2-dimensional walking patterns upon many local walkers. The local walker has 3 levels (outer, middle, and inner) while the global walker has 2 levels (outer and inner). Thus, the algorithm has 5-nested loops that modify local state based on user-defined unit step vectors.

The Walker’s programmability is derived from:

- The walker traverses a unit-normalized surface. Some example unit sizes:
    - 1x1: Walking pixels
    - 4x4: Walking sub-blocks
    - 16x16: Walking macro-blocks
    - 32x16: Walking macro-block-pairs
- The use of unit step vectors to describe the motion at each of level of nesting
- Starting locations for the local and global walkers
- Block sizes of the local and global walker
- And a small number of special mode controls for the inner-most loop which are aimed at efficiently dividing an image into two balanced workloads for dual-slice designs.

1.4.3.1 Walker Parameter Description

The global and local loops are both described by the same four parameters:

- Resolution,
- Starting location,
- Outer unit vector,
- Inner unit vector

The local inner loop has some special modes that will be described later. A table of the user inputs and some example values are given below:

GLOBAL LOOP PARAMETERS							
Global Resolution		Global Start		Outer Loop Unit Vector		Inner Loop Unit Vector	
X	Y	X	Y	X	Y	X	Y
120	68	0	0	32	0	0	32

LOCAL LOOP PARAMETERS							
Block Resolution		Local Start		Outer Loop Unit Vector		Inner Loop Unit Vector	
X	Y	X	Y	X	Y	X	Y
32	32	0	0	1	0	-2	2

LOCAL INNER LOOP SPECIAL MODE SELECTS							
Dual Mode	Repel	Attract			Extra Steps	X	Y
TRUE	FALSE	FALSE			1	0	1

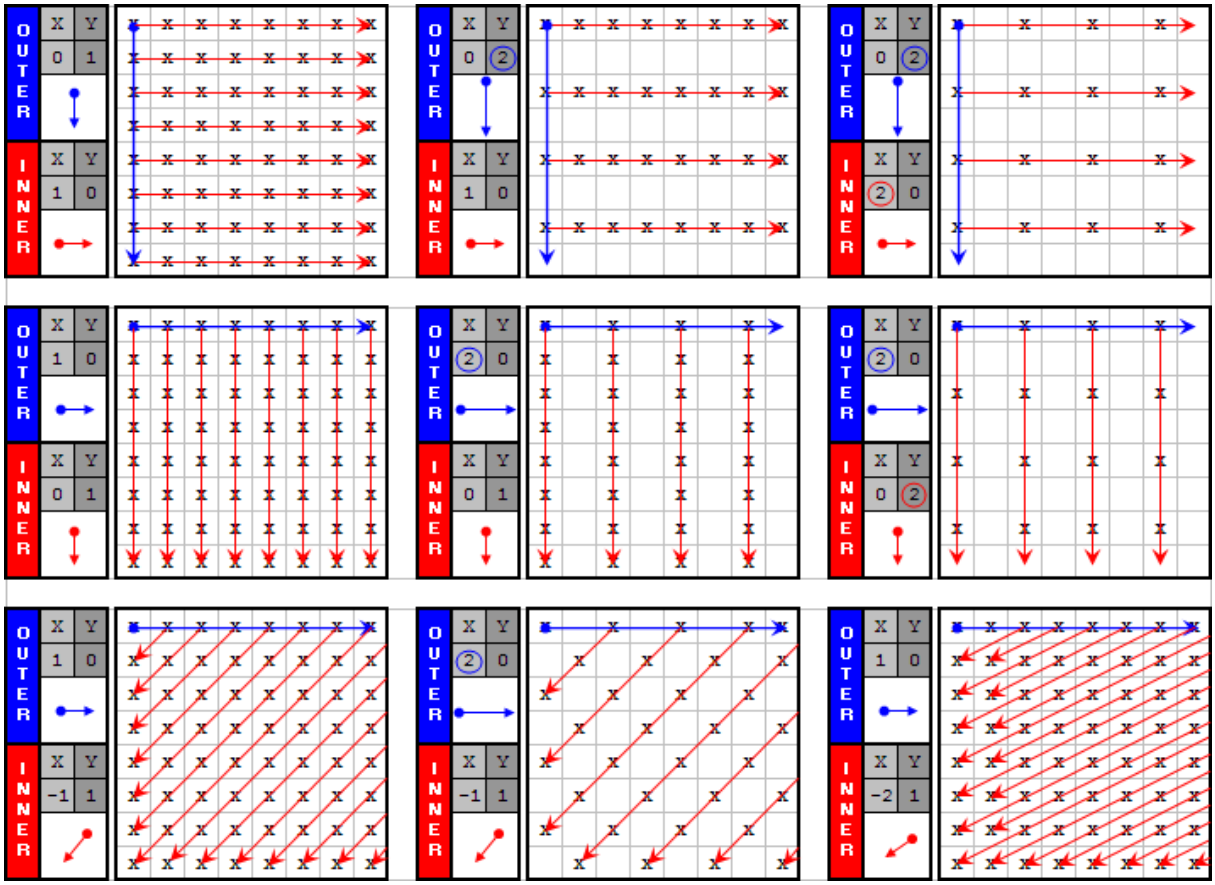
It should be emphasized that the value of what a “unit” represents is implicitly defined by the user. In other words, the walker traverses a “unit normalized space” that is not inherently bound to pixel walking. If the smallest unit of work the user wants to walk is a 4x3 block of pixels, you can program the inner loop to step (4,3) or (1,1):

- In the first case (4,3) the user is walking in units of pixels
- In the second case (1,1) the user is walking in units of 4x3 blocks of pixels.

It should be noted that hardware doesn’t contain enough bits for pixel walking for pixel resolution like 1920x1088. The intended usage of the walker is for block walking whereas the block size is not relevant to the walker parameters.

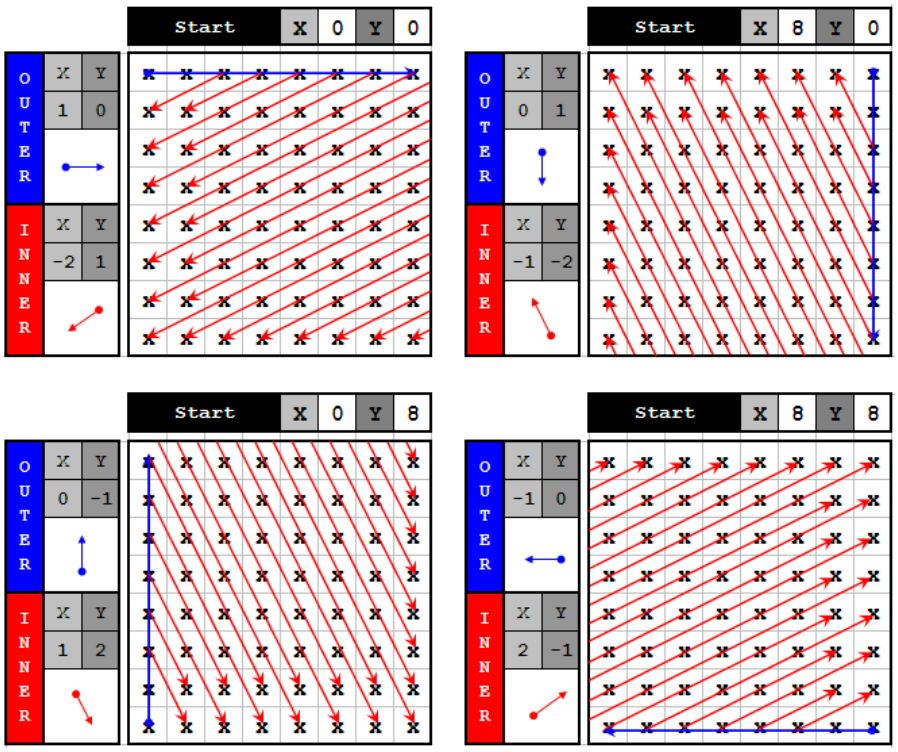
1.4.3.2 Basic Parameters for the Local Loop

The local inner and outer loop xy-pair parameters alone can describe a large variety of primitive walking patterns. Below are 9 primitive walking patterns generated by varying only the inner and outer unit step vectors of the local loop:



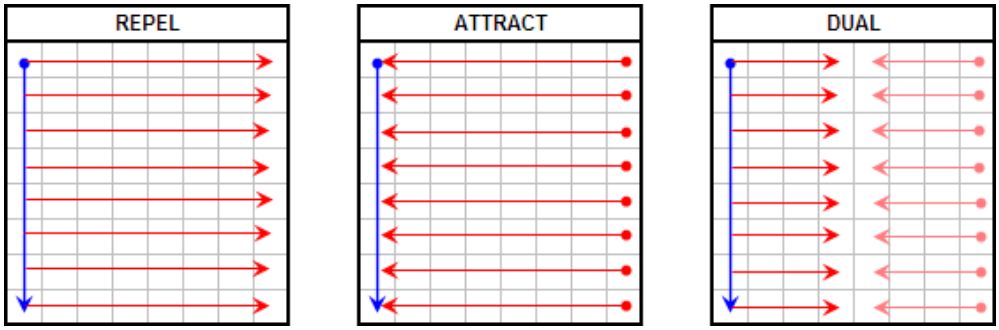
- The top row shows the outer unit vector pointing down (+Y) and the inner unit vector pointing right (+X). Rows and columns can easily be skipped by increasing the unit step vectors above one.
- The middle row the outer unit vector pointing right (+X) and the inner unit vector pointing down (+Y). Again, rows and columns are skipped by increasing the unit step vectors beyond one.
- The last row shows the capability to walk angles not perpendicular to the edge. The 1st shows a 45º walking pattern by setting the inner unit vector to (-1,1). The 2nd shows a checkerboard pattern by skipping every other outer loop and retaining the inner unit vector of (-1,1). The 3rd shows a 26.5º walking pattern by setting the inner unit vector to (-2,1).

The block resolution, shown as [8,8], and the starting location, currently [0,0], can be varied and the above patterns can be stretched and rotated many ways. The diagram below shows an example of where the start position and unit step vectors can be set to achieve a full rotation of the same pattern:

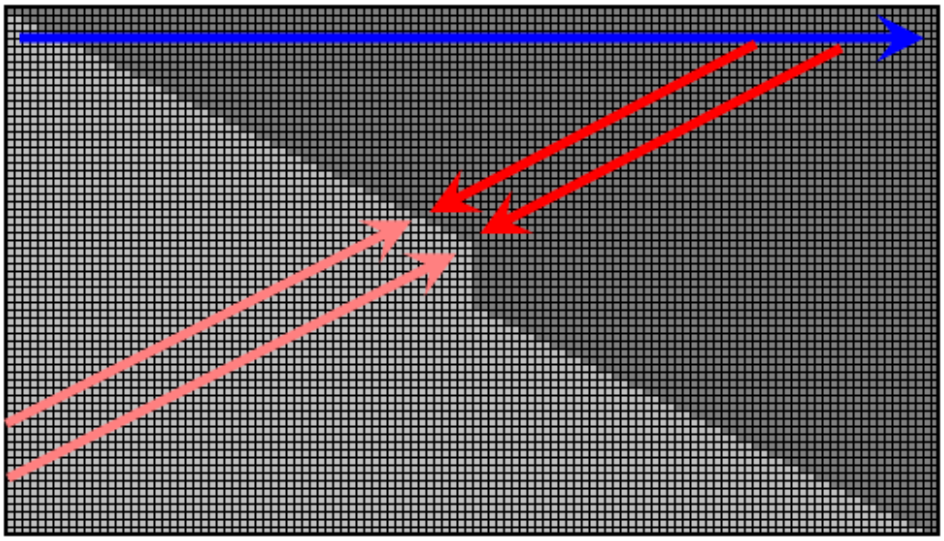


1.4.3.3 Dual Mode of Local Loop

The local Inner Loop Special mode selects are included to aid in the distribution of work, specifically with two slices in mind. Essentially, the local inner loop can be bisected and each half-walk can be directed inward towards the center of the image (dual). The local inner loop need not be bisected, and can either move away from the outer loop (repel) or move towards it (attract) when an even split is not desired:

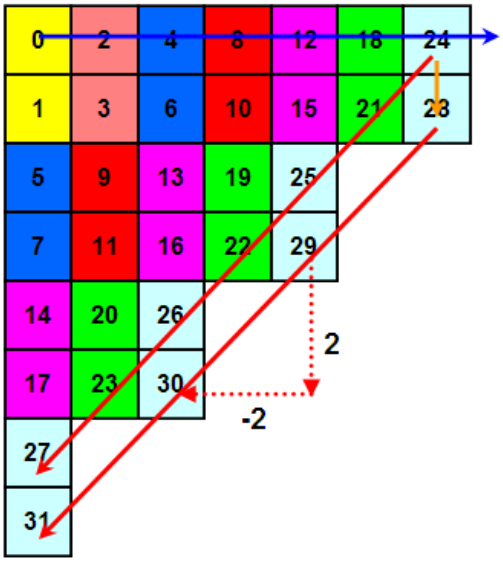


In Dual mode, the sequence will alternate between two half-walks such that every-other output would go to the same slice. This effect will produce a more balanced workload to two slices as shown in the example below where the color of the block represents which slice it was dispatched to. This is the walker's approach to fine-grained parallelism.



1.4.3.4 MbAff-Like Special Case in Local Loop

The local loop has an additional middle loop that is used to achieve some specific walking patterns, with MBAFF mode especially in mind. A pattern to handle MBAFF AVC content is to walk the top macroblocks of all macroblock pairs (MB-pairs) on a wavefront followed by the respective bottom macroblocks. The pattern is shown below.

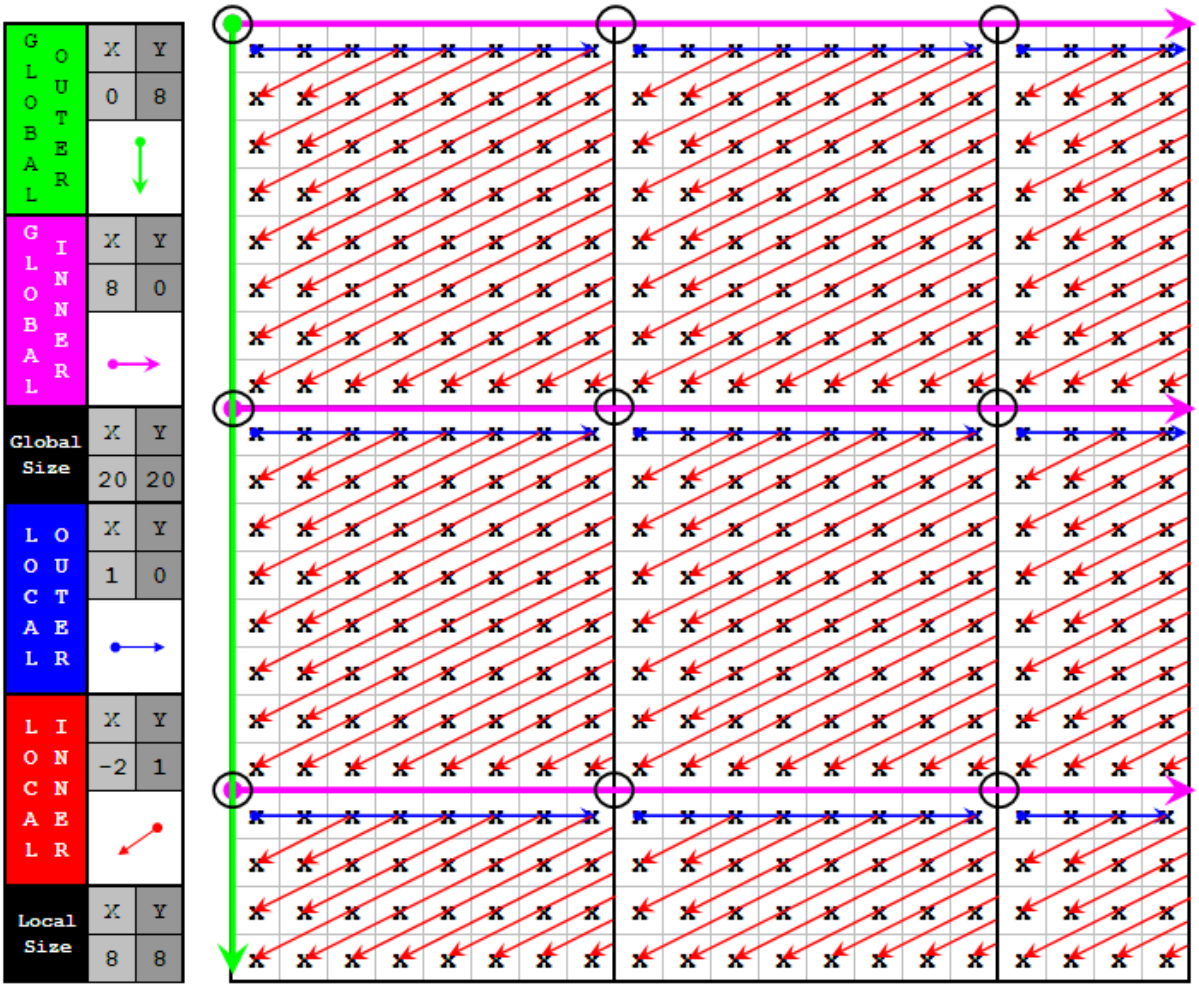


The outer loop unit step vector would be [1, 0] and the inner loop unit step vector would be [-2, 2]. A third loop is necessary to repeat the inner loop, only shifted down a unit before restarting. Thus, a middle loop with a unit step vector of [0,1] would achieve this MBAFF pattern. Additionally, the number of "extra steps" taken by the middle loop would be 1 in this case.

The addition of a middle loop also creates more overall flexibility, which seems necessary due to the integer-based unit step vector solution proposed (Manhattan distance issues etc.).

1.4.3.5 Global Loop

The same set of general parameters is used to describe the global loop as well. Thus, a global loop that is walking a raster-scan pattern can be combined with a local loop that is walking a 26.5º pattern (or vice-versa). As shown in the example below, if the local block size ([8,8]) is not an even multiple of the global resolution ([20,20]), the slack is still processed by dynamically changing the local block resolution.



The global loop will always resolve to be the upper-left corner of the local loop, shown above black circles. Note that local loop can still start in any corner of the local block, but the local (0,0) will always be the location where global loop begins the local loop, hence the upper-left corner.

The user can specify where the staring location of the global loop as with the local loop. If the user were to set the global starting location to (16,16) in the previous example, after inverting the global outer and global inner unit step vectors the same pattern would be achieved in the reverse order. Note that the slack would still be handled along the right and bottom edge of the global image in that case. The user could have also started at (12,12) in which case the slack would be handled on the left and top faces.

1.4.3.6 Walker Algorithm Description

The walker algorithm has been tested and optimized in software. A high-level pseudo-code description is given below:

```
Walker(){ //C-Style Pseudo-Code of Walker Algorithm
  Load_Inputs_And_Initialize();
  While (Global_Outer_Loop_In_Bounds()){
    Global_Inner_Loop_Intialization();
    While (Global_Inner_Loop_In_Bounds()){
      Local_Block_Boundary_Adjustment();
      Local_Outer_Loop_Initialization();
      While (Local_Outer_Loop_In_Bounds()){
        Local_Middle_Loop_Initialization();
        While (Local_Middle_Steps_Remaining()){
          Local_Inner_Loop_Initialization();
          While (Local_Inner_Loop_Is_Shrinking()){
            Execute();
            Calculate_Next_Local_Inner_X_Y();
          } //End Local Inner Loop
          Calculate_Next_Local_Middle_X_Y();
        } //End Local Middle Loop
        Calculate_Next_Local_Outer_X_Y();
        Calculate_Next_Local_Inverse_Outer_X_Y();
      } //End Local Outer Loop
      Calculate_Next_Global_Inner_X_Y();
    } //End Global Inner Loop
    Calculate_Next_Global_Outer_X_Y();
  } //End Global Outer Loop
} //End Walker
```

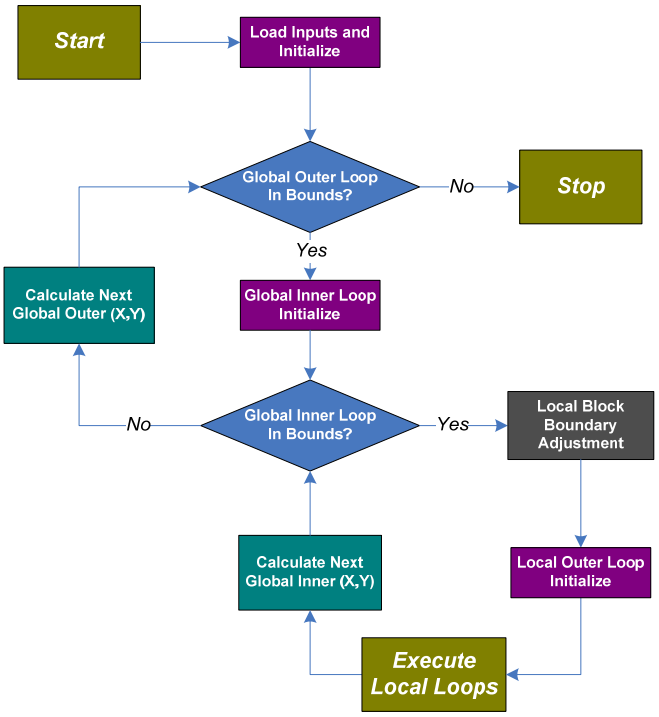
The pseudo-code has the following characteristics:

- There are 5 levels of iteration
- The highest 2 levels are called “global” and the lowest 3 levels are called “local”
    - The global loop is split into an outer and an inner loop.
    - The local loop is split into an outer, a middle, and an inner loop.
    - A bounding box for the global and local resolution is defined by the user.
    - The starting location within each bounding box is also specified by the user.
- Each of the 5 loops has its own persistent
    - Current position (x,y)
    - Unit step vector (x,y)
- The final output (x,y) is a summation of the global x,y and the local x,y.
- The next (x,y) for given level can be calculated while the next lower level is still executing. Additionally, the result can be used to check to see if the current level will execute again once control is returned.

The flow of the global outer and inner loops is:

1. Check a bound condition
2. Initialize the next level loop
3. Execute the next level loop
4. When the next level loop fails its condition, calculate the next position for the current loop level and repeat.

Figure 1-3. Walker algorithm flowchart for the Global Loop

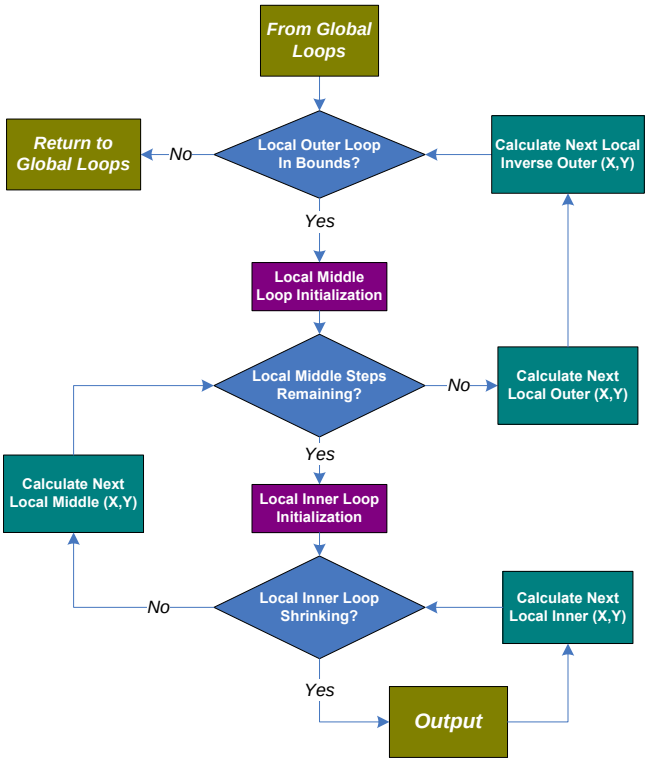


Take note of the grey box “Local Block Boundary Adjustment”. This logic is necessary to adjust the local block size when the distance between the current global position to the edge of the image is less than the local resolution. Additionally, the local starting positions might be modified here as well if the defined starting position is larger than the new local block size.

The flow of the 3 local loops does not vary much from the 2 global loops. The differences are:

- In addition to a boundary check, the local middle loop also ensures the number of middle steps is less than or equal to the user defined “number of extra steps”.
- The local inner loop only checks to see if the prior distance between the x,y starting and ending points are greater than their current distance. If this is true, it implies that the two inner loops are converging towards each other.
- When the middle loop check fails, both the starting points (local outer) and ending points (local inner) are updated.

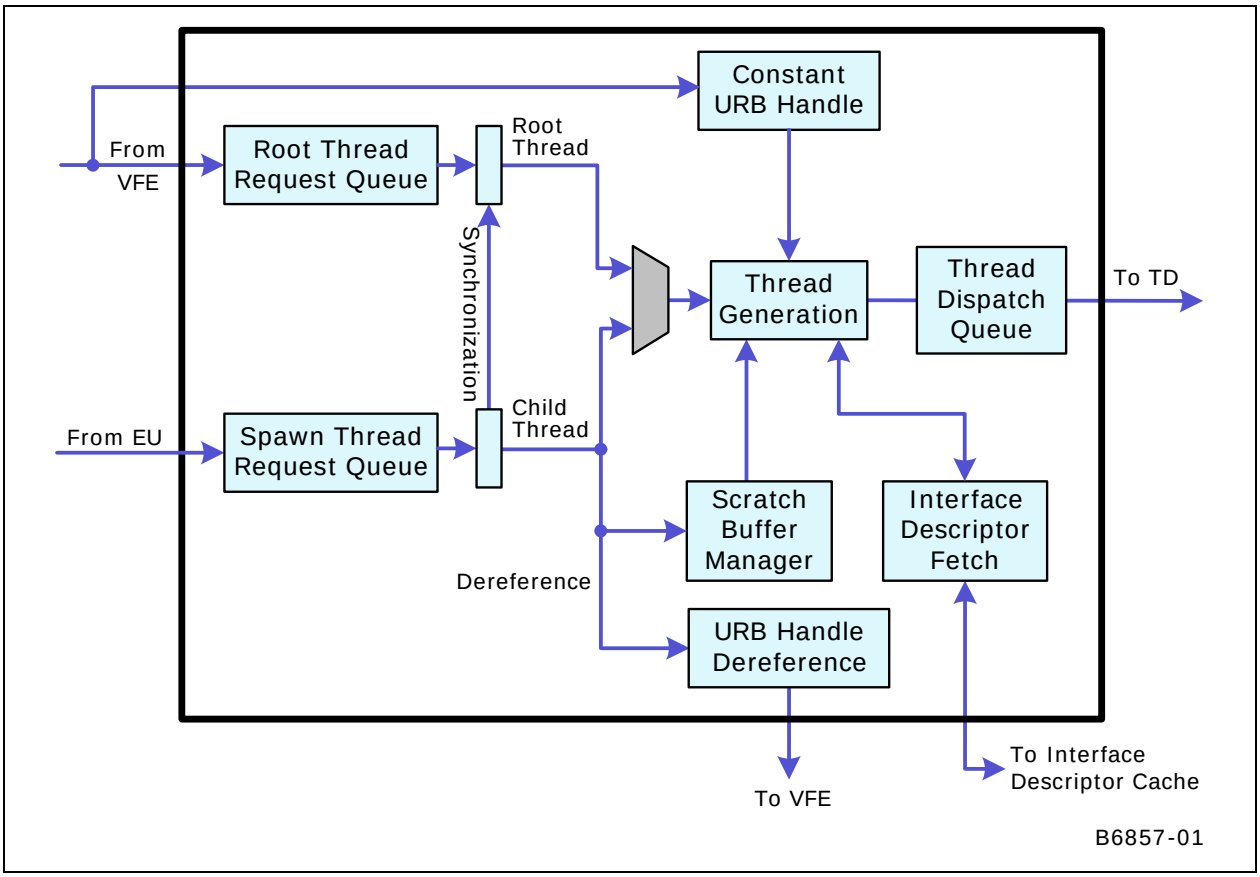
Figure 1-4. Walker algorithm flowchart for the Local Loop



1.5 Thread Spawner Unit

The Thread Spawner (TS) unit is responsible for making thread requests (root and child) to the Thread Dispatcher, managing scratch memory, maintaining outstanding root thread counts, and monitoring the termination of threads.

Figure 1-5. Thread Spawner block diagram



1.5.1 Basic Functions

1.5.1.1 Root Threads Lifecycle

Thread requests sourced from VFE are called **root threads**, since these threads may be creating subsequent (child) threads. A root thread may a macroblock thread created by VFE as in VLD mode, or may be a general-purpose thread assembled by VFE according to full description provided by host software in Generic mode.

Thread requests are stored in the Root Thread Queue. TS keeps everything needed to get the root threads ready for dispatch and then tracks dispatched threads until their retirement.

TS arbitrates between root thread and child thread. The root thread request queue is in the arbitration only if the number of outstanding threads does not exceed the maximum root thread state variable. Otherwise, the root thread request queue is stalled until some other root threads retire/terminate.

Once a root thread is selected to be dispatched, its lifecycle can be described by the following steps:

1. TS forwards the interface descriptor pointer to the L1 interface descriptor cache (a small fully associated cache containing up to 4 interface descriptors). The interface descriptor is either

found in the cache or a corresponding request is forwarded to the L2 cache. Interface descriptors return back to TS in requesting order.

- Once TS receives the interface descriptor, it checks whether maximum concurrent root thread number has reached to determine whether to make a thread dispatch request or to stall the request until some other root threads retire. If the thread requests the use of scratch memory, it also generates a pointer into the scratch space.

2. TS then builds the transparent header and the R0 header.

3. Finally, TS makes a thread request to the Thread Dispatcher.

4. TS keeps track of dispatched thread, and monitors messages from the thread (resource dereference and/or thread termination). When it receives a root thread termination message, it can recover the scratch space and thread slot allocated to it. The URB handle may also be dereferenced for a terminated root thread for future reuse. It should be noted that URB handle dereference may occur before a root thread terminates. See detailed description in the Media Message section.

   - It is the root thread's responsibility (software) to guarantee that all its children have retired before the root thread can retire.

1.5.1.2 URB Handles

VFE is in charge of allocating URB handles for root threads. One URB handle is assigned to each root thread. The handle is used for the payload into the root thread.

If Children Present is not set (root-without-child case), TS signals VFE to dereference the URB handle immediately after it receives acknowledgement from TD that the thread is dispatched.

If Children Present is set (root-with-child case), the URB handle is forwarded to the root thread and serves as the return URB handle for the root thread. TS does not signal deference at the time of dispatch. TS signals URB handle deference only when it receives a resource dereference message from the thread.

Children Present is a command variable in the _OBJECT command

1.5.1.3 Root to Child Responsibilities

Any thread created by another thread running in an EU is called a **child thread**. Child threads can create additional threads, all under the tree of a root which was requested via the VFE path.

A root thread is responsible of managing pre-allocated resources such as URB space and scratch space for its direct and indirect child threads. For example, a root thread may split its URB space into sections. It can use one section for delivering payload to one child thread as well as forwarding the section to the child thread to be used as return URB space. The child thread may further subdivide the URB section into subsections and use these subsections for its own child threads. Such process may be iterated. Similarly, a root thread may split its scratch memory space into sections and give one scratch section for one child thread.

TS unit only enforces limitation on number of outstanding root threads. It is the root threads' responsibility to limit the number of child threads in their respected trees to balance performance and avoid deadlock.

1.5.1.4 Multiple Simultaneous Roots

Multiple root threads are allowed concurrently running in GEN4 execution units. As there is only one scratch space state variable shared for all root threads, all concurrent root thread requiring scratch space share the same scratch memory size. Figure 1-6 depicts two examples of thread-thread relationship. The left graph shows one single tree structure. This tree starts with a single root thread that generates many child threads. Some child threads may create subsequent child threads. The right graph shows a case with multiple disconnected trees. It has multiple root threads, showing sibling roots of disconnected trees. Some roots may have child threads (branches and leafs) and some may not.

There is another case (as shown in Figure 1-7) where multiple trees may be connected. If a root is a synchronized root thread, it may be dependent on a preceding sibling root thread or on a child thread.

Figure 1-6. Examples of thread relationship

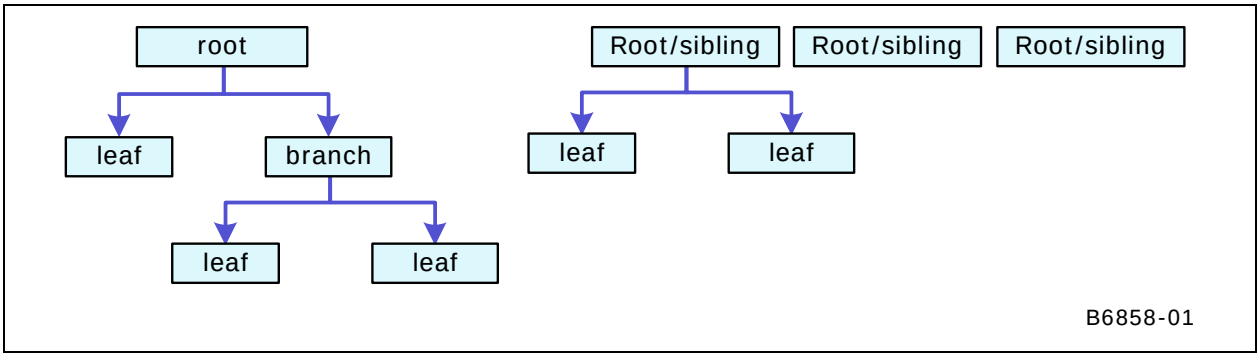
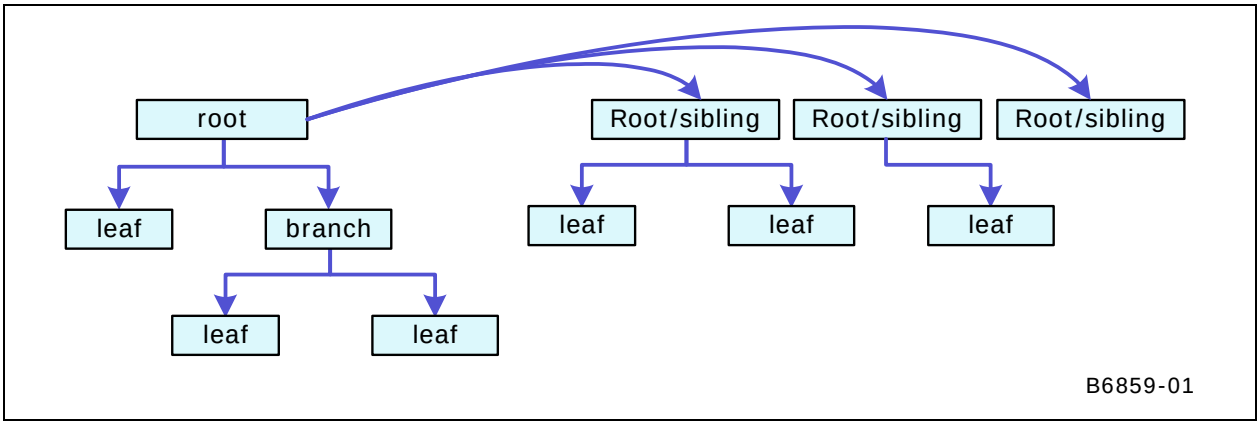


Figure 1-7. A example of thread relationship with root sibling dependency



1.5.1.5 Synchronized Root Threads

A synchronized root thread (SRT) originates from a MEDIA_OBJECT command with Thread Synchronization field set. Synchronized root threads share the same root thread request queue with the non-synchronized roots. A SRT is not automatically dispatched. Instead, it stays in the root thread request queue until a spawn-root message is at the head of the child thread request queue. Conversely, a spawn-root message in the child thread request queue will block the child thread request queue until the head of

root thread request queue is a SRT. When they are both at the head of queues, they are taken out from the queue at the same time.

A spawn-root message may be issued by a root thread or a child thread. There is no restriction. However, the number of spawn-root messages and the number of SRT must be identical between state changes. Otherwise, there can be a deadlock. Furthermore, as both requests are blocking, synchronized root threads must be used carefully to avoid deadlock.

When Scoreboard Control is enabled, the dispatch of a SRT originated from a MEDIA_OBJECT_EX command is still managed by the same way in addition to the hardware scoreboard control.

1.5.1.6 Deadlock Prevention

Root threads must control deadlock within their own child set. Each root is given a set of preallocated URB space; to prevent deadlock it must make sure that all the URB space is not allocated to intermediate children who must create more children before they can exit.

There are limits to the number of concurrent threads. The upper bound is determined by the number of execution units and the number of threads per EU. The actual upper bound on number of concurrent threads may be smaller if the GRF requirement is large. Deadlock may occur if a root or intermediate parent cannot exit until it has started its children but there is no space (for example, available thread slot in execution units) for its children to start.

To prevent deadlock, the maximum number of root threads is provided in VFE state. The Thread Spawner keeps track of how many roots have been spawned and prevents new roots if the maximum has been reached. When child threads are present, it is software's responsible of constraining child thread generation, particularly the generation of child threads that may also spawn more child threads.

Child thread dispatch queue in TS is another resource that needs to be considered in preventing deadlock. The child thread dispatch queue in TS is used for (1) message to spawn a child thread, (2) message to spawn a synchronized root thread, and (3) thread termination message. If this queue is full, it will prevent any thread to terminate, causing deadlock.

For example, if an application only has one root thread (max # of root threads is programmed to be one). This root thread spawns child threads. In order to avoid deadlock, the maximum number of outstanding child thread that this root thread can spawn is the sum of the maximum available thread slots plus the depth of the child thread dispatch queue minus one.

Max_Outstanding_Child_Threads = (Thread Slot Number – 1) + (TS Child Queue Depth – 1)

Adding other root threads (synchronized and/or non-synchronized) to the above example, the situation is more complicated. A conservative measure may have to use to prevent deadlock. For example, the root thread spawning child threads may have to exclude the max number of root threads as in the following equation to compute the maximum number of outstanding child threads to be dispatched.

Max_Outstanding_Child_Threads = (Thread Slot Number – 1) + (TS Child Queue Depth – 1) – (Max Root Threads-1)

Table 1-2. TS Resource Available in Device Hardware

Device	Child Thread Dispatch Queue Depth
[DevSNB+]	OPEN

1.5.1.7 Child Thread Lifecycle

When a (parent) thread creates a child thread, the parent thread behaves like a fixed function. It provides all necessary information to start the child thread, by assembling the payload in URB (including R0 header) and then sending a spawn thread message to TS with following data:

- An interface descriptor pointer for the child thread.
- A pointer for URB data

The interface descriptor for a child may be different from the parent – how the parent determines the child interface descriptor is up to the parent, but it must be one from the interface descriptor array on the same interface descriptor base address.

The URB pointer is not the same as a URB handle. It does not have an URB handle number and does not appear in any handle table. This is acceptable because the URB space is never reclaimed by TS after a child is dispatched, but rather when the parent releases its original handles and/or retires.

The child request is stored in the child thread queue. The depth of the queue is limited to 8, overrun is prevented by the message bus arbiter which controls the message bus. The arbiter knows the depth of the queue and will only allow 8 requests to be outstanding until the TS signals an entry has been removed.

As mentioned previously, child threads have higher priority over root threads. Once TS selects a child thread to dispatch, it follows these steps:

1. TS forwards the interface descriptor pointer to the L1 interface descriptor cache (a small fully associated cache containing up to 4 interface descriptors). The interface descriptor is either found in the cache or a corresponding request is forwarded to the L2 cache. Interface descriptors return back to TS in requesting order.
2. TS then builds the transparent header but not the R0 header.
3. Finally, TS makes a thread request to the Thread Dispatcher.
4. Once the dispatch is done, TS can forget the child – unlike roots, no bookkeeping is done that has to be updated when the child retires.

If more data needs to be transferred between a parent thread and its child thread than that can fit in a single URB payload, extra data must be communicated via shared memory through data port.

1.5.1.8 Arbitration between Root and Child Threads

When both root thread queue and child thread queue are both non-empty, TS serves the child thread queue. In other words, child threads have higher priority over root threads. The only condition that the child thread queue is stalled by the root thread queue is that the head of child thread queue is a root-synchronization message and the head of root thread queue is not a synchronized root thread.

1.5.1.9 Persistent Root Thread

Persistent Root Thread (PRT) is in the context of multi-context scheduling, where the thread supports midstream interruptability for fine grain context switch. A persistent root thread in general stays in the system for a long period of time. It is normally a parent thread, and only one PRT is allowed in the system at a time. Upon context switch interrupt, instead of proceeding to completion, a PRT can save its software context and terminate. The PRT can be restarted later, *even if it had completed normally the last time it was executed*. Therefore, the PRT must always save enough context (via data port messages to a predefined surface) to allow it to restart from where it left off (including determining that it has nothing left to do). However, since only one PRT can execute at a time, once the next PRT starts, the previous one will never be restarted, thus the context save surface can be reused from one PRT to the next.

A PRT may check the Thread Restart Enable bit in the R0 header to find out whether it is a fresh start or resumed from a previous interrupt and then can continue operations from that previously saved context.

PRT can be interleaved with other root (such as parent root thread, or synchronized root thread) and child threads. A parent root thread is not necessarily a PRT, and doesn't have to be as long as it can be finished in deterministic time that is shorter than required for fine-grain context switch interrupt.

Use of PRT must follow the following rule:

- There can only be one PRT in the media pipeline at a given time. That means, there shall not be any other media primitive commands (MEDIA_OBJECT or MEDIA_OBJECT_EX) between it and the previous MI_FLUSH command. In other words, when multiple such PRTs are used in a sequence of media primitive commands, MI_FLUSH must be inserted.

1.5.2 Interfaces

1.5.2.1 Interface to VFE

TS receives an interface descriptor pointer and a URB handle from VFE. It uses the interface descriptor pointer to fetch the interface descriptor. TS uses the information in the interface descriptor along with the URB handle to fill out the transparent header in the message to TD for all threads. For root thread, TS also generate the R0 header.

TS transmits URB handle dereference signal to VFE. As described previously, the derefernce signal may be at dispatch time or at later time depending on Children Present. No matter which case, there is one and only one URB handle dereference for a thread.

1.5.2.2 Interface to Thread Dispatcher

TS creates the transparent header, assembles the URB handles and calls TD to dispatch a new thread. For an unsynchronized root thread, there is one URB handle managed by VFE and optionally one Constant URB handle managed by CS. For a synchronized root thread, there is one URB handle managed by VFE, a URB handle created by the synchronizing thread (the one that sends the 'spawn root thread' message, and optionally one Constant URB handle managed by CS. For a child thread, there is one URB handle managed by the parent thread plus an optional Constant URB handle.

1.6 Media State Model [DevSNB+]

The media state model is based on in-line state load mechanism. VFE state, URB configuration and Interface Descriptors are loaded to VFE hardware through state commands.

All Interface Descriptors have the same size and are organized as a contiguous array in memory. They can be selected by Interface Descriptor Index for a given kernel. This allows different kinds of kernels to coexist in the system.

1.7 Media State and Primitive Commands

1.7.1 MEDIA_VFE_STATE Command [DevSNB+]

An MI_FLUSH is required before MEDIA_VFE_STATE unless the only bits that are changed are scoreboard related: Scoreboard Enable, Scoreboard Type, Scoreboard Mask, Scoreboard * Delta. For these scoreboard related states, a MEDIA_STATE_FLUSH is sufficient.

- MEDIA_STATE_FLUSH (optional, only if barrier dependency is needed)
- MEDIA_INTERFACE_DESCRIPTOR_LOAD (optional)

DWord	Bit	Description
0	31:29	**Command Type =** GFXPIPE = 3h
	28:16	**Media Command Opcode =** MEDIA_VFE_STATE Pipeline[28:27] = Media = 2h; Opcode[26:24] = 0h; Subopcode[23:16] **=** 0h
	15:0	**DWord Length** (Excludes DWords 0,1) = 06h
1	31:10	**Scratch Space Base Pointer.** Specifies the 1k-byte aligned address offset to scratch space for use by the kernel. This pointer is relative to the **General State Base Address**. Format = GeneralStateOffset[31:10]
	9:8	Reserved: MBZ
	7:4	Stack Size Range = [0,11] indicating [1KBytes, 2MBytes]
	3:0	**Per Thread Scratch Space.** Specifies the amount of scratch space allowed to be used by each thread. The driver must allocate enough contiguous scratch space, pointed to by the Scratch Space Pointer, to ensure that the Maximum Number of Threads each get Per Thread Scratch Space size without exceeding the driver-allocated scratch space. **Note:** The definition of this field is different from that in 3D fixed functions, where the per-thread scratch space is specified in powers of 2. Format = U4 Range = [0,11] indicating [1k bytes, 12k bytes] [DevSNB]

DWord	Bit	Description
2	31:16	**Maximum Number of Threads.** Specifies the maximum number of simultaneous root threads allowed to be active. Used to avoid using up the scratch space, or to avoid potential deadlock. Note that MSB will be zero due to the range limit below. Format = U16 representing (thread count – 1) Range = [0, n-1] where n = (# EUs) * (# threads/EU). See *Graphics Processing Engine* for listing of #EUs and #threads in each device.
	15:8	**Number of URB Entries.** Specifies the number of URB entries that are used by the unit. Format = U8 Range = [0,64] → [0,64] Entries [DevSNB]
	7	**Reset Gateway Timer.** This field controls the reset of the timestamp counter maintained in Message Gateway. 0 – Maintaining the existing timestamp state 1 – Resetting relative timer and latching the global timestamp
	6	**Bypass Gateway Control.** This field configures Gateway to use a simple message protocol. 0 – Maintaining OpenGateway/ForwardMsg/CloseGateway protocol (legacy mode) 1 – Bypassing OpenGateway/CloseGateway protocol
	5	**Fast Preempt.** This field controls when an preempt signal is sent (broadcast) to EUs upon a context switch interrupt. If this field is set to 1, TS hardware will send the preempt signal to EUs, when the media pipe is ready (drained). If this field is set to 0, TS hardware will wait until the number of outstanding thread to be 1 (only the _PRT) before sending the preempt signal to EUs. 0 – Preempt only _PRT 1 – Fast Preempt
	4:3	**Reserved: MBZ**
	2	**Reserved:MBZ**
	1:0	**Reserved**
3	31:8	**Reserved**
	7:0	Reserved : MBZ
4	31:16	**URB Entry Allocation Size.** Specifies the length of each URB entry used by the unit, in 256-bit register increments. ROB address for URB starts after CURBE Allocated region Format = U12 Range = [0, 991] 256-bit register increments. ROB has 32KB of storage. (URB Entry Allocation Size * Number of URB Entries) + CURBE Allocation Size + 32 must be less than or equal to 1024 [DevSNB+]

DWord	Bit	Description
	15:0	**CURBE Allocation Size.** Specifies the total length allocated for CURBE, in 256-bit register increments – 1. ROB address for CURBE starts at address 32. Format = U12 Range = [0, 991] 256-bit register increments. ROB has 32KB of storage. (URB Entry Allocation Size * Number of URB Entries) + CURBE Allocation Size + 32 must be less than 1024 [Dev-SNB+] Range = [0, 2015] 256-bit register increments. ROB has 64KB of storage. (URB Entry Allocation Size * Number of URB Entries) + CURBE Allocation Size + 32 must be less than 2048 [DevSNB+]
5	31	**Scoreboard Enable.** This field enables and disables the hardware scoreboard in the Media Pipeline. If this field is cleared, hardware ignores the following scoreboard state fields. 0 – Scoreboard disabled 1 – Scoreboard enabled
	30	**Scoreboard Type.** This field selects the type of scoreboard in use. 0 – Stalling scoreboard 1 – Non-stalling scoreboard
	29:8	Reserved : MBZ
	7:0	**Scoreboard Mask.** Each bit indicates the corresponding dependency scoreboard is enabled. The scoreboard is based on the relative (X, Y) distance from the current threads' (X, Y) position. **Bit n (for n = 0…7):** Score n is enabled Format = TRUE/FALSE
6	31:28	**Scoreboard 3 Delta Y.** Relative vertical distance of the dependent instance assigned to scoreboard 3, in the form of 2's compliment. Format = S3
	27:24	**Scoreboard 3 Delta X.** Relative horizontal distance of the dependent instance assigned to scoreboard 3, in the form of 2's compliment. Format = S3
	23:16	**Scoreboard 2 Delta (X, Y)**
	15:8	**Scoreboard 1 Delta (X, Y)**
	7:0	**Scoreboard 0 Delta (X, Y)**
7	31:24	**Scoreboard 7 Delta (X, Y)**
	23:16	**Scoreboard 6 Delta (X, Y)**
	15:8	**Scoreboard 5 Delta (X, Y)**
	7:0	**Scoreboard 4 Delta (X, Y)**

1.7.2 MEDIA_CURBE_LOAD Command [DevSNB+]

MEDIA_CURBE_LOAD			
Project:	[DevSNB+]	**Length Bias:**	2
Desc			

DWord	Bit	Description
0	31:29	**Command Type** Default Value: 3h GFXPIPE Format: OpCode
	28:27	**Media Command Opcode** Default Value: 2h MEDIA_CURBE_LOAD Format: OpCode
	26:24	**Media Command Opcode** Default Value: 0h MEDIA_CURBE_LOAD Format: OpCode
	23:16	**Media Command Opcode** Default Value: 1h MEDIA_CURBE_LOAD Format: OpCode
	15:0	**DWord Length** Default Value: 2h Excludes DWord (0,1) Format: =n Total Length - 2 Project: All
1	31:0	**Reserved** Project: All Format: MBZ
2	31:17	**Reserved** Project: All Format: MBZ
	16:0	**CURBE Total Data Length** Project: All Format: U17 In bytes This field provides the length in bytes of the CURBE data. This field must have the same alignment as the Curbe Object Data Start Address. It must be DQWord (32-byte) aligned. As the CURBE data are sent directly to ROB, range is limited to CURBE Allocation Size.
3	31:0	**CURBE Data Start Address** Project: All Address: DynamicBaseAddress[31:0] Surface Type: CURBE Range 0..4GB Specifies the 32-byte aligned address of the CURBE data. This pointer is relative to the **Dynamics Base Address**. It must be DWord aligned.

1.7.3 MEDIA_INTERFACE_DESCRIPTOR_LOAD Command [DevSNB+]

MEDIA_INTERFACE_DESCRIPTOR_LOAD			
Project:	[DevSNB+]	**Length Bias:**	2
Desc			

DWord	Bit	Description
0	31:29	**Command Type** Default Value: 3h GFXPIPE Format: OpCode
	28:27	**Media Command Opcode** Default Value: 2h MEDIA_INTERFACE_DESCRIPTOR_LOAD Format: OpCode
	26:24	**Media Command Opcode** Default Value: 0h MEDIA_INTERFACE_DESCRIPTOR_LOAD Format: OpCode
	23:16	**Media Command Opcode** Default Value: 2h MEDIA_INTERFACE_DESCRIPTOR_LOAD Format: OpCode
	15:0	**DWord Length** Default Value: 2h Excludes DWord (0,1) Format: =n Total Length - 2 Project: All
1	31:5	**Reserved** Project: All Format: MBZ
	4:0	**Reserved** Project: All Format: MBZ
2	31:17	**Reserved** Project: All Format: MBZ
	16:0	**Interface Descriptor Total Length** Project: All Format: U17 In bytes Range: 32..992 [DevSNB] This field provides the length in bytes of the Interface Descriptor data. This field must have the same alignment as the Interface Descriptor Data Start Address. It must be DQWord (32-byte) aligned. As the Interface Descriptor data are sent directly to ROB, range is limited to CURBE Allocation Size. Range = [32 – 992] → [1- 31] interface descriptor entries [DevSNB]

MEDIA_INTERFACE_DESCRIPTOR_LOAD		
3	31:0	**Interface Descriptor Data Start Address** Project: All Address: DynamicBaseAddress[31:0] Surface Type: INTERFACE_DESCRIPTOR_DATA Range: 0..4GB Specifies the 32-byte aligned address of the Interface Descriptor data. This pointer is relative to the **Dynamics Base Address**.

1.7.4 INTERFACE_DESCRIPTOR_DATA [DevSNB+]

Interface Descriptor Data payload as pointed by the Interface Descriptor Data Start Address:

INTERFACE_DESCRIPTOR_DATA		
Project:	[DevSNB+]	
Default Value:	00000000h	
Desc		
DWord	**Bit**	**Description**
0	31:6	**Kernel Start Pointer** Project: All Address: InstructionBaseAddress[31:6] Surface Type: Kernel Specifies the 64-byte aligned address offset of the first instruction in the kernel. This pointer is relative to the **Instruction Base Address**.
	5:0	**Reserved** Project: All Format: MBZ
1	31:26	**Reserved** Project: All Format: MBZ
	25:20	**Reserved** Project: All Format: MBZ
	19	**Reserved** Project: All Format: MBZ
	18	**Single Program Flow (SPF)** Project: All Specifies whether the kernel program has a single program flow (SIMDnxm with m = 1) or multiple program flows (SIMDnxm with m > 1).

Value	Name	Description	Project
0h	Multiple Program Flow		All
1h	Single Program Flow		All

<table>
<tr><td></td><td>17</td><td>Thread Priority
Project: All
Specifies the priority of the thread for dispatch
<table><tr><th>Value</th><th>Name</th><th>Description</th><th>Project</th></tr><tr><td>0h</td><td>Normal Priority</td><td></td><td>All</td></tr><tr><td>1h</td><td>High Priority</td><td></td><td>All</td></tr></table></td></tr>
<tr><td></td><td>16</td><td>Floating Point Mode
Project: All
Specifies the floating point mode used by the dispatched thread.
<table><tr><th>Value</th><th>Name</th><th>Description</th><th>Project</th></tr><tr><td>0h</td><td>Use IEEE-754 Rules</td><td></td><td>All</td></tr><tr><td>1h</td><td>Use alternate rules</td><td></td><td>All</td></tr></table></td></tr>
<tr><td></td><td>15:14</td><td>Reserved Project: All Format: MBZ</td></tr>
<tr><td></td><td>13</td><td>Illegal Opcode Exception Enable
Project: All
Format: Enable
This bit gets loaded into EU CR0.1[12] (note the bit # difference). See Exceptions and ISA Execution Environment.</td></tr>
<tr><td></td><td>12</td><td>Reserved Project: All Format: MBZ</td></tr>
<tr><td></td><td>11</td><td>MaskStack Exception Enable
Project: All
Format: Enable
This bit gets loaded into EU CR0.1[11]. See Exceptions and ISA Execution Environment.</td></tr>
<tr><td></td><td>10:8</td><td>Reserved Project: All Format: MBZ</td></tr>
<tr><td></td><td>7</td><td>Software Exception Enable
Project: All
Format: Enable
This bit gets loaded into EU CR0.1[13] (note the bit # difference). See Exceptions and ISA Execution Environment.</td></tr>
<tr><td></td><td>6:0</td><td>Reserved Project: All Format: MBZ</td></tr>
<tr><td>2</td><td>31:5</td><td>Sampler State Pointer
Project: All
Address: DynamicStateOffset[31:5]
Surface Type: SAMPLER_STATE
Specifies the 32-byte aligned address offset of the sampler state table. This pointer is relative to the Dynamic State Base Address.
This field is ignored for child threads.</td></tr>
</table>

<table>
<tr><td></td><td>4:2</td><td>Sampler Count
Project: All
Format: U3
Range: 0..4
Specifies how many samplers (in multiples of 4) the kernel uses. Used only for prefetching the associated sampler state entries.
This field is ignored for child threads.
If this field is not zero, sampler state is prefetched for the first instance of a root thread upon the startup of the media pipeline.

<table>
<tr><th>Value</th><th>Name</th><th>Description</th><th>Project</th></tr>
<tr><td>0h</td><td>No samplers used</td><td></td><td>All</td></tr>
<tr><td>1h</td><td>Between 1 and 4 samplers used</td><td></td><td>All</td></tr>
<tr><td>2h</td><td>Between 5 and 8 samplers used</td><td></td><td>All</td></tr>
<tr><td>3h</td><td>Between 9 and 12 samplers used</td><td></td><td>All</td></tr>
<tr><td>4h</td><td>Between 13 and 16 samplers used</td><td></td><td>All</td></tr>
</table></td></tr>
<tr><td></td><td>1:0</td><td>Reserved Project: All Format: MBZ</td></tr>
<tr><td>3</td><td>31:5</td><td>Binding Table Pointer
Project: All
Address: SurfaceStateOffset[31:5] – [DevSNB]

Surface Type: BINDING_TABLE
Specifies the 32-byte aligned address of the binding table. This pointer is relative to the Surface State Base Address.
This field is ignored for child threads.</td></tr>
<tr><td></td><td>4:0</td><td>Binding Table Entry Count
Project: All
Format: U5
Range: 0..31
Specifies how many binding table entries the kernel uses. Used only for prefetching of the binding table entries and associated surface state.
Note: The maximum number of prefetched binding table entries is limited to 31. For kernels using a large number of binding table entries, it may be wise to set this field to zero to avoid prefetching too many entries and thrashing the state cache.
This field is ignored for child threads.
If this field is not zero, binding table and surface state are prefetched for the first instance of a root thread upon the startup of the media pipeline.</td></tr>
</table>

INTERFACE_DESCRIPTOR_DATA		
4	31:16	**Constant URB Entry Read Length** Project: All Format: U6 Range: 0..63 Specifies the amount of URB data read and passed in the thread payload for the Constant URB entry, in 8-DW register increments. A value 0 means that no Constant URB Entry will be loaded. The Constant URB Entry Read Offset field will then be ignored.
	15:0	**Constant URB Entry Read Offset** Project: All Format: U6 Range 0..2015 Specifies the offset (in 8-DW units) at which Constant URB data is to be read from the URB before being included in the thread payload. Range = [0,2015] indicating [0,2015] 256-bit register increments. ROB has 64KB of storage; 2048 entries. However, lowest 32 entries are reserved for VFE/TS to store interface descriptor data. Hence, URB Entry Read Offset plus Read Length shall not exceed 2015.
5	31:4	**Reserved** Project: [DevSNB] Format: MBZ
	3:0	**Barrier ID** Project: [DevSNB] Format: U4 Range 0..15 Specifies the barrier id that is associated with this interface descriptor
6	31:8	**Reserved** Project: All Format: MBZ
	7:0	**Reserved**
7	31:0	**Reserved** Project: All Format: MBZ

1.7.5 MEDIA_GATEWAY_STATE Command [DevSNB]

MEDIA_GATEWAY_STATE			
Project:	[DevSNB]	**Length Bias:**	2
This command updates the Message Gateway state. In particular, it updates the state for a selected Barrier.			

DWord	Bit	Description
0	31:29	**Command Type** Default Value: 3h GFXPIPE Format: OpCode

MEDIA_GATEWAY_STATE		
	28:27	**Media Command Opcode** Default Value: 2h MEDIA_GATEWAY_STATE Format: OpCode
	26:24	**Media Command Opcode** Default Value: 0h MEDIA_GATEWAY_STATE Format: OpCode
	23:16	**Media Command Opcode** Default Value: 3h MEDIA_GATEWAY_STATE Format: OpCode
	15:0	**DWord Length** Default Value: 0h Excludes DWord (0,1) Format: =n Total Length - 2 Project: All
1	31:24	**Reserved** Project: All Format: MBZ
	23:16	**BarrierID** Project: All Format: U4 This field indicates which one from the 16 Barrier States is updated.
	15:8	**Barrier.Byte** Project: All Format: U8 This is the initial value to be delivered by the Message Gateway to the requester threads when the Barrier Thread Count is reached.
	7:0	**Barrier.ThreadCount** Project: All Format: U8 Range 1..MaxThreadCount Barrier Thread Count is the total number of threads in a group that share the barrier function. It cannot be greater than the maximum number of threads, MaxThreadCount (DevID), supported by the product. For example, MaxThreadCount (DevSNB) = 60.

1.7.6 MEDIA_STATE_FLUSH Command [DevSNB]

MEDIA_STATE_FLUSH			
Project:	[DevSNB]	**Length Bias:**	2

This command updates the Message Gateway state. In particular, it updates the state for a selected Barrier.
This command can be considered same as a MI_Flush except that only media parser will get flushed instead of the entire 3D/media render pipeline. The command should be programmed prior to new Media state, curbe and/or interface descriptor commands when switching to a new context or programming new state for the same context.

With this command, pipelined state change is allowed for the media pipe.

It should be cautious when using this command when child_present flag in the media state is enabled. This is because that CURBE state as well as Interface Descriptor state are shared between root threads and child threads. Changing these states while child threads are generated on the fly may cause unexpected behavior.

Combining with MI_ARB_ON/OFF command, it is possible to support interruptability with the following command sequence where interrupt may be allowed only when MI_ARB_ON_OFF is ON:

```
MEDIA_STATE_FLUSH
VFE_STATE          // VFE will hold CS if watermark isn't met
MI_ARB_OFF         // There must be at least one VFE command before this one
MEDIA_OBJECT
….
MI_ARB_ON
```

DWord	Bit	Description
0	31:29	**Command Type** Default Value: 3h GFXPIPE Format: OpCode
	28:27	**Media Command Opcode** Default Value: 2h MEDIA_STATE_FLUSH Format: OpCode
	26:24	**Media Command Opcode** Default Value: 0h MEDIA_STATE_FLUSH Format: OpCode
	23:16	**Media Command Opcode** Default Value: 4h MEDIA_STATE_FLUSH Format: OpCode
	15:0	**DWord Length** Default Value: 0h Excludes DWord (0,1) Format: =n Total Length - 2 Project: All

<table>
<tr><th colspan="3">MEDIA_STATE_FLUSH</th></tr>
<tr><td>1</td><td>31:24</td><td>Reserved Project: All Format: MBZ</td></tr>
<tr><td></td><td>23:16</td><td>ThreadCountWaterMark
Project: All
Format: U8 Zero is Valid
The state flush is subject to the number of spawn threads to be dispatched to be less than or equal to the Thread Count water Mark. If the number of unoccupied threads in the system is smaller than this number, this command will stall the media pipeline.
Usage example: in certain usage a predetermined set of threads must be dispatched all together as a work group and interrupt is only allowed between groups. For that, a media state flush with the watermark set such that media pipe doesn't proceed with the next group of threads until there are enough hardware thread slots available.</td></tr>
<tr><td></td><td>15:0</td><td>BarrierMask
Project: All
This is a bit mask, with the bit location mapping with BarrierID, indicating whether the state flush is subject to the corresponding Barrier. When Barrier is not used, all bits must be set to 0.
<table>
<tr><th>Value</th><th>Name</th><th>Description</th><th>Project</th></tr>
<tr><td>0h</td><td>Not Waiting</td><td>Not waiting for the corresponding Barrier state to clear</td><td>All</td></tr>
<tr><td>1h</td><td>Waiting</td><td>Waiting for the corresponding Barrier state to clear</td><td>All</td></tr>
</table></td></tr>
</table>

1.7.7 MEDIA_OBJECT Command [DevSNB+]

The MEDIA_OBJECT command is the basic media primitive command for the media pipeline. It supports loading of inline data as well as indirect data.

Dword	Bits	Description
0	31:29	**Command Type =** GFXPIPE = 3h
	28:16	**Media Command Opcode =** MEDIA_OBJECT Pipeline[28:27] = Media = 2h; Opcode[26:24] **=** 1h; Subopcode[23:16] = 0h
	15:0	**Dword Length** (Excludes DWords 0,1) **Generic Mode:** DWord Length = N+4, where N is in the range of [0,504]. The maximum is 504 DW (equivalent to 63 8-DW registers). When both inline and indirect data are fetched for this command, the total size in 8-DW registers must be less than 112 (with both inline data length N and indirect data length rounded up to 8-DW aligned individually). The minimal inline data length is 0.
1	31:8	**Reserved**
	7:6	**Reserved.** MBZ

Dword	Bits	Description
	5:0	**Interface Descriptor Offset.** This field specifies the offset from the interface descriptor base pointer to the interface descriptor which will be applied to this object. It is specified in units of interface descriptors. Format: U5 [DevSNB] Range = [0,30] [DevSNB only]
2	31	**Children Present.** Indicates that the root thread may send spawn messages to spawn child threads and/or synchronized root threads. If Children Present is not set, TS signals VFE to dereference the URB handle immediately after it receives acknowledgement from TD that the thread is dispatched. If Children Present is set, the URB handle is forwarded to the root thread and serves as the return URB handle for the root thread. TS does not signal deference at the time of dispatch. TS signals URB handle deference only when it receives a resource dereference message from the thread. ***In order avoid deadlock, such dereference must be issued once and only once for each URB handle.*** Format = Enable
	30:25	**Reserved.** MBZ
	24	**Thread Synchronization.** This field when set indicates that the dispatch of the thread originated from this command is based on the “spawn root thread” message. 0 = No thread synchronization 1 = Thread dispatch is synchronized by the “spawn root thread” message
	23:22	**Reserved.** MBZ
	21	**Use Scoreboard.** This field specifies whether the thread associated with this command uses hardware scoreboard. Only when this field is set, the scoreboard control fields in the VFE Dword are valid. If this field is cleared, the thread associated with this command bypasses hardware scoreboard. 0 = Not using scoreboard 1 = Using scoreboard
	20	**Reserved.** MBZ
	19	**Reserved**
	18:17	**Half-Slice Destination Select** This field selects the half slice that this thread must be sent to.

Value	Name	Description	Project
10	Half-Slice 1		
01	Half-Slice 0		
00	Either half-slice		

Dword	Bits	Description
	16:0	**Indirect Data Length.** This field provides the length in bytes of the indirect data. A value zero indicates that indirect data fetching is disabled – subsequently, the Indirect Data Start Address field is ignored. This field must have the same alignment as the Indirect Object Data Start Address. It must be DQWord (32-byte) aligned. As the indirect data are sent directly to URB, range is limited to 496 DW. When both inline and indirect data are fetched for this command, the total size in 8-DW registers must be less than 112 (with both inline data length and indirect data length rounded up to 8-DW aligned). Format = U17 in bytes
3	31:0	**Indirect Data Start Address.** This field specifies the Graphics Memory starting address of the data to be loaded into the kernel for processing. This pointer is relative to the **Indirect Object Base Address**. Hardware ignores this field if indirect data is not present. Alignment of this address depends on the mode of operation. It is the DWord aligned address of the indirect data. Range = [0 - 512MB] (Bits 31:29 MBZ)
4	31:25	**Reserved.** MBZ
	24:16	**Scoreboard Y** This field provides the Y term of the scoreboard value of the current thread. Format = U9
	15:9	**Reserved.** MBZ
	8:0	**Scoreboard X** This field provides the X term of the scoreboard value of the current thread. Format = U9
5	31:20	**Reserved.** MBZ
	19:16	**Scoreboard Color:** This field specifies which dependency color the current thread belongs to. It affects the dependency scoreboard control. Format = U4
	15:8	**Reserved.** MBZ
	7:0	**Scoreboard Mask:** Each bit indicates the corresponding dependency scoreboard is dependent on. This field is AND'd with the corresponding Scoreboard Mask field in the MEDIA_VFE_STATE command. **Bit n (for n = 0…7):** Scoreboard n is dependent, where bit 0 maps to n = 0. Format = TRUE/FALSE
6..N	31:0	**Inline Data** **Generic Mode:** The format of this data is specified by software. Hardware does not interpret this data; it merely passes it to the kernel for processing. The total size for the inline data and indirect data must not exceed 112 registers.

1.7.7.1 Inline Data Format in AVC-IT Mode

Each MEDIA_OBJECT_EX command in “AVC-IT mode” corresponds to the processing of one macroblock. Macroblock parameters are passed in as inline data and the non-zero DCT coefficient data (as well as motion vectors and weight/offset) for the macroblock is passed in as indirect data.

Error! Reference source not found. depicts the inline data format in AVC-IT mode. All fields in inline data are forwarded to the thread as thread payload, except the QP fields, where the derived macroblock information is filled in. Starting at GRF location, inline data are stored in GRF contiguously with the tail-end partial GRF, if present, zero-filled. Some fields are merely forwarded. Some fields are also used by VFE as indicated in the following table by a mark of [Used by VFE]. As shown, inline data starts at dword 4 of MEDIA_OBJECT_EX command.

Table 1-3 shows dwords 4+3, 4+4 and 4+5 of the inline data. Luma intra prediction mode (LumaIntraPredModes) is provided as a fixed-size data structure. Details can be found in Section 1.7.7.1.1.

Table 1-3. Inline data subfields for an Intra Macroblock in AVC-IT mode (and AVC-MC mode)

<table>
<tr><th>Dword</th><th>Bit</th><th>Description</th></tr>
<tr><td>4+3</td><td>31:8</td><td>Reserved</td></tr>
<tr><td></td><td>7:0</td><td>MbIntraStruct (Macroblock Intra Structure)
<table>
<tr><th>Bits</th><th>MotionVerticalFieldSelect Index</th></tr>
<tr><td>7:6</td><td>ChromaIntraPredMode</td></tr>
<tr><td>5</td><td>[DevILK] IntraPredAvailFlagF – F (Pixel [-1, 7] available for intra prediction) F = Is_Left_MB_Field & Is_Left_Bottom_MB_Intra</td></tr>
<tr><td>4</td><td>IntraPredAvailFlagE – E (left neighbor bottom half)</td></tr>
<tr><td>3</td><td>IntraPredAvailFlagD – D (Upper right neighbor)</td></tr>
<tr><td>2</td><td>IntraPredAvailFlagC – C (Upper left neighbor)</td></tr>
<tr><td>1</td><td>IntraPredAvailFlagB – B (Upper neighbor)</td></tr>
<tr><td>0</td><td>IntraPredAvailFlagA – A (Left neighbor top half)</td></tr>
</table></td></tr>
<tr><td>4+4</td><td>31:16</td><td>LumaIndraPredModes[1] (Luma Intra Prediction Modes)</td></tr>
<tr><td></td><td>15:0</td><td>LumaIndraPredModes[0]</td></tr>
<tr><td>4+5</td><td>31:16</td><td>LumaIndraPredModes[3]</td></tr>
<tr><td></td><td>15:0</td><td>LumaIndraPredModes[2]</td></tr>
</table>

Dwords 4+3, 4+4 and 4+5 of the inline data for an inter-predicted macroblock is detailed in Table 1-4.

Table 1-4. Inline data subfields for an Inter Macroblock in AVC-IT mode (and AVC-MC mode)

DWord	Bit	Description
4+3	31:24	**Log2WeightDenomChroma**

DWord	Bit	Description
	23:16	**Log2WeightDenomLuma**
	15:8	**SubMbPredMode (Sub Macroblock Prediction Mode)** This field describes the prediction mode of the sub macroblocks. It contains four subfields each with 2-bits, corresponding to the 4 fixed size 8x8 sub macroblocks in sequential order. Details can be found in Section 1.7.7.1.3. This field is derived from sub_mb_type for a BP_8x8 macroblock. This field is derived from **MbType** for a non-BP_8x8 inter macroblock, and carries redundant information as **MbType**) Bits [1:0]: SubMbPredMode[0] Bits [3:2]: SubMbPredMode[1] Bits [5:4]: SubMbPredMode[2] Bits [7:6]: SubMbPredMode[3]
	7:0	**SubMbShape (Sub Macroblock Shape)** This field describes the subdivision of the sub macroblocks. It contains four subfields each with 2-bits, corresponding to the 4 fixed size 8x8 sub macroblocks in sequential order. Details can be found in Section 1.7.7.1.3. This field is derived from sub_mb_type for a BP_8x8 macroblock. This field is forced to 0 for a non-BP_8x8 inter macroblock, and effectively carries redundant information as **MbType**). Bits [1:0]: SubMbShape[0] Bits [3:2]: SubMbShape[1] Bits [5:4]: SubMbShape[2] Bits [7:6]: SubMbShape[3]
4+4	31:24	**BindingTableIndexForward – Block 3**
	23:16	**BindingTableIndexForward – Block 2**
	15:8	**BindingTableIndexForward – Block 1**
	7:0	**BindingTableIndexForward – Block 0**
4+5	31:24	**BindingTableIndexBackward – Block 3**
	23:16	**BindingTableIndexBackward – Block 2**
	15:8	**BindingTableIndexBackward – Block 1**
	7:0	**BindingTableIndexBackward – Block 0**

1.7.7.1.1 Luma Intra Prediction Modes

Luma Intra Prediction Modes (LumaIntraPredModes) is defined in Table 1-5. It is further categorized as Intra16x16PredMode (Table 1-6), Intra8x8PredMode (Table 1-7) and Intra4x4PredMode (Table 1-8),

operating on 16x16, 8x8 and 4x4 block sizes, respectively. Figure 1-8 illustrates the intra prediction directions geometrically for the Intra4x4 prediction. When a macroblock is subdivided, the intra prediction is performed for the subdivision in a predetermined order. For example, Figure 1-9 shows the block order for Intra4x4 prediction. And Figure 1-10 shows the block order of Block8x8 in a 16x16 region or Block4x4 in a 8x8 region.

Table 1-5. Definition of LumaIntraPredModes

LumaIntraPredModes [index]		Intra16x16PredMode	Intra8x8PredMode	Intra4x4PredMode
Index	**Bit**	**MbType = [1…24] Transform8x8Flag = 0**	**MbType = 0 Transform8x8Flag = 1**	**MbType = 0 Transform8x8Flag = 0**
0	15:12	MBZ	**Block8x8 3**	**Block4x4 3 (0_0)**
	11:8	MBZ	**Block8x8 2**	**Block4x4 2 (0_1)**
	7:4	MBZ	**Block8x8 1**	**Block4x4 1 (0_2)**
	3:0	**Block16x16**	**Block8x8 0**	**Block4x4 0 (0_3)**
1	15:12	MBZ	MBZ	**Block4x4 7 (1_0)**
	11:8	MBZ	MBZ	**Block4x4 6 (1_1)**
	7:4	MBZ	MBZ	**Block4x4 5 (1_2)**
	3:0	MBZ	MBZ	**Block4x4 4 (1_3)**
2	15:12	MBZ	MBZ	**Block4x4 11 (2_0)**
	11:8	MBZ	MBZ	**Block4x4 10 (2_1)**
	7:4	MBZ	MBZ	**Block4x4 9 (2 2)**
	3:0	MBZ	MBZ	**Block4x4 8 (2_3)**
3	15:12	MBZ	MBZ	**Block4x4 15 (3_0)**
	11:8	MBZ	MBZ	**Block4x4 14 (3_1)**
	7:4	MBZ	MBZ	**Block4x4 13 (3_2)**
	3:0	MBZ	MBZ	**Block4x4 12 (3_3)**

Table 1-6. Definition of Intra16x16PredMode

Intra16x16PredMode	Description
0	**Intra_16x16_Vertical**
1	**Intra_16x16_Horizontal**
2	**Intra_16x16_DC**
3	**Intra_16x16_Plane**
4 – 15	Reserved

Table 1-7. Definition of Intra8x8PredMode

Intra8x8PredMode	Description
0	**Intra_8x8_Vertical**
1	**Intra_8x8_Horizontal**
2	**Intra_8x8_DC**
3	**Intra_8x8_Diagonal_Down_Left**
4	**Intra_8x8_Diagonal_Down_Right**
5	**Intra_8x8_Vertical_Right**
6	**Intra_8x8_Horizontal_Down**
7	**Intra_8x8_Vertical_Left**
8	**Intra_8x8_Horizontal_Up**
9 – 15	Reserved

Table 1-8. Definition of Intra4x4PredMode

Intra4x4PredMode	Description
0	**Intra_4x4_Vertical**
1	**Intra_4x4_Horizontal**
2	**Intra_4x4_DC**

Intra4x4PredMode	Description
3	**Intra_4x4_Diagonal_Down_Left**
4	**Intra_4x4_Diagonal_Down_Right**
5	**Intra_4x4_Vertical_Right**
6	**Intra_4x4_Horizontal_Down**
7	**Intra_4x4_Vertical_Left**
8	**Intra_4x4_Horizontal_Up**
9 – 15	Reserved

Figure 1-8. Intra_4x4 prediction mode directions

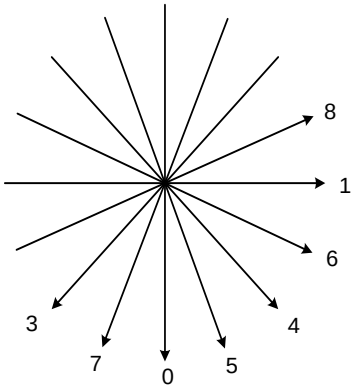


Figure 1-9. Numbers of Block4x4 in a 16x16 region

0	1	4	5
2	3	6	7
8	9	12	13
10	11	14	15

Figure 1-10. Numbers of Block4x4 in an 8x8 region or numbers of Block8x8 in a 16x16 region

0	1
2	3

1.7.7.1.2 Macroblock Type

Macroblock Type, MbType, is defined as a unified parameter for all slice types (I, P or B slices) as shown in Table 1-9. Furthermore, MbType has the same meaning for a P macroblock and a B macroblock. For example, BP_L0_16x16 can be viewed as a P_L0_16x16 macroblock in a P slice or a B_L0_16x16 macroblock in a B slice.

As shown in Table 1-10, Macroblock Type (MbType) is derived from *mb_type*, as defined in AVC spec, for an I-, P- or B-slice.

Table 1-9. Definition of MbType

MbType	For an Intra Macroblock (IntraMbFlag = 1)	For an Inter Macroblock (IntraMbFlag = 1)
0	I_NxN	Reserved
1	I_16x16_0_0_0	**BP**_L0_16x16
2	I_16x16_1_0_0	B_L1_16x16
3	I_16x16_2_0_0	B_Bi_16x16
4	I_16x16_3_0_0	**BP**_L0_L0_16x8
5	I_16x16_0_1_0	**BP**_L0_L0_8x16
6	I_16x16_1_1_0	B_L1_L1_16x8
7	I_16x16_2_1_0	B_L1_L1_8x16
8	I_16x16_3_1_0	B_L0_L1_16x8
9	I_16x16_0_2_0	B_L0_L1_8x16
10	I_16x16_1_2_0	B_L1_L0_16x8
11	I_16x16_2_2_0	B_L1_L0_8x16
12	I_16x16_3_2_0	B_L0_Bi_16x8
13	I_16x16_0_0_1	B_L0_Bi_8x16
14	I_16x16_1_0_1	B_L1_Bi_16x8
15	I_16x16_2_0_1	B_L1_Bi_8x16
16	I_16x16_3_0_1	B_Bi_L0_16x8
17	I_16x16_0_1_1	B_Bi_L0_8x16
18	I_16x16_1_1_1	B_Bi_L1_16x8
19	I_16x16_2_1_1	B_Bi_L1_8x16
20	I_16x16_3_1_1	B_Bi_Bi_16x8

MbType	For an Intra Macroblock (IntraMbFlag = 1)	For an Inter Macroblock (IntraMbFlag = 1)
21	I_16x16_0_2_1	B_Bi_Bi_8x16
22	I_16x16_1_2_1	**BP**_8x8
23	I_16x16_2_2_1	Reserved
24	I_16x16_3_2_1	Reserved
25	I_PCM	Reserved
26	Reserved (for SI)	Reserved
27-63	Reserved	Reserved

Table 1-10. Deriving MbType from mb_type for I, P and B slices

mb_type	I Slice		P Slice		B Slice	
	MbType	Description	MbType	Description	MbType	Description
0	0	I_NxN	1	BP_L0_16x16	22	B_Direct_16x16 mapped to BP_8x8
1	1	I_16x16_0_0_0	**4**	BP_L0_L0_16x8	1	BP_L0_16x16
2	2	I_16x16_1_0_0	**5**	BP_L0_L0_8x16	2	B_L1_16x16
3	3	I_16x16_2_0_0	**22**	BP_8x8	3	B_Bi_16x16
4	4	I_16x16_3_0_0	**22**	BP_8x8	4	BP_L0_L0_16x8
5	5	I_16x16_0_1_0	0	I_NxN	5	BP_L0_L0_8x16
6	6	I_16x16_1_1_0	1	I_16x16_0_0_0	6	B_L1_L1_16x8
7	7	I_16x16_2_1_0	2	I_16x16_1_0_0	7	B_L1_L1_8x16
8	8	I_16x16_3_1_0	3	I_16x16_2_0_0	8	B_L0_L1_16x8
9	9	I_16x16_0_2_0	4	I_16x16_3_0_0	9	B_L0_L1_8x16
10	10	I_16x16_1_2_0	5	I_16x16_0_1_0	10	B_L1_L0_16x8
11	11	I_16x16_2_2_0	6	I_16x16_1_1_0	11	B_L1_L0_8x16
12	12	I_16x16_3_2_0	7	I_16x16_2_1_0	12	B_L0_Bi_16x8

mb_type	I Slice		P Slice		B Slice	
	MbType	Description	MbType	Description	MbType	Description
13	13	I_16x16_0_0_1	8	I_16x16_3_1_0	13	B_L0_Bi_8x16
14	14	I_16x16_1_0_1	9	I_16x16_0_2_0	14	B_L1_Bi_16x8
15	15	I_16x16_2_0_1	10	I_16x16_2_0_1	15	B_L1_Bi_8x16
16	16	I_16x16_3_0_1	11	I_16x16_2_2_0	16	B_Bi_L0_16x8
17	17	I_16x16_0_1_1	12	I_16x16_3_2_0	17	B_Bi_L0_8x16
18	18	I_16x16_1_1_1	13	I_16x16_0_0_1	18	B_Bi_L1_16x8
19	19	I_16x16_2_1_1	14	I_16x16_1_0_1	19	B_Bi_L1_8x16
20	20	I_16x16_3_1_1	15	I_16x16_2_0_1	20	B_Bi_Bi_16x8
21	21	I_16x16_0_2_1	16	I_16x16_3_0_1	21	B_Bi_Bi_8x16
22	22	I_16x16_1_2_1	17	I_16x16_0_1_1	22	BP_8x8
23	23	I_16x16_2_2_1	18	I_16x16_1_1_1	0	I_NxN
24	24	I_16x16_3_2_1	19	I_16x16_2_1_1	1	I_16x16_0_0_0
25	25	I_PCM	20	I_16x16_3_1_1	2	I_16x16_1_0_0
26	n/a	n/a	21	I_16x16_0_2_1	3	I_16x16_2_0_0
27			22	I_16x16_1_2_1	4	I_16x16_3_0_0
28			23	I_16x16_2_2_1	5	I_16x16_0_1_0
29			24	I_16x16_3_2_1	6	I_16x16_1_1_0
30			25	I_PCM	7	I_16x16_2_1_0
31			n/a	n/a	8	I_16x16_3_1_0
32					9	I_16x16_0_2_0
33					10	I_16x16_2_0_1
34					11	I_16x16_2_2_0
35					12	I_16x16_3_2_0

mb_type	I Slice		P Slice		B Slice	
	MbType	Description	MbType	Description	MbType	Description
36					13	I_16x16_0_0_1
37					14	I_16x16_1_0_1
38					15	I_16x16_2_0_1
39					16	I_16x16_3_0_1
40					17	I_16x16_0_1_1
41					18	I_16x16_1_1_1
42					19	I_16x16_2_1_1
43					20	I_16x16_3_1_1
44					21	I_16x16_0_2_1
45					22	I_16x16_1_2_1
46					23	I_16x16_2_2_1
47					24	I_16x16_3_2_1
48					25	I_PCM
49-63					n/a	n/a

1.7.7.1.3 Sub Macroblock Shape and Sub Macroblock Prediction Mode

Sub Macroblock Shape, SubMbShape, describes the shape of the sub divisions of an 8x8 sub macroblock of a BP_8x8 macroblock. Sub Macroblock Prediction Mode, SubMbPredMode, indicates the prediction mode for the sub macroblock. They are defined in Table 1-11 and Table 1-12. Both of these parameters can be derived from sub_mb_type field as defined in AVC spec according to Table 1-14 and Table 1-14.

For a non-BP_8x8 inter macroblock (IntraMbFlag = 0), the sub macroblocks will be greater than and equal to 8x8. Both SubMbShape and SubMbPredMode must be filled to match with the MbType. In particular, SubMbShape is 0 and SubMbPredMode is determined based on MbType according to Table 1-15.

Table 1-11. Definition of SubMbShape for an 8x8 region of a BP_8x8 macroblock

SubMbShape	NumSubMbPart	SubMbPartWidth	SubMbPartHeight
0	1	8	8

SubMbShape	NumSubMbPart	SubMbPartWidth	SubMbPartHeight
1	2	8	4
2	2	4	8
3	4	4	4

Table 1-12. Definition of SubMbPredMode for an 8x8 region of a BP_8x8 macroblock

SubMbPredMode	Description	Comments
0	Pred_L0	P_8x8 and B_8x8
1	Pred_L1	B_8x8 only
2	BiPred	B_8x8 only
3	Reserved	

Table 1-13. Mapping sub_mb_type to SubMbType and SubMbPredMode in P macroblocks (BP_8x8)

sub_mb_type [i]	name	SubMb Prediction	SubMbPartWidth	SubMbPartHeight	SubMbShape [i]	SubMbPredMode [i]
0	P_L0_8x8	Pred_L0	8	8	**0**	**0**
1	P_L0_8x4	Pred_L0	8	4	**1**	**0**
2	P_L0_4x8	Pred_L0	4	8	**2**	**0**
3	P_L0_4x4	Pred_L0	4	4	**3**	**0**
Inferred	n/a	n/a	n/a	n/a	n/a	n/a

Table 1-14. Mapping sub_mb_type to SubMbType and SubMbPredModd in B macroblocks (BP_8x8)

sub_mb_type [i]	name	SubMb Prediction	SubMbPartWidth	SubMbPartHeight	SubMbShape [i]	SubMbPredMode [i]
0	B_Direct_8x8	Direct	4	4	**3**	
1	B_L0_8x8	Pred_L0	8	8	**0**	**0**

sub_mb_type [i]	name	SubMb Prediction	SubMbPartWidth	SubMbPartHeight	SubMbShape [i]	SubMbPredMode [i]
2	B_L1_8x8	Pred_L1	8	8	**0**	**1**
3	B_Bi_8x8	BiPred	8	8	**0**	**2**
4	B_L0_8x4	Pred_L0	8	4	**1**	**0**
5	B_L0_4x8	Pred_L0	4	8	**2**	**0**
6	B_L1_8x4	Pred_L1	8	4	**1**	**1**
7	B_L1_4x8	Pred_L1	4	8	**2**	**1**
8	B_Bi_8x4	BiPred	8	4	**1**	**2**
9	B_Bi_4x8	BiPred	4	8	**2**	**2**
10	B_L0_4x4	Pred_L0	4	4	**3**	**0**
11	B_L1_4x4	Pred_L1	4	4	**3**	**1**
12	B_Bi_4x4	BiPred	4	4	**3**	**2**
inferred	mb_type	Direct	4	4	**3**	

Table 1-15. SubMbPredMode[] for non BP_8x8 macroblocks (when IntraMbFlag = 0)

		SubMbPredMode[i]			
MbType	Name	i = 0	i = 1	i = 2	i = 3
0	Reserved	Reserved	Reserved	Reserved	Reserved
1	**BP**_L0_16x16	0	0	0	0
2	B_L1_16x16	1	1	1	1
3	B_Bi_16x16	2	2	2	2
4	**BP**_L0_L0_16x8	0	0	0	0
5	**BP**_L0_L0_8x16	0	0	0	0
6	B_L1_L1_16x8	1	1	1	1

		SubMbPredMode[i]			
MbType	Name	i = 0	i = 1	i = 2	i = 3
7	B_L1_L1_8x16	1	1	1	1
8	B_L0_L1_16x8	0	0	1	1
9	B_L0_L1_8x16	0	1	0	1
10	B_L1_L0_16x8	1	1	0	0
11	B_L1_L0_8x16	1	0	1	0
12	B_L0_Bi_16x8	0	0	2	2
13	B_L0_Bi_8x16	0	2	0	2
14	B_L1_Bi_16x8	1	1	2	2
15	B_L1_Bi_8x16	1	2	1	2
16	B_Bi_L0_16x8	2	2	0	0
17	B_Bi_L0_8x16	2	0	2	0
18	B_Bi_L1_16x8	2	2	1	1
19	B_Bi_L1_8x16	2	1	2	1
20	B_Bi_Bi_16x8	2	2	2	2
21	B_Bi_Bi_8x16	2	2	2	2

1.7.7.1.4 Motion Vector Size

In AVC, a macroblock may have 0 or 32 motion vectors and many other combinations in between. In order to simplify the AVC-IT interface, the motion vectors of a macroblock are regrouped. As shown in Table 1-16, only 5 distinct combined motion vector states (cMvState) B0, B1, B2, P3 and B3, are derived, corresponding the MvSize of 0, 2, 8, 16, and 32, respectively.

The maximum value of MvSize depends on the profile and level of the input AVC data. According to AVC Spec Table A-4 in section A.3.3.2, for Main and High Profiles at Level greater than 3.0, MinLumaBiPreSize is set to 8x8 (i.e. sub_mb_type in B macroblocks shall not be equal to B_Bi_8x4, B_Bi_4x8, or B_Bi_4x4). Therefore, B3 state is not valid for the given profile and level.

Programming Notes: *Programmers may (and should) take advantage of such profile and level restriction to conserve memory foot print for indirect data, memory bandwidth for delivering data as well as possibly the GRF register space storing motion vectors. For example, when the maximum possible MvSize is 16, only 16 dwords need to be allocated for motion vectors in both indirect data buffer and GRF space.*

Table 1-16. Motion vector regroup

Mblk Type	MV State	Max # MVs	Reference Lists	Combined MV State (cMvState)	Comments
P	P0	0	n/a	B0	Merged with B0
P	P1	1	L0	B1	Merged with B1
P	P2	4	L0	B2	Merged with B2
P	P3	**16**	L0	**P3**	Sub-macroblock partition smaller than 8x8
B	B0	**0**	n/a	**B0**	
B	B1	**2**	L0, L1, or Bi	**B1**	
B	B2	**8**	L0, L1, or Bi	**B2**	Sub-macroblock partition down to 8x8
B	B3	**32**	L0, L1, or Bi	**B3**	For a High Profile AVC data, only encountered with level <= 3.1

cMvState can be derived based on the following macroblock parameters: MbType, SubMbShape, and SubMbPredMode. Table 1-17 provides the detailed mapping.

Table 1-17. Regrouped motion vector states for an Inter Macroblock

MbType	Inter Macroblock Type	Max (sub_mb_type[])	Max (SubMBPredMode[])	Extact MV #	MV State	MvSize	Commends
1	BP_L0_16x16	n/a	n/a	1	B1	2MV	
2	B_L1_16x16	n/a	n/a	1	B1	2MV	
3	B_Bi_16x16	n/a	n/a	2	B1	2MV	
4	BP_L0_L0_16x8	n/a	n/a	2	B2	8MV	
5	BP_L0_L0_8x16	n/a	n/a	2	B2	8MV	
6	B_L1_L1_16x8	n/a	n/a	2	B2	8MV	

MbType	Inter Macroblock Type	Max (sub_mb_type[])	Max (SubMBPredMode[])	Extact MV #	MV State	MvSize	Commends
7	B_L1_L1_8x16	n/a	n/a	2	B2	8MV	
8	B_L0_L1_16x8	n/a	n/a	2	B2	8MV	
9	B_L0_L1_8x16	n/a	n/a	2	B2	8MV	
10	B_L1_L0_16x8	n/a	n/a	2	B2	8MV	
11	B_L1_L0_8x16	n/a	n/a	2	B2	8MV	
12	B_L0_Bi_16x8	n/a	n/a	3	B2	8MV	
13	B_L0_Bi_8x16	n/a	n/a	3	B2	8MV	
14	B_L1_Bi_16x8	n/a	n/a	3	B2	8MV	
15	B_L1_Bi_8x16	n/a	n/a	3	B2	8MV	
16	B_Bi_L0_16x8	n/a	n/a	3	B2	8MV	
17	B_Bi_L0_8x16	n/a	n/a	3	B2	8MV	
18	B_Bi_L1_16x8	n/a	n/a	3	B2	8MV	
19	B_Bi_L1_8x16	n/a	n/a	3	B2	8MV	
20	B_Bi_Bi_16x8	n/a	n/a	4	B2	8MV	
21	B_Bi_Bi_8x16	n/a	n/a	4	B2	8MV	
22	**BP**_8x8	0	1	4	B2	8MV	Without sub-partition, no BiPred
22	**BP**_8x8	0	2	5,6,7,8	B2	8MV	Without sub-partition, with BiPred
22	**BP**_8x8	> 0	1	5-16	P3	16MV	With sub-partition, no BiPred
22	**BP**_8x8	> 0	2	6-32	B3	32MV	With sub-partition, with BiPred

1.7.7.1.5 Binding Table Index Data in AVC-IT Mode

There are always 8 binding table indices transferred in the inline data for an Inter Macroblock, a forward and backward index for each 8x8 block in the macroblock. This data is derived from the reference index sent with each motion vector; since between 0 and 32 motion vectors can be sent, a mapping scheme is specified here to indicate which reference index is to be used for which block in the inline data.

The general scheme is that whenever the motion vectors are for partitions smaller than 8x8 then pick the upper right, since all binding table indices are guarenteed to be the same for all sub-blocks in an 8x8. If the motion vectors are for partitions larger than 8x8, then replicate the single binding table index for all 8x8s in the partition. If there is only a forward or backward motion vector specified, then replicate the binding table indices for the missing direction.

MbType	Inter Macroblock	Binding Table Replication Rule
1	**BP**_L0_16x16	L0 binding table index replicated to all 4 forward and all 4 backward
2	B_L1_16x16	L1 binding table index replicated to all 4 forward and all 4 backward
3	B_Bi_16x16	L0 replicated to all 4 forward, L1 replicated to all 4 backward
4	**BP**_L0_L0_16x8	First L0 (top) replicated to blocks 0 & 1, both forward and backward, 2nd L0 replicated to blocks 2 & 3, both forward and backward.
5	**BP**_L0_L0_8x16	First L0 (left) replicated to blocks 0 & 2, both forward and backward, 2nd L0 replicated to blocks 1 & 3, both forward and backward.
6	B_L1_L1_16x8	First L1 (top) replicated to blocks 0 & 1, both forward and backward, 2nd L1 replicated to blocks 2 & 3, both forward and backward.
7	B_L1_L1_8x16	First L1 (left) replicated to blocks 0 & 2, both forward and backward, 2nd L1 replicated to blocks 1 & 3, both forward and backward.
8	B_L0_L1_16x8	First L0 (top) replicated to blocks 0 & 1, both forward and backward, 2nd L1 replicated to blocks 2 & 3, both forward and backward.
9	B_L0_L1_8x16	First L0 (left) replicated to blocks 0 & 2, both forward and backward, 2nd L1 replicated to blocks 1 & 3, both forward and backward.
10	B_L1_L0_16x8	First L1 (top) replicated to blocks 0 & 1, both forward and backward, 2nd L0 replicated to blocks 2 & 3, both forward and backward.
11	B_L1_L0_8x16	First L1 (left) replicated to blocks 0 & 2, both forward and backward, 2nd L0 replicated to blocks 1 & 3, both forward and backward.
12	B_L0_Bi_16x8	First L0 (top) replicated to blocks 0 & 1, both forward and backward, 2nd L0 replicated to blocks forward 2 & 3, 2nd L1 to backward blocks 2 & 3
13	B_L0_Bi_8x16	First L0 (left) replicated to blocks 0 & 2, both forward and backward, 2nd L0 replicated to blocks forward 1 & 3, 2nd L1 to backward blocks 2 & 3
14	B_L1_Bi_16x8	First L1 (top) replicated to blocks 0 & 1, both forward and backward, 2nd L0 replicated to blocks forward 2 & 3, 2nd L1 to backward blocks 2 & 3
15	B_L1_Bi_8x16	First L1 (left) replicated to blocks 0 & 2, both forward and backward, 2nd L0 replicated to blocks forward 1 & 3, 2nd L1 to backward blocks 2 & 3
16	B_Bi_L0_16x8	First L0 replicated to blocks forward 0 & 1, 1st L1 to backward blocks 0 & 1, 2nd L0 replicated to blocks 2 & 3, both forward and backward
17	B_Bi_L0_8x16	First L0 replicated to blocks forward 0 & 2, 1st L1 to backward blocks 0 & 2, 2nd L0 replicated to blocks 1 & 3, both forward and backward

MbType	Inter Macroblock	Binding Table Replication Rule
18	B_Bi_L1_16x8	First L0 replicated to blocks forward 0 & 1, 1st L1 to backward blocks 0 & 1, 2nd L1 replicated to blocks 2 & 3, both forward and backward
19	B_Bi_L1_8x16	First L0 replicated to blocks forward 0 & 2, 1st L1 to backward blocks 0 & 2, 2nd L1 replicated to blocks 1 & 3, both forward and backward
20	B_Bi_Bi_16x8	First L0 replicated to forward blocks 0 & 1, 1st L1 to backward blocks 0 & 1, 2nd L0 replicated to forward blocks 2 & 3, 2nd L1 to backward blocks 2 & 3
21	B_Bi_Bi_8x16	First L0 replicated to forward blocks 0 & 2, 1st L1 to backward blocks 0 & 2, 2nd L0 replicated to forward blocks 2 & 3, 2nd L1 to backward blocks 2 & 3
22	**BP**_8x8	1) If the MvSize is 8, then the Binding Table Indices can be directly derived from the reference indices in the 8 motion vectors. 2) If MvSize is 16, then the macroblock is being split into 4x4 sub-blocks and biprediction is off (only 1 motion vector per 4x4). In this case, each 4x4 can either be forward or backward predicted, but the table reference for each set of 4 in an 8x8 is the same. Each of the 4 motion vectors in an 8x8 needs to be looked at – if one of them is forward predicted then the associated table reference can be used for that 8x8 block, and if one is backward predicted then that can be used for the backward reference for the 8x8. If all 4 motion vectors are forward, then the backward reference is not used and the forward table reference can be used as the default. 3) If MvSize is 32, then BiPred for the 4x4 sub-blocks. In this case between 4 and 8 motion vectors are sent per 8x8 block depending on whether the prediction is Bi or forward or backward. These motion vectors have to be searched in a simular way to the MvSize=16 case to find both the forward and backward reference or to replicate the existing reference if one of them is missing entirely.

1.7.7.2 Indirect Data Format in AVC-IT Mode

Indirect data in AVC-IT mode consist of Motion Vectors, Weight/Offset and Transform-domain Residue (Coefficient). All three data blocks have variable size. Sizes of Motion Vector block and the Weight-Offset block are determined by the MvSize value as shown in Table 1-18. Weight-Offset block, if present, is always packed behind the Motion Vector block. Coefficient data block can be either packed behind the Weight-Offset block or start at a predetermined offset, controlled by the fields in VFE_STATE_EX.

When coefficient data block is packed behind, it starts at the next 8-dword aligned offset from the indirect object data address. This 8-dword alignment doesn't leave any gap between the coefficient data block from the motion vector data block and weight-offset data block with one exception. When MvSize = 2 and weight-offset is not present, there is a 4-dword gap. Hardware ignores the value in the gap.

Table 1-18. Indirect subfield size in AVC-IT mode (and AVC-MC mode)

MvSize	MV		Weight/Offset		Examples
	Count	DW	Count	DW	
0	0	0	0	0	Intra macroblock in a picture containing P and/or B slices
2	2	4	1	4	P or B macroblocks with 16x16 sub macroblock
8	8	8	4	16	P or B macroblocks with minimal sub macroblock at 8x8

MvSize	MV		Weight/Offset		Examples
	Count	DW	Count	DW	
16	16	16	4	16	P macroblocks with minimal sub macroblock at less than 8x8
32	32	32	4	16	B macroblocks with minimal sub macroblock at less than 8x8

1.7.7.2.1 Motion Vector Block of Indirect Data in AVC-IT and AVC-MC Modes

Motion Vector block contains motion vectors in an intermediate format that is partially expanded according to the smallest subdivisions within an inter-predicted macroblock. During the expansion (done by AVC BSD engine or done by host software), a place that does not contain a motion vector is filled by replicating the most relevant motion vector according to the following motion vector replication rules. The intent of such motion vector replication is to allow a simpler kernel programming with fewer conditions to check. This would likely reduce the kernel footprint; however, it may or may not achieve better performance.

Motion Vector Replication Rules:

- Rule #1
    - #1.1: For L0 MV, for any partition or subpartition where there is at least one motion vector
        - If L0 inter prediction exists, the corresponding L0 MV is used
        - Else if L1 inter prediction exits (of the same block), set to the same as L1 MV
        - (Note that there is no ‘else’ here. If the partition or subpartition doesnot contain a motion vector, it will be filled according to the following replication rules)
    - #1.2: For L1 MV, for any partition or subpartition where there is at least one motion vector
        - If L1 inter prediction exists, the corresponding L1 MV is used
        - Else if L0 inter prediction exits (of the same block), set to the same as L0 MV
        - (Note that there is no ‘else’ here. If the partition or subpartition doesnot contain a motion vector, it will be filled according to the following replication rules)
- For a 16x16 partitioned macroblocked, MvSize = 2. The two MV fields follow Rule #1.
- For a macroblock with partition down to 8x8, MvSize = 8. The eight MV fields follow Rule #1.
    - For an 8x16 partition, each 8x16 is broken down into 2 8x8 stacking vertically. The 8x16 MVs (after rule #1) are replicated into both 8x8 blocks.
    - For an 16x8 partition, each 16x8 is broken down into 2 8x8 stacking horizontally. The 16x8 MVs (after rule #1) are replicated into both 8x8 blocks.
    - For an 8x8 partition, each 8x8 has its own MVs (after rule #1).
- For P macroblock with subpartition below 8x8, MvSize = 16,

- For an 8x8 partition, the 8x8 L0 MV is replicated into all the four 4x4 blocks.
- For an 4x8 subpartition within an 8x8 partition, each 4x8 is broken down into 2 4x4 stacking vertically. The 4x8 L0 MV is replicated into both 4x4 blocks.
- For an 8x4 subpartition within an 8x8 partition, each 8x4 is broken down into 2 4x4 stacking horizontally. The 8x4 MV is replicated into both 4x4 blocks.
- For a 4x4 subpartition within an 8x8 partition, each 4x4 has its own L0 MV.

- For B macroblock with subpartition below 8x8, MvSize = 32,
  - For an 8x8 partition, the 8x8 MVs (after rule #1) is replicated into all the four 4x4 blocks.
  - For an 4x8 subpartition within an 8x8 partition, each 4x8 is broken down into 2 4x4 stacking vertically. The 4x8 MVs (after rule #1) are replicated into both 4x4 blocks.
  - For an 8x4 subpartition within an 8x8 partition, each 8x4 is broken down into 2 4x4 stacking horizontally. The 8x4 MVs (after rule #1) are replicated into both 4x4 blocks.
  - For a 4x4 subpartition within an 8x8 partition, each 4x4 has its own MVs (after rule #1).

Table 1-19. Indirect data Motion Vector block in AVC-IT mode (and AVC-MC mode)

DWord	Bit	MvSize				
		0	2	8	16	32
0	31:16	n/a	MVVert_L0	MVVert_Y0_L0	MVVert_Y0_L0	MVVert_Y0_L0
	15:0	n/a	MVHorz_L0	MVHorz_Y0_L0	MVHorz_Y0_L0	MVHorz_Y0_L0
1	31:16	n/a	MVVert_L1	MVVert_Y0_L1	MVVert_Y1_L0	MVVert_Y0_L1
	15:0	n/a	MVHorz_L1	MVHorz_Y0_L1	MVHorz_Y1_L0	MVHorz_Y0_L1
2	31:0	n/a	Reserved: MBZ	MV_Y1_L0	MV_Y2_L0	MV_Y1_L0
3	31:0	n/a	Reserved: MBZ	MV_Y1_L1	MV_Y3_L0	MV_Y1_L1
4	31:0	n/a	n/a	MV_Y2_L0	MV_Y4_L0	MV_Y2_L0
5	31:0	n/a	n/a	MV_Y2_L1	MV_Y5_L0	MV_Y2_L1
6	31:0	n/a	n/a	MV_Y3_L0	MV_Y6_L0	MV_Y3_L0
7	31:0	n/a	n/a	MV_Y3_L1	MV_Y7_L0	MV_Y3_L1
8	31:0	n/a	n/a	n/a	MV_Y8_L0	MV_Y4_L0
9	31:0	n/a	n/a	n/a	MV_Y9_L0	MV_Y4_L1

DWord	Bit	MvSize				
		0	2	8	16	32
10	31:0	n/a	n/a	n/a	MV_Y10_L0	MV_Y5_L0
11	31:0	n/a	n/a	n/a	MV_Y11_L0	MV_Y5_L1
12	31:0	n/a	n/a	n/a	MV_Y12_L0	MV_Y6_L0
13	31:0	n/a	n/a	n/a	MV_Y13_L0	MV_Y6_L1
14	31:0	n/a	n/a	n/a	MV_Y14_L0	MV_Y7_L0
15	31:0	n/a	n/a	n/a	MV_Y15_L0	MV_Y7_L1
16	31:0	n/a	n/a	n/a	n/a	MV_Y8_L0
17	31:0	n/a	n/a	n/a	n/a	MV_Y8_L1
18	31:0	n/a	n/a	n/a	n/a	MV_Y9_L0
19	31:0	n/a	n/a	n/a	n/a	MV_Y9_L1
20	31:0	n/a	n/a	n/a	n/a	MV_Y10_L0
21	31:0	n/a	n/a	n/a	n/a	MV_Y10_L1
22	31:0	n/a	n/a	n/a	n/a	MV_Y11_L0
23	31:0	n/a	n/a	n/a	n/a	MV_Y11_L1
24	31:0	n/a	n/a	n/a	n/a	MV_Y12_L0
25	31:0	n/a	n/a	n/a	n/a	MV_Y12_L1
26	31:0	n/a	n/a	n/a	n/a	MV_Y13_L0
27	31:0	n/a	n/a	n/a	n/a	MV_Y13_L1
28	31:0	n/a	n/a	n/a	n/a	MV_Y14_L0
29	31:0	n/a	n/a	n/a	n/a	MV_Y14_L1
30	31:0	n/a	n/a	n/a	n/a	MV_Y15_L0
31	31:0	n/a	n/a	n/a	n/a	MV_Y15_L1

1.7.7.2.2 Weight-Offset Block of Indirect Data in AVC-IT and AVC_MC Modes for WeightedBiPredFlag ≠ 10

Table 1-20. Indirect data Weight-Offset block in AVC-IT mode (and AVC-MC mode)

DWord	Bit	MvSize		
		0	2	8, 16, 32
0	31:24	**n/a**	**Offset_Y_L1**	**Offset_Y_Block0_L1**
	23:16	**n/a**	**Weight_Y_L1**	**Weight_Y_Block0_L1**
	15:8	**n/a**	**Offset_Y_L0**	**Offset_Y_Block0_L0**
	7:0	**n/a**	**Weight_Y_L0**	**Weight_Y_Block0_L0**
1	31:16	**n/a**	**WO_Cb_L1**	**WO_Cb_Block0_L1**
	15:0	**n/a**	**WO_Cb_L0**	**WO_Cb_Block0_L0**
2	31:16	**n/a**	**WO_Cr_L1**	**WO_Cr_Block0_L1**
	15:0	**n/a**	**WO_Cr_L0**	**WO_Cr_Block0_L0**
3	31:4	**n/a**	Reserved: MBZ	Reserved: MBZ
	3:0		[**DevILK**] Weight is 128 [ChromaL1, Chroma L0, Luma L1, Luma L0]	[**DevILK**] Weight is 128 [ChromaL1, Chroma L0, Luma L1, Luma L0]
4	31:16	**n/a**	**n/a**	**WO_Y_Block1_L1**
	15:0	**n/a**	**n/a**	**WO_Y_Block1_L0**
5	31:16	**n/a**	**n/a**	**WO_Cb_Block1_L1**
	15:0	**n/a**	**n/a**	**WO_Cb_Block1_L0**
6	31:16	**n/a**	**n/a**	**WO_Cr_Block1_L1**
	15:0	**n/a**	**n/a**	**WO_Cr_Block1_L0**
7	31:4	**n/a**	**n/a**	Reserved: MBZ
	3:0			[**DevILK**] Weight is 128[ChromaL1, Chroma L0, Luma L1, Luma L0]
8	31:16	**n/a**	**n/a**	**WO_Y_Block2_L1**
	15:0	**n/a**	**n/a**	**WO_Y_Block2_L0**
9	31:16	**n/a**	**n/a**	**WO_Cb_Block2_L1**

DWord	Bit	MvSize		
		0	2	8, 16, 32
	15:0	n/a	n/a	**WO_Cb_Block2_L0**
10	31:16	n/a	n/a	**WO_Cr_Block2_L1**
	15:0	n/a	n/a	**WO_Cr_Block2_L0**
11	31:4	n/a	n/a	Reserved: MBZ
	3:0			[**DevILK**] Weight is 128[ChromaL1, Chroma L0, Luma L1, Luma L0]
12	31:16	n/a	n/a	**WO_Y_Block3_L1**
	15:0	n/a	n/a	**WO_Y_Block3_L0**
13	31:16	n/a	n/a	**WO_Cb_Block3_L1**
	15:0	n/a	n/a	**WO_Cb_Block3_L0**
14	31:16	n/a	n/a	**WO_Cr_Block3_L1**
	15:0	n/a	n/a	**WO_Cr_Block3_L0**
15	31:4	n/a	n/a	Reserved: MBZ
	3:0			[**DevILK**] Weight is 128[ChromaL1, Chroma L0, Luma L1, Luma L0]

1.7.7.2.3 Weight-Offset Block of Indirect Data in AVC-IT and AVC_MC Modes for WeightedBiPredFlag = 10

Implicit weights are used for B-slices when WeightedBiPredFlag = 10. In this mode the offsets are always zero and the weights are 9-bits. To fit this in the same memory footprint, the offsets are not sent and the 9-bit weights are sign extended into the 16-bit block used for the weight/offset pair in explicit mode.

Table 1-38. Indirect data Implicit Weight block in AVC-IT mode (and AVC-MC mode)

DWord	Bit	MvSize		
		0	2	8, 16, 32
	31:16	n/a	**Weight_Y_L1**	**Weight_Y_Block0_L1**
	15:0	n/a	**Weight_Y_L0**	**Weight_Y_Block0_L0**
1	31:16	n/a	**Weight_Cb_L1**	**Weight_Cb_Block0_L1**

DWord	Bit	MvSize		
		0	2	8, 16, 32
	15:0	n/a	**Weight_Cb_L0**	**Weight_Cb_Block0_L0**
2	31:16	n/a	**Weight_Cr_L1**	**Weight_Cr_Block0_L1**
	15:0	n/a	**Weight_Cr_L0**	**Weight_Cr_Block0_L0**
3	31:0	n/a	Reserved: MBZ	Reserved: MBZ
4	31:16	n/a	n/a	**Weight_Y_Block1_L1**
	15:0	n/a	n/a	**Weight_Y_Block1_L0**
5	31:16	n/a	n/a	**Weight_Cb_Block1_L1**
	15:0	n/a	n/a	**Weight_Cb_Block1_L0**
6	31:16	n/a	n/a	**Weight_Cr_Block1_L1**
	15:0	n/a	n/a	**Weight_Cr_Block1_L0**
7	31:0	n/a	n/a	Reserved: MBZ
8	31:16	n/a	n/a	**Weight_Y_Block2_L1**
	15:0	n/a	n/a	**Weight_Y_Block2_L0**
9	31:16	n/a	n/a	**Weight_Cb_Block2_L1**
	15:0	n/a	n/a	**Weight_Cb_Block2_L0**
10	31:16	n/a	n/a	**Weight_Cr_Block2_L1**
	15:0	n/a	n/a	**Weight_Cr_Block2_L0**
11	31:0	n/a	n/a	Reserved: MBZ
12	31:16	n/a	n/a	**Weight_Y_Block3_L1**
	15:0	n/a	n/a	**Weight_Y_Block3_L0**
13	31:16	n/a	n/a	**Weight_Cb_Block3_L1**
	15:0	n/a	n/a	**Weight_Cb_Block3_L0**
14	31:16	n/a	n/a	**Weight_Cr_Block3_L1**

DWord	Bit	MvSize		
		0	2	8, 16, 32
	15:0	**n/a**	**n/a**	**Weight_Cr_Block3_L0**
15	31:0	**n/a**	**n/a**	Reserved: MBZ

The weights for MvSize = 8,16,32 are replicated is exactly the same manner as the binding table indices. See section 1.7.3.1.5 for the description of the replication method. For MvSize=2 the replication is described in the following table:

MbType	Inter Macroblock	Implicit Weight Replication Rule
1	BP_L0_16x16	L0 weight and offset replicated to L1 entries
2	B_L1_16x16	L1 weight and offset replicated to L0 entries
3	B_Bi_16x16	No replication needed.

1.7.7.2.4 Transform Residual Block of Indirect Data in AVC-IT Mode

Transform-domain residual data block in AVC-IT mode is similar to that in IS mode. Only the non-zero coefficients are present in the data buffer and they are packed in the 8x8 block sequence of Y0, Y1, Y2, Y3, Cb4 and Cr5, as shown in **Error! Reference source not found.**. When an 8x8 block is further subdivided into 4x4 subblocks, the coefficients, if present, are organized in the subblock order. The smallest subblock division is referred to as a **transform block**. The indirect data length in MEDIA_OBJECT_EX includes all the non-zero coefficients for the macroblock. It must be doubleword aligned.

Each non-zero coefficient in the indirect data buffer is contained in a doubleword-size data structure consisting of the coefficient index, end of block (EOB) flag and the fixed-point coefficient value in 2's compliment form. As shown in Table 1-21, *index* is the row major 'raster' index of the coefficient within a transform block. A coefficient is a 16-bit value in 2's complement.

Table 1-21. Structure of a transform-domain residue unit

DWord	Bit	Description
0	31:16	**Transform-Domain Residual (coefficient) Value.** This field contains the value of the non-zero transform-domain residual data in 2's compliment.
	15:7	Reserved: MBZ

DWord	Bit	Description
	6:1	**Index.** This field specifies the raster-scan address (raw address) of the coefficient within the transform block. For a coefficient at Cartesian location (row, column) = (y, x) in a transform block of width W, Index is equal to (y * W + x). For example, coefficient at location (row, column) = (0, 0) in a 4x4 transform block has an index of 0; that at (2, 3) has an index of 2*4 + 3 = 11. The valid range of this field depends on the size of the transform block. Format = U6 Range = [0, 63]
	0	**EOB (End of Block).** This field indicates whether the transform-domain residue is the last one of the current transform block.

1.7.7.3 Inline Data Format in AVC-MC Mode

Each MEDIA_OBJECT_EX command in “AVC-MC mode” corresponds to the processing of one macroblock. Macroblock parameters are passed in as inline data and the pixel-domain residual data (as well as motion vectors and weight/offset) for the macroblock is passed in as indirect data.

Inline data format in AVC-MC mode follows the exact same format like the one in AVC-IT mode. Specifically, the common fields required by VFE are at the same locations and have the same meaning.

The following table depicts the inline data format in AVC-MC mode. Unlike AVC-IT, all fields in inline data are forwarded to the thread. Starting at GRF location, inline data are stored in GRF contiguously with the tail-end partial GRF, if present, zero-filled. Some fields are merely forwarded. Some fields are also used by VFE as indicated in the following table by a mark of [Used by VFE]. As shown, inline data starts at dword 4 of MEDIA_OBJECT_EX command.

1.7.7.4 Indirect Data Format in AVC-MC Mode

Indirect data in AVC-IT mode consist of Motion Vectors, Weight/Offset and pixel-domain residual data. All three data blocks have variable size.

Sizes of Motion Vector block and the Weight-Offset block are determined by the MvSize value. They are the same as in AVC-IT mode and are depicted in Table 1-18. Weight-Offset block, if present, is always packed behind the Motion Vector block. See Section 1.7.7.2 for more details.

Residual data block must start at a predetermined offset, controlled by the fields in VFE_STATE_EX.

1.7.7.5 Inline Data Format in VC1-IT Mode

Each MEDIA_OBJECT_EX command in “VC1-IT mode” corresponds to the processing of one macroblock. Macroblock parameters, including motion vectors, are passed in as inline data and the non-zero DCT coefficient data for the macroblock is passed in as indirect data.

Error! Reference source not found. depicts the inline data format in VC1-IT mode. All fields in inline data are forwarded to the thread as thread payload. Inline data are stored in GRF contiguously with the tail-end partial GRF, if present, zero-filled. Some fields are merely forwarded. Some fields are also used by VFE as indicated in the following table by a mark of [Used by VFE]. As shown, inline data starts at dword 4 of MEDIA_OBJECT_EX command.

Table 1-22. Inline data in VC1-IT mode

<table>
<tr><th>DWord</th><th>Bit</th><th>Description</th></tr>
<tr><td>4+0</td><td>31:28</td><td>MvFieldSelect. A bit-wise representation indicating which field in the reference frame is used as the reference field for current field. It's only used in decoding interlaced pictures.
This field is valid for non-intra macroblock only.
<table>
<tr><th>Bit</th><th>Description</th></tr>
<tr><td>28</td><td>Forward predict of current frame/field or TOP field of interlace frame, or block 0 in 4MV mode.</td></tr>
<tr><td>29</td><td>Backward predict of current frame/field or TOP field of interlace frame, or forward predict for block 1 in 4MV mode.</td></tr>
<tr><td>30</td><td>Forward predict of BOTTOM field of interlace frame, or block 2 in 4MV mode.</td></tr>
<tr><td>31</td><td>Backward predict of BOTTOM field of interlace frame, or forward predict for block 3 in 4MV mode.</td></tr>
</table>
Each corresponding bit has the following indication.
0 = The prediction is taken from the top reference field.
1 = The prediction is taken from the bottom reference field.</td></tr>
<tr><td></td><td>27</td><td>Reserved. MBZ</td></tr>
<tr><td></td><td>26</td><td>MvFieldSelectChroma . This field specifies the polarity of reference field for chroma blocks when their motion vector is derived in Motion4MV mode for interlaced (field) picture.
Non-intra macroblock only. This field is derived from MvFieldSelect.
0 = The prediction is taken from the top reference field.
1 = The prediction is taken from the bottom reference field.</td></tr>
<tr><td></td><td>25:24</td><td>MotionType – Motion Type
For frame picture, a macroblock may only be either 00 or 10.
For interlace picture, a macroblock may be of any motion types. It can be 01 if and only if DctType is 1.
This field is 00 if and only if IntraMacroblock is 1.
00 = Intra
01 = Field Motion.
10 = Frame Motion or no motion.
Others = Reserved.</td></tr>
<tr><td></td><td>23</td><td>Reserved. MBZ</td></tr>
<tr><td></td><td>22</td><td>MvSwitch. This field specifies whether the prediction needs to be switched from forward to backward or vice versa for single directional prediction for top and bottom fields of interlace frame B macroblocks.
0 = No directional prediction switch from top field to bottom field
1 = Switch directional prediction from top field to bottom field</td></tr>
</table>

DWord	Bit	Description
	21	**DctType.** This field specifies whether the residual data is coded as field residual or frame residual for interlaced picture. This field can be 1 only if MotionType is 00 (intra) or 01 (field motion). For progressive picture, this field must be set to '0', i.e. all macrobalcoks are frame macroblock. 0 = Frame residual type. 1 = Field residual type.
	20	**OverlapTransform.** This field indicates whether overlap smoothing filter should be performed on I-block boundaries. 0 = No overlap smoothing filter. 1 = Overlap smoothing filter performed.
	19	**Motion4MV.** This field indicates whether current macroblock a progressive P picture uses 4 motion vectors, one for each luminance block. It's only valid for progressive P-picture decoding. Otherwise, it is reserved and MBZ. For example, with MotionForward is 0, this field must also be set to 0. 0 = 1MV-mode. 1 = 4MV-mode.
	18	**MotionBackward.** This field specifies whether the backward motion vector is active for B-picture. This field must be 0 if Motion4MV is 1 (no backward motion vector in 4MV-mode). 0 = No backward motion vector. 1 = Use backward motion vector(s).
	17	**MotionForward.** This field specifies whether the forward motion vector is active for P and B pictures. 0 = No forward motion vector. 1 = Use forward motion vector(s).
	16	**IntraMacroblock.** This field specifies if the current macroblock is intra-coded. When set, Coded Block Pattern is ignored and no prediction is performed (i.e., no motion vectors are used). For field motion, this field indicates whether the top field of the macroblock is coded as intra. 0 = Non-intra macroblock. 1 = Intra macroblock.
	15:12	**LumaIntra8x8Flag – Luma Intra 8x8 Flag** This field specifies whether each of the four 8x8 luminance blocks are intra or inter coded when Motion4MV is set to 4MV-Mode. Each bit corresponds to one block. "0" indicates the block is inter coded and '1' indicates the block is intra coded. When Motion4MV is not 4MV-Mode, this field is reserved and MBZ. Bit 15: Y0 Bit 14: Y1 Bit 13: Y2 Bit 12: Y3

DWord	Bit	Description
	11:6	**CBP - Coded Block Pattern** This field specifies whether the 8x8 residue blocks in the macroblock are present or not. Each bit corresponds to one block. “0” indicates residue block isn’t present, “1” indicates residue block is present. Note: For each block in an intra-coded macroblock or an intra-coded block in a P macroblock in 4MV-Mode, the corresponding CBP must be 1. Subsequently, there must be at least one coefficient (this coefficient might be zero) in the indirect data buffer associated with the bock (i.e. residue block must be present). Bit 11: Y0 Bit 10: Y1 Bit 9: Y2 Bit 8: Y3 Bit 7: Cb4 Bit 6: Cr5
	5	**ChromaIntraFlag - Derived Chroma Intra Flag** This field specifies whether the chroma blocks should be treated as intra blocks based on motion vector derivation process in 4MV mode. 0 = Chroma blocks are not coded as intra. 1 = Chroma blocks are coded as intra
	4	**LastRowFlag – Last Row Flag** This field indicates that the current macroblock belongs to the last row of the picture. This field may be used by the kernel to manage pixel output when overlap transform is on. 0 = Not in the last row 1 = In the last row
	3:0	**Reserved.** MBZ
4+1	32:26	**Reserved.** MBZ
	25:24	**OSEdgeMaskChroma** This field contains the overscan edge mask for the Chroma blocks, the bit order of this field matches the overscan edge numbers shown in Figure 1-11. The left edge masks are used by VFE hardware and the top edge masks are used by the kernel software. Bit 24: Top edge of block Cb/Cr Bit 25: Left edge of block Cb/Cr

DWord	Bit	Description
	23:16	**OSEdgeMaskLuma** This field contains the overscan edge mask for the Luma blocks, the bit order of this field matches the overscan edge numbers shown in Figure 1-11. The left edge masks are used by VFE hardware and the top edge masks are used by the kernel software. Bit 16: Top edge of block Y0 Bit 17: Top edge of block Y1 Bit 18: Top edge of block Y2 Bit 19: Top edge of block Y3 Bit 20: Left edge of block Y0 Bit 21: Left edge of block Y1 Bit 22: Left edge of block Y2 Bit 23: Left edge of block Y3 ***Programming Note**: In order to create 8 predication bits from each edge mask bit, software may first create a 0, 1 vector by using a **shr** instruction with a step shift vector like 0, 1, 2, 3 (e.g. using immediate of type :v. Then each 0 or 1 of the LSB can be repeated by an **and** instruction to create 8 bits to the flag register. Alternatively, this can be achieved with one and instruction using a CURBE constant map of bit 0 and bit 1 mask.*
	15:8	**VertOrigin (Vertical Origin)** In unit of macroblocks relative to the current picture (frame or field).
	7:0	**HorzOrigin (Horizontal Origin)** In unit of macroblocks.
4+2	31:16	**MotionVector[0].Vert**
	15:0	**MotionVector[0].Horz**
4+3	31:0	**MotionVector[1]**
4+4	31:0	**MotionVector[2]**
4+5	31:0	**MotionVector[3]**
4+6	31:0	**MotionVectorChroma** Notes: This field is derived from MotionVector[3:0] as described in the following section.

<table>
<tr><th>DWord</th><th>Bit</th><th>Description</th></tr>
<tr><td>4+7</td><td>32:24</td><td>Subblock Code for Y3 [Used by VFE]
The following subblock coding definition applies to all 6 subblock coding bytes. Bits 7:6 are reserved.
<table>
<tr><th colspan="2">Subblock Partitioning
(Bits [1:0])</th><th colspan="4">Subblock Present
(0 means not present, 1 means present)</th></tr>
<tr><th>Code</th><th>Meaning</th><th>Bit 2</th><th>Bit 3</th><th>Bit 4</th><th>Bit 5</th></tr>
<tr><td>00</td><td>Single 8x8 block (sb0)</td><td>Sb0</td><td>Don’t care</td><td>Don’t care</td><td>Don’t care</td></tr>
<tr><td>01</td><td>Two 8x4 subblocks (sb0-1)</td><td>Sb0</td><td>Sb1</td><td>Don’t care</td><td>Don’t care</td></tr>
<tr><td>10</td><td>Two 4x8 subblocks (sb0-1)</td><td>Sb0</td><td>Sb1</td><td>Don’t care</td><td>Don’t care</td></tr>
<tr><td>11</td><td>Four 4x4 subblocks (sb0-3)</td><td>Sb0</td><td>Sb1</td><td>Sb2</td><td>Sb3</td></tr>
</table></td></tr>
<tr><td></td><td>23;16</td><td>Subblock Code for Y2 [Used by VFE]</td></tr>
<tr><td></td><td>15:8</td><td>Subblock Code for Y1 [Used by VFE]</td></tr>
<tr><td></td><td>7:0</td><td>Subblock Code for Y0 [Used by VFE]</td></tr>
<tr><td>4+8</td><td>31:16</td><td>Reserved. MBZ</td></tr>
<tr><td></td><td>15:8</td><td>Subblock Code for Cr [Used by VFE]</td></tr>
<tr><td></td><td>7:0</td><td>Subblock Code for Cb [Used by VFE]</td></tr>
</table>

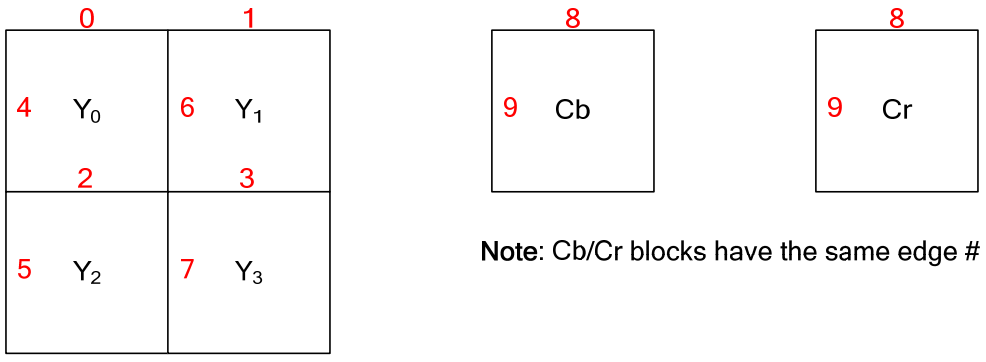


Figure 1-11 Indexing Block Edges for Overlapped Smoothing

1.7.7.5.1 Deriving Motion Vectors and Field Select for Interlaced Frame Picture

In MPEG2, the motion vectors are related to the decoded picture. For field picture, it is related to the field and which field of the reference frame that a motion vector points to is given in the bitstream by syntax

element called Motion Vector Field Select (MVFS). In contrary, motion vectors defined in VC1 standard for an interlaced frame is frame based and there is no such MVFS syntax. The VC1-IT interface defines the motion vector and MVFS following the MPEG2 convention. Therefore, motion vectors and MVFS must be derived from the frame-based motion vector values and the current macroblock position (in the top or bottom field).

The derivation of the picture-based motion vectors (luma) and MVFS is provided by the following pseudo-code. The idea is that MVFS comes from the LSB of the final pointer to the reference frame (note here it is frame based not field base). The final pointer is the addition of the frame based motion vector and the current macroblock position in the current frame (again, it is relative to the frame, not picture).

- Let (MV_X, MV_Y) be the original frame based luma motion vector in quarter-pel representation.
- Let (LMV_X, LMV_Y) be the derived field based luma motion vector and (CMV_X, CMV_Y) be the derived field based chroma motion vector, both in quarter-pel precision as well.
- Let MVFS be the derived motion vertical field select field.
- Then
    - LMV_X = MV_X;
    - if (Current_field != BOTTOM_FIELD)
        - iy = MV_Y >> 2; // Interger portion of MV_Y
    - else // Current_field == BOTTOM_FIELD
        - iy = (MV_Y >> 2) + 1; // Interger portion of MV_Y adjusted
    - MVFS = iy & 1; // 0 – top field, 1 – bottom field
    - LMV_Y = ((iy >>1)<< 2) + (MV_Y & 3);

1.7.7.5.2 Chroma Interpolations for Motion Prediction

There are two different interpolation modes are used for generating chroma samples according to the picture level parameter **bMVprecisionAndChromaRelation**: *the quarter-pel chroma motion prediction*, and *the half-pel chroma motion prediction*. The bilinear interpolation is applied for both cases.

For the quarter-pel case, the motion vectors for the chroma components are derived from the corresponding luma motion vectors according to the following pseudocodes.

For the case of 1-MV,

```
cmv.x = (mv.x + (mv.x&3==3))>>1;

cmv.y = (mv.y + (mv.y&3==3))>>1;
```

For the case of 4-MV,

```
switch(number of inter-coded blocks)

case 3: // median of three
```

```
    if(mv0.x<mv1.x && mv0.x<mv2.x)
      mv0.x = (mv1.x<mv2.x) ? mv1.x : mv2.x;
    else if(mv0.x>mv1.x && mv0.x>mv2.x)
      mv0.x = (mv1.x>mv2.x) ? mv1.x : mv2.x;
    cmv.x = (mv0.x + (mv0.x&3==3))>>1;
    cmv.y = (mv0.y + (mv0.y&3==3))>>1;
    break;
  case 4: // average of middle two
    if(mv0.x<mv1.x && mv0.x<mv2.x && mv0.x<mv3.x){
      if(mv2.x<mv3.x){ mv0.x = mv2.x;  if(mv1.x>mv3.x) mv1.x=mv3.x; }
      else          { mv0.x = mv3.x;  if(mv1.x>mv2.x) mv1.x=mv2.x; }
    }
    else if(mv0.x>mv1.x && mv0.x>mv2.x && mv0.x>mv3.x){
      if(mv2.x>mv3.x){ mv0.x = mv2.x;  if(mv1.x<mv3.x) mv1.x=mv3.x; }
      else          { mv0.x = mv3.x;  if(mv1.x<mv2.x) mv1.x=mv2.x; }
    }
    else if(mv1.x<mv2.x && mv1.x<mv3.x)
      mv1.x = (mv2.x<mv3.x) ? mv2.x : mv3.x;
    else if(mv1.x>mv2.x && mv1.x>mv3.x)
      mv1.x = (mv2.x>mv3.x) ? mv2.x : mv3.x;
  case 2: // average of two
    x = (mv0.x + mv1.x)>>1; cmv.x = (x + (x&3==3))>>1;
    y = (mv0.y + mv1.y)>>1; cmv.y = (y + (y&3==3))>>1;
    break;
  case 0: case 1: chroma should be coded as intra-blocks.
  }
```

For the half–pel case, the motion vectors for the chroma components are derived from one more extra shifting operation by rounding to the nearest full-pel if they are not currently in the half-grid.

For simple and main profile, the motion vectors are truncated so that the reference block is not totally off the picture frame:

```
// For luma:

if((mv.x>>2)<-16) mv.x = -64    +(mv.x&3);

if((mv.x>>2)> PW) mv.x = (PW<<2)+(mv.x&3);

if((mv.y>>2)<-16) mv.y = -64    +(mv.y&3);

if((mv.y>>2)> PH) mv.y = (PH<<2)+(mv.y&3);


// For chroma:

if((cmv.x>>2)<- 8) cmv.x = -32     +(cmv.x&3);

if((cmv.x>>2)>CPW) cmv.x = (CPW<<2)+(cmv.x&3);

if((cmv.y>>2)<- 8) cmv.y = -32     +(cmv.y&3);

if((cmv.y>>2)>CPH) cmv.y = (CPH<<2)+(cmv.y&3);
```

1.7.7.6 Indirect Data Format in VC1-IT Mode

Indirect data format in VC1-IT mode is identical to the transform-domain residual data block portion of the indirect data format in AVC-IT mode.

The indirect data start address in MEDIA_OBJECT_EX specifies the doubleword aligned address of the first non-zero transform-domain residue (referred to as ‘coefficient’) of the first block of the macroblock. The indirect data length in MEDIA_OBJECT_EX includes all the non-zero coefficients for the macroblock. It must be doubleword aligned.

Each non-zero coefficient in the indirect data buffer is contained in a doubleword-size data structure as shown in Table 1-21.

1.7.7.7 Inline Data Format in Generic Mode

MEDIA_OBJECT_EX command can also be used in “Generic mode” in place of MEDIA_OBJECT command. The only difference of the usage is to allow interface descriptor remap. MEDIA_OBJECT_EX command cannot be used together with MEDIA_OBJECT command.

1.7.8 MEDIA_OBJECT_PRT Command

The MEDIA_OBJECT_PRT command is for generating Persistent Root Thread for the media pipeline. It only supports loading of inline data but not indirect data.

This command should be used for a root thread that might have to be present in the system for a period longer than the certain minimal context-switch interrupt latency. It has to honor the context interrupt signal to terminate upon request. It should also handle replay from the interrupted point upon context restore (the same thread being dispatched more than once). In contrary, if a thread is not a Persistent Root Thread, if dispatched, it must run to completion.

The command can be used in all VFE modes, except VLD mode.

[DevSNB+]:

MEDIA_OBJECT_PRT

Project:	[DevSNB+]	**Length Bias:**	2

For simplification, _PRT command has a fixed size of 16 DWORD

DWord	Bit	Description
0	31:29	**Command Type** Default Value: 3h GFXPIPE Format: OpCode
	28:27	**Media Command Opcode** Default Value: 2h MEDIA_OBJECT_PRT Format: OpCode
	26:24	**Media Command Opcode** Default Value: 1h MEDIA_OBJECT_PRT Format: OpCode
	23:16	**Media Command Opcode** Default Value: 2h MEDIA_OBJECT_PRT Format: OpCode
	15:0	**DWord Length** Default Value: 14h Excludes DWord (0,1) Format: =n Total Length - 2 Project: All **Note:** Regardless of the mode, inline data must be present in this command. The command size must fit within 16 dwords.
1	31:6	**Reserved** Project: All Format: MBZ
	5:0	**Interface Descriptor Offset** Project: All Format: U5 [DevSNB] This field specifies the offset from the interface descriptor base pointer to the interface descriptor which will be applied to this object. It is specified in units of interface descriptors.
2	31	**Children Present** Project: All Format: Enable Indicates that the root thread may send spawn messages to spawn child threads and/or synchronized root threads. If Children Present is not set, TS signals VFE to dereference the URB handle immediately after it receives acknowledgement from TD that the thread is dispatched. If Children Present is set, the URB handle is forwarded to the root thread and serves as the return URB handle for the root thread. TS does not signal deference at the time of dispatch. TS signals URB handle deference only when it receives a resource dereference message from the thread. ***In order avoid deadlock, such de-reference must be issued once and only once for each URB handle.***

MEDIA_OBJECT_PRT

	30:24	**Reserved** Project: All Format: MBZ
	23	**PRT_Fence Needed** Project: All Format: Enable This field specifies that a PRT_Fence is generated after dispatching the thread associated with this MEDIA_OBJECT_PRT. The PRT_Fence prevents additional threads following this persistent root thread until a thread spawn message is sent. The PRT_Fence is generated on first dispatch of the persistent root, as well as on re-dispatches of the persistent root after context restore.
	22	**PRT_FenceType** Project: All This field specifies the type of fence the PRT thread uses. If this field is set to 0, the fence is set at the end of the root thread queue. It will block the dispatch of the next root thread, but allowed these root threads to be populated through VFE to the root thread queue in TS. If this field is set to 1, the fence is set at the entry of VFE, similar to the fence set by the MEDIA_STATE_FLUSH command. No more command can go into the media pipe until a thread spawn message is sent (by the PRT). This field is only valid when PRT_Fence Needed is set to 1. Otherwise, it is ignored by hardware.
	21:17	**Reserved** Project: All Format: MBZ was **Indirect Data Length**
	16:0	**Indirect Data Length.** This field provides the length in bytes of the indirect data. A value zero indicates that indirect data fetching is disabled – subsequently, the Indirect Data Start Address field is ignored. This field must have the same alignment as the Indirect Object Data Start Address. It must be DQWord (32-byte) aligned. As the indirect data are sent directly to URB, range is limited to 496 DW. When both inline and indirect data are fetched for this command, the total size in 8-DW registers must be less than 112 (with both inline data length and indirect data length rounded up to 8-DW aligned). Format = U17 in bytes
3	31:0	**Indirect Data Length** Project: All Format: MBZ was **Indirect Data Start Address**
4..15	31:0	**Inline Data** Project: All Format: U32 FormatDesc

Value	Name	Description	Project
0h	Root thread queue	Root thread queue fence	All
1h	VFE state flush	VFE state flush fence	All

1.7.9 MEDIA_OBJECT_WALKER Command [DevSNB+]

The MEDIA_OBJECT_WALKER command uses the hardware walker in VFE for generating threads associated with a rectangular shaped object. It only supports loading of inline data (optionally) but not indirect data. Control of scoreboards (up to 8) is implicit based on the (X, Y) address of the generated thread and the scoreboard control state.

The command can be used only in Generic modes.

When **Use Scoreboard** field is set, the (X, Y) address and the Color field of the generated thread are used in the hardware scoreboard and the thread dependencies are set by states from the MEDIA_VFE_STATE command.

One or more threads may be generated by this command. This command doesn't support indirect object load. When inline data is present, it is repeated for all threads it generates. Unlike CURBE, which requires pipeline flush for change, continued change of this kind of 'global' (in the sense of shared by multiple threads from this command) data is supported when MEDIA_OBJECT_WALKER commands are issued without a pipeline flush in between.

Dword	Bits	Description
0	31:29	**Command Type =** GFXPIPE = 3h
	28:16	**Media Command Opcode =** MEDIA_OBJECT_WALKER Pipeline[28:27] = Media = 2h; Opcode[26:24] **=** 1h; Subopcode[23:16] = 03h
	15:0	**DWord Length** (Excludes DWords 0,1) Valid range: [16…N] **Note:** If this field is greater than 15, it indicates that inline data is present. If present, inline data is common for all threads generated from this command, If this field is 15, it indicates that inline data is not present. It should be noted that unlike other media object command, inline data is optional for this command.
1	31:0	**Reserved**
	7:6	**Reserved.** MBZ
	5:0	**Interface Descriptor Offset.** This field specifies the offset from the interface descriptor base pointer to the interface descriptor which will be applied to this object. It is specified in units of interface descriptors. Format: U5 [DevSNB]

Dword	Bits	Description
2	31	**Children Present.** Indicates that the root thread may send spawn messages to spawn child threads and/or synchronized root threads. If Children Present is not set, TS signals VFE to dereference the URB handle immediately after it receives acknowledgement from TD that the thread is dispatched. If Children Present is set, the URB handle is forwarded to the root thread and serves as the return URB handle for the root thread. TS does not signal deference at the time of dispatch. TS signals URB handle deference only when it receives a resource dereference message from the thread. ***In order avoid deadlock, such dereference must be issued once and only once for each URB handle.*** Format = Enable
	30:25	**Reserved. MBZ**
	24	**Thread Synchronization.** This field when set indicates that the dispatch of the thread originated from this command is based on the "spawn root thread" message. 0 = No thread synchronization 1 = Thread dispatch is synchronized by the "spawn root thread" message
	23:22	**Reserved.** MBZ
	21	**Use Scoreboard.** This field specifies whether the thread associated with this command uses hardware scoreboard. Only when this field is set, the scoreboard control fields in the VFE Dword are valid. If this field is cleared, the thread associated with this command bypasses hardware scoreboard. 0 = Not using scoreboard 1 = Using scoreboard
	20:17	**Reserved.** MBZ
	16:0	**Indirect Data Length.** This field provides the length in bytes of the indirect data. A value zero indicates that indirect data fetching is disabled – subsequently, the Indirect Data Start Address field is ignored. This field must have the same alignment as the Indirect Object Data Start Address. It must be DQWord (32-byte) aligned. As the indirect data are sent directly to URB, range is limited to 496 DW. When both inline and indirect data are fetched for this command, the total size in 8-DW registers must be less than or equal to 63 (with both inline data length and indirect data length rounded up to 8-DW aligned). Format = U17 in bytes
3	31:0	**Indirect Data Start Address.** This field specifies the Graphics Memory starting address of the data to be loaded into the kernel for processing. This pointer is relative to the **Indirect Object Base Address**. Hardware ignores this field if indirect data is not present. Alignment of this address depends on the mode of operation. It is the DWord aligned address of the indirect data. Range = [0 - 512MB] (Bits 31:29 MBZ)
4	31:0	**Reserved.** MBZ
5	31:8	**Reserved.** MBZ

Dword	Bits	Description
	7:0	**Scoreboard Mask:** Each bit indicates the corresponding dependency scoreboard is dependent on. This field is AND'd with the corresponding Scoreboard Mask field in the MEDIA_VFE_STATE. All threads generated by this walker command share the same dynamic mask. **Bit n (for n = 0…7):** Scoreboard n is dependent, where bit 0 maps to n = 0. Format = TRUE/FALSE
6	31	**Dual Mode** Format: TRUE / FALSE
	30	**Repel** Format: TRUE / FALSE Note: Repel should not be combined with either Dual Mode or Quad Mode
	29	**Reserved**
	28	**Reserved.** MBZ
	29:24	**Color Count Minus One.** This field specifies the number of repeat of the inner most loop of the walker. Each repeated walk position is assigned with an incremental Color number. The Color number together with the X and Y position of the thread is used for dependency scoreboard control. **Usage Example:** This allows multiple sets of dependency threads to be dispatched. Format: U4
	23:21	**Reserved.** MBZ
	20:16	**Middle Loop Extra Steps** Format = U5
	15:14	**Reserved.** MBZ
	13:12	**Local Mid-Loop Unit Y** Format = S1
	11:10	**Reserved.** MBZ
	9:8	**Mid-Loop Unit X** Format = S1
	7:0	**Reserved.** MBZ
7	31:26	**Reserved.** MBZ
	25:16	**Global Loop Exec Count** Format = U10
	15:10	**Reserved.** MBZ
	9:0	**Local Loop Exec Count** Format = U10
8	31:25	**Reserved.** MBZ

Dword	Bits	Description
	24:16	**Block Resolution Y :** Vertical resolution of the local loop. Format = U9
	15:9	**Reserved.** MBZ
	8:0	**Block Resolution X :** Horizontal resolution of the local loop. Format = U9
9	31:25	**Reserved.** MBZ
	24:16	**Local Start Y :** Starting vertical position of the local loop. Format = U9
	15:9	**Reserved.** MBZ
	8:0	**Local Start X :** Starting horizontal position of the local loop. Format = U9
10	31:25	**Reserved.** MBZ
	24:16	**[DevSNB]Local End Y :** Ending vertical position of the local loop. Format = U9
	15:9	**Reserved.** MBZ
	8:0	**[DevSNBLocal End X :** Ending horizontal position of the local loop. Format = U9
11	31:26	**Reserved.** MBZ
	25:16	**Local Outer Loop Stride Y :** Vertical stride of the local outer loop, in 2's complement. Format = S9
	15:12	**Reserved.** MBZ
	9:0	**Local Outer Loop Stride X :** Horizontal stride of the local outer loop, in 2's complement. Format = S9
112	31:26	**Reserved.** MBZ
	25:16	**Local Inner Loop Unit Y :** Vertical stride of the local inner loop, in 2's complement. Format = S9
	15:12	**Reserved.** MBZ
	9:0	**Local Inner Loop Unit X :** Horizontal stride of the local inner loop, in 2's complement. Format = S9
13	31:25	**Reserved.** MBZ
	24:16	**Global Resolution Y :** Vertical resolution of the global loop. Format = U9
	15:9	**Reserved.** MBZ

Dword	Bits	Description
	8:0	**Global Resolution X :** Horizontal resolution of the global loop. Format = U9
14	31:26	**Reserved.** MBZ
	25:16	**Global Start Y :** Starting vertical location of the global loop, in 2's complement. Format = S9
	15:10	**Reserved.** MBZ
	9:0	**Global Start X :** Starting horizontal location of the global loop, in 2's complement. Format = S9
15	31:26	**Reserved.** MBZ
	25:16	**Global Outer Loop Stride Y :** Vertical stride of the global outer loop, in 2's complement. Format = S9
	15:10	**Reserved.** MBZ
	9:0	**Global Outer Loop Stride X :** Horizontal stride of the global outer loop, in 2's complement. Format = S9
16	31:26	**Reserved.** MBZ
	25:16	**Global Inner Loop Unit Y :** Vertical stride of the global inner loop, in 2's complement. Format = S9
	15:10	**Reserved.** MBZ
	9:0	**Global Inner Loop Unit X :** Horizontal stride of the global inner loop, in 2's complement. Format = S9
17…N	31:0	**Inline Data**

1.8 Media Messages

All message formats are given in terms of dwords (32 bits) using the following conventions which are detailed in GEN4 Subsystem Chapter.

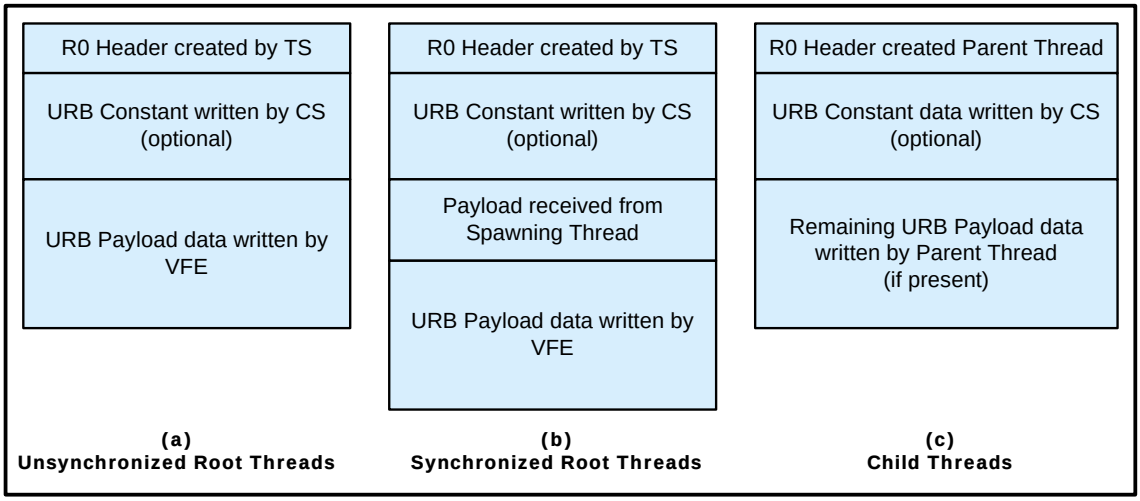
Dispatch Messages: **R**p.d

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

1.8.1 Thread Payload Messages

The root thread's register contents differ from that of child threads, as shown in Figure 1-12. The register contents for a synchronized root thread (also referred to as 'spawned root thread') and an unsynchronized one are also different. Whether the URB Constant data field is present or not is determined by the interface descriptor of a given thread. This applies to both root and child threads. When URB Constant data field is present for a synchronized root thread, URB constant data field is before the data field received from the spawning thread, which is also before the URB payload data.

Figure 1-12. Thread payload message formats for root and child threads



1.8.1.1 Generic Mode Root Thread

The following table shows the R0 register contents for a Generic mode root thread, which is generated by TS. The remaining payloads are application dependent.

Table 1-23. R0 header of a generic mode root thread

DWord	Bit	Description
R0.7	31	**Reserved**
	27:24	Reserved
	23:0	Reserved Root threads should have zero in this field.
R0.6	31:24	Reserved
	23:0	Reserved
R0.5	31:10	**Scratch Space Pointer.** Specifies the 1k-byte aligned pointer to the scratch space. This field is only valid when Scratch Space is enabled. Format = GeneralStateOffset[31:10]
	9:8	Reserved : MBZ
	7:0	**FFTID.** This ID is assigned by TS and is a unique identifier for the thread in comparison to other concurrent root threads. It is used to free up resources used by the thread upon thread completion.
R0.4	31:5	**Binding Table Pointer:** Specifies the 32-byte aligned pointer to the Binding Table. It is specified as an offset from the **Surface State Base Address**. Format = SurfaceStateOffset[31:5]

DWord	Bit	Description
	4:0	Reserved : MBZ
R0.3	31:5	**Sampler State Pointer.** Specifies the 32-byte aligned pointer to the sampler state table. Format = GeneralStateOffset[31:5]
	4	Reserved : MBZ
	3:0	**Per Thread Scratch Space.** Specifies the amount of scratch space, in 16-byte quantities, allowed to be used by this thread. The value specifies the power that two will be raised to, to determine the amount of scratch space. Format = U4 Range = [0,11] indicating [1k bytes, 2M bytes] in powers of two
R0.2 [DevSNB]	31:28	Reserved : MBZ
	27:24	**BarrierID**. This field indicates which one from the 16 Barriers this kernel is associated. Format: U4
	23:16	**Barrier.Offset.** This is the offset for the Barrier to indicate the offset from the requester's RegBase (which may be 0 if Bypass Gateway Control is set to 1) for the broadcast barrier message. Barrier.Offset + RegBase must be in the valid GRF range. Otherwise, hardware behavior is undefined. It is in unit of 256-bit GRF register. The most significant bit of this field must be zero. Format = U8 Range = [0,127]
	15:9	Reserved : MBZ
	8:4	**Interface Descriptor Offset.** This field specifies the offset from the interface descriptor base pointer to the interface descriptor which will be applied to this object. It is specified in units of interface descriptors. Format = U5
	3:0	**Scoreboard Color** (only with MEDIA_OBJECT_EX)**:** This field specifies which dependency color the current thread belongs to. It affects the dependency scoreboard control. Format = U4
	3:0	**Scoreboard Color** (only with MEDIA_OBJECT_EX)**:** This field specifies which dependency color the current thread belongs to. It affects the dependency scoreboard control. Format = U4
R0.1	31:28	[DevSNB+] Reserved : MBZ
	27:26	[DevSNB+] Reserved : MBZ
	25	**Reserved.** MBZ
	24:16	[DevILK+] **Scoreboard Y** This field provides the Y term of the scoreboard value of the current thread. Format = U9

DWord	Bit	Description
	15:12	[DevSNB+] Reserved : MBZ
	11:9	**Reserved.** MBZ
	8:0	[DevILK+] **Scoreboard X** This field provides the X term of the scoreboard value of the current thread. Format = U9
R0.0	31:24	[DevSNB+] **Scoreboard Mask:** Each bit indicates the corresponding dependency scoreboard is dependent on. This field is AND'd with the corresponding Scoreboard Mask field in the MEDIA_VFE_STATE. **Bit n (for n = 0…7):** Scoreboard n is dependent, where bit 24 maps to n = 0. Format = TRUE/FALSE
	23:16	Reserved : MBZ
	15:0	**URB Handle.** This is the URB handle where indicating the URB space for use by the root thread and its children.

1.8.1.2 Root Thread from MEDIA_OBJECT_PRT [DevCTG+]

The root thread payload message for an MEDIA_OBJECT_PRT command has a fixed format independent of the VFE mode (e.g. Generic mode or AVC-IT mode). One example GRF register location is given for the condition that CURBE is disabled.

Table 1-24. Root thread payload layout for a MEDIA_OBJECT_PRT command

GRF Register	Example	Description
R0	R0	**R0 header**
R1 – R(m)	n/a	**Constants from CURBE when CURBE is enabled** m is a non-negative value
R(m+1)	R1	**In-line Data block**.

The R0 header field is as the following, which is the same as in other modes except the Thread Restart Enable bit (bit 0 of R0.2).

Table 1-25. R0 header of the thread payload of a MEDIA_OBJECT_PRT command

DWord	Bit	Description
R0.7	31	**Reserved**
	27:24	Reserved

DWord	Bit	Description
	23:0	Reserved
R0.6	31:24	**Reserved**
	23:0	**Reserved**
R0.5	31:10	**Scratch Space Pointer.** Specifies the 1k-byte aligned pointer to the scratch space. This field is only valid when Scratch Space is enabled. Format = GeneralStateOffset[31:10]
	9:8	Reserved : MBZ
	7:0	**FFTID.** This ID is assigned by TS and is a unique identifier for the thread in comparison to other concurrent root threads. It is used to free up resources used by the thread upon thread completion.
R0.4	31:5	**Binding Table Pointer:** Specifies the 32-byte aligned pointer to the Binding Table. It is specified as an offset from the **Surface State Base Address**. Format = SurfaceStateOffset[31:5]
	4:0	Reserved : MBZ
R0.3	31:5	**Sampler State Pointer.** Specifies the 32-byte aligned pointer to the sampler state table. Format = GeneralStateOffset[31:5]
	4	Reserved : MBZ
	3:0	**Per Thread Scratch Space.** Specifies the amount of scratch space, in 16-byte quantities, allowed to be used by this thread. The value specifies the power that two will be raised to, to determine the amount of scratch space. Format = U4 Range = [0,11] indicating [1k bytes, 2M bytes] in powers of two
R0.2	31:4	**Interface Descriptor Pointer.** Specifies the 16-byte aligned pointer to *this thread's* interface descriptor. Can be used as a base from which to offset child thread's interface descriptor pointers from. Format = GeneralStateOffset[31:4]
	3:1	Reserved : MBZ
	0	**Thread Restart Enable.** If set, indicates that the persistent root thread (PRT) is being restarted, and context should be restored from the context save area before executing. Format = Enable
R0.1	31:0	Reserved : MBZ
R0.0	31:16	Reserved : MBZ
	15:0	**URB Handle.** This is the URB handle where indicating the URB space for use by the root thread and its children.

The inline data block field is the same as in the MEDIA_OBJECT_EX command with zero-filled partial GRF.

1.8.1.3 Root Thread from MEDIA_OBJECT_WALKER [DevSNB+]

The root thread payload message for an MEDIA_OBJECT_WALKER command, which must be in Generic mode, has the same format as that of the generic mode root thread format.

Table 1-26. Root thread payload layout for a MEDIA_OBJECT_WALKER command

GRF Register	Example	Description
R0	R0	**R0 header**
R1 – R(m)	n/a	**Constants from CURBE when CURBE is enabled** m is a non-negative value
R(m+1)	R1	**In-line Data block**.

The R0 header field is identical to that of Generic Mode Root Thread.

The inline data block field is the same as in the MEDIA_OBJECT command with zero-filled partial GRF.

There is no indirect data block field.

1.8.2 Thread Spawn Message

The thread spawn message is issued to the TS unit by a thread running on an EU. This message contains only one 8-DW register. The thread spawn message may be used to

- Spawn a child thread
- Spawn a root thread (start dispatching a synchronized root thread)
- Dereference URB handle
- Indicate a thread termination, dereference other TS managed resource and may or may not dereference URB handle
- Release a PRT_Fence (**[DevCTG+]**)

In order to end a root thread, the end of thread message must be targeted at the thread spawner. In this case, the root thread sends a message with a “dereference resource” in the Opcode field. The thread spawner does *not* snoop the messages sideband to determine when a root thread has ended. Thread Spawner does not track when a child thread terminates, to be consistent a child thread should also terminate with a “dereference resource” message to the Thread Spawner. Software must set the Requester Type (root or child thread) field correctly.

As TS dispatches one synchronized root thread upon receiving a ‘spawn root thread’ message (from a synchronization thread). The synchronizing thread must send the number of ‘spawn root thread’ message exactly the same as the subsequent ‘synchronized root thread’. No more, no less. Otherwise, hardware behavior is undefined.

URB Handle Offset field in this message (in M0.4) has 10 bits, allowing addressing of a large URB space. However, when a parent thread writes into the URB, it subjects to the maximum URB offset limitation of the URB write message, which is only 6 bits (see Unified Return Buffer Chapter for details). In this case, the parent thread may have to modify the URB Return Handle 0 field of the URB write message in order to subdivide the large URB space that the thread manages.

[DevIL-B+]: In addition to monitor 'end of thread message' targeted to Thread Spawner, Thread Spawner also monitors the message targeting to Message Gateway for EOT signal. Therefore, a child thread, who doesn't hold any hardware resource (URB handle or scratch memory) that Thread Spawner manages, can terminate with a Gateway message with EOT on. The reason of this new TS feature is to avoid a possible risk condition as described below.

In a system running child threads, a parent thread is monitoring the status of the child threads by communications through Message Gateway. When a child thread is about to terminate, it sends a message to the parent through Message Gateway and then sends a second message of EOT (end of thread) to TS.

There is a latency between sending a message to parent thread and the EOT to TS due to message bus arbitration. The parent thread may acknowledge the GW message and issue a new child dispatch before the EOT was processed; basically threads are issued faster than retired.

Because the messages for new child dispatch and EOT go to the same queue in TS, if the queue gets full, EOTs will get blocked. In the case when all the EUs/Threads are full, this will create a system deadlock: no EOTs can be acknowledged by TS (to free up EU resource) and no child threads can be dispatched (to free up TS queue to receive EOT message).

1.8.2.1 Message Descriptor

The following table shows the lower 16 bits of the message descriptor (lower 20 bits for **[DevIL+]**) within the SEND instruction for a thread spawn message.

Bit	Description
19	**[DevILK+]: Header Present** This bit must be set to **zero** for all Thread Spawner messages.
18:5	Reserved: MBZ
4	**Resource Select.** This field specifies the resource associated with the action taken by the Opcode. If Opcode is "Spawn thread", this field selects whether it is a child thread or a root thread. 0: spawn a *child* thread 1: spawn a *root* thread or (**[DevCTG]** only) release a PRT_Fence If Opcode == "Dereference Resource", this field indicates whether the URB handle is to be dereferenced. The URB handle can only be dereferenced once. 0: The URB handle is dereferenced 1: The URB handle is NOT dereferenced
3:2	**Reserved:** MBZ
1	**Requester Type.** This field indicates whether the requesting thread is a root thread or a child

Bit	Description
	thread. If it is a root thread, when Opcode is 0, FF managed resources will be dereferenced. If it is a child thread and Opcode is 0, no resource will be dereferenced – basically no action is required by the TS. 0: Root thread 1: Child thread
0	**Opcode.** Indicates the operation performed by the message. A root thread must terminate with a message to TS (Opcode == 0 and EOT == 1). A child thread *should* also terminate with such a message. A thread cannot terminate with an Opcode of “spawn thread”. 0: dereference resource (also used for end of thread) 1: spawn thread

1.8.2.2 Message Payload

DWord	Bit	Description
M0.7	31:0	**Reserved**:
M0.6	31:0	**Reserved**:
M0.5	31:8	Ignored
	7:0	**FFTID.** This ID is assigned by TS and is a unique identifier for the thread in comparison to other concurrent root threads. It is used to free up resources used by a root thread upon thread completion. This field is valid only if the **Opcode** is “dereference resource”, and is ignored by hardware otherwise.
M0.4	31:16	Ignored
	15:10	Dispatch URB Length. Indicates the number of 8-DW URB entries contained in the Dispatch URB Handle that will be dispatched. When spawning a child thread, the URB handle contains most of the child thread's payload including R0 header. When spawning a root thread, the URB handle contains the message passed from the requesting thread to the spawned "peer" root thread. The number of GRF registers that will be initialized at the start of the spawned child thread is the addition of this field and the number of URB constants if present. The number of GRF registers that will be initialized at the start of a spawned root thread is the addition of this field, the number of URB constants if present, and the URB handle received from VFE. This field is ignored if the Opcode is "dereference resource". Length of 0 can be used while spawning child threads to indicate that there is no payload beyond the required R0 header. Length of 0 while spawning a root thread indicates that there is no payload at all from the parent thread. A spawned root has R0 supplied by the Media_Object command indirect/inline data. Format = U6 Range = [0,63] for child threads

DWord	Bit	Description
	9:0	**URB Handle Offset.** Specifies the 8-DW URB entry offset into the URB handle that determines where the associated dispatch payload will be retrieved from when the spawned child or root thread is dispatched. This field is ignored if the **Opcode** is “dereference resource”. Format = U10 Range = [0,1023]
M0.3	31:0	Ignored
M0.2 [DevSNB+]	31:28	Ignored
	27:24	**BarrierID**. This field indicates which one from the 16 Barriers this kernel is associated. Format: U4
	23:16	Reserved
	15:10	Ignored
	9:4	**Interface Descriptor Offset.** This field specifies the offset from the interface descriptor base pointer to the interface descriptor which will be applied to this object. It is specified in units of interface descriptors. Format = U5 [DevSNB]
	3:0	**Scoreboard Color** (only with MEDIA_OBJECT_EX)**:** This field specifies which dependency color the current thread belongs to. It affects the dependency scoreboard control. Format = U4
M0.1	31:0	Ignored
M0.0	31:28	Ignored
	27:24	Reserved: MBZ
	23:16	Reserved : MBZ
	15:0	**Dispatch URB Handle** If **Opcode** (and **Requester Type**) is “spawn a child thread”: Specifies the URB handle for the child thread. If **Opcode** (and **Requester Type**) is “spawn a root thread”: Specifies the URB handle containing message (e.g. requester’s gateway information) from the requesting thread to the spawned root thread. If **Opcode** is “dereference resource”: This field is required on end of thread messages if the **Children Present** bit is set, as the handle must be dereferenced, otherwise this field is ignored.

Revision History

Revision Number	Description	Revision Date
1.0	First 2011 OpenSource edition	May 2011

§§