Intel® OpenSource HD Graphics PRM

Volume 2 Part 2: 3D / Media - Media

For the all new 2010 Intel Core Processor Family Programmer's Reference Manual (PRM)

March 2010

Revision 1.0

Doc Ref #: IHD_OS_V2Pt2_3_10

Revision History

Document Number	Revision Number	Description	Revision Date
IHD_OS_V2Pt2_3_10	1.0	First Release.	March 2010

§§

Contents

§§

1. Media and General Purpose Pipeline

1.1.1 Hardware Feature Map in Products

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

Video Front End Features in Device Hardware

Features/ Device	[DevBW]	[DevCL]	[DevCTG]	[DevILK]
Generic Mode	**Y**	**Y**	**Y**	**Y**
Root Threads	**Y**	**Y**	**Y**	**Y**
Parent/Child Threads	**Y**	**Y**	**Y**	**Y**
SRT (Synchronized Root Threads)	**Y**	**Y**	**Y**	**Y**
PRT (Persistent Root Thread)	**N**	**N**	**Y**	**Y**
Interface Descriptor Remapping	**N**	**N**	**Y**	**Y**
Interface Descriptor Remapping	**N**	**N**	**Y**	**Y**
IS Mode (HW Inverse Scan)	**Y**	**Y**	**Y**	**Y**
VLD Mode (HW MPEG2 VLD)	**N**	**Y**	**Y**	**Y**
AVC MC Mode	**N**	**N**	**Y**	**Y**
AVC IT Mode (HW AVC IT)	**N**	**N**	**Y**	**Y**
AVC ILDB Filter (in Data Port)	**N**	**N**	**Y**	**Y**
VC1 MC Mode	**N**	**N**	**Y**	**Y**
VC1 IT Mode (HW VC1 IT)	**N**	**N**	**Y**	**Y**
Stalling HW Scoreboard	**N**	**N**	**N**	**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).

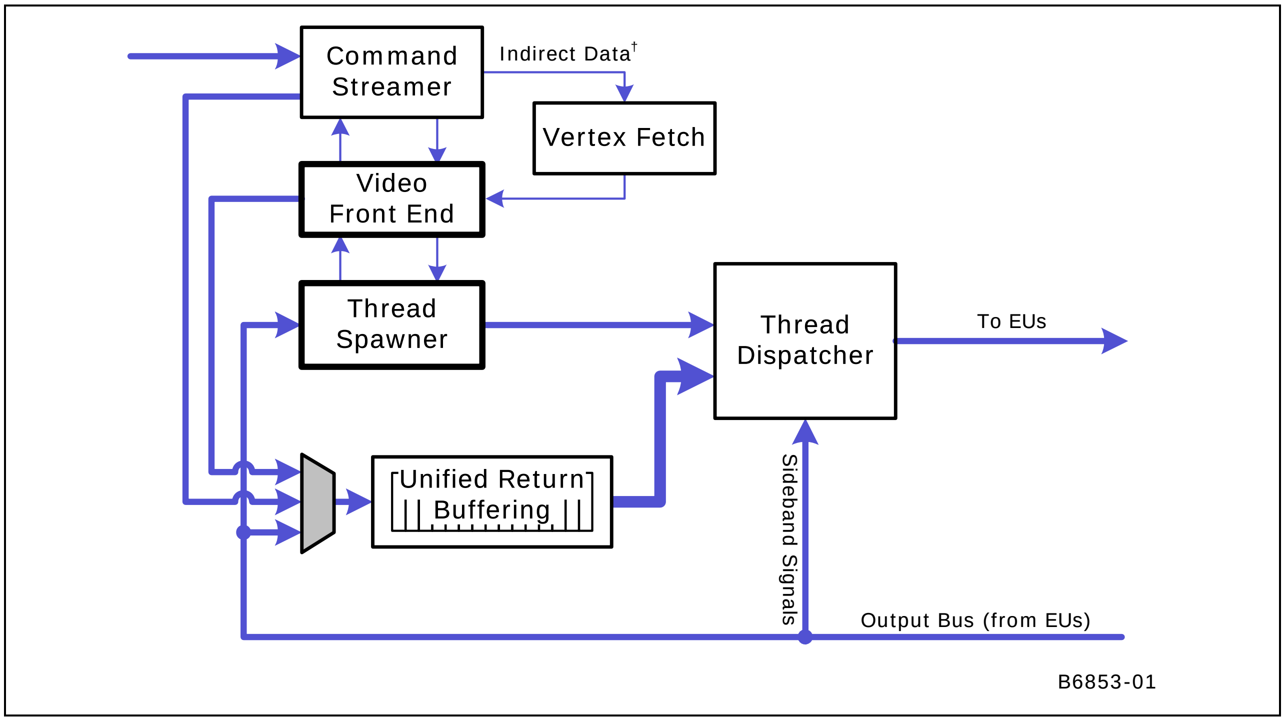
The fixed functions, VFE and TS, in the media pipeline, in most cases, share the same basic building blocks as the fixed functions in the 3D pipeline. However, there are some unique features in media fixed functions as highlighted by the followings.

- 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).

[**DevBW**] **Erratum**: DevBW does not have MPEG-2 VLD hardware. Therefore, software cannot use the VLD mode of the Media_Object command.

[**DevBW-A**] **Erratum**: Using vertex cache in Vertex Fetch unit to speed up streaming of indirect media data load is not available on DevBW-A. On DevBW-A, indirect media data are loaded directly from CS to VFE.

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.

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.

- ## Special Requirements for Each Context Initialization
    - Always initialize the URB fence (with a URB_FENCE command) before the first pipeline select command (PIPELINE_SELECT).
    - Always initialize the pipeline state pointer (with a STATE_BASE_ADDRESS command) before the first pipeline select command.
- Step 1: MI_FLUSH/PIPE_CONTROL
    - This step is mandatory.
    - Programmer may choose not to flush certain caches to improve performance.
    - Multiple such commands in step 1 are allowed, but not recommended for performance reason.
    - [**DevBW-B, DevBW-C, DevCLN**]
        - MI_LOAD_REGISTER_IMM
        - It is used to load an MMIO register to disable the vertex cache for indirect media object load. The register is 0x2124 and the bit is 15.
            - Address = 0x2124
            - Data = 0x10000000
        - MI_FLUSH
        - MI_LOAD_REGISTER_IMM
        - This command is optional for this step. It is only required when indirect object load is used subsequently by MEDIA_OBJECT commands.
        - IF present, it is used to load an MMIO register to enable the vertex cache for indirect media object load. The register is 0x2124 and the bit is 15.
            - Address = 0x2124
            - Data = 0x10001000
- Step 1.5: State commands SF_STATE + URB_FENCE **[Errata: Pre-DevILK]**
    - When switching from 3D context to Media context, the following sequence must be sent before the PIPE_SELECT command.
        - SF_STATE command must be sent down to set the “Number of URB Entries” to “0”.
        - URB_FENCE command must be sent down to set the “URB Fence” for all 3D units to “0”, including CS, VS, GS, CL, ans SF.
- Step 2: 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.
    - If this command is issued, it must be followed by a URB_FENCE command (step 3).

- Step 3: State command URB_FENCE
    - This step is optional. This command can be omitted if URB fence needs not to be changed. However, as mentioned above, if a PIPELINE_SELECT command is issued, this command is then required.
    - If present, only one URB_FENCE command in step 3 is allowed. Hardware behavior is undefined if more than one URB_FENCE commands are issued in this step.
- Step 4: 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 object load.
        - *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.*
    - The following state commands can be issued in arbitrary order.
    - MEDIA_STATE_POINTERS
        - 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.
    - CS_URB_STATE
        - This command is optional for this step. Note that if CS_URB_STATE command is present, there will be at least one MEDIA_STATE_POINTERS command in this step (as mentioned above).
        - Multiple such commands in this step are allowed. The last one overwrites previous ones.
        - If present, “Number of URB Entries” must be 0 if no URB entry is allowed to CS by URB_FENCE command.
        - “Number of URB Entries” must be set to 1 as media pipeline does not support pipelined CONSTANT_BUFFER command (see step 5).
    - STATE_PREFETCH
        - This command is optional for this step.
    - STATE_SIP
        - This command is optional for this step. It is only required when SIP is used by the kernels.

- 3DSTATE_VERTEX_ELEMENTS ([**DevBW-B, DevBW-C, DevCLN**] only. For other products, this command cannot be issued as indirect object load is fully described by each MEDIA_OBJECT command.)
  - This command is optional for this step. It is only required when indirect object load is used subsequently by MEDIA_OBJECT commands.
  - If present, only the following programming is allowed. Hardware behavior with other programming is undefined,
    - Two elements need to be programmed
    - Vertex Element 0
      - Vertex Buffer Index = 0
      - Valid = True
      - Surface Format = 0x002
      - Source Element Offset = 0x0
      - Component Control 0,1,2,3 = 0x1
      - Destination Offset = 0x0
    - Vertex Element 1
      - Vertex Buffer Index = 0
      - Valid = True
      - Surface Format = 0x002
      - Source Element Offset = 0x10
      - Component Control 0,1,2,3 = 0x1
      - Destination Offset = 0x10
- 3DSTATE_VERTEX_BUFFERS ([**DevBW-B, DevBW-C, DevCLN**] only. For other products, this command cannot be issued as indirect object load is fully described by each MEDIA_OBJECT command.)
  - This command is optional for this step. It is only required when indirect object load is used subsequently by MEDIA_OBJECT commands.
  - If present, only the following programming is allowed. Hardware behavior with other programming is undefined,
    - Only 1 vertex buffer
    - Buffer Access Type : Vertex Data
    - Buffer Pitch : 0x20
    - Buffer Start Address : <Indirect Data Address>
      - When VFE is in Generic Mode, the vertex buffer base address can be byte aligned. The restriction is that indirect data size of each MEDIA_OBJECT command must be a multiple of 32 bytes.
      - When VFE is either in VLD mode or IS mode, the indirect data size may not be multiple of 32 bytes (that's OK). However, it is required that the vertex buffer to be programmed to be 32-byte aligned. All indirect data must be included in the vertex buffer programmed.
    - Max index is always set to 0 (ie disabled)

- Step 5: State command CONSTANT_BUFFER
  - This step is optional. However, it is required (as a software workaround) when 3DPRIMITIVE commands are used subsequently to load indirect object data.
  - If present, only one such command is allowed. Hardware behavior is undefined if more than one CONSTANT_BUFFER commands are issued in the program sequence without a FLUSH in between.
- Step 6: Primitive commands
  - 3DPRIMITIVE ([**DevBW-B, DevBW-C, DevCLN**] only. For other products, this command cannot be issued as indirect object load is fully described by each MEDIA_OBJECT command.)
    - This command is optional for this step. It is only required when indirect object load is used subsequently by MEDIA_OBJECT commands.
    - If present, this command must precede one or many MEDIA_OBJECT commands. If more than one MEDIA_OBJECT commands are followed, the indirect object data for these commands must be stored in memory contiguously (with certain 32-byte aligned overlaps allowed, see XXX for details).
    - If present, only the following programming is allowed. Hardware behavior with other programming is undefined,
      - Sequential access for the Vertex Buffer
      - Primitive topology type is 0x01h = PointList
      - Vertex Count per instance = Size of the block to transfer for the media indirect command in 32 byte quantities.
      - Start Vertex Location = 0
      - Instance Count = 1
      - Start instance Location = 0
      - Base Vertex Location = 0
  - 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 6 can be issued to continue processing media primitives.

Programming Notes on Improving Indirect Media Object Load Performance [DevBW-C, DevCL]: *The large vertex cache is used to stream indirect media object loads for one or many MEDIA_OBJECT commands. By grouping multiple such commands together significant streaming performance can be achieve. Here is an example.*

- *1 Vertex Buffer programmed with 2 Vertex components*
- *Vertex format is fixed to A32R32B32G32_UINT, this format is left untouched by the vertex fetch*

Here the number of vertices equal to the total size to be transferred for MEDIA_OBJECT commands in 32-byte chunks If the first MEDIA_OBJECT command transfers indirect data size of 4 64 byte quantities, the number of vertices would be 8, If the second MEDIA_OBJECT command to be transferred and the total size is 8 64 byte quantities, number of vertices is 16.

However, using the Vertex Buffer in sequential mode as described above does post a restriction that data for multiple MEDIA_OBJECT commands sharing the same 3DPRIMITIVE command must be stored in memory sequentially. When data are not stored sequentially in memory, there are several approaches as listed below. Certain experiments may be required in order to find which approach provides the best performance for a given application.

- *Preceding each MEDIA_OBJECT command with one 3DPRIMITIVE command.*
- *Grouping several 3DPRIMITIVE commands together followed by the MEDIA_OBJECT commands using the fetched data from the 3DPRIMITIVE commands.*
- *Using indexed vertex buffer to gather indirect media object data from non-contiguous memory locations.*

As a side effect, when vertex cache is used for media indirect object load, the statistical counters in the VF unit may be affected during media operations. When 3D operations and media operations are from different contexts, this side effect is not an issue as the statistical counters are context save/restored. However, if 3D and media operations are mixed within one context, it is advisable to turn off the statistical counters before entering media operation (using vertex cache for indirect object load) and turn them back on before returning to 3D operations. This can be achieved using the 3DSTATE_VF_STATISTICS command.

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.

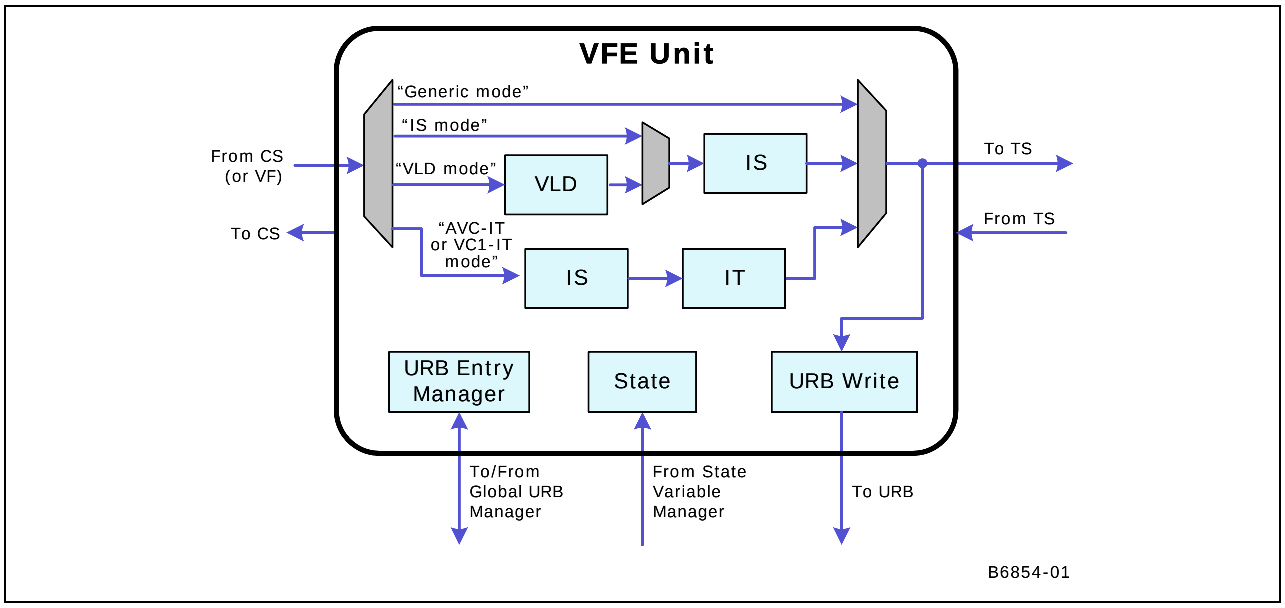
VFE supports three modes of operation: Generic mode, Inverse Scan mode and VLD mode.

- **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.

- **IS (Inverse Scan) mode**: The IS mode is a special mode for video decoding when off-host IDCT acceleration is supported by kernels running on GENx execution units.
- **VLD mode**: It is a special mode for video decoding when MPEG-2 off-host VLD acceleration is supported by GENx hardware.
- **[DevCTG, DevILK], AVC-IT (AVC Inverse Integer Transform) mode**: The AVC-IT mode is a special mode for AVC video decoding when off-host IDCT acceleration is supported by VFE hardware and MC and Loop Filter are supported by kernels running on GENx execution units.
- **[DevCTG, DevILK] AVC-MC (AVC Motion Compensation) mode**: The AVC-MC mode is a special mode for AVC video decoding with host-based IDCT and MC and Loop Filter are supported by kernels running on GENx execution units.
- **[DevCTG, DevILK] VC1-IT (VC1 Inverse Integer Transform) mode**: The VC1-IT mode is a special mode for VC1 video decoding when off-host IDCT acceleration is supported by VFE hardware and MC and Loop Filter are supported by kernels running on GENx execution units.

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

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.

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.

1.4.2.2 IS Mode

In Inverse Scan (IS) mode, the Inverse Scan unit is used. In particular, GENx architecture can be used to support off-host IDCT acceleration for MPEG-2.

In this mode, a new thread is generated for each MEDIA_OBJECT command. One MEDIA_OBJECT command corresponds to a macroblock. The indirect payload in the command contains the non-zero DCT coefficients for all coded blocks in the macroblock. Detailed data format can be found in section 1.7.2.2.

The indirect payload is streamed into the IS unit. IS unit process the non-zero DCT coefficients on a block by block basis. The 16-bit non-zero coefficients are placed in its location within an 8x8 block according to the (x,y) addresses. Hardware fills the rest of the coefficients in the block to zero. The assembled DCT data blocks are then written into URB.

Note that the index for a non-zero coefficient is in row-major order (x, y) address. Host software is responsible of converting the coding scan order to this unified row-major order (e.g. zig-zag scan or alternative scan order such as vertical scan found in MPEG-2 or other coding standard).

Blocks that are not coded will not have coefficient data in the message to the kernel, and the coded blocks are packed back to back. As the message size is variable, VFE calculates the final message size according to the coded block pattern field before sending it to TS.

Interface descriptor select and scratch memory allocation are handled in the same way as in Generic mode.

1.4.2.3 VLD Mode

In VLD mode, both the VLD unit and the IS unit in VFE are used. The VLD unit is specifically designed to support MPEG-2 variable length decoding; it does not support other standards such as WMV. Each MEDIA_OBJECT command contains compressed bitstream data associated with a slice. A slice, according to MPEG-2 compressed video bitstream syntax is the smallest unit that marked by byte-aligned start-code, allowing easy parsing by host software. A slice contains one or more macroblocks. Unlike the other two modes, one or many threads may be generated for each MEDIA_OBJECT command. Each thread corresponds to a macroblock. The indirect payload in the MEDIA_OBJECT command contains the bitstream data for a slice. The indirect payload is streamed into the VLD unit. The decoded non-zero coefficients are then sent to the IS unit. And then the IDCT data blocks (8x8 size) output from the IS unit are then written into URB. For each macroblock, VFE generates the interface descriptor pointer based the decoded macroblock type.

VFE partially decodes (VLD and IS) the MPEG-2 bitstream for a slice and assembles resulting data on a macroblock by macroblock basis for threads running on EUs to complete the rest of the work. The macroblock-based thread (referred to as a post-VLD thread hereafter) performs inverse quantization, inverse DCT, and motion compensation in order to generate the final output picture. VFE also handles skipped macroblocks so that each post-VLD thread operates on one and only one macroblock.

For bitstreams that are MPEG-2 standard compliant, the output from the VFE fixed function hardware is bit accurate. Bit precision difference may be caused by the IDCT implementation in the kernel. The IDCT kernel must meet the IEEE standard requirement for IDCT.

For bitstreams that are not MPEG-2 standard compliant due to, for example, data corruption, output from VFE fixed function may be unpredictable. That may result in data corruption in the destination buffer after kernel operation. However, VFE fixed function will continue functioning (without hanging).

VFE decodes the slice through the following three major stages: variable length decode, inverse scan and output formatting.

1.4.2.3.1 Variable Length Decode

Variable Length Decode (VLD) stage contains the following sub-stages: data parser, symbol decoder, and motion vector (MV) predictor.

Data Parser

Slice data are processed a dword at a time. Using the byte offset and bit offset provided by the MEDIA_OBJECT command, data parser determines the start bit and sends the slice data to the decoding stage.

Data parser tracks the length of the slice, which is provided by the MEDIA_OBJECT command. Data parser uses the slice length and the starting offsets to calculate the end of slice. When the end of slice is reached, data parser indicates end of slice to symbol decoder and does not pass on any more data that comes from the command stream until a new slice begins.

Symbol Decoder

Symbol decoder performs variable length decoding of the slice bitstream according to the MPEG-2 standard. The decoder analyzes symbols in the bitstream and separates them for further processing. For example, motion vector differentials are sent to motion vector predictor but DCT coefficients are sent directly to IS stage.

Motion Vector Predictor

Motion Vector (MV) Predictor calculates the motion vectors based on the motion vector differentials received from symbol decoder and the motion vector prediction values maintained within MV Predictor, updates the motion vector prediction values accordingly and performs additional arithmetic for dual prime motion vectors to convert them to uni/bi-directional motion vectors. The output motion vectors are relative to the current macroblock position.

1.4.2.3.2 Inverse Scan

IS unit process the non-zero DCT coefficients with their (x, y) location within an 8x8 block received from VLD on a block by block basis. For each new block of data, IS initializes the 8x8 block storage to zero. For each non-zero coefficient received from VLD, IS first sign-extend it to a 16-bit signed value and then place it in the block storage at the location identified by its (x, y) address. When the end of block signal is received from VLD, IS writes the assembled DCT data block into URB.

Only the coded blocks are assembled in the URB, and they are assembled back to back. As the thread payload size is variable, VFE calculates the final message size according to the coded block pattern field before sending the payload size to TS.

1.4.2.3.3 Output Formatting

Additional functionality after inverse scan formats the data that is sent as thread payload to the kernel. Some of this functionality, such as expansion of skip macroblocks and determination of second P field, is done by hardware to make the kernel more efficient.

Skip Macroblocks

VFE processes skip macroblocks by separating them into individual macroblocks and forming one thread for one macroblock. For each skip macroblock, hardware sets its coded block pattern to 0, indicating that no error data is present. All contiguous skip macroblocks have the same relative motion vector. Hardware also handles the difference of skipped macroblocks in a P picture or a B picture as defined by MPEG-2 specification. According to MPEG-2 specification, skip macroblocks cannot extend beyond the end of the current line.

Second Field

According to MPEG-2 specification, field prediction for a P field picture uses the most recently decoded two fields as reference, namely, the most recently decoded reference top field and the most recently decoded reference bottom field. When the current P field is the first field of a frame, both of its reference fields come from the same frame. When the current P field is the second field of a frame, one of its reference fields comes from the same frame.

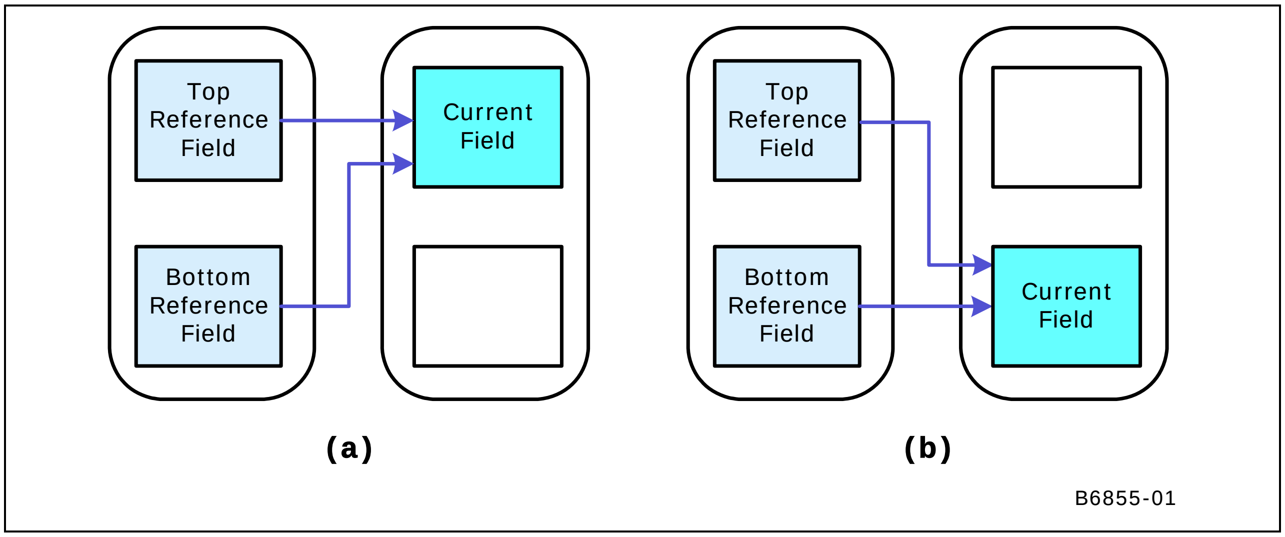
Detecting second field is important if reference frame selection is required. This is no longer true for GEN4 as each reference field is specified by unique binding table index. Each binding table index contains the pointer to the surface state, which contains not only the field indication but also the base address of the frame buffer. Therefore, it is up to the kernel developer to determine whether to use the Second Field information provided by the hardware.

VFE sets the second field indicator under the following conditions:

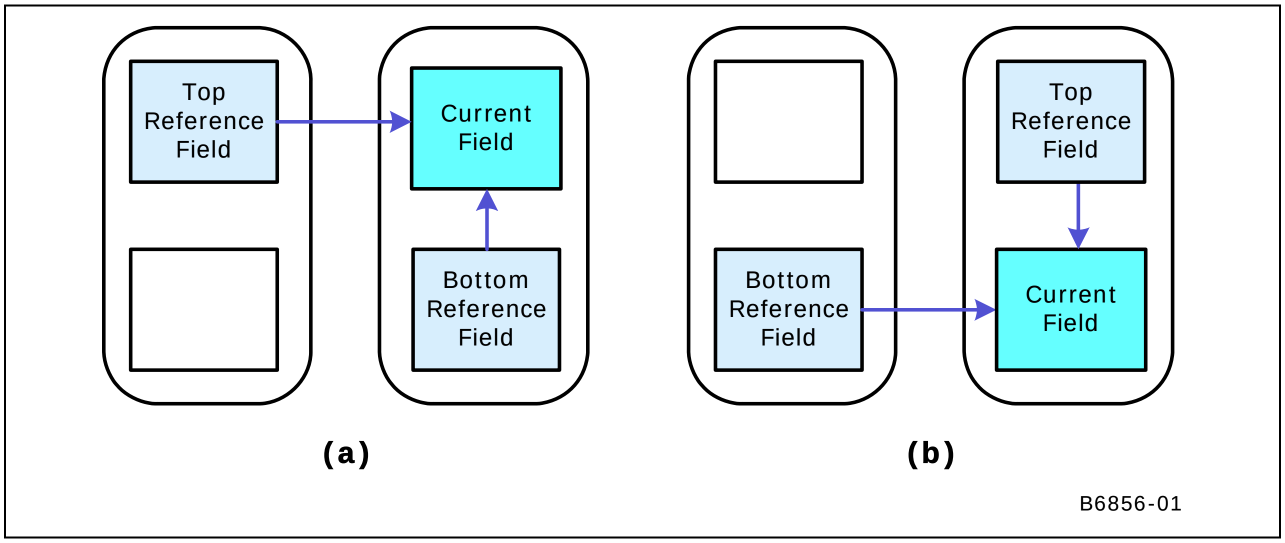
- Picture coding type is P picture
- Destination format is field
- Motion type may be field, 16x8 or dual prime
- Either:
    - Top field is first, and
    - Current field is field 1, and

  OR
    - Top field is not first, and
    - Current field is field 0, and

Prediction for a P field picture that is a first field, which is (a) a top field, or (b) a bottom field.



Prediction for a P field picture that is a second field, which is (a) a top field, or (b) a bottom field



1.4.2.3.4 Handling Motion Vectors

The Table below provides a summary of different motion types and associated properties. For Frame_Motion_Type, there are three types of Prediction_Type: frame-based prediction, field-based prediction and dual-prime prediction. For Field_Motion_Type, there are three types of Prediction_Type: field-based prediction, dual-prime prediction and 16x8 prediction.

The second table below details the motion compensation operations for various frame motion types and the third table depicts the motion compensation operations for various field motion types.

Summary of Motion Types

*_Motion_Type	Prediction_Type	Vector [r][s]	Possible MV Combinations in Bitstream	Uses Motion Vertical Field Select
Frame	Frame-based	[0][0] – 0 [0][1] – 1 [1][0] – 2 [1][1] – 3	None,0,1,0+1	No
Frame	Field-based	[0][0] – 0 [0][1] – 1 [1][0] – 2 [1][1] – 3	0+2,1+3, 0+1+2+3	Yes
Frame	Dual-Prime	[0][0] – 0 [0][1] – 1 [1][0] – 2 [1][1] – 3 [2][0] – 4 [3][0] – 6	0+4+6 (See Frame-Dual Prime table)	No
Field	Field-based	[0][0] – 0 [0][1] – 1 [1][0] – 2 [1][1] – 3	None, 0, 1, 0+1	Yes
Field	Dual-Prime	[0][0] – 0 [0][1] – 1 [1][0] – 2 [1][1] – 3 [2][0] – 4	0+4 (See Field-Dual Prime table)	Yes
Field	16x8	[0][0] – 0 [0][1] – 1 [1][0] – 2 [1][1] – 3	0+2, 1+3,0+1+2+3	Yes

*Vectors 4 and 6 are the derived motion vectors (DMVs) for dual-prime prediction that are calculated by PR and placed in the thread payload in the specified MVector position.

Motion Comp Operation for Pictures with Frame Motion Type

frame_motion_type	forward	backward	intra	Motion vector (v'[r][s][t])	Command	HW Mvector (MV[r][s])	**Prediction Map**	Prediction formed for	MVFS
Frame-based‡	-	-	1	v'[0][0][1:0]	0	-	-	None (motion vector is for concealment)	-
Frame-based	1	1	0	v'[0][0][1:0]	2	MV[0][0]	Fwd	frame, forward	-
				v'[0][1][1:0]		MV[0][1]	Back	frame, backward	-
Frame-based	1	0	0	v'[0][0][1:0]	2	MV[0][0]	Fwd	frame, forward	-
Frame-based	0	1	0	v'[0][1][1:0]	2	MV[0][1]	Back	frame, backward	-
Frame-based‡	0 (1)	0	0	v'[0][0][1:0]*§	2	MV[0][1]	Fwd	frame, forward	-
Field-based	1	1	0	v'[0][0][1:0]	4	MV[0][0]	Fwd	top field, forward	[0][0]
				v'[1][0][1:0]		MV[1][0]	Fwd	bottom field, forward	[1][0]
				v'[0][1][1:0]		MV[0][1]	Back	top field, backward	[0][1]
				v'[1][1][1:0]		MV[1][1]	Back	bottom field, backward	[1][1]
Field-based	1	0	0	v'[0][0][1:0]	4	MV[0][0]	Fwd	top field, forward	[0][0]
				v'[1][0][1:0]		MV[1][0]	Fwd	bottom field, forward	[1][0]
Field-based	0	1	0	v'[0][1][1:0]	4	MV[0][1]	Back	top field, backward	[0][1]
				v'[1][1][1:0]		MV[1][1]	Back	bottom field, backward	[1][1]
Dual prime	1	0 (1)	0	v'[0][0][1:0]	4	MV[0][0]	Fwd	top field, from same parity, forward	[0][0] = 0
				v'[0][0][1:0]		MV[1][0]	Fwd	bottom field, from same parity, forward	[1][0] = 1
				v'[2][0][1:0]*†		MV[0][1]	Fwd	top field, from opposite parity, forward	[0][1] = 1
				v'[3][0][1:0]*†		MV[1][1]	Fwd	bottom field, from opposite parity, forward	[1][1] = 0

NOTE -	Motion vectors are listed in the order they appear in the bitstream
?	the motion vector is only present if concealment_motion_vectors is one
‡	frame_motion_type is not present in the bitstream but is assumed to be Frame-based
*	These motion vectors are not present in the bitstream
†	These motion vectors are derived from vector'[0][0][1:0] as described in 7.6.3.6
§	The motion vector is taken to be (0; 0) as explained in 7.6.3.5

Motion Comp Operation with Field Motion Type

							Prediction Map (SecondPField|BottomField)					
field_motion _type	forward	backward	intra	Motion vector	Command	HW MVector	00	01	10	11	Prediction formed for	
Field-based‡	-	-	1	v'[0][0][1:0]?		None	N/A	-	-	-	None (motion vector is for concealment)	
Field-based	1	1	0	v'[0][0][1:0]	2	MV[0][0]	Fwd	Fwd	x	x	whole field, forward	B-Pict only
				v'[0][1][1:0]		MV[0][1]	Back	Back	x	x	whole field, backward	
Field-based, P picture	1	0	0	v'[0][0][1:0]	2	MV[0][0]	Fwd (M0)	Fwd (M0)	Fwd (M0) if MVFS[0][0]=0	Dst (M1) if MVFS[0][0]=0	whole field, forward	
									Dst (M1) if MVFS[0][0]=1	Fwd (M0) if MVFS[0][0]=1		
Field-based, B picture	1	0	0	v'[0][0][1:0]	2	MV[0][0]	Fwd	Fwd	x	x	whole field, forward	
Field-based	0	1	0	v'[0][1][1:0]	2	MV[0][1]	Back	Back	x	x	whole field, backward	B-Pict only
Field-based‡	0 (1) %	0	0	v'[0][0][1:0]*§	**2**	MV[0][0]	Fwd	Fwd	Fwd w/ MVFS[0][0]=0	Fwd w/ MVFS[0][0]=1	whole field, forward	P-Pict only, Same parity.
16x8 MC	1	1	0	v'[0][0][1:0]	4	MV[0][0]	Fwd	Fwd	x	x	upper 16x8 field, forward	B-Pict only
				v'[1][0][1:0]		MV[1][0]	Fwd	Fwd	x	x	lower 16x8 field, forward	
				v'[0][1][1:0]		MV[0][1]	Back	Back	x	x	upper 16x8 field, backward	
				v'[1][1][1:0]		MV[1][1]	Back	Back	x	x	lower 16x8 field, backward	
16x8 MC	1	0	0	v'[0][0][1:0]	4	MV[0][0]	Fwd	Fwd	Fwd (M0) if MVFS[0][0]=0	Dst (M1) if MVFS[0][0]=0	upper 16x8 field, forward	
									Dst (M1) if MVFS[0][0]=1	Fwd (M0) if MVFS[0][0]=1		
				v'[1][0][1:0]		MV[1][0]	Fwd (M0)	Fwd (M0)	Fwd (M0) if MVFS[1][0]=0	Dst (M1) if MVFS[1][0]=0	lower 16x8 field, forward	
									Dst (M1) if MVFS[1][0]=1	Fwd (M0) if MVFS[1][0]=1		
16x8 MC	0	1	0	v'[0][1][1:0]	4	MV[0][1]	Back	Back	x	x	upper 16x8 field, backward	B-Pict only
				v'[1][1][1:0]		MV[1][1]	Back	Back	x	x	lower 16x8 field, backward	
Dual prime	1	0 (1) %	0	v'[0][0][1:0]	2	MV[0][0]	Fwd (M0)	Fwd (M0)	Fwd (M0) w/ MVFS[0][0]=0	Fwd (M0) w/ MVFS[0][0]=1	whole field, from same parity, forward	P-Pict only (SW forces MVFS[0][0], MVFS[0][1])
				v'[2][0][1:0]*†		MV[0][1]	Fwd (M1)	Fwd (M1)	Dest (M1) w/ MVFS[0][1]=1	Dest (M1) w/ MVFS[0][1]=0	whole field, from opposite parity, forward	

Notes: Motion vectors are listed in the order they appear in the bitstream.

? — The motion vector is only present if concealment_motion_vectors is one.

‡ — Field_motion_type is not present in the bitstream but is assumed to be Field-based.

* — These motion vectors are not present in the bitstream.

† — These motion vectors are derived from vector'[0][0][1:0] as described in 7.6.3.6.

§ — The motion vector is taken to be (0; 0) as explained in 7.6.3.5.

% — Software converts the motion type. For Dual-prime case, software converts it to a bidirectional prediction.

Remarks:

- Forward, Backward MV pairs always used for combined prediction (Bi-dir or Dual-Prime)
- MVs [0][0] and [0][1] can be to Fwd reference, Backward reference, or Destinsation buffer (for 2nd field case)
- MV [1][0] can be to a Fwd or Dest
- MV [1][1] is always to a Back
- (M0), (M1) stands for MIP_INFO setting for the first reference picture and the second reference picture. It is particularly important to set correct M1 for P-pictures to deal with SecondField and DualPrime cases.
- Software converts the dual-prime case to a field-based bidirectional prediction with 2 MVs.

1.4.2.3.5 Dual Prime Handling

Dual prime prediction is only valid for a P-picture. In dual prime mode, each field will have two predictions similar to the forward and backward predictions in a B-picture, as the final prediction value for the field is the average of the two. One of the motion vectors is provided by the bitstream and the other one is derived. Motion Vector Predictor unit is responsible for converting all dual prime predictions to a forward and backward field prediction according the for P frame picture and for P field picture.

Converting Frame-Dual Prime Motion to 4MV

Prediction formed for: Field / Parity	MPEG-2 MV[r][s]	Thread Payload MV[r][s]	Motion Vertical Field Select (MVFS)	
Top / Same	[0][0]	[0][0]	Bit 0 = 0	MVFS = 6h
Bottom / Same	[0][0]	[1][0]	Bit 2 = 1	
Not Used	[0][1]	-		
Not Used	[1][0]	-		
Not Used	[1][1]	-		
Top / Opposite	[2][0]	[0][1]	Bit 1 = 1	
Bottom / Opposite	[3][0]	[1][1]	Bit 3 = 0	

Converting Field-Dual Prime Motion to 2MV

Prediction formed for: Field / Parity	MPEG-2 MV[r][s]	Thread Payload MV[r][s]	MotionVerticalFieldSelect	
			Top Field	Bottom Field
Whole field / Same	[0][0]	[0][0]	Bit 0 = 0	Bit 0 = 1
Whole field / Opposite	[2][0]	[0][1]	Bit 1 = 1	Bit 1 = 0

1.4.2.3.6 Interface Descriptor Selection

In VLD mode, the Interface Descriptor Offset field in the MEDIA_OBJECT command is ignored by hardware. Instead, the interface descriptor offset is computed by hardware based on the decoded macroblock parameters and a remapping table.

First a macroblock index is computed based on parameters such as picture structure, motion type, prediction type, DCT type, intra-coding type and motion vector present information. The table below provides the macroblock index table for a frame-picture destination buffer (with Picture Structure = 11). The next table shows macroblock indices for a field-picture destination buffer (with Picture Structure = 01 or 10). As Picture Structure is a state variable that will not be changed until a pipeline flush, the macroblock indices can be computed separately for different Picture Structure.

After the macroblock index is computed, it is used as the index into the Interface Descriptor Remap Table to derive the final interface descriptor offset value. The Interface Descriptor Remap Table is provided as part of the VLD state.

The interface descriptor offset value multiplied by the interface descriptor size is then added to the interface descriptor base pointer to generate the interface descriptor pointer for the post-VLD thread.

The last three columns in the third table indicate whether a macroblock index is applicable for a given Picture Coding type (I, P or B). A ‘Y’ (or a ‘N’) means the macroblock index on the row is valid (or invalid) for the Picture Type shown on the column. Taking a frame picture destination for example, only macroblock indices 0 and 8 are valid for an I-picture; indices 0-3 and 8-11 are valid for a P-picture; and for a B-picture, only indices 3 and 11 are not valid.

Developers can use the remap table for kernel development to fine-tune system performance and reduce software complexity. For example, if the destination is a frame picture, the kernel for a macroblock with dual-prime motion in a P-picture (macroblock index = 3) may be identical to that for a macroblock with bidirection field motion in a B picture (macroblock index = 7). A common set of interface descriptors can be configured once for frame picture destinations, and reused without change when the destination is of I-, P- and B- picture coding type.

In another case, if it is determined that kernel software will be responsible of handling DCT types for a frame picture destination, then macroblock index *i* and *i*+8 , for *i* = 0 to 7, can be mapped to the same interface descriptor.

Macroblock indices for frame picture destination

Macroblock Index	Interface Descriptor Kernel Function (Frame Picture Destination)	I	P	B
0	I macroblock	Y	Y	Y
1	Forward frame motion	N	Y	Y
2	Forward field motion	N	Y	Y
3	P picture, dual-prime motion	N	Y	N
4	Backward frame motion	N	N	Y
5	Backward field motion	N	N	Y
6	Bidirectional frame motion	N	N	Y
7	Bidirectional field motion	N	N	Y
8	I macroblock w/ field DCT	Y	Y	Y
9	Forward frame motion w/ field DCT	N	Y	Y
10	Forward field motion w/ field DCT	N	Y	Y

Macroblock Index	Interface Descriptor Kernel Function (Frame Picture Destination)	I	P	B
11	P picture, dual-prime motion w/ field DCT	N	Y	N
12	Backward frame motion w/ field DCT	N	N	Y
13	Backward field motion w/ field DCT	N	N	Y
14	Bidirectional frame motion w/ field DCT	N	N	Y
15	Bidirectional field motion w/ field DCT	N	N	Y

Macroblock indices for field picture destination

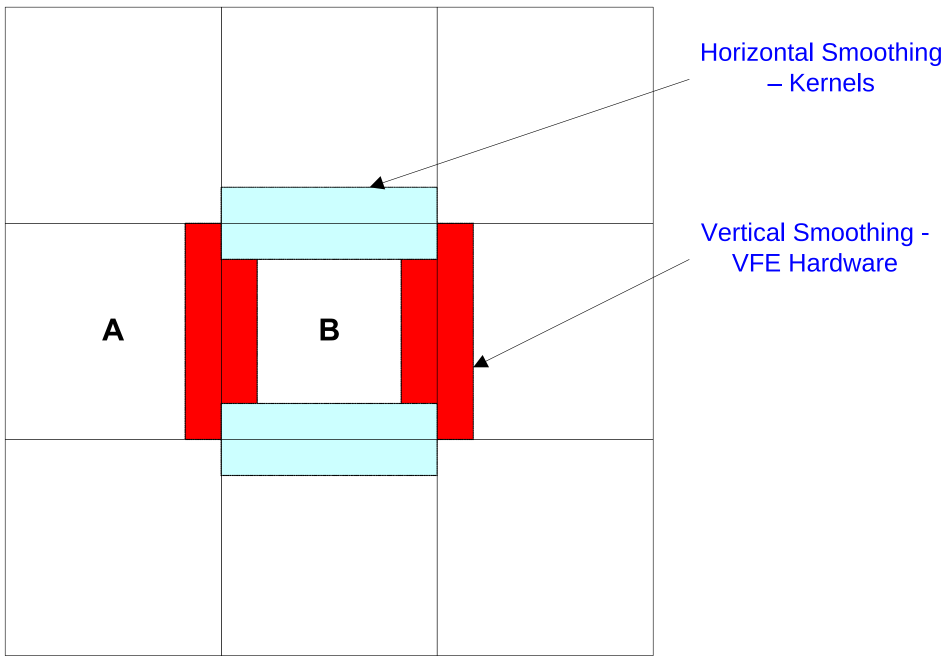
Macroblock Index	Interface Descriptor Kernel Function (Field Picture Destination)	I	P	B
0	I macroblock	Y	Y	Y
1	Forward field motion	N	Y	Y
2	Forward 16x8 motion	N	Y	Y
3	P picture, dual-prime motion	N	Y	N
4	Backward field motion	N	N	Y
5	Backward 16x8 motion	N	N	Y
6	Bidirectional field motion	N	N	Y
7	Bidirectional 16x8 motion	N	N	Y

1.4.2.4 VC1-IT Mode [DevCTG, DevILK]

In VC1-IT mode, Inverse Transform is performed by VFE hardware and Motion Compensation is done by GEN4 kernels. Overlap Smoothing is performed jointly by VFE hardware and GEN4 kernels.

1.4.2.4.1 Overlap Smoothing

One unique feature required by VC1 standard is the Overlap Smoothing operation. Overlap Smoothing involves filtering of the edges of two neighbor Intra blocks after Interface Transform. A hybrid solution is employed in GEN4 architecture with vertical edge filtering done by VFE hardware and horizontal edge filtering done by kernels, where ‘A’ and ‘B’ are two (8x8) blocks. The shaded areas around Intra block ‘B’ are filtered if the corresponding neighbor block is also an Intra block. Information regarding whether an edge requires filter is carried in the VC1-IT in-line data.


Horizontal Smoothing – Kernels
Vertical Smoothing - VFE Hardware
A
B

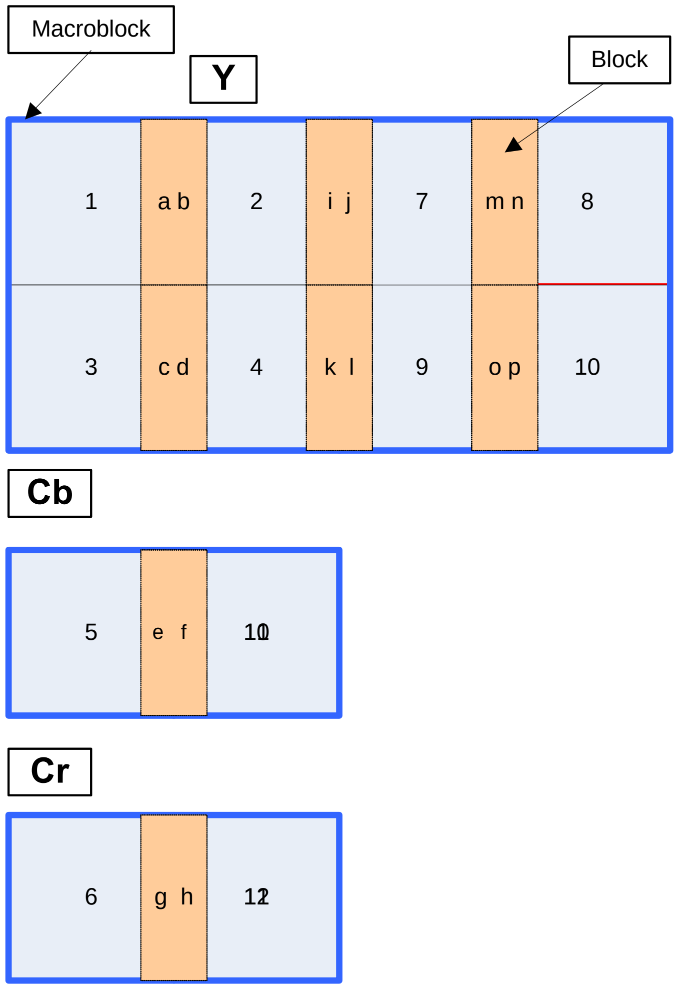
Overlap Smoothing along the edges of neighboring Intra blocks

As each MEDIA_OBJECT_EX command corresponds to one macroblock, vertical edge filtering within a macroblock can be performed in stream by VFE. For edges cross macroblocks, VFE stores a 2x8 strip for each block along the macroblock right-hand-side edges and uses the stored strips for blocks from the next macroblocks.

The next diagram shows an example of processing two adjacent macroblocks. The blocks are delivered to VFE in order shown below from 1 to 12. The shaded strips, namely ‘a’ to ‘p’, are 2x8 block boundaries that need to be filtered. The edges within a macroblock boundary are filtered when Inverse Transform is processed for the current macroblock. The edges of a macroblock are stored in VFE and modified when the next macroblock is transformed and then written to the URB. Note that VFE stores four right-most block edge strips for the current macroblock (*i*,*k*,*e*,*g*).

In summary, VFE hardware has a delayed operation of Overlap Smoothing such that at the end, each block is still output in 8x8 quantity with its left and right edges filtered. Storing such edge strips means that VFE maintain an internal state, which is not saved or restored during context switch. This has the following implications

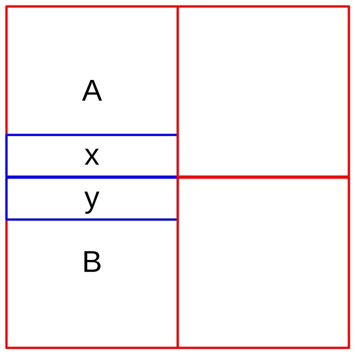
- Software must deliver the macroblocks in row-major order (the natural macroblock decoding order) whenever Overlap Smoothing is turned on.
- Software must avoid context switch between two Intra macroblocks.



Overlap Smoothing vertical filter operation by VFE

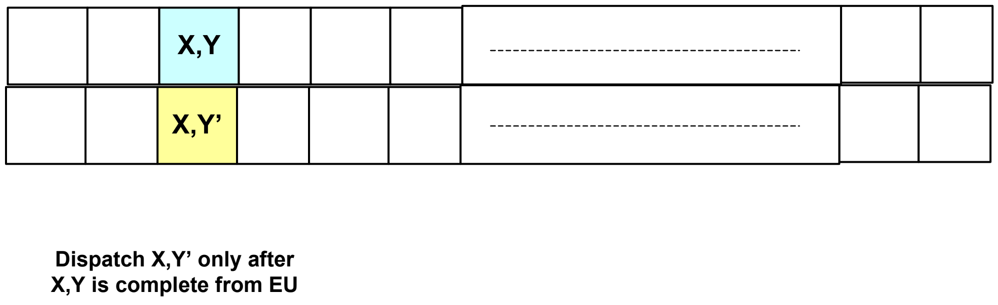
The Overlap Smoothing for horizontal edges is performed by kernels in the EU. Indicators of the edges to be smoothing are sent as control in the in-line data. As higher precision (16-bit) are required to store the edge pixels comparing to the 8-bit precision for the final destination, a separate surface must be used.

When processing a macroblock that contains the block with pixels in region 'A' and region 'x', the kernel may write the final pixel data (in 8-bit precision) in region A to the destination surface, while keeping a copy of the pixel data in region 'x' (in 16-bit precision) to a difference buffer, as pixels in region 'x' may or may not require Overlap Smoothing depending on future macroblocks containing pixels in regions 'y' and 'B'. Later on, Macroblock containing region B is processed. If Overlap Smoothing is enabled across the horizontal edges between A and B, the kernel (processing B) reads data from region x, performs overlap smoothing on x and y blocks and writes macroblock B (including region 'y') to the destination buffer and updates the two rows for macroblock A as well. The LastRowFlag in the VC1-IT inline data can be used by kernel to know if the current macroblock is in the last row and therefore should write the last two rows of pixels to the destination surface.



Overlap Smoothing horizontal filter operation by kernels

In addition, kernels must also take care of the thread-to-thread dependencies between vertically adjacent macroblocks. As shown in XXX, if macroblock at location (X,Y') requires Overlap Smoothing filtering on its upper edge, it can not proceed Overlap Smoothing operation until the thread operating on macroblock at (X, Y) has terminated with output data reached coherent space.



Thread-thread Dependency in Overlap Smoothing horizontal filter operation by kernels

1.4.3 Scoreboard Control [DevILK+]

A hardware mechanism is provided to control the dispatch of root threads. Without using this hardware mechanism, only the dispatch of a SRT is managed by a parent root thread using the SRT message to TS.

There is a scoreboard hardware in TS unit. The scoreboard is addressed by the 18-bit (X, Y) scoreboard field in VFE DWord, where (X, Y) is typically used as the Cartesian coordinate of the working unit in a 2D frame but may be interpolated in other ways. When a root thread is dispatched, the entry at (X, Y) is marked. When the root thread is terminated, the corresponding bit in the scoreboard is cleared.

Each root thread may have up to eight dependencies. The dependency relation is described by the state value of Scoreboard Controls in terms of related distance of (deltaX, deltaY). There is a global scoreboard enabling in the state as well as the-per thread enabling for each dependency.

TS stalls the dispatch of a root thread if any scoreboard entry, which is denoted by (Scoreboard X + deltaX, Scoreboard Y + deltaY), matching with any enabled dependencies is marked as in-flight. The thread is dispatched only after all dependencies are cleared.

For a root thread, TS only stalls the dispatch of the thread only if the dependent scoreboard entries of the thread marked. It does not automatically stalls the dispatch for destination collision if (deltaX = 0, deltaY=0) is not set in the scoreboard state. This kind of scoreboard destination collision is due to the scoreboard wrap-around (or aliasing), which must be avoided. With 9-bit per X, Y field, the hardware scoreboard can support a frame that is subdivided up to 512x512 threads without a scoreboard aliasing.

In addition to the above 'stalling scoreboard', Media Pipe may also support a non-stalling scoreboard. With the non-stalling, a thread is dispatched with the dependent threads marked. The thread dependency affects the issuing of a SENDC instruction. See ISA chapter for details.

Scoreboard Support in Device Hardware

Device	Stalling scoreboard	Non-Stalling scoreboard
[DevILK]	Yes	No

Restrictions:

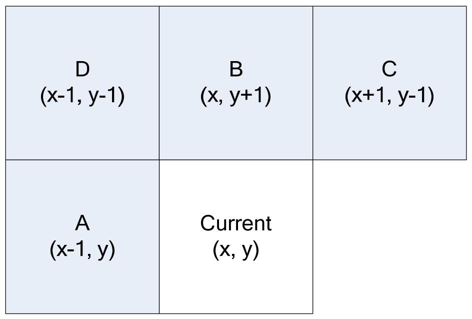
- *The hardware scoreboard only handles root threads, but not child threads. This limitation may be revisited when future application requirement changes.*
- *The usage of hardware scoreboard and SRT are maturely exclusive. In other words, when hardware scoreboard is used, SRT should not be issued.*

1.4.3.1 AVC-Style Dependency Example

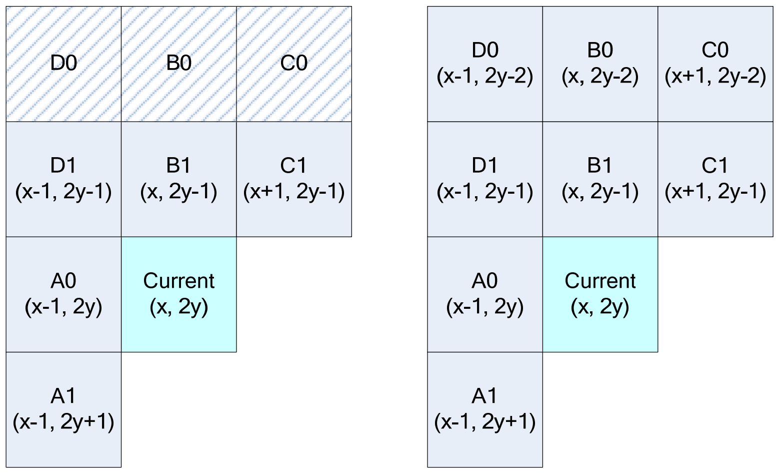
For AVD decoding, dependencies for a given macroblock may be set based on the availability of neighbor macroblocks, namely A, B, C, D and left-bottom neighbors (left-bottom only if MbAff = 1), as well as the current macroblock's address, MbAff flag and FieldMbFlag. For a macroblock in a progressive frame picture or a field picture, one macroblock may depend on up to four neighbors, A, B, C and D. For a macroblock in a MbAff pair, it may depend on up to three, five or eight neighbors, based on the current macroblock's address and FieldMbFlag.

The neighbor's availability depends on the corresponding **IntraPredAvailFlagA|B|C|D|E** flags for the macroblock (or the macroblock pair). Hardware assumes that the flags are set correctly in the MEDIA_OBJECT_EX command. For simplicity, the left neighbor pair (A0 and A1) availability for a MbAff macroblock can be determined as a group by **IntraPredAvailFlagA | IntraPredAvailFlagE**. For the second macroblock in a 'frame' MbAff pair, it depends on the first macroblock in the pair and it is always available.

Neighbor addresses of a macroblock in a progressive frame picture (MbAff = 0) or a field picture with up to 4 dependencies



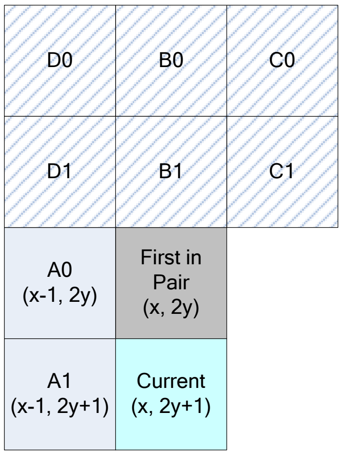
Neighbor addresses of the first macroblock in a MbAff frame picture (MbAff = 1) with up to 8 dependencies



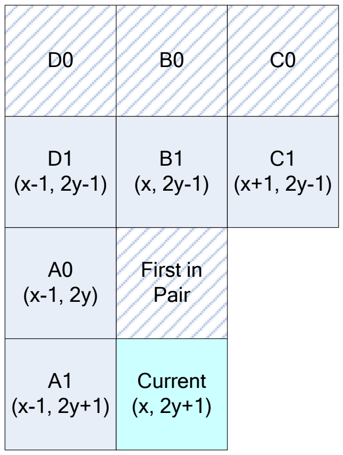
(a) Neighbors for the first macroblock in a 'frame' MbAff pair

(b) Neighbors for the first macroblock in a 'field' MbAff pair

Neighbor addresses of the second macroblock in a MbAff frame picture (MbAff = 1) with up to 8 dependencies



(b) Neighbors for the second macroblock in a 'frame' MbAff pair



(b) Neighbors for the second macroblock in a 'field' MbAff pair

Neighbor Availability

MbAff	FieldMbFlag	VertOrigin[0]	A	B	C	D	LB	Comments
0	0/1	0/1	√	√	√	√	0	Progressive or Field picture
1	0	0	√	√	√	√	√	1st Frame MbAff macroblock
1	0	1	√	na	0	na	√	2nd Frame MbAff macroblock
1	1	0	√	√	√	√	√	1st Field MbAff macroblock
1	1	1	√	√	√	√	√	2nd Field MbAff macroblock

1.4.3.2 VC1-Style Dependency Example

For VC1, only one dependency may be set depending on the availability of the upper neighbor macroblock.

Macroblock sequence order in a VC-1 picture with WidthInMblk = 5 and HeightInMblk = 6

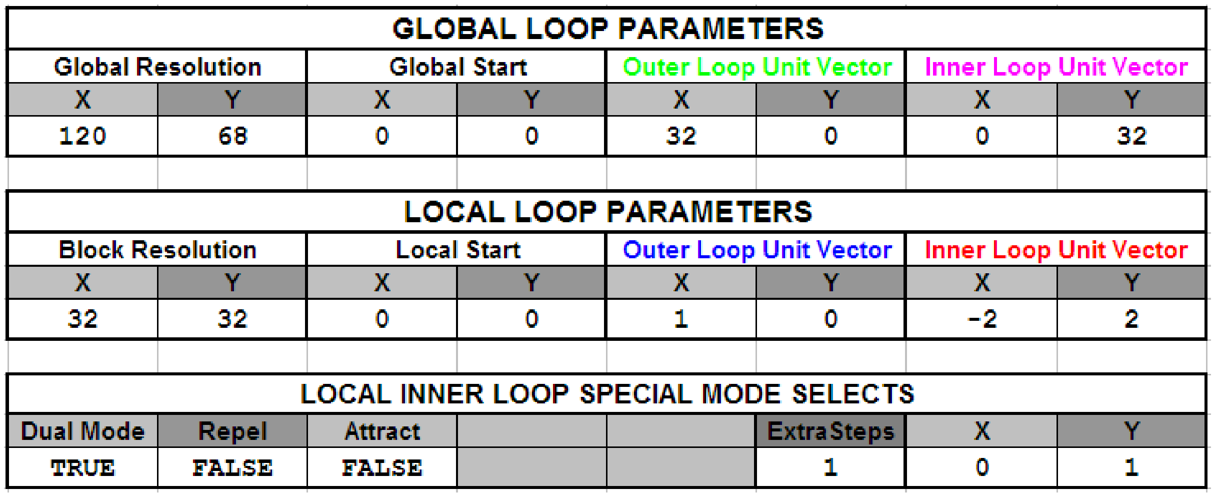
	0	**1**	**2**	**3**	**4**
0	0	1	2	3	4
1	5	6	7	8	9
2	10	11	12	13	14
3	15	16	17	18	19
4	20	21	22	23	24
5	25	26	27	28	29

1.4.3.3 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

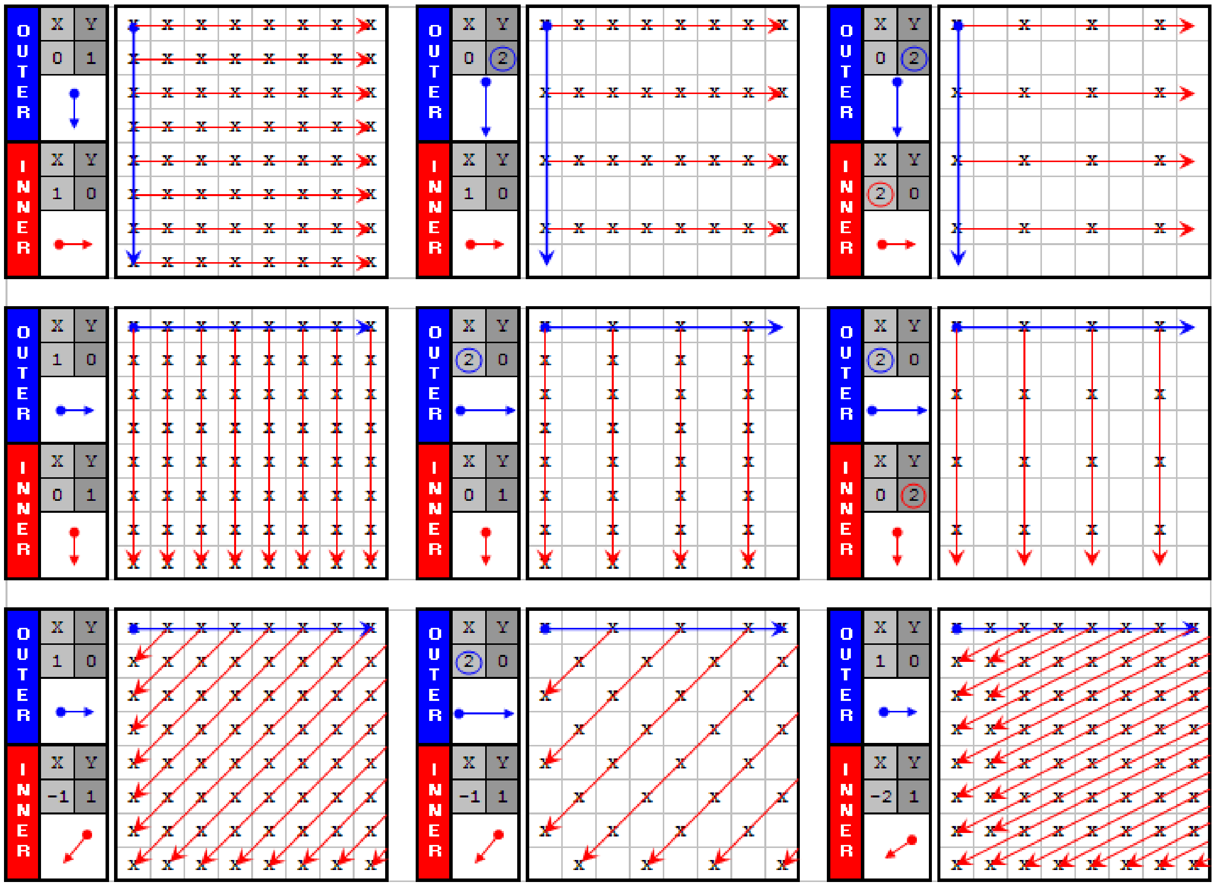
It should be emphasized that the value of what a “unit” represents is implicitly defined by the user. In other words, the walker traverses a “unit normalized space” that is not inherently bound to pixel walking. If the smallest unit of work the user wants to walk is a 4x3 block of pixels, you can program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

1.4.3.4 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:

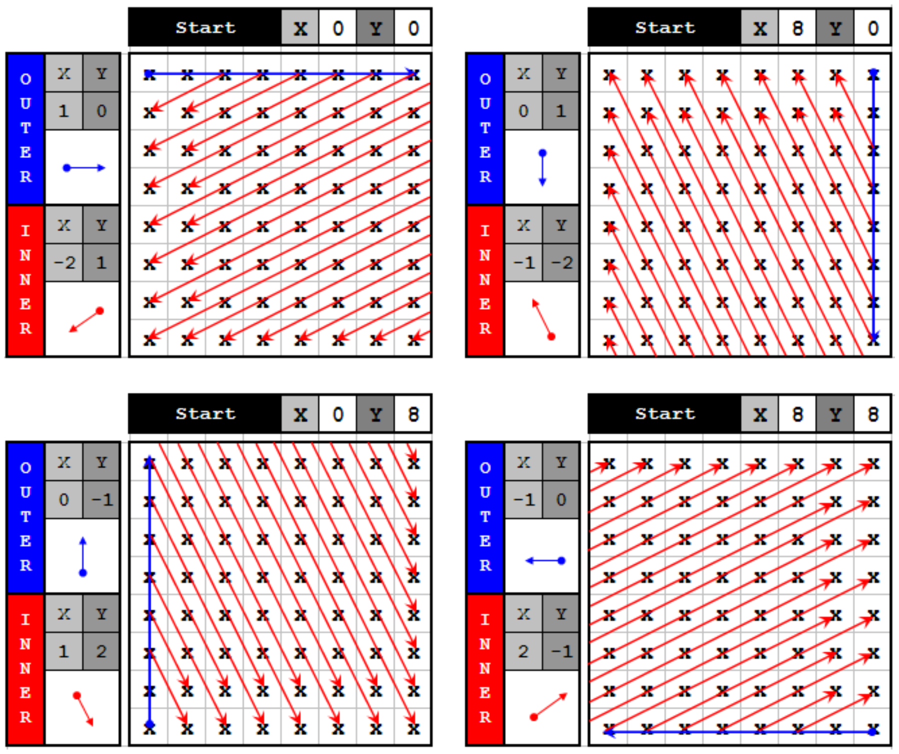


- The top row shows the outer unit vector pointing down (+Y) and the inner unit vector pointing right (+X). Rows and columns can easily be skipped by increasing the unit step vectors above one.

- 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.

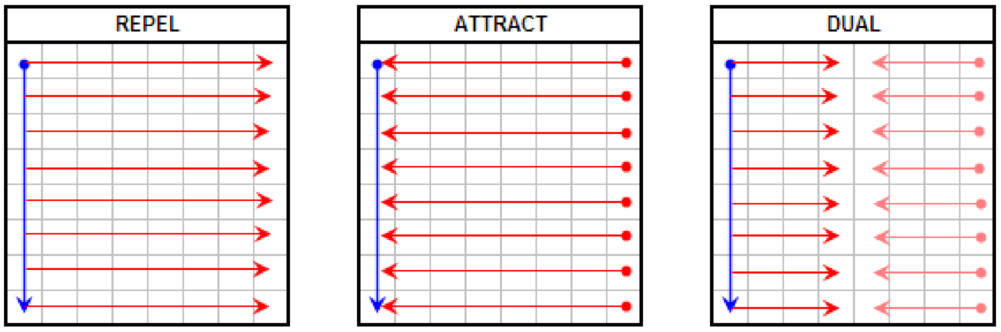
- The last row shows the capability to walk angles not perpendicular to the edge. The 1st shows a 45° walking pattern by setting the inner unit vector to (-1,1). The 2nd shows a checkerboard pattern by skipping every other outer loop and retaining the inner unit vector of (-1,1). The 3rd shows a 26.5° 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:

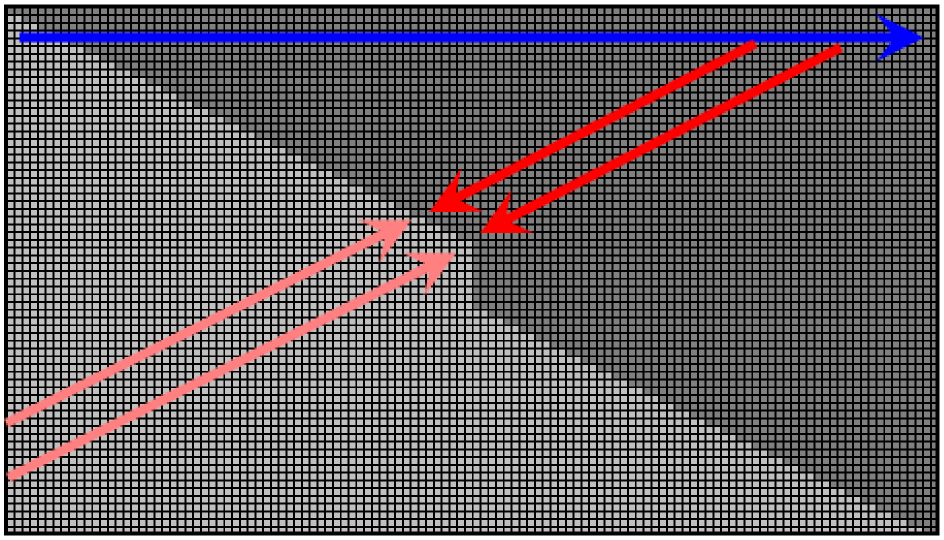


1.4.3.5 Dual Mode of Local Loop

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:

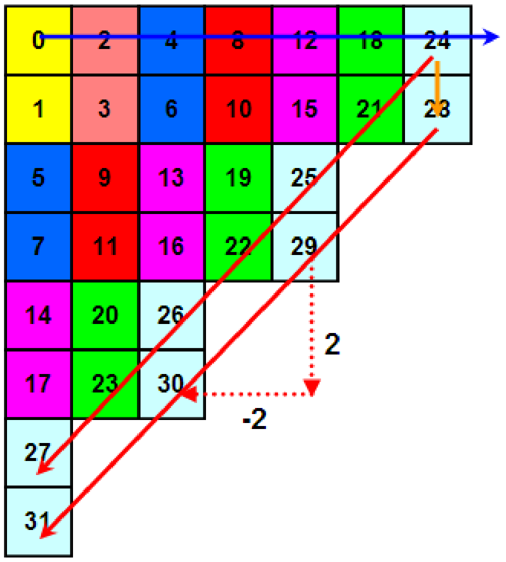


In Dual mode, the sequence will alternate between two half-walks such that every-other output would go to the same slice. This effect will produce a more balanced workload to two slices as shown in the example below where the color of the block represents which slice it was dispatched to. This is the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



1.4.3.6 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.

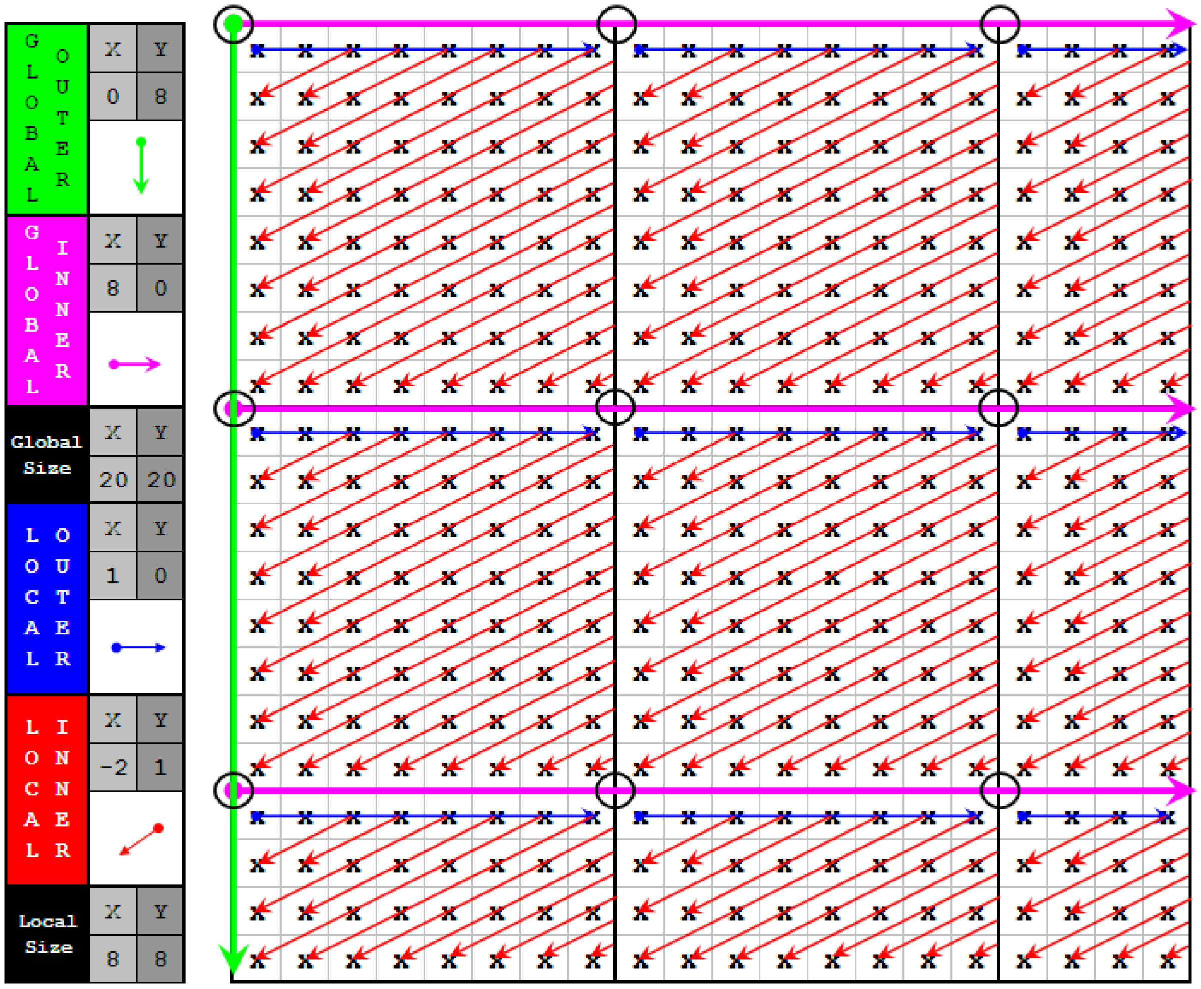


The outer loop unit step vector would be [1, 0] and the inner loop unit step vector would be [-2, 2]. A third loop is necessary to repeat the inner loop, only shifted down a unit before restarting. Thus, a middle loop with a unit step vector of [0,1] would achieve this MBAFF pattern. Additionally, the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

1.4.3.7 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 26.5° 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.

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
```

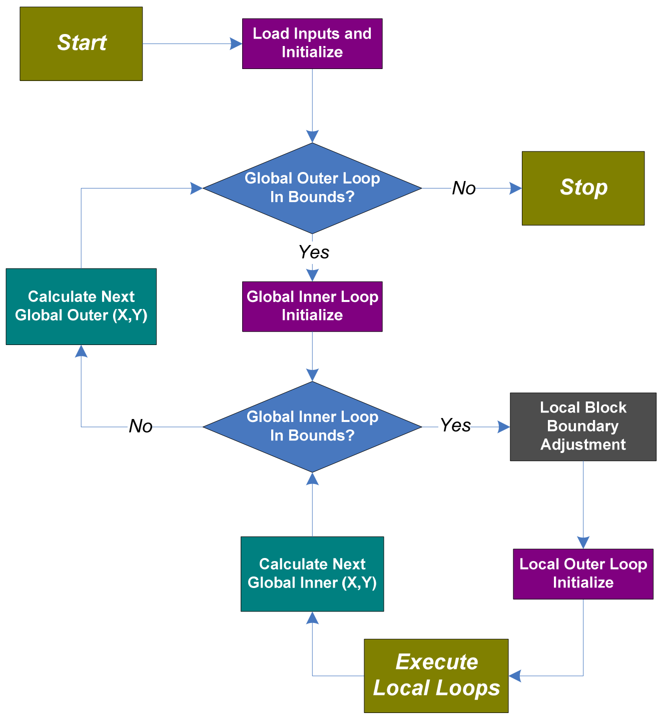
The pseudo-code has the following characteristics:

- 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 1-2. Walker algorithm flowchart for the Global Loop

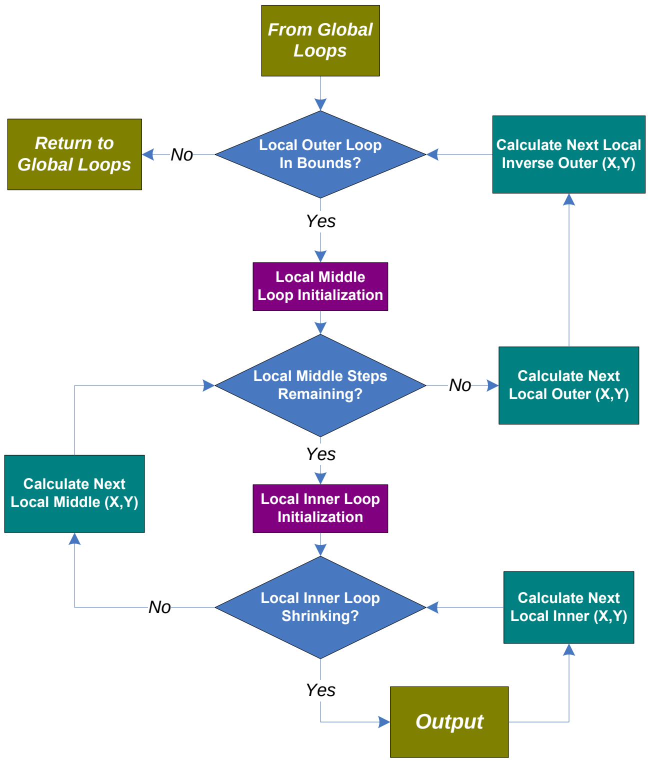

Start
Load Inputs and Initialize
Global Outer Loop In Bounds?
No
Stop
Yes
Calculate Next Global Outer (X,Y)
Global Inner Loop Initialize
No
Global Inner Loop In Bounds?
Yes
Local Block Boundary Adjustment
Calculate Next Global Inner (X,Y)
Local Outer Loop Initialize
Execute Local Loops

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.

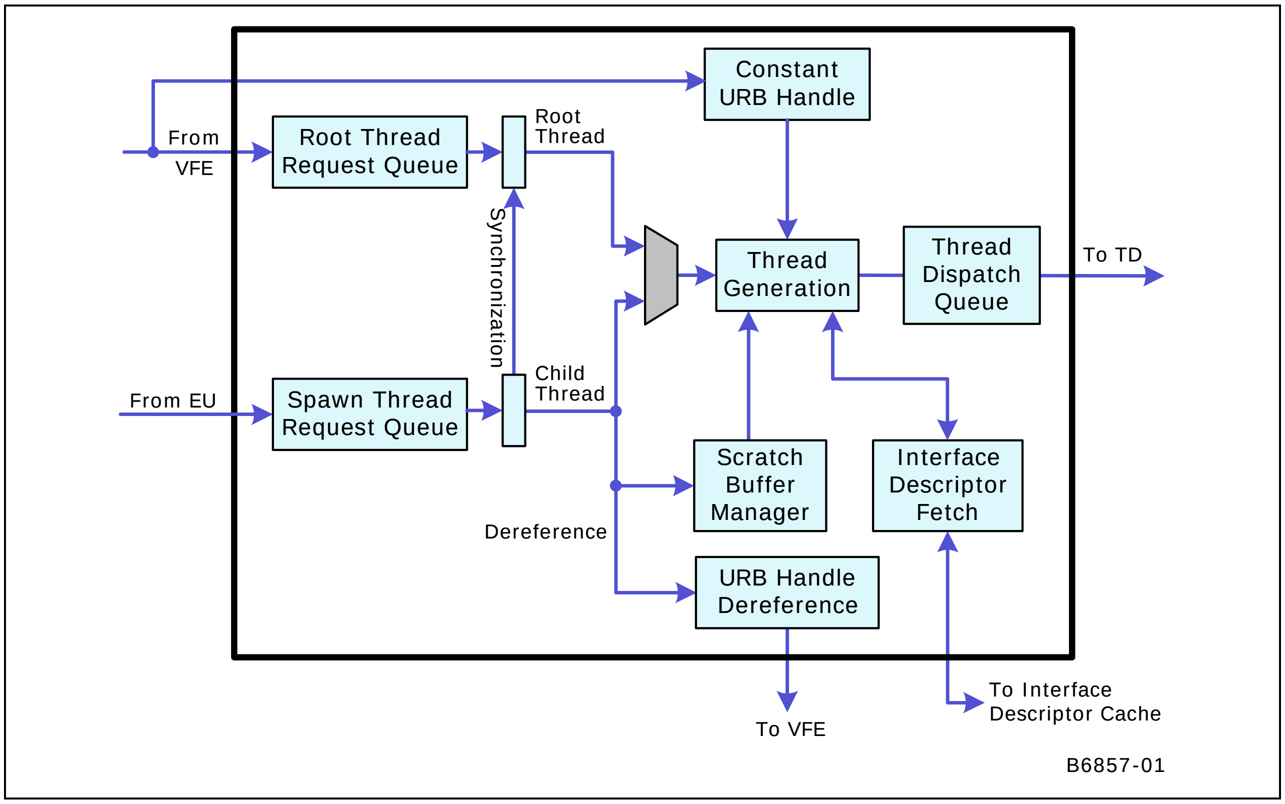
Figure 1-3. Walker algorithm flowchart for the Local Loop


From Global Loops
Return to Global Loops
No
Local Outer Loop In Bounds?
Calculate Next Local Inverse Outer (X,Y)
Yes
Local Middle Loop Initialization
Local Middle Steps Remaining?
No
Calculate Next Local Outer (X,Y)
Yes
Calculate Next Local Middle (X,Y)
Local Inner Loop Initialization
No
Local Inner Loop Shrinking?
Calculate Next Local Inner (X,Y)
Yes
Output

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.

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

If Children Present state variable is set (root-with-child mode), 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.

[Pre-SNB] Children Present is a state variable

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.

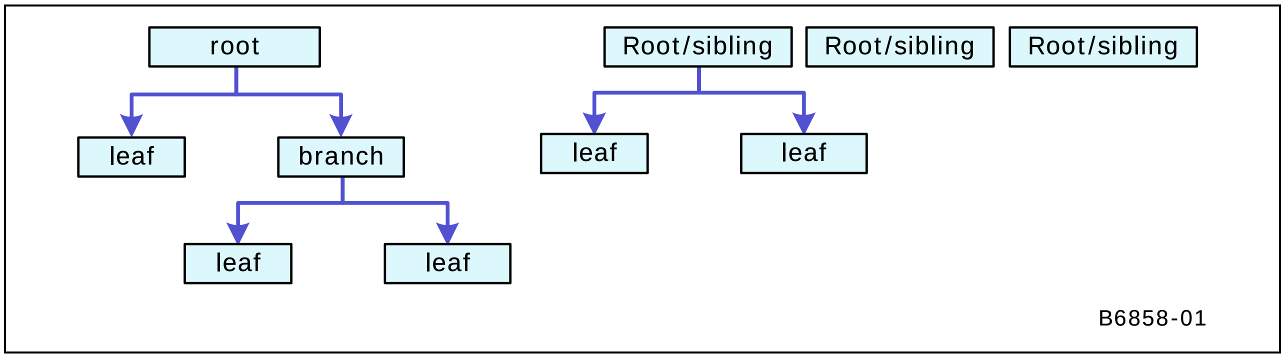
TS unit only enforces limitation on number of outstanding root threads. It is the root threads' responsibility to limit the number of child threads in their respected trees to balance performance and avoid deadlock.

1.5.1.4 Multiple Simultaneous Roots

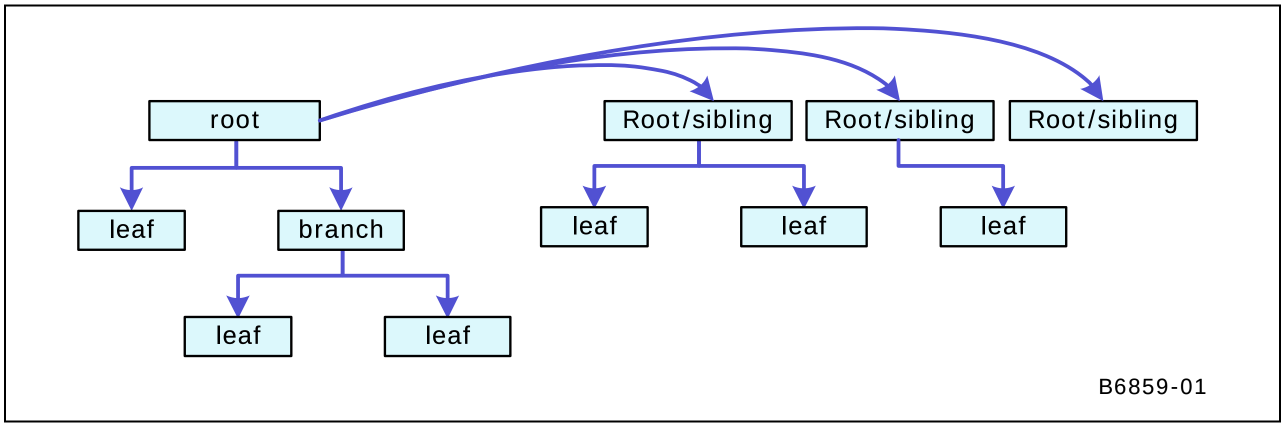
Multiple root threads are allowed concurrently running in GEN4 execution units. As there is only one scratch space state variable shared for all root threads, all concurrent root thread requiring scratch space share the same scratch memory size. The samples below depict two examples of thread-thread 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 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.

Examples of thread relationship



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)

TS Resource Available in Device Hardware

Device	Child Thread Dispatch Queue Depth
[DevBW]	**8**
[DevCL]	**8**
[DevCTG]	**8**
[DevILK]	**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 R0 header for a child, as part of the URB payload, consisting of the 32-bit field from the parent and a parent created field to uniquely identify the child.

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 [DevCTG+]

Persistent Root Thread (PRT) is 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 rules:

- 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 state variable. 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

1.6.1 Media State Model

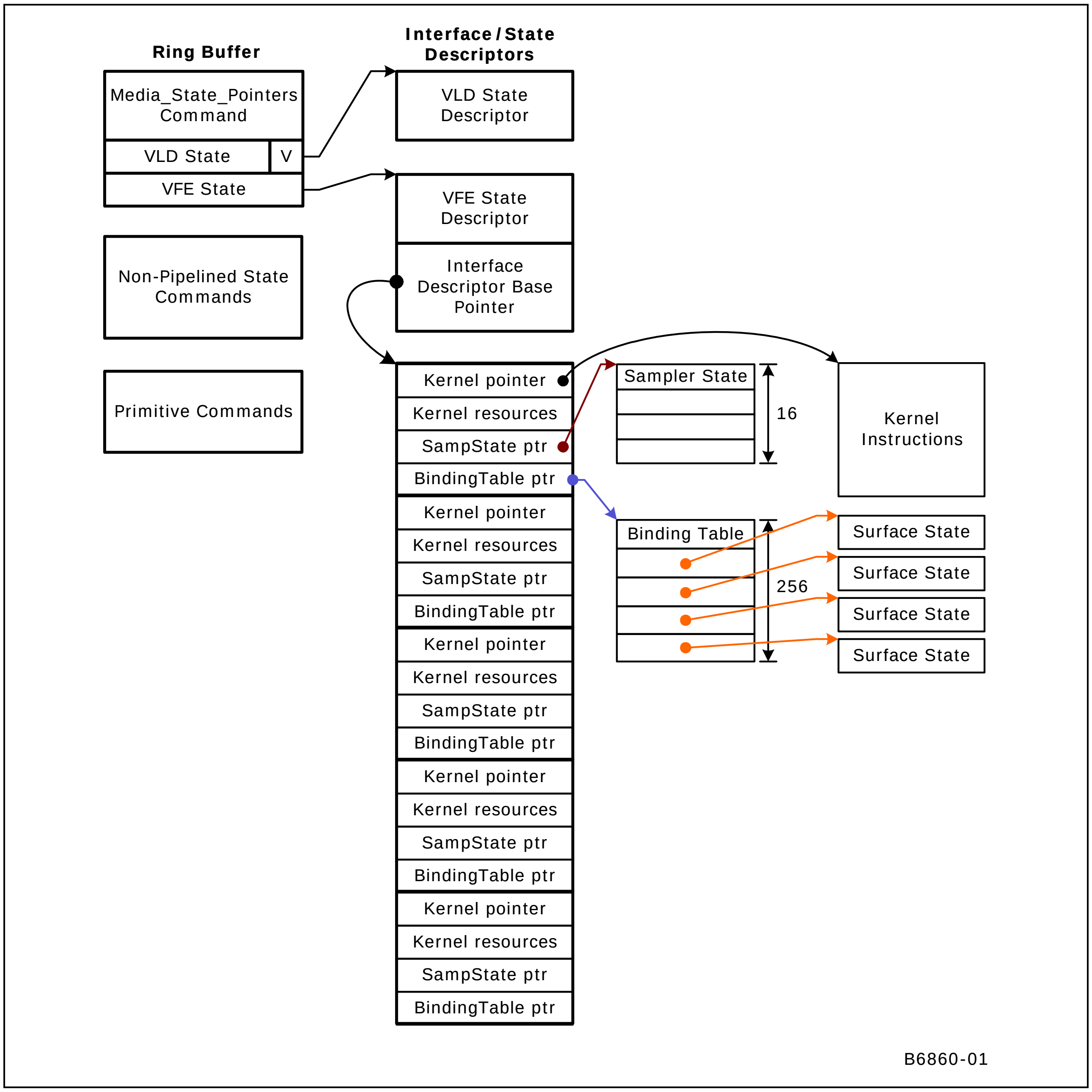
The media state model is based on an indirect state fetching mechanism. State Descriptors provide state information for fixed function units of the media pipeline. Interface Descriptors provide state information for kernels (threads) dispatched from the media pipeline. There are organized in different memory locations.

VFE State Descriptor contains states for both VFE unit and TS unit. The special purpose VLD state information is provided by a separate VLD State Descriptor.

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.

The MEDIA_STATE_POINTERS command provides the memory pointers to the Descriptors.

Media State Model



1.6.2VFE_STATE

Dword	Bit	Description
0	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	**Extended VFE State Present.** This field specifies whether extended VFE state is present or not. It must be programmed with the same value as the **Extended VFE State Enable** field in MEDIA_STATE_POINTERs command. 0 = Disabled. No extended VFE state (and Extension State Pointer is ignored). 1 = Enabled. The extended VFE state pointed by Extended State Pointer is loaded [**DevBW, DevCL**] This field is reserved and MBZ.
	6:4	Reserved : MBZ

	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]

1	31:25	**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 = U7 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.

	24:16	**URB Entry Allocation Size.** Specifies the length of each URB entry used by the unit, in 512-bit register increments - 1.
		Format = U9
		Range = [0,255] indicating [1,256] 512-bit register increments

	15:9	**Number of URB Entries.** Specifies the number of URB entries that are used by the unit.
		Format = U7
		Range = [1,64]

	8:7	Reserved : MBZ

Dword	Bit	Description
	6:3	**VFE Mode** 0000 – **Generic** Mode 0001 – **VLD** Mode (MPEG-2 only) 0010 – **IS** Mode (supporting WMV IDCT) 0100 – **AVC-MC** Mode 0111 – **AVC-IT** Mode 1011 – **VC1-IT** Mode All other encodings are reserved [**DevBW, DevCL**] AVC-MC, AVC-IT and VC1-IT modes are not supported [**DevBW**] VLD mode is not supported
	2	**Children Present.** Indicates that the root thread may send spawn messages to spawn child threads and/or synchronized root threads. In VLD Mode, this field must be 0. Format = Enable
2	31:4	**Interface Descriptor Base Pointer.** Specifies the 16-byte aligned address of the interface descriptor base pointer. This pointer is relative to the **General State Base Address**. Format = GeneralStateOffset[31:4]
	3:0	Reserved : MBZ

1.6.3 VLD_STATE

Dword	Bit	Description
0	31:28	**f_code[1][1].** Used for backward motion vector prediction. See ISO/IEC 13818-2 §7.6.3.1 for details
	27:24	**f_code[1][0].** Used for backward motion vector prediction. See ISO/IEC 13818-2 §7.6.3.1 for details
	23:20	**f_code[0][1].** Used for forward motion vector prediction. See ISO/IEC 13818-2 §7.6.3.1 for details
	19:16	**f_code[0][0].** Used for forward motion vector prediction. See ISO/IEC 13818-2 §7.6.3.1 for details
	15:14	**Intra DC Precision.** See ISO/IEC 13818-2 §6.3.10 for details.
	13:12	**Picture Structure.** This field specifies whether the picture is encoded in the form of a frame picture or one field (top or bottom) picture. See *ISO/IEC 13818-2* §6.3.10 for details. Format = MPEG_PICTURE_STRUCTURE 00 = Reserved 01 = MPEG_TOP_FIELD 10 = MPEG_BOTTOM_FIELD 11 = MPEG_FRAME

Dword	Bit	Description
	11	**TFF (Top Field First).** When two fields are stored in a picture, this bit indicates if the top field is the first field. For a frame P picture, the value 1 indicates that the top field of the reconstructed frame is the first field output by the decoding process, the same as defined in ISO/IEC 13818-2 §6.3.10. Particularly, it is used by the hardware to calculate derivative motion vectors from the dual-prime motion vectors. For a field P picture, hardware uses this bit together with the Picture Structure to determine if the current picture is the Second Field. In this case, the definition of this bit differs from ISO/IEC 13818-2 §6.3.10 – software must derive the value for this bit according to the following relation: | | Picture Structure = top field | Picture Structure = bottom field | |---|---|---| | Second Field = 0 | TFF = 1 | TFF = 0 | | Second Field = 1 | TFF = 0 | TFF = 1 |
	10	**Frame Prediction Frame DCT.** This field provides constraints on the DCT type and prediction type. It affects the syntax of the bitstream.
	9	**Concealment Motion Vector Flag.** This field indicates if the concealment motion vectors are coded in intra macroblocks. It affects the syntax of the bitstream.
	8	**Quantizer Scale Type**. This field specifies the quantizer scaling type. Format = MPEG_Q_SCALE_TYPE 0: MPEG_QSCALE_LINEAR 1: MPEG_QSCALE_NONLINEAR
	7	**Intra VLC Format.** This field is used by VLD.
	6	**Scan Order.** This field specifies the Inverse Scan method for the DCT-domain coefficients in the blocks of the current picture. Format = MPEG_INVERSESCAN_TYPE 0 = MPEG_ZIGZAG_SCAN 1 = MPEG_ALTERNATE_VERTICAL_SCAN
	5:0	**Reserved.**
1	31:14	**Reserved.**
	13	**Reserved (was Concealment Enable)**
	12	**Reserved (was Concealment Reference)**
	11	**Reserved (was Concealment Type)**
	10:9	**Picture Coding Type.** This field identifies whether the picture is an intra-coded picture (I), predictive-coded picture (P) or bi-directionally predictive-coded picture (B). See *ISO/IEC 13818-2* §6.3.9 for details. Format = MPEG_PICTURE_CODING_TYPE 00 = Reserved 01 = MPEG_I_PICTURE 10 = MPEG_P_PICTURE 11 = MPEG_B_PICTURE
	8:1	**Reserved (was Slice Error Control)**
	0	**Reserved.** (was Disable Mismatch)

Dword	Bit	Description
2	31:0	**Interface Descriptor Remap Table [7:0].** This field contains the interface descriptor remap table entries for the first 8 kernel indices. Each table entry has 4 bits, providing a remapping range of [0, 15]. This field is applicable to both frame picture destination (Picture Structure = 11) and field picture destination (Picture Structure = 01 or 10). Bits 31:28: Remap for index = 7 Bits 27:24: Remap for index = 6 Bits 23:20: Remap for index = 5 Bits 19:16: Remap for index = 4 Bits 15:12: Remap for index = 3 Bits 11:8: Remap for index = 2 Bits 7:4: Remap for index = 1 Bits 3:0: Remap for index = 0 **[DevCL] Errata:** This field is reserved.
3	31:0	**Interface Descriptor Remap Table [15:8].** This field contains the interface descriptor remap table entries for the last 8 kernel indices. Each table entry has 4 bits, providing a remapping range of [0, 15]. This field is only applicable to frame destination. It is ignored when the destination is a field picture. **[DevCL] Errata:** This field is reserved.

1.6.4 VFE_STATE_EX [DevCTG+]

Dword	Bit	Description
	7:0	Reserved : MBZ
1	31:0	**VFE Control.** This field is used by VFE depending on the mode of operation. See the following tables for details. If VFE Mode = AVC-IT or AVC-MC, this field is valid .If VFE Mode = VC1-IT, this field is valid. Otherwise, this field is reserved.
2	31:0	**Interface Descriptor Remap Table.** This field contains the interface descriptor remap table entries for the first 8 kernel indices. Each table entry has 4 bits, providing a remapping range of [0, 15]. The input of this table is the Interface Descriptor Offset within the MEDIA_OBJECT or MEDIA_OBJECT_EX command. As the table is limited to map the first 16 values, any Interface Descriptor Offset greater than 15 is not remapped. Bits 31:28: Remap for index = 7 Bits 27:24: Remap for index = 6 Bits 23:20: Remap for index = 5 Bits 19:16: Remap for index = 4 Bits 15:12: Remap for index = 3 Bits 11:8: Remap for index = 2 Bits 7:4: Remap for index = 1 Bits 3:0: Remap for index = 0
3	31:0	**Interface Descriptor Remap Table (cont).** This field contains the interface descriptor remap table entries for the next 8 kernel indices (index = 8…15). Each table entry has 4 bits, providing a remapping range of [0, 15]. Bits 31:28: Remap for index = 15 Bits 27:24: Remap for index = 14 Bits 23:20: Remap for index = 13 Bits 19:16: Remap for index = 12 Bits 15:12: Remap for index = 11 Bits 11:8: Remap for index = 10 Bits 7:4: Remap for index = 9 Bits 3:0: Remap for index = 8
4	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** [**DevCTG**] Reserved : MBZ

Dword	Bit	Description
	30	**Scoreboard Type.** This field selects the type of scoreboard in use. 0 – **Stalling scoreboard** 1 – **Non-stalling scoreboard** [**DevCTG**] Reserved : MBZ [**DevILK**] This field must be zero (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 [DevCTG] Reserved : MBZ
5	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 [DevCTG] Reserved : MBZ
	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 [DevCTG] Reserved : MBZ
	23:16	**Scoreboard 2 Delta (X, Y)** [DevCTG] Reserved : MBZ
	15:8	**Scoreboard 1 Delta (X, Y)** [DevCTG] Reserved : MBZ
	7:0	**Scoreboard 0 Delta (X, Y)** [DevCTG] Reserved : MBZ
6	31:24	**Scoreboard 7 Delta (X, Y)** [DevCTG] Reserved : MBZ
	23:16	**Scoreboard 6 Delta (X, Y)** [DevCTG] Reserved : MBZ
	15:8	**Scoreboard 5 Delta (X, Y)** [DevCTG] Reserved : MBZ
	7:0	**Scoreboard 4 Delta (X, Y)** [DevCTG] Reserved : MBZ
7	31:0	Reserved : MBZ

VFE Control in AVC-IT or AVC-MC Mode

Bit	Description
31:27	Reserved : MBZ
26	**Residual Data Fixed Source Offset Flag** 0: Residual data are packed right behind the Motion Vector and Weight-Offset blocks 1: Residual data start at the fixed offset provided by **Residual Data Source Offset** field. This field must be 1 if VFE_Mode = AVC-MC
25:24	**Indirect Sub-Field Present Flag**. This field indicates if any sub-field in the indirect data buffer is present. **NoMV** may only be used for an I-picture, where all macroblocks must have MvSize = 0. 00: **NoMV**. Motion Vector and Weight/Offset fields are not present in the indirect data buffer. 01: **NoWO**. Motion Vectors may be present in the indirect data buffer, but Weight/Offset is not. 10: Reserved 11: Both Motion Vector field and Weight/Offset field may be present
23:16	**Residual Data Source Offset.** This field specifies the fixed distance of the residual data from **Indirect Data Start Address**. It is in unit of dwords and must be 8-dword aligned. It is only valid when **Residual Data Fixed Source Offset Flag** is set and must be zero otherwise.
15:13	Reserved : MBZ
12:8	**Inter Weight-Offset Residual Data GRF Destination Offset**: This field specifies the offset in unit of GRF registers of the Weight/Offset data relative to the leading GRF location where the indirect data (where Motion Vectors, if present, are stored) will be placed. This field doesn't apply to intra-predicted (or I_PCM) macroblocks. In other words, this field is ignored by hardware for macroblocks with MvSize = 0. This field must be programmed to be greater than or equal to the maximum size for Motion Vectors data block. This field must be zero if **Indirect Sub-Field Present Flag** is **NoMV**.
7:5	Reserved : MBZ
4:0	**Inter Residual Data GRF Destination Offset**: This field specifies the offset in unit of GRF registers of the residual data of an inter-predicted macroblock relative to the leading GRF location of the indirect data (where Motion Vectors, if present, are stored). This field must be programmed to be greater than or equal to the maximum size for Motion Vectors and Weight-Offset data blocks. Therefore, it must be zero if **Indirect Sub-Field Present Flag** is **NoMV**. Note that this field doesn't affect intra-predicted macroblocks (or I_PCM macroblocks). As there is no motion vector for an I-macroblock (or I_PCM), the residual data will be placed at the beginning of the indirect data GRF location. In other words, this field is ignored by hardware for macroblocks with MvSize = 0.

VFE Control in VC1-IT Mode

Bit	Description
31:0	Reserved : MBZ

1.6.4.1 Interface Descriptor Remapping

When Extended VFE State VFE_STATE_EX is selected, the Interface Descriptor Offset field of the primitive command goes through the Interface Descriptor Remap Table, when Interface Descriptor Offset is within [0, 15]. When no Extended VFE State is selected, the interface descriptor remap is bypassed.

1.6.5INTERFACE_DESCRIPTOR

DWord	Bit	Description
0	31:6	**Kernel Start Pointer.** Specifies the 64-byte aligned address offset of the first instruction in the kernel. This pointer is relative to the **General State Base Address [Pre-DevILK]** or **Instruction Base Address [Dev ILK] [DevBW] Errata BWT007:** Instructions pointed at by offsets from General State Base must be contained within 32-bit physical address space (that is, must map to memory pages under 4G.) **[Pre-DevILK]:** Format = GeneralStateOffset[31:6] **[DevILK+]:** Format = InstructionBaseOffset[31:6]
	5:4	Reserved : MBZ

	3:0	**GRF Register Blocks.** Defines the number of GRF Register Blocks used by the kernel. A register block contains 8 registers. A kernel using a register count that is not a multiple of 8 must round up to the next multiple of 8.
		Format = U4 register block count - 1
		Range = [0,15] corresponding to [1,16] 8-register blocks
		Restriction: LSB must be zero, indicating that GRF assignment is in granularity of 16 GRF registers.

1	31:26	**Constant URB Entry Read Length.** 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.
		Format = U6
		Range = [0,63]

	25:20	**Constant URB Entry Read Offset.** 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.
		Format = U6
		Range = [0,63]

	19	Reserved : MBZ
	18	**Single Program Flow (SPF).** Specifies whether the kernel program has a single program flow (SIMDnxm with m = 1) or multiple program flows (SIMDnxm with m > 1). 0: Multiple Program Flows 1: Single Program Flow

DWord	Bit	Description
	17	**Thread Priority.** Specifies the priority of the thread for dispatch 0: Normal Priority 1: High Priority **Programming Notes:** • **[Pre-DevILK]**: This field must be set to zero.
	16	**Floating Point Mode.** Specifies the floating point mode used by the dispatched thread. 0: Use IEEE-754 Rules 1: Use alternate rules
	15:14	Reserved: MBZ
	13	**Illegal Opcode Exception Enable.** This bit gets loaded into EU CR0.1[12] (note the bit # difference). See *Exceptions* and *ISA Execution Environment*. Format: Enable
	12	Reserved: MBZ
	11	**MaskStack Exception Enable.** This bit gets loaded into EU CR0.1[11]. See *Exceptions* and *ISA Execution Environment*. Format: Enable
	10:8	Reserved: MBZ
	7	**Software Exception Enable.** This bit gets loaded into EU CR0.1[13] (note the bit # difference). See *Exceptions* and *ISA Execution Environment*. Format: Enable
	6:0	Reserved: MBZ
2	31:5	**Sampler State Pointer.** Specifies the 32-byte aligned address offset of the sampler state table. This pointer is relative to the **General State Base Address**. **[DevBW-A] Errata BWT007:** Sampler state pointed at by offsets from General State Base must be contained within 32-bit physical address space (that is, must map to memory pages under 4G.) Format = GeneralStateOffset[31:5] *This field is ignored for child threads.*
	4:2	**Sampler Count.** Specifies how many samplers (in multiples of 4) the kernel uses. Used only for prefetching the associated sampler state entries. Format = U3 Range = [0,4] 0: no samplers used 1: between 1 and 4 samplers used 2: between 5 and 8 samplers used 3: between 9 and 12 samplers used 4: between 13 and 16 samplers used *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.*
	1:0	Reserved : MBZ

DWord	Bit	Description
3	31:5	**Binding Table Pointer.** Specifies the 32-byte aligned address of the binding table. This pointer is relative to the **Surface State Base Address**. Format = SurfaceStateOffset[31:5] *This field is ignored for child threads.*
	4:0	**Binding Table Entry Count.** 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. Format = U5 Range = [0,31] *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.*

.

1.7 Media State and Primitive Commands

1.7.1 MEDIA_STATE_POINTERS Command

The MEDIA_STATE_POINTERS command is used to set up the pointers to the VFE states (VFE state, VLD state or VFE state extension). This command is issued prior to a set of media primitive commands, and points to the Generic mode VFE state and VLD decode mode VLD state (or VFE extended state).

[DevBW] Errata BWT007: State data pointed at by offsets from General State Base must be contained within 32-bit physical address space (that is, must map to memory pages under 4G.)

DWord	Bit	Description
0	31:29	**Command Type =** GFXPIPE = 3h
	28:16	**Media Command Opcode =** MEDIA_STATE_POINTERS Pipeline[28:27] = Media = 2h; Opcode[26:24] = 0h; Subopcode[23:16] **=** 0h
	15:0	**DWord Length** (Excludes DWords 0,1) = 01h
1	31:5	**Extended State Pointer.** Specifies the 32-byte aligned address of the extended VFE state (either VLD_STATE or VFE_STATE_EX). This pointer is relative to the **General State Base Address**. Which extended VFE state is used depends on **VFE Mode**. If **VFE Mode** is set to VLD Mode (0001), VLD_STATE is used. Otherwise, VFE_STATE_EX is used. Format = GeneralStateOffset[31:5] [**DevBW, DevCL**] Note that VFE_STATE_EX is reserved
	4:1	Reserved : MBZ
	0	**Extended VFE State Enable (**was **VLD Enable).** This field specifies whether extended VFE state is loaded. 0 = Disabled. No extended VFE state (and Extension State Pointer is ignored). 1 = Enabled. The extended VFE state pointed by Extended State Pointer is loaded [**DevBW, DevCL**] Note that VFE_STATE_EX is reserved
2	31:5	**Pointer to VFE_STATE.** Specifies the 32-byte aligned address of the VFE_STATE. This pointer is relative to the **General State Base Address**. Format = GeneralStateOffset[31:5]
	4:0	Reserved : MBZ

1.7.2 MEDIA_OBJECT Command

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.

The MEDIA_OBJECT command can be used in the following three VFE modes: Generic mode, IS mode and VLD mode.

The MEDIA_OBJECT command cannot be used in the following VFE modes: AVC-IT, AVC-MC, and VC1-IT.

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) **VLD Mode:** DWord Length = 4. There are 2 DW of inline data in this mode. **IS Mode:** DWord Length = N+2, where N is the number of DW of inline data (N>= 10). According to the inline format table shown in the following section, N is 10. However, hardware must be able to handle different size of N, as software may determine later to transfer additional driver/kernel information inline. **Generic Mode:** DWord Length = N+2, 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 or equal to 63 (with both inline data length N and indirect data length rounded up to 8-DW aligned individually). If indirect data are fetched, the minimal inline data length is 0. If indirect data are not fetched, the minimal inline data is 1DW. **Note:** Regardless of the mode, inline data must be present in this command.
	7	**Reserved.** MBZ
	6:0	**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. *In VLD mode, this field is ignored by hardware.* Format = U7
2	31:29	**Reserved.** MBZ
	28	**Retain Bit.** The hardware will keep the last 256-bit quantity of this indirect object in-use after the object transfer is complete. A subsequent Indirect object packet will use the retained 256bit quantity as the first piece of data. Format = Enable (1) /Disable (0) [**DevBW**] Erratum: this field is reserved: MBZ
	27:25	**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. *In VLD mode, this field must be programmed as 0, because the **Children Present** field in VFE_STATE must be 0 in this mode.* 0 = No thread synchronization 1 = Thread dispatch is synchronized by the “spawn root thread” message

Dword	Bits	Description
	23:17	**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. **VLD Mode:** It is the length in bytes of the bitstream data for the current slice. It includes the first byte of the first macroblock and the last non-zero byte of the last macroblock in the slice. Specifically, the zero-padding bytes (if present) and the next start-code are excluded. Hardware ignores the contents after the last non-zero byte. This field is sized to support MPEG-2 MP@HL bitstream. According to Table 8-6 of *ISO/IEC 13818-2,* the maximum number of bits per macroblock for 4:2:0 is 4608. So the maximum slice size for MP@HL (e.g. 1080i) is 4608 * 120 / 8 = 69120 bytes (0x10E00), which requires 17 bits. **Programming Restriction**: hardware has the limitation that the maximum allowed value is 0x1FFE0. As MPEG-2 spec does not post any limitation of the size of zero-padding bytes, it is possible to have a slice data with large length (including zero-padding bytes). As the data beyond 0x10E00 would only be zero bytes for a valid slice data, it is recommended that host software truncates the indirect data length to, say, 1x12000. Hardware can take care of the zero-padding bytes beyond the last non-zero byte of the slice. **IS Mode:** It must be DWord aligned. **Generic Mode:** 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). [**DevBW-A**] Erratum: In Generic Mode, the length alignment restrict is relaxed to be DWord alignment.. 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. **VLD Mode:** It is the byte aligned address for the VLD bitstream data. **IS Mode:** It is the DWord aligned address for the first IDCT coefficients. **Generic Mode:** It is the DWord aligned address of the indirect data. Range = [0 - 512MB] (Bits 31:29 MBZ)
4..N	31:0	**Inline Data** **IS and VLD Modes:** Hardware interprets this data in the specified format. **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 the URB allocation size.

1.7.2.1 Inline and Indirect Data Format in Generic Mode

In Generic mode, inline data must be present. All inline data will be delivered to the thread's payload starting and ending on the 8-DW aligned register boundary. Inline data starts on dword 4 of the MEDIA_OBJECT command. If the dword length field of the MEDIA_OBJECT command is N+2, the size of the inline data will N. VFE always zero-pads inline data into 8-DW before delivering to URB. If N is multiple of 8-DW, the inline data corresponds to exactly N/8 GRF registers. If N is not multiple of 8-DW, there will be (N/8 + 1) registers written for the inline data with the last register containing the last a few dwords of inline data with remaining dwords zeroed out by VFE.

Indirect data, if present, will be written into GRF registers immediately following the inline data in the thread's payload. Alignment and padding for indirect data are the same as that for inline data. In short, indirect data are also starting and ending on 8-DW aligned register boundary. If indirect data length is not multiple of 8-DW, VFE hardware will zero pad the last GRF register.

1.7.2.2 Inline and Indirect Data Format in IS Mode

Each MEDIA_OBJECT command in "IS mode" corresponds to the processing of one macroblock. Macroblock parameters are passed in as inline data and the non-zero DCT coefficient data for the macroblock is passed in as indirect data.

The table below depicts the inline data format in IS mode. All fields in inline data are forwarded to the thread as thread payload. Alignment and padding is identical to that described for Generic mode. 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 command. There are 10 dwords total.

Inline data in IS mode

DWord	Bit	Description
4+0	31:28	**Motion Vertical Field Select.** A bit-wise representation of a long [2][2] array as defined in §6.3.17.2 of the *ISO/IEC 13818-2* (see also §7.6.4). Bit | MVector [r] | MVector [s] | MotionVerticalFieldSelect Index 28 | 0 | 0 | 0 29 | 0 | 1 | 1 30 | 1 | 0 | 2 31 | 1 | 1 | 3 Format = MC_MotionVerticalFieldSelect. 0 = The prediction is taken from the top reference field. 1 = The prediction is taken from the bottom reference field.
	27	**Second Field.** This bit indicates that this is the second field in the current frame. The prediction for this macroblock, if it belongs to a field P-picture, should use this bit to determine which frame contains the reference field as described in §7.6.2.1 of the *ISO/IEC 13818-2*. When the picture type is not P or the prediction type is not field, this value should be 0. Format = MC_SecondPField 0 = This is not the second field. 1 = This is the second field.
	26	**Reserved.** (HWMC mode)

DWord	Bit	Description
	25:24	**Motion Type.** When combined with the destination picture type (field or frame) this Motion Type field indicates the type of motion to be applied to the macroblock. See *ISO/IEC 13818-2* §6.3.17.1, Tables 6-17, 6-18. In particular, the device supports dual-prime motion prediction (11) in both frame and field picture type. Format = MC_MotionType Value | Destination = Frame Picture_Structure = 11 | Destination = Field Picture_Structure != 11 '00' | Reserved | Reserved '01' | Field | Field '10' | Frame | 16x8 '11' | Dual-Prime | Dual-Prime
	23:22	**Reserved.** (Scan method)
	21	**DCT Type.** This field specifies the DCT type of the current macroblock. The kernel should ignore this field when processing Cb/Cr data. See *ISO/IEC 13818-2* §6.3.17.1. This field is zero if Coded Block Pattern is also zero (no coded blocks present). 0 = MC_FRAME_DCT (Macroblock is frame DCT coded). 1 = MC_FIELD_DCT (Macroblock is field DCT coded).
	20	**Overlap Transform (H261 Loop Filter).** This field, when set, indicates that overlap smoothing filter is performed after motion compensation and before in-loop deblocking.
	19	**4MV Mode:** (H263/WMV) This field indicates if the current macroblock is coded with 4 motion vectors, one for each 8x8 block.
	18	**Macroblock Motion Backward.** This field specifies if the backward motion vector is active. See *ISO/IEC 13818-2* Tables B-2 through B-4. 0 = No backward motion vector. 1 = Use backward motion vector(s).
	17	**Macroblock Motion Forward.** This field specifies if the forward motion vector is active. See *ISO/IEC 13818-2* Tables B-2 through B-4. 0 = No forward motion vector. 1 = Use forward motion vector(s).
	16	**Macroblock Intra Type.** 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). See *ISO/IEC 13818-2* Tables B-2 through B-4. 0 = Non-intra macroblock. 1 = Intra macroblock.
	15:0	**Reserved.** (MB address)
4+1	31:24	**Reserved.** (Skip Macroblocks)
	23:0	**Reserved.** (Offset into error data)
4+2	31:24	**Subblock Coding for Block Y1**

DWord	Bit	Description
	23:16	**Subblock Coding for Block Y0.** This field specifies the subblock partition and subblock coding pattern for the block. The definition of the 8 bits of this field is listed below. Bits [7:6]: reserved Bits [5:2]: Subblock present Bits [1:0]: Subblock partitioning
	15:12	**Reserved.**
	11:6	**Coded Block Pattern.** This field specifies whether blocks are present or not. Format = 6-bit mask. Bit 11: Y0 Bit 10: Y1 Bit 9: Y2 Bit 8: Y3 Bit 7: Cb4 Bit 6: Cr5 [Used by VFE]
	5:0	**Reserved.** (Quantization Scale Code)
4+3	31:24	**Subblock Coding for Block Cr5**
	23:16	**Subblock Coding for Block Cb4**
	15:8	**Subblock Coding for Block Y3**
	7:0	**Subblock Coding for Block Y2**
4+4	31:16	**Motion Vectors – Field 0, Forward, Vertical Component.** Each vector component is a 16-bit two's-complement value. The vector is relative to the current macroblock location. According to ISO/IEC 13818-2 Table 7-8, the valid range of each vector component is [-2048, +2047.5], implying a format of s11.1. However, it should be noted that motion vector values are sign extended to 16 bits.
	15:0	**Motion Vectors – Field 0, Forward, Horizontal Component**
4+5	31:16	**Motion Vectors – Field 0, Backward, Vertical Component**
	15:0	**Motion Vectors – Field 0, Backward, Horizontal Component**
4+6	31:16	**Motion Vectors – Field 1, Forward, Vertical Component**
	15:0	**Motion Vectors – Field 1, Forward, Horizontal Component**
4+7	31:16	**Motion Vectors – Field 1, Backward, Vertical Component**
	15:0	**Motion Vectors – Field 1, Backward, Horizontal Component**
4+8	31:30	**Reserved.**
	29	**May need this for WMV.** (Interpolation Rounder Control)
	28	**May need this for WMV.** (Bidirectional Averaging Control)
	27:20	**Reserved.**

DWord	Bit	Description
	19:18	**Picture Coding Type.** This field identifies whether the picture is an intra-coded picture (I), predictive-coded picture (P) or bi-directionally predictive-coded picture (B). See *ISO/IEC 13818-2* §6.3.9 for details. Format = MPEG_PICTURE_CODING_TYPE 00 = Reserved 01 = MPEG_I_PICTURE 10 = MPEG_P_PICTURE 11 = MPEG_B_PICTURE
	17:16	**Picture Structure.** This field specifies whether the picture is encoded in the form of a frame picture or one field (top or bottom) picture. See *ISO/IEC 13818-2* §6.3.10 for details. Format = MPEG_PICTURE_STRUCTURE 00 = Reserved 01 = MPEG_TOP_FIELD 10 = MPEG_BOTTOM_FIELD 11 = MPEG_FRAME
	15	**Reserved.** (8-bit Intra)
	14:13	**Reserved.** (Intra DC Precision)
	12:0	**Reserved.**
4+9	31:27	**Reserved.**

	26:20	**Vertical Origin.** Set the vertical origin of the next macroblock in the destination picture in units of macroblocks. (Valid range is 0 to 120).
		Format = U7 in macroblock units.
		Range = [0, 120]

	19:11	**Reserved:** MBZ

	10:4	**Horizontal Origin.** Set the horizontal origin of the next macroblock in the destination picture in units of macroblocks.
		Format = U7 in macroblock units.
		Range = [0, 127]

	3:0	**Reserved.**

The control parameters for inverse-scan are carried in the inline data packet. In particular, the Coded Block Pattern field in DW6 is used to determine how many blocks are coded and therefore how many blocks will be output from the inverse-scan.

Besides that dword 6 (containing Coded Block Pattern field) is used by VFE hardware as control parameter for inverse-scan, the rest of the inline data are determined between the host software and the kernel software. Therefore, the exact format and size of the inline data may differ, as long as the Coded Block Pattern field in dword 6 remains the same.

Subblock coding (bits [7:6] are reserved).

Subblock Partitioning (Bits [1:0])		Subblock Present (0 means not present, 1 means present)			
Code	**Meaning**	**Bit 2**	**Bit 3**	**Bit 4**	**Bit 5**
00	Single 8x8 block (sb0)	Sb0	Don't care	Don't care	Don't care
01	Two 8x4 subblocks (sb0-1)	Sb0	Sb1	Don't care	Don't care
10	Two 4x8 subblocks (sb0-1)	Sb0	Sb1	Don't care	Don't care
11	Four 4x4 subblocks (sb0-3)	Sb0	Sb1	Sb2	Sb3

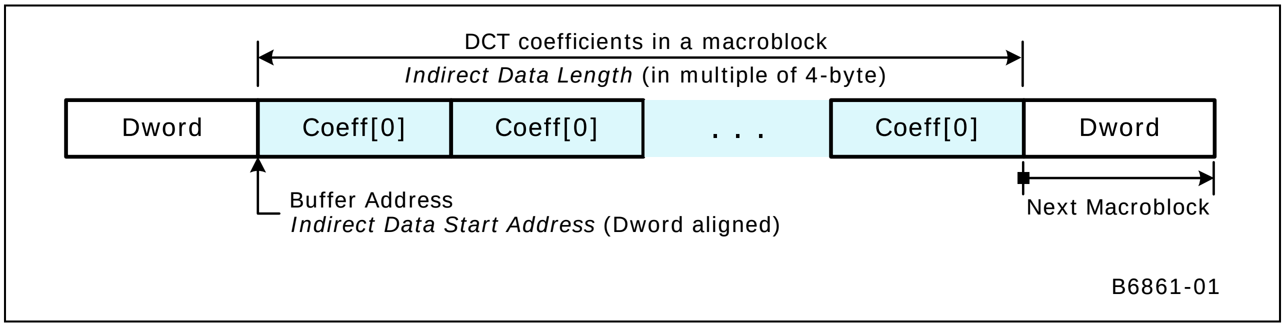
The block data output from the inverse scan will follow immediately after the inline data in the thread's payload, again, aligning to GRF register. Block data output by nature is 8-DW aligned. The actual size depends on the coded block pattern. As each block contains 8x8 16-bit DCT coefficients, if the total number of coded block is M, the block data will take 8 * 8 * 2 * M / 32 = 4 * M GRE registers.

As VFE performs inverse-scan on the indirect data, the indirect data must follow the exact format described in Figure 1-4 and

.

The indirect data start address in MEDIA_OBJECT specifies the doubleword aligned address of the first non-zero DCT coefficient of the first block of the macroblock. Only the non-zero DCT coefficients are present in the data buffer and they are packed in the block sequence of Y0, Y1, Y2, Y3, Cb4 and Cr5, as shown in Figure 1-4. The indirect data length in MEDIA_OBJECT includes all the non-zero coefficients for the macroblock. It must be doubleword aligned.

Figure 1-4. Structure of the IDCT Compressed Data Buffer


DCT coefficients in a macroblock
Indirect Data Length (in multiple of 4-byte)
Dword
Coeff[0]
Coeff[0]
. . .
Coeff[0]
Dword
Buffer Address
Indirect Data Start Address (Dword aligned)
Next Macroblock
B6861-01

Each non-zero coefficient in the indirect data buffer is contained in a doubleword-size data structure containing the coefficient index, end of block (EOB) flag and the fixed-point coefficient value in 2's compliment form. As shown in

, *index* is the row major 'raster' index of the coefficient within an 8x8 block. DCT coefficient is a 16-bit value in 2's complement, which is clamped to a 12-bit signed value by the host. Effectively, bit 27 is the sign bit. However, as the kernel software consumes these data as 16-bit quantities any way, VFE simply forwards these exact 16-bit DCT coefficients to the thread's payload.

Structure of a DCT coefficient unit

DWord	Bit	Description
0	31:16	**DCT Coefficient Value.** This field contains the value of the non-zero DCT coefficient in 2's compliment.
	15:7	Reserved: MBZ

	6:1	**Index.** This field specifies the raster-scan address (raw address) of the DCT coefficient within the 8x8 block. For example, coefficient at location (row, column) = (0, 0) has an index of 0; that at (2, 3) has an index of 2*8 + 3 = 19.
		Format = U6
		Range = [0, 63]

	0	**EOB (End of Block).** This field indicates whether the DCT coefficient is the last one of the current block.

1.7.2.3 Inline and Indirect Data Format in VLD Mode

A MEDIA_OBJECT command in "VLD mode" is used to process a slice using the VFE hardware. Slice header parameters are passed in as inline data and the bitstream data for the slice is passed in as indirect data. Of the inline data, slice_horizontal_position and slice_vertical_position determines the location within the destination picture of the first macroblock in the slice.

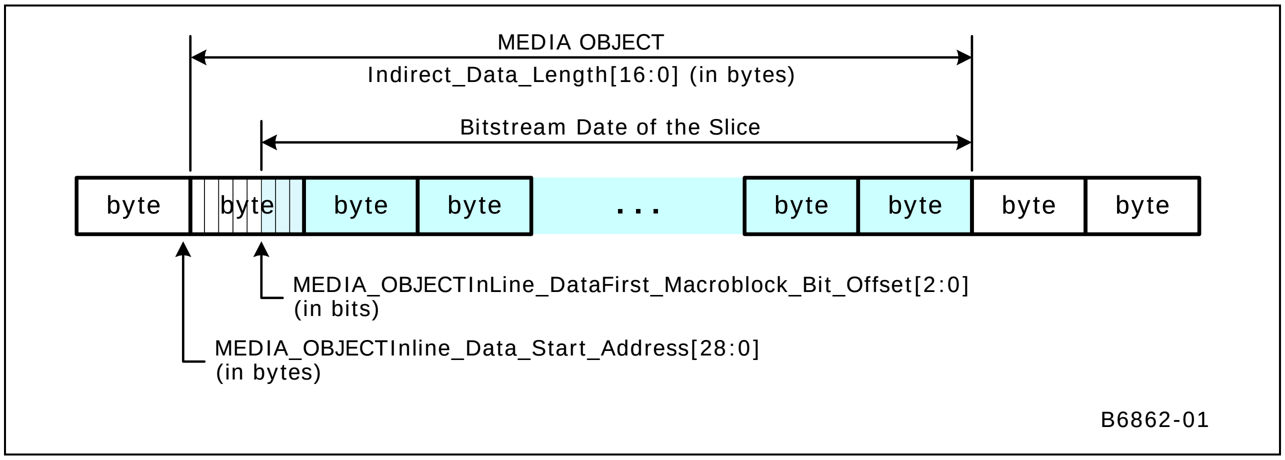
DWord	Bits	Description
4	31	**Reserved.** MBZ
	30:24	**Slice Horizontal Position.** This 7-bit field indicates the horizontal position (in macroblock units) of the first macroblock in the slice. Format = U7 in macroblocks
	23	**Reserved.** MBZ
	22:16	**Slice Vertical Position.** This 7-bit field indicates the vertical position (in macroblock units) of the first macroblock in the slice. Format = U7 in macroblocks
	15	**Reserved.** MBZ
	14:8	**Macroblock Count.** This 7-bit field indicates the number of macroblocks in the slice, including skipped macroblocks.
	7:3	**Reserved.** MBZ.
	2:0	**First Macroblock Bit Offset.** This field provides the bit offset of the first macroblock in the first byte of the input bitstream. Format = U3
5	31:29	**Reserved.** MBZ.

DWord	Bits	Description
	28:24	**Quantizer Scale Code.** This field sets the quantizer scale code of the inverse quantizer. It remains in effect until changed by a decoded quantizer scale code in a macroblock. This field is decoded from the slice header by host software. Format = U5 (0 is Reserved)
	23:0	**Reserved.** MBZ.

The indirect data start address in MEDIA_OBJECT specifies the starting Graphics Memory address of the bitstream data that follows the slice header. It provides the byte address for the first macroblock of the slice. Together with the First Macroblock Bit Offset field in the inline data, it provides the bit location of the macroblock within the compressed bitstream.

The indirect data length in MEDIA_OBJECT provides the length in bytes of the bitstream data for this slice. It includes the first byte of the first macroblock and the last **non-zero** byte of the last macroblock in the slice. Specifically, the zero-padding bytes (if present) and the next start-code are excluded. Hardware ignores the contents after the last non-zero byte. The diagram below illustrates these parameters for a slice data.

Indirect data buffer for a slice


MEDIA OBJECT
Indirect_Data_Length[16:0] (in bytes)
Bitstream Date of the Slice
byte
byte
byte
byte
. . .
byte
byte
byte
byte
MEDIA_OBJECTInLine_DataFirst_Macroblock_Bit_Offset[2:0]
(in bits)
MEDIA_OBJECTInline_Data_Start_Address[28:0]
(in bytes)
B6862-01

1.7.3 MEDIA_OBJECT_EX Command [DevCTG-DevILK]

[**DevBW/DevCL]** This command is not supported.

[**DevBW/DevCL**] This command is not supported.

The MEDIA_OBJECT_EX command is the extended media primitive command for the media pipeline. The command can be used for AVC and VC1 decode in different modes. It supports loading of inline data as well as indirect data.

This command can be used in the following VFE modes: AVC-IT, AVC-MC, VC1-IT and Generic modes.

This command cannot be used in the following VFE modes: IS and VLD modes.

Dword	Bits	Description
0	31:29	**Command Type =** GFXPIPE = 3h
	28:16	**Media Command Opcode =** MEDIA_OBJECT_EX Pipeline[28:27] = Media = 2h; Opcode[26:24] **=** 1h; Subopcode[23:16] = 01h
	15:0	**DWord Length** (Excludes DWords 0,1) **Note:** Regardless of the mode, inline data must be present in this command.
1	31:0	**VFE Dword.** This field contains the data specific to the mode of operation. This field is consumed by VFE, and in general is not forwarded to the thread.
2	31	**Reserved.** MBZ
	30:24	**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. *Actual Interface Descriptor is selected post the Interface descriptor Remapping.* Format = U7
	23:22	**Reserved.** MBZ
	21	**[DevILK] 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 [DevCTG] **Reserved**. MBZ
	20	**Thread Synchronization.** This field indicates whether the thread is a Synchronized Root Thread (SRT). 0 = Not a SRT 1 = SRT
	19:17	**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. **AVC-IT Mode:** It must be DWord aligned. **AVC-MC Mode:** It must be 8-DWord aligned. **VC1-IT Mode:** It must be DWord aligned. **Generic Mode:** It must be DWord aligned. Format = U17 in bytes

Dword	Bits	Description
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. **AVC-IT Mode:** It is the DWord aligned address for the first field, if available, in the order of Motion Vectors, Weights/Offsets, transform-domain residual data (IDCT coefficients). **AVC-MC Mode:** It is the DWord aligned address for the first field, if available, in the order of Motion Vectors, Weights/Offsets, pixel-domain residual data. **VC1-IT Mode:** It is the DWord aligned address for the first IDCT coefficients. **Generic Mode:** It is the DWord aligned address. Range = [0 - 512MB] (Bits 31:29 MBZ)
4..N	31:0	**Inline Data** **AVC-IT Modes:** Hardware interprets this data in the specified format. **AVC-MC Modes:** Hardware interprets this data in the specified format. **VC1-IT Modes:** Hardware interprets this data in the specified format. **Generic Modes:** Hardware does not interpret this data

[DevILK+] Up to eight dependencies are provided in Generic mode, which are explicitly controlled by software programming using the VFE Dword field.

The VFE Dword [DevCTG]

DWord	Bit	Description
1	31:0	**Reserved: MBZ**

The VFE Dword [DevILK]

DWord	Bit	Description
1	31:28	**Scoreboard Mask 4-7 :** Each bit indicates the corresponding dependency scoreboard is dependent on. This field is AND'd with the corresponding Scoreboard Mask field in the VFE_STATE_EX. **Bit n (for n = 4…7):** Scoreboard n is dependent, where bit 28 maps to n = 4. Format = TRUE/FALSE
	27:26	**Scoreboard Color :** This field specifies which dependency color the current thread belongs to. It affects the dependency scoreboard control. Format = U2
	25	**Reserved.** MBZ
	24:16	**Scoreboard Y** This field provides the Y term of the scoreboard value of the current thread. Format = U9

DWord	Bit	Description
	15:12	**Scoreboard Mask 0-3 :** Each bit indicates the corresponding dependency scoreboard is dependent on. This field is AND'd with the corresponding Scoreboard Mask field in the VFE_STATE_EX. **Bit n (for n = 0…3):** Scoreboard n is dependent, where bit 12 maps to n = 0. Format = TRUE/FALSE
	11:9	**Reserved.** MBZ
	8:0	**Scoreboard X** This field provides the X term of the scoreboard value of the current thread. Format = U9

1.7.3.1 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.

The table below 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.

Inline data in AVC-IT mode ([DevCTG])

DWord	Bit	Description
4+0	31:27	**Reserved.** MBZ
	26:25	**MbAffFieldFlag** This field indicates that the current macroblock is a field macroblock within a MbAff frame picture. It is provided as **Flag = MbaffFrame** & **FieldMbFlag.** 00 = if (Flag == 0) 11 = if (Flag == 1) Other encodings are reserved

DWord	Bit	Description
	24	**FieldMbPolarityFlag** This field indicates the field polarity of the current macroblock. Within a MbAff frame picture, this field may be different per macroblock and is set to 1 only for the second macroblock in a MbAff pair if FieldMbFlag is set. Otherwise, it is set to 0. Within a field picture, this field is set to 1 if the current picture is the bottom field picture. Otherwise, it is set to 0. It is a constant for the whole field picture. This field is reserved and MBZ for a progressive frame picture. 0 = Current macroblock is a field macroblock from the **top** field 1 = Current macroblock is a field macroblock from the **bottom** field *Programming Note: Here bits [26:24] (MbAffFieldFlag and FiedlMbPolarityFlag) match with bits [10:8] of the Media Block Read message descriptor, simplifying the programming for message generation, as when MbAffFieldFlag is "1", kernels need to override the original "frame" surface state set for MBAFF frame picture.*
	23	**Reserved**: MBZ
	22:20	**MvSize (Motion Vector Size)** [Used by VFE]. This field specifies the size of motion vectors for the macroblock stored in the indirect data buffer. The valid numbers are listed below indicating the size of the regrouped motion vectors. Details are provided in Section 1.7.3.1.4. This field is reserved (MBZ) when **IntraMbFlag** = 1. 000 = 0: No motion vector 001 reserved 010 = 2MV: One motion vector pair 011 reserved 100 = 8MV: Four motion vector pairs 101 = 16MV: 16 motion vectors 110 = 32MV: 16 motion vector pairs 111 reserved
	19	**DcBlockCodedYFlag** [Used by VFE]. This field is consumed by VFE and is not delivered to the thread.
	18	**DcBlockCodedCbFlag** [Used by VFE]. This field is consumed by VFE and is not delivered to the thread.
	17	**DcBlockCodedCrFlag** [Used by VFE]. This field is consumed by VFE and is not delivered to the thread.
	16	**Reserved.** MBZ
	15	**Transform8x8Flag** [Used by VFE]. This field equals to the value of *transform_size_8x8_flag* as defined in AVC spec. If set, it indicates that luma samples are in residual 8x8 blocks. Otherwise, it indicates that luma samples are in residual 4x4 blocks. 0: luma residual 4x4 blocks 1: luma residual 8x8 blocks
	14	**FieldMbFlag (Field Macroblock Flag).** This field specifies whether current macroblock is field macroblock. 0 = Frame macroblock. 1 = Field macroblock.

DWord	Bit	Description
	13	**IntraMbFlag (Intra Macroblock Flag).** This field specifies whether the current macroblock is an Intra (I) macroblock. A collective macroblock type in AVC standard includes I, SI, P and B. Type SI is not supported. 0 = P or B macroblock. 1 = I macroblock.
	12:8	**MbType (Macroblock Type).** This field, along with **IntraMbFlag** specifies the macroblock types. Further details can be found in Section 1.7.3.1.2.
	7:6	**WeightedBiPredFlag (Weighted Bidirectional Prediction Flag)** (from Picture State). This field specifies the bidirectional prediction mode and is derived from syntax elements *weighted_bipred_flag* and *weighted_pred_flag* as defined in AVC spec. It is valid only for inter predicted macroblock. Otherwise (intra macroblock), this field is reserved and MBZ. For B-macroblock, this field is the same as *weighted_bipred_flag* as defined in AVC spec. 00 = Default weighted prediction 01 = Explicit weighted prediction 10 = Implicit weighted prediction 11 = Reserved. For P-macroblock, the MSB is always 0 and the LSB is the same as *weighted_pred_flag* as defined in AVC spec. 00 = Default weighted prediction 01 = Explicit weighted prediction 10 and 11 are reserved
	5	**WeightedPredFlag (Weighted Prediction Flag)** [from Picture State]. It is valid only for inter predicted macroblock. Otherwise (intra macroblock), this field is reserved and MBZ. 0 = Default weighted prediction 1 = Explicit weighted prediction *Note: Information in this field is also carried in* **WeightedBiPredFlag***.*
	4	**IntraPredAvailFlagF – F** (Pixel [-1, 7] available for intra prediction) ***F = Is_Left_MB_Field & Is_Left_Bottom_MB_Intra***
	3:2	**ChromaFormatIdc (Chroma Format Indicator).** This field is equal to the value of *chroma_format_idct* as defined in AVC (§7.4.2.1). It specifies the chroma sampling relative to the luma sampling. This field is constant within a picture. 00 = Luma only (monochrome) 01 = YUV420 sampling 10 is reserved (for YUV422 sampling) 11 is reserved (for YUV444 sampling)

DWord	Bit	Description
	1	**MbaffFrameFlag (MB-AFF Frame Flag)** [PPS]. This field indicates whether the current picture is a progressive frame or a MB-AFF (macroblock adaptive frame field) frame picture. This field is frame Picture Parameter Set. This field is reserved (MBZ) if **FieldPicFlag** = 1. 0 = Progressive frame picture 1 = MB-AFF frame picture
	0	**FieldPicFlag (Field Picture Flag)** [PPS]. This field indicates whether the current picture is a field or a frame picture. A frame picture may be a progressive frame picture or an MB-AFF frame picture depending on the value of **MbaffFrame**. This field is frame Picture Parameter Set. 0 = Frame picture 1 = Field picture
4+1	31:16	**CbpY (Coded Block Pattern Y)** [Used by VFE]
	15:8	**VertOrigin (Vertical Origin).** This field specifies the vertical origin of current macroblock in the destination picture in units of macroblocks. For field macroblock pair in MBAFF frame, the vertical origins for both macroblocks should be set as if they were located in corresponding field pictures. For example, for field macroblock pair originated at (16, 64) pixel location in an MBAFF frame picture, the Vertical Origin for both macroblocks should be set as 2 (macroblocks). Format = U8 in unit of macroblock.
	7:0	**HorzOrigin (Horizontal Origin).** This field specifies the horizontal origin of current macroblock in the destination picture in units of macroblocks. Format = U8 in unit of macroblock.
4+2	31:24	**QpPrimeCr** [Used by VFE]
	23:16	**QpPrimeCb** [Used by VFE]
	15:8	**QpPrimeY** [Used by VFE]
	7:4	**CbpCr** [Used by VFE]
	3:0	**CbpCb** [Used by VFE]
4+3 to 4+5	31:0	intra macroblocks intra macroblocks Error! Reference source not found.
4+6	31:16	**LevelScaleCb.** [Used by VFE] This field is for inverse transform of the Chroma (Cb) DC block. The LevelScale field is consumed by VFE and is not needed by the thread, but since the GRF has to be filled to the end of the block it should be sent anyways.
	15:0	**LevelScaleCr.** [Used by VFE] This field is for inverse transform of the Chroma (Cr) DC block.
4+7	31:16	**Reserved.** MBZ
	15:0	**LevelScaleY.** [Used by VFE] This field is for inverse transform of the Luma DC block.

Inline data in AVC-IT mode ([DevILK])

DWord	Bit	Description
4+0	31:27	**Reserved.** MBZ
	26:25	**MbAffFieldFlag** This field indicates that the current macroblock is a field macroblock within a MbAff frame picture. It is provided as **Flag = MbaffFrame** & **FieldMbFlag.** 00 = if (Flag == 0) 11 = if (Flag == 1) Other encodings are reserved
	24	**FieldMbPolarityFlag** This field indicates the field polarity of the current macroblock. Within a MbAff frame picture, this field may be different per macroblock and is set to 1 only for the second macroblock in a MbAff pair if FieldMbFlag is set. Otherwise, it is set to 0. Within a field picture, this field is set to 1 if the current picture is the bottom field picture. Otherwise, it is set to 0. It is a constant for the whole field picture. This field is reserved and MBZ for a progressive frame picture. 0 = Current macroblock is a field macroblock from the **top** field 1 = Current macroblock is a field macroblock from the **bottom** field *Programming Note: Here bits [26:24] (MbAffFieldFlag and FiedlMbPolarityFlag) match with bits [10:8] of the Media Block Read message descriptor, simplifying the programming for message generation, as when MbAffFieldFlag is “1”, kernels need to override the original “frame” surface state set for MBAFF frame picture.*
	23	**Reserved**: MBZ
	22:20	**MvSize (Motion Vector Size)** [Used by VFE]. This field specifies the size of motion vectors for the macroblock stored in the indirect data buffer. The valid numbers are listed below indicating the size of the regrouped motion vectors. Details are provided in Section 1.7.3.1.4. This field is reserved (MBZ) when **IntraMbFlag** = 1. 000 = 0: No motion vector 001 reserved 010 = 2MV: One motion vector pair 011 reserved 100 = 8MV: Four motion vector pairs 101 = 16MV: 16 motion vectors 110 = 32MV: 16 motion vector pairs 111 reserved
	19	**DcBlockCodedYFlag** [Used by VFE]. This field is consumed by VFE and is not delivered to the thread.
	18	**DcBlockCodedCbFlag** [Used by VFE]. This field is consumed by VFE and is not delivered to the thread.
	17	**DcBlockCodedCrFlag** [Used by VFE]. This field is consumed by VFE and is not delivered to the thread.
	16	**Reserved.** MBZ

DWord	Bit	Description
	15	**Transform8x8Flag** [Used by VFE]. This field equals to the value of *transform_size_8x8_flag* as defined in AVC spec. If set, it indicates that luma samples are in residual 8x8 blocks. Otherwise, it indicates that luma samples are in residual 4x4 blocks. 0: luma residual 4x4 blocks 1: luma residual 8x8 blocks
	14	**FieldMbFlag (Field Macroblock Flag).** This field specifies whether current macroblock is field macroblock. 0 = Frame macroblock. 1 = Field macroblock.
	13	**IntraMbFlag (Intra Macroblock Flag).** This field specifies whether the current macroblock is an Intra (I) macroblock. A collective macroblock type in AVC standard includes I, SI, P and B. Type SI is not supported. 0 = P or B macroblock. 1 = I macroblock.
	12:8	**MbType (Macroblock Type).** This field, along with **IntraMbFlag** specifies the macroblock types. Further details can be found in Section 1.7.3.1.2.
	7:6	**WeightedBiPredFlag (Weighted Bidirectional Prediction Flag)** (from Picture State). This field specifies the bidirectional prediction mode and is derived from syntax elements *weighted_bipred_flag* and *weighted_pred_flag* as defined in AVC spec. It is valid only for inter predicted macroblock. Otherwise (intra macroblock), this field is reserved and MBZ. For B-macroblock, this field is the same as *weighted_bipred_flag* as defined in AVC spec. 00 = Default weighted prediction 01 = Explicit weighted prediction 10 = Implicit weighted prediction 11 = Reserved. For P-macroblock, the MSB is always 0 and the LSB is the same as *weighted_pred_flag* as defined in AVC spec. 00 = Default weighted prediction 01 = Explicit weighted prediction 10 and 11 are reserved
	5	**WeightedPredFlag (Weighted Prediction Flag)** [from Picture State]. It is valid only for inter predicted macroblock. Otherwise (intra macroblock), this field is reserved and MBZ. 0 = Default weighted prediction 1 = Explicit weighted prediction *Note: Information in this field is also carried in* **WeightedBiPredFlag**.
	4	**Reserved.** MBZ

DWord	Bit	Description
	3:2	**ChromaFormatIdc (Chroma Format Indicator).** This field is equal to the value of *chroma_format_idct* as defined in AVC (§7.4.2.1). It specifies the chroma sampling relative to the luma sampling. This field is constant within a picture. 00 = Luma only (monochrome) 01 = YUV420 sampling 10 is reserved (for YUV422 sampling) 11 is reserved (for YUV444 sampling)
	1	**MbaffFrameFlag (MB-AFF Frame Flag)** [PPS]. This field indicates whether the current picture is a progressive frame or a MB-AFF (macroblock adaptive frame field) frame picture. This field is frame Picture Parameter Set. This field is reserved (MBZ) if **FieldPicFlag** = 1. 0 = Progressive frame picture 1 = MB-AFF frame picture
	0	**FieldPicFlag (Field Picture Flag)** [PPS]. This field indicates whether the current picture is a field or a frame picture. A frame picture may be a progressive frame picture or an MB-AFF frame picture depending on the value of **MbaffFrame**. This field is frame Picture Parameter Set. 0 = Frame picture 1 = Field picture
4+1	31:16	**CbpY (Coded Block Pattern Y)** [Used by VFE]
	15:8	**VertOrigin (Vertical Origin).** This field specifies the vertical origin of current macroblock in the destination picture in units of macroblocks. For field macroblock pair in MBAFF frame, the vertical origins for both macroblocks should be set as if they were located in corresponding field pictures. For example, for field macroblock pair originated at (16, 64) pixel location in an MBAFF frame picture, the Vertical Origin for both macroblocks should be set as 2 (macroblocks). Format = U8 in unit of macroblock.
	7:0	**HorzOrigin (Horizontal Origin).** This field specifies the horizontal origin of current macroblock in the destination picture in units of macroblocks. Format = U8 in unit of macroblock.
4+2	31:24	**QpPrimeCr** [Used by VFE]
	23:16	**QpPrimeCb** [Used by VFE]
	15:8	**QpPrimeY** [Used by VFE]
	7:4	**CbpCr** [Used by VFE]
	3:0	**CbpCb** [Used by VFE]
4+3 to 4+5	31:0	intra macroblocks intra macroblocks Error! Reference source not found.
4+6	31:16	**LevelScaleCb.** [Used by VFE] This field is for inverse transform of the Chroma (Cb) DC block. The LevelScale field is consumed by VFE and is not needed by the thread, but since the GRF has to be filled to the end of the block it should be sent anyways.

DWord	Bit	Description
	15:0	**LevelScaleCr.** [Used by VFE] This field is for inverse transform of the Chroma (Cr) DC block.
4+7	31:16	**Reserved.** MBZ
	15:0	**LevelScaleY.** [Used by VFE] This field is for inverse transform of the Luma DC block.
4+8	31:0	**BSD_Pass_Thru_Data(Reserved to 0 in BSD mode)**
4+9	31:0	**BSD_Pass_Thru_Data(Reserved to 0 in BSD mode)**
4+10	31:0	**BSD_Pass_Thru_Data(Reserved to 0 in BSD mode)**
4+11	31:0	**BSD_Pass_Thru_Data(Reserved to 0 in BSD mode)**

This table 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 0.

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

Dword	Bit	Description
4+3	31:8	Reserved
	7:0	**MbIntraStruct (Macroblock Intra Structure)**

Bits	MotionVerticalFieldSelect Index
7:6	**ChromaIntraPredMode**
5	[DevILK] **IntraPredAvailFlagF – F** (Pixel [-1, 7] available for intra prediction) F = Is_Left_MB_Field & Is_Left_Bottom_MB_Intra [DevCTG] Reserved
4	**IntraPredAvailFlagE – E** (left neighbor bottom half)
3	**IntraPredAvailFlagD – D** (Upper right neighbor)
2	**IntraPredAvailFlagC – C** (Upper left neighbor)
1	**IntraPredAvailFlagB – B** (Upper neighbor)
0	**IntraPredAvailFlagA – A** (Left neighbor top half)

4+4	31:16	**LumaIndraPredModes[1] (Luma Intra Prediction Modes)**
	15:0	**LumaIndraPredModes[0]**
4+5	31:16	**LumaIndraPredModes[3]**
	15:0	**LumaIndraPredModes[2]**

Dwords 4+3, 4+4 and 4+5 of the inline data for an inter-predicted macroblock is detailed in **Error! Reference source not found.**.

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 1.7.3.1.3. 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 1.7.3.1.3. 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**

1.7.3.1.1 Luma Intra Prediction Modes

Luma Intra Prediction Modes (LumaIntraPredModes) is defined in this table. It is further categorized as Intra16x16PredMode, Intra8x8PredMode (When a macroblock is subdivided, the intra prediction is performed for the subdivision in a predetermined order. For example, the figure below shows the block order for Intra4x4 prediction. And the figure below shows the block order of Block8x8 in a 16x16 region or Block4x4 in a 8x8 region.

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)**

Definition of Intra16x16PredMode

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

Definition of Intra8x8PredMode

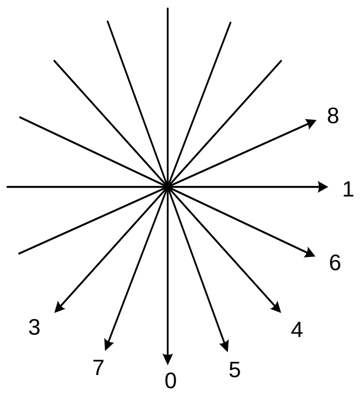
Intra8x8PredMode	Description
0	**Intra_8x8_Vertical**

Intra8x8PredMode	Description
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

Definition of Intra4x4PredMode

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

Intra_4x4 prediction mode directions



Numbers of Block4x4 in a 16x16 region

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

Numbers of Block4x4 in an 8x8 region or numbers of Block8x8 in a 16x16 region

0	1
2	3

1.7.3.1.2 Macroblock Type

Macroblock Type, MbType, is defined as a unified parameter for all slice types (I, P or B slices). 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.

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

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
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

MbType	For an Intra Macroblock (IntraMbFlag = 1)	For an Inter Macroblock (IntraMbFlag = 1)
26	Reserved (for SI)	Reserved
27-63	Reserved	Reserved

Deriving MbType from mb_type for I, P and B slices

	I Slice		P Slice		B Slice	
mb_type	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
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

mb_type	I Slice		P Slice		B Slice	
	MbType	Description	MbType	Description	MbType	Description
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
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

1.7.3.1.3 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. Both of these parameters can be derived from sub_mb_type field as defined in AVC spec .

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.

Definition of SubMbShape for an 8x8 region of a BP_8x8 macroblock

SubMbShape	NumSubMbPart	SubMbPartWidth	SubMbPartHeight
0	1	8	8
1	2	8	4
2	2	4	8
3	4	4	4

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	

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

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_8x 8	Direct	4	4	**3**	**?**
1	B_L0_8x8	Pred_L0	8	8	**0**	**0**
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**	**?**

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
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

		SubMbPredMode[i]			
MbType	Name	i = 0	i = 1	i = 2	i = 3
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

1.7.3.1.4 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 the table below, 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.*

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.

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	
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	

MbType	Inter Macroblock Type	Max (sub_mb_type[])	Max (SubMBPredMode[])	Extact MV #	MV State	MvSize	Commends
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

1.7.3.1.5 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.

1.7.3.2 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. 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.

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
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

1.7.3.2.1 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).

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**
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**

DWord	Bit	MvSize				
		0	2	8	16	32
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**

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

Indirect data Weight-Offset block in AVC-IT mode (and AVC-MC mode)

Weight-Offset programming model[Errata – Dev Ctg/Dev EL] Driver needs to look ahead all slices for the frame and check if any weight value is 128. This can only happen when luma_weight_l0/l1/chroma_weight_l0/l1 flags are 0. If it finds a 128 then it needs to remap this value to a non used value within a 16 bit twos compliment range and then give the remapped number to the kernel for identifying this case.

Indirect data Weight-Offset block in AVC-IT mode (and AVC-MC mode) – [DevCTG]

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		[**DevCTG**] Reserved: MBZ [**DevILK**] Weight is 128 [ChromaL1, Chroma L0, Luma L1, Luma L0]	[**DevCTG**] Reserved: MBZ [**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

DWord	Bit	MvSize		
		0	2	8, 16, 32
	3:0			[**DevCTG**] Reserved: MBZ [**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**
	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			[**DevCTG**] Reserved: MBZ [**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			[**DevCTG**] Reserved: MBZ [**DevILK**] Weight is 128[ChromaL1, Chroma L0, Luma L1, Luma L0]

[DevILK

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**

DWord	Bit	MvSize		
		0	2	8, 16, 32
	15:0	**n/a**	**WO_Cr_L0**	**WO_Cr_Block0_L0**
3	31:4	**n/a**	Reserved MBZ	Reserved MBZ
	3:0	**n/a**	Weight is 128[ChromaL1, Chroma L0, Luma L1, Luma L0]	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	**n/a**	**n/a**	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**
	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	**n/a**	**n/a**	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	**n/a**	**n/a**	Weight is 128[ChromaL1, Chroma L0, Luma L1, Luma L0]

1.7.3.2.3 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.

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**
	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**
	15:0	**n/a**	**n/a**	**Weight_Cr_Block3_L0**

DWord	Bit	MvSize		
		0	2	8, 16, 32
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.

1.7.3.2.4 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 Figure 1-4. 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. *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.

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

	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.

1.7.3.3 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 table below 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.

Inline data in AVC-MC mode ([DevCTG])

DWord	Bit	Description
4+0	31:27	**Reserved.** MBZ
	26:25	**MbAffFieldFlag** This field indicates that the current macroblock is a field macroblock within a MbAff frame picture. It is provided as **Flag = MbaffFrame** & **FieldMbFlag.** 00 = if (Flag == 0) 11 = if (Flag == 1) Other encodings are reserved
	24	**FieldMbPolarityFlag** This field indicates the field polarity of the current macroblock. Within a MbAff frame picture, this field may be different per macroblock and is set to 1 only for the second macroblock in a MbAff pair if FieldMbFlag is set. Otherwise, it is set to 0. Within a field picture, this field is set to 1 if the current picture is the bottom field picture. Otherwise, it is set to 0. It is a constant for the whole field picture. This field is reserved and MBZ for a progressive frame picture. 0 = Current macroblock is a field macroblock from the **top** field 1 = Current macroblock is a field macroblock from the **bottom** field *Programming Note: Here bits [26:24] (MbAffFieldFlag and FiedlMbPolarityFlag) match with bits [10:8] of the Media Block Read message descriptor, simplifying the programming for message generation, as when MbAffFieldFlag is "1", kernels need to override the original "frame" surface state set for MBAFF frame picture.*
	23	**Reserved**: MBZ

DWord	Bit	Description
	22:20	**MvSize (Motion Vector Size)** [Used by VFE]. This field specifies the size of motion vectors for the macroblock stored in the indirect data buffer. The valid numbers are listed below indicating the size of the regrouped motion vectors. This field is reserved (MBZ) when **IntraMbFlag** = 1. 000 = 0: No motion vector 001 reserved 010 = 2MV: One motion vector pair 011 reserved 100 = 8MV: Four motion vector pairs 101 = 16MV: 16 motion vectors 110 = 32MV: 16 motion vector pairs 111 reserved *This field is identical to that in AVC-IT mode.*
	19:16	**Reserved.** MBZ
	15	**Transform8x8Flag.** This field equals to the value of *transform_size_8x8_flag* as defined in AVC spec. If set, it indicates that luma samples are in residual 8x8 blocks. Otherwise, it indicates that luma samples are in residual 4x4 blocks. 0: luma residual 4x4 blocks 1: luma residual 8x8 blocks
	14	**FieldMbFlag (Field Macroblock Flag).** This field specifies whether current macroblock is field macroblock. 0 = Frame macroblock. 1 = Field macroblock.
	13	**IntraMbFlag (Intra Macroblock Flag).** This field specifies whether the current macroblock is an Intra (I) macroblock. A collective macroblock type in AVC standard includes I, SI, P and B. Type SI is not supported. 0 = P or B macroblock. 1 = I macroblock.
	12:8	**MbType (Macroblock Type).** This field, along with **IntraMbFlag** specifies the macroblock types. Further details can be found in Section 1.7.3.1.2.
	7:6	**WeightedBiPredFlag (Weighted Bidirectional Prediction Flag)** (from Picture State). Valid only for macroblock in inter mode. Otherwise (intra macroblock), this field is reserved.
	5	**WeightedPredFlag (Weighted Prediction Flag)** [from Picture State]. Valid only for macroblock in inter mode. Otherwise (intra macroblock), this field is reserved.
	4	**Reserved.** MBZ
	3:2	**ChromaFormatIdc (Chroma Format Indicator).** This field is equal to the value of *chroma_format_idct* as defined in AVC (§7.4.2.1). It specifies the chroma sampling relative to the luma sampling. This field is constant within a picture. 00 = Luma only (monochrome) 01 = YUV420 sampling 10 is reserved (for YUV422 sampling) 11 is reserved (for YUV444 sampling)

DWord	Bit	Description
	1	**MbaffFrameFlag (MB-AFF Frame Flag)** [PPS]. This field indicates whether the current picture is a progressive frame or a MB-AFF (macroblock adaptive frame field) frame picture. This field is frame Picture Parameter Set. This field is reserved (MBZ) if **FieldPicFlag** = 1. 0 = Progressive frame picture 1 = MB-AFF frame picture
	0	**FieldPicFlag (Field Picture Flag)** [PPS]. This field indicates whether the current picture is a field or a frame picture. A frame picture may be a progressive frame picture or an MB-AFF frame picture depending on the value of **MbaffFrame**. This field is frame Picture Parameter Set. 0 = Frame picture 1 = Field picture
4+1	31:16	**CbpY (Coded Block Pattern Y)** *Not expected to be used by Kernel. Thus, programming this field by host software is optional.*
	15:8	**VertOrigin (Vertical Origin).** This field specifies the vertical origin of current macroblock in the destination picture in units of macroblocks. For field macroblock pair in MBAFF frame, the vertical origins for both macroblocks should be set as if they were located in corresponding field pictures. For example, for field macroblock pair originated at (16, 64) pixel location in an MBAFF frame picture, the Vertical Origin for both macroblocks should be set as 2 (macroblocks). Format = U8 in unit of macroblock.
	7:0	**HorzOrigin (Horizontal Origin).** This field specifies the horizontal origin of current macroblock in the destination picture in units of macroblocks. Format = U8 in unit of macroblock.
4+2	31:14	**Reserved.** MBZ
	13:8	**Coded Block Pattern** [Used by VFE). This field specifies whether blocks (8x8) are present or not. Each bit corresponds to one block. “0” indicates error block isn’t present, “1” indicates error block is present. Bit 13: Y0 Bit 12: Y1 Bit 11: Y2 Bit 10: Y3 Bit 9: Cb Bit 8: Cr *This field is different than the input in AVC-IT mode, but it is the same as the corresponding Root Thread payload field in AVC-IT mode.*
	7:4	**CbpCr** *Not expected to be used by Kernel. Thus, programming this field by host software is optional.*
	3:0	**CbpCb** *Not expected to be used by Kernel. Thus, programming this field by host software is optional.*

DWord	Bit	Description
4+3 to 4+5	31:0	For intra macroblocks, see Section 0(Inline Data Format in AVC-IT Mode) For intra macroblocks, see Error! Reference source not found. in Section 0 (Inline Data Format in AVC-IT Mode) *This field is identical to that in AVC-IT mode.*

Inline data in AVC-MC mode ([DevILK])

DWord	Bit	Description
4+0	31:27	**Reserved.** MBZ
	26:25	**MbAffFieldFlag** This field indicates that the current macroblock is a field macroblock within a MbAff frame picture. It is provided as **Flag = MbaffFrame** & **FieldMbFlag.** 00 = if (Flag == 0) 11 = if (Flag == 1) Other encodings are reserved
	24	**FieldMbPolarityFlag** This field indicates the field polarity of the current macroblock. Within a MbAff frame picture, this field may be different per macroblock and is set to 1 only for the second macroblock in a MbAff pair if FieldMbFlag is set. Otherwise, it is set to 0. Within a field picture, this field is set to 1 if the current picture is the bottom field picture. Otherwise, it is set to 0. It is a constant for the whole field picture. This field is reserved and MBZ for a progressive frame picture. 0 = Current macroblock is a field macroblock from the **top** field 1 = Current macroblock is a field macroblock from the **bottom** field *Programming Note: Here bits [26:24] (MbAffFieldFlag and FiedlMbPolarityFlag) match with bits [10:8] of the Media Block Read message descriptor, simplifying the programming for message generation, as when MbAffFieldFlag is "1", kernels need to override the original "frame" surface state set for MBAFF frame picture.*
	23	**Reserved**: MBZ
	22:20	**MvSize (Motion Vector Size)** [Used by VFE]. This field specifies the size of motion vectors for the macroblock stored in the indirect data buffer. The valid numbers are listed below indicating the size of the regrouped motion vectors. Details are provided in Section 1.7.3.1.4. This field is reserved (MBZ) when **IntraMbFlag** = 1. 000 = 0: No motion vector 001 reserved 010 = 2MV: One motion vector pair 011 reserved 100 = 8MV: Four motion vector pairs 101 = 16MV: 16 motion vectors 110 = 32MV: 16 motion vector pairs 111 reserved

DWord	Bit	Description
	19	**DcBlockCodedYFlag** [Used by VFE]. This field is consumed by VFE and is not delivered to the thread.
	18	**DcBlockCodedCbFlag** [Used by VFE]. This field is consumed by VFE and is not delivered to the thread.
	17	**DcBlockCodedCrFlag** [Used by VFE]. This field is consumed by VFE and is not delivered to the thread.
	16	**Reserved.** MBZ
	15	**Transform8x8Flag** [Used by VFE]. This field equals to the value of *transform_size_8x8_flag* as defined in AVC spec. If set, it indicates that luma samples are in residual 8x8 blocks. Otherwise, it indicates that luma samples are in residual 4x4 blocks. 0: luma residual 4x4 blocks 1: luma residual 8x8 blocks
	14	**FieldMbFlag (Field Macroblock Flag).** This field specifies whether current macroblock is field macroblock. 0 = Frame macroblock. 1 = Field macroblock.
	13	**IntraMbFlag (Intra Macroblock Flag).** This field specifies whether the current macroblock is an Intra (I) macroblock. A collective macroblock type in AVC standard includes I, SI, P and B. Type SI is not supported. 0 = P or B macroblock. 1 = I macroblock.
	12:8	**MbType (Macroblock Type).** This field, along with **IntraMbFlag** specifies the macroblock types. Further details can be found in Section 1.7.3.1.2.
	7:6	**WeightedBiPredFlag (Weighted Bidirectional Prediction Flag)** (from Picture State). This field specifies the bidirectional prediction mode and is derived from syntax elements *weighted_bipred_flag* and *weighted_pred_flag* as defined in AVC spec. It is valid only for inter predicted macroblock. Otherwise (intra macroblock), this field is reserved and MBZ. For B-macroblock, this field is the same as *weighted_bipred_flag* as defined in AVC spec. 00 = Default weighted prediction 01 = Explicit weighted prediction 10 = Implicit weighted prediction 11 = Reserved. For P-macroblock, the MSB is always 0 and the LSB is the same as *weighted_pred_flag* as defined in AVC spec. 00 = Default weighted prediction 01 = Explicit weighted prediction 10 and 11 are reserved

DWord	Bit	Description
	5	**WeightedPredFlag (Weighted Prediction Flag)** [from Picture State]. It is valid only for inter predicted macroblock. Otherwise (intra macroblock), this field is reserved and MBZ. 0 = Default weighted prediction 1 = Explicit weighted prediction *Note: Information in this field is also carried in* **WeightedBiPredFlag**.
	4	**Reserved.** MBZ
	3:2	**ChromaFormatIdc (Chroma Format Indicator).** This field is equal to the value of *chroma_format_idct* as defined in AVC (§7.4.2.1). It specifies the chroma sampling relative to the luma sampling. This field is constant within a picture. 00 = Luma only (monochrome) 01 = YUV420 sampling 10 is reserved (for YUV422 sampling) 11 is reserved (for YUV444 sampling)
	1	**MbaffFrameFlag (MB-AFF Frame Flag)** [PPS]. This field indicates whether the current picture is a progressive frame or a MB-AFF (macroblock adaptive frame field) frame picture. This field is frame Picture Parameter Set. This field is reserved (MBZ) if **FieldPicFlag** = 1. 0 = Progressive frame picture 1 = MB-AFF frame picture
	0	**FieldPicFlag (Field Picture Flag)** [PPS]. This field indicates whether the current picture is a field or a frame picture. A frame picture may be a progressive frame picture or an MB-AFF frame picture depending on the value of **MbaffFrame**. This field is frame Picture Parameter Set. 0 = Frame picture 1 = Field picture
4+1	31:16	**CbpY (Coded Block Pattern Y)** [Used by VFE]
	15:8	**VertOrigin (Vertical Origin).** This field specifies the vertical origin of current macroblock in the destination picture in units of macroblocks. For field macroblock pair in MBAFF frame, the vertical origins for both macroblocks should be set as if they were located in corresponding field pictures. For example, for field macroblock pair originated at (16, 64) pixel location in an MBAFF frame picture, the Vertical Origin for both macroblocks should be set as 2 (macroblocks). Format = U8 in unit of macroblock.
	7:0	**HorzOrigin (Horizontal Origin).** This field specifies the horizontal origin of current macroblock in the destination picture in units of macroblocks. Format = U8 in unit of macroblock.
4+2	31:24	**QpPrimeCr** [Used by VFE]
	23:16	**QpPrimeCb** [Used by VFE]
	15:8	**QpPrimeY** [Used by VFE]
	7:4	**CbpCr** [Used by VFE]
	3:0	**CbpCb** [Used by VFE]

DWord	Bit	Description
4+3 to 4+5	31:0	intra macroblocks intra macroblocks Error! Reference source not found.
4+6	31:16	**LevelScaleCb.** [Used by VFE] This field is for inverse transform of the Chroma (Cb) DC block. The LevelScale field is consumed by VFE and is not needed by the thread, but since the GRF has to be filled to the end of the block it should be sent anyways.
	15:0	**LevelScaleCr.** [Used by VFE] This field is for inverse transform of the Chroma (Cr) DC block.
4+7	31:16	**Reserved.** MBZ
	15:0	**LevelScaleY.** [Used by VFE] This field is for inverse transform of the Luma DC block.
4+8	31:0	**BSD_Pass_Thru_Data**
4+9	31:0	**BSD_Pass_Thru_Data**
4+10	31:0	**BSD_Pass_Thru_Data**
4+11	31:0	**BSD_Pass_Thru_Data**

1.7.3.4 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. Weight-Offset block, if present, is always packed behind the Motion Vector block. See Section 1.7.3.2 for more details.

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

1.7.3.5 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.

The table below 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.

Inline data in VC1-IT mode

DWord	Bit	Description
4+0	31:28	**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 <u>top</u> reference field. 1 = The prediction is taken from the <u>bottom</u> reference field.
	27	**Reserved.** MBZ
	26	**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 <u>top</u> reference field. 1 = The prediction is taken from the <u>bottom</u> reference field.
	25:24	**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.
	23	**Reserved.** MBZ
	22	**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

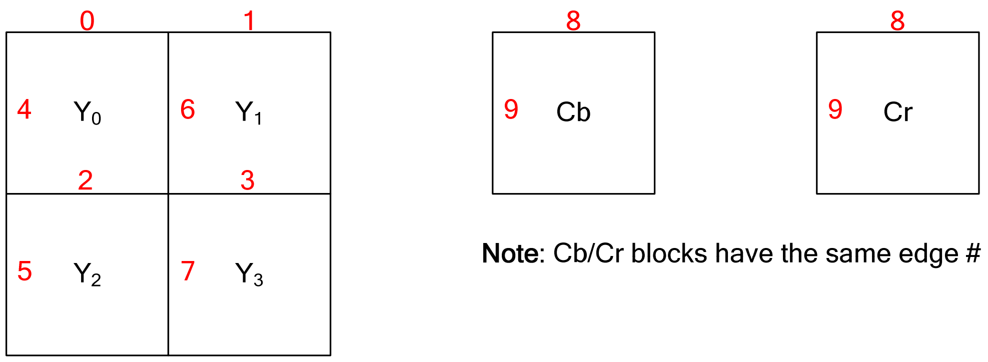
DWord	Bit	Description
	21	**DctType.** This field specifies whether the residual data is coded as field residual or frame residual for interlaced picture. This field can be 1 only if MotionType is 00 (intra) or 01 (field motion). For progressive picture, this field must be set to '0', i.e. all macrobalcoks 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
	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** OPEN: This field is not valid for a field motion in an interlaced frame picture where 4 MVs for chroma blocks. Notes: This field is derived from MotionVector[3:0] as described in the following section.

DWord	Bit	Description
4+7	32:24	**Subblock Code for Y3** [Used by VFE] The following subblock coding definition applies to all 6 subblock coding bytes. Bits 7:6 are reserved.

Subblock Partitioning (Bits [1:0])		**Subblock Present (0 means not present, 1 means present)**			
Code	**Meaning**	**Bit 2**	**Bit 3**	**Bit 4**	**Bit 5**
00	Single 8x8 block (sb0)	Sb0	Don't care	Don't care	Don't care
01	Two 8x4 subblocks (sb0-1)	Sb0	Sb1	Don't care	Don't care
10	Two 4x8 subblocks (sb0-1)	Sb0	Sb1	Don't care	Don't care
11	Four 4x4 subblocks (sb0-3)	Sb0	Sb1	Sb2	Sb3

DWord	Bit	Description
	23;16	**Subblock Code for Y2** [Used by VFE]
	15:8	**Subblock Code for Y1** [Used by VFE]
	7:0	**Subblock Code for Y0** [Used by VFE]
4+8	31:16	**Reserved.** MBZ
	15:8	**Subblock Code for Cr** [Used by VFE]
	7:0	**Subblock Code for Cb** [Used by VFE]



Note: Cb/Cr blocks have the same edge #

Indexing Block Edges for Overlapped Smoothing

1.7.3.5.1 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

```
o  LMV_X = MV_X;
o  if (Current_field != BOTTOM_FIELD)
       ▪  iy = MV_Y >> 2;               // Interger portion of MV_Y
o  else                                 // Current_field == BOTTOM_FIELD
       ▪  iy = (MV_Y  >> 2) + 1;        // Interger portion of MV_Y adjusted
o  MVFS = iy & 1;                       // 0 – top field, 1 – bottom field
o  LMV_Y = ((iy >>1)<< 2) + (MV_Y & 3);
```

1.7.3.5.2 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);
```

1.7.3.6 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.

1.7.3.7 Inline Data Format in Generic Mode [DebCTG+]

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.

1.7.4 MEDIA_OBJECT_PRT Command [DevCTG+]

[**DevBW/DevCL**] This command is not supported.

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.

Note for **[DevCTG+]**: This command is supported for generating a general-purpose root thread independent of the VFE mode.

Note for **[DevILK+]**: Thread generated by this command always bypasses the hardware scoreboard if the hardware scoreboard is enabled.

Dword	Bits	Description
0	31:29	**Command Type =** GFXPIPE = 3h
	28:16	**Media Command Opcode =** MEDIA_OBJECT_PRT Pipeline[28:27] = Media = 2h; Opcode[26:24] **=** 1h; Subopcode[23:16] = 02h
	15:0	**DWord Length** (Excludes DWords 0,1) Valid range: [3, 14] **Note:** Regardless of the mode, inline data must be present in this command. The command size must fit within 16 dwords.
1	31:0	**Reserved.** MBZ
2	31	**Reserved.** MBZ
	30:24	**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 = U7

Dword	Bits	Description
	23	**PRT_Fence Needed.** 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. Format = Enable
	22:21	**Reserved.** MBZ
	20	**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. *In VLD mode, this field must be programmed as 0, because the* ***Children Present*** *field in VFE_STATE must be 0 in this mode.* 0 = No thread synchronization 1 = Thread dispatch is synchronized by the "spawn root thread" message
	19:0	**Reserved.** MBZ **(**was **Indirect Data Length)**
3	31:0	**Reserved.** MBZ **(**was **Indirect Data Start Address)**
4..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.

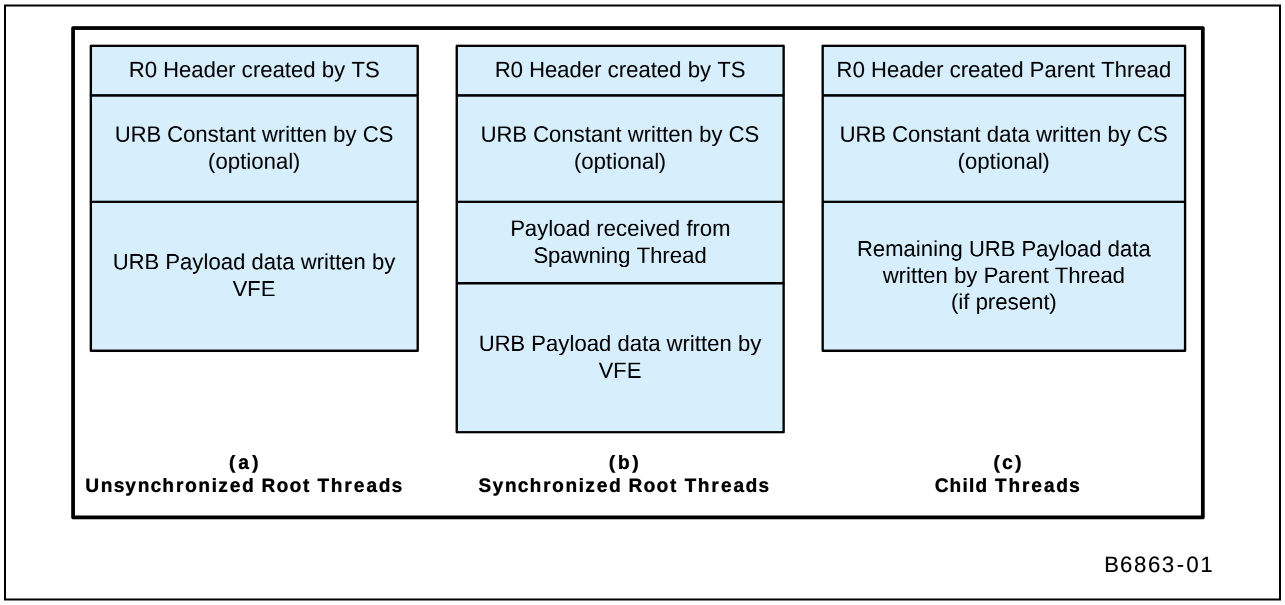
Dispatch Messages: **R**p.d

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 1-5. 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 1-5. Thread payload message formats for root and child threads


R0 Header created by TS
URB Constant written by CS (optional)
URB Payload data written by VFE
(a) Unsynchronized Root Threads
R0 Header created by TS
URB Constant written by CS (optional)
Payload received from Spawning Thread
URB Payload data written by VFE
(b) Synchronized Root Threads
R0 Header created Parent Thread
URB Constant data written by CS (optional)
Remaining URB Payload data written by Parent Thread (if present)
(c) Child Threads
B6863-01

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.

R0 header of a generic mode root thread

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

DWord	Bit	Description
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]
	3:0	Reserved : MBZ
	27:24	**BarrierID**. This field indicates which one from the 16 Barriers this kernel is associated. Format: U4
	23:21	Reserved : MBZ
	20:16	**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
	15:4	Reserved : MBZ
	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	**[DevILK**] **Scoreboard Mask 4-7** (only with MEDIA_OBJECT_EX)**:** Each bit indicates the corresponding dependency scoreboard is dependent on. This field is AND'd with the corresponding Scoreboard Mask field in the VFE_STATE_EX. **Bit n (for n = 4…7):** Scoreboard n is dependent, where bit 28 maps to n = 4. Format = TRUE/FALSE **[Pre-DevILK**] Reserved : MBZ
	27:26	**[DevILK**] **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 = U2 [Pre-DevILK] Reserved : MBZ
	25	**Reserved.** MBZ

DWord	Bit	Description
	24:16	**[DevILK+] Scoreboard Y** This field provides the Y term of the scoreboard value of the current thread. Format = U9 [DevILK] only with MEDIA_OBJECT_EX. [Pre-DevILK] Reserved : MBZ
	15:12	[DevILK] **Scoreboard Mask 0-3** (only with MEDIA_OBJECT_EX)**:** Each bit indicates the corresponding dependency scoreboard is dependent on. This field is AND'd with the corresponding Scoreboard Mask field in the VFE_STATE_EX. **Bit n (for n = 0…3):** Scoreboard n is dependent, where bit 12 maps to n = 0. Format = TRUE/FALSE [Pre-DevILK] 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 [DevILK] only with MEDIA_OBJECT_EX. [Pre-DevILK] Reserved : MBZ
	23:16	[DevILK+] **Thread Dependency Identifier (TDID)**. This field is assigned by TS to be used for hardware scoreboard. Format = U8 [Pre-DevILK] 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.

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).

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

1.8.1.3 IS-Mode Root Thread

The following table shows the root thread payload messages when VFE is in IS mode and URB push constant is not enabled.

DWord	Bit	Description
R0.7	31	**Reserved**
	27:24	**Reserved**
	23:0	**Reserved** : MBZ.
R0.6	31:24	**Reserved**
	23:0	**Reserved**
R0.5	31:10	**NOT USED** (was Scratch Space Pointer).
	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. Note: Nothing to free up in this case.
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	**NOT USED** (was Sampler State Pointer).

DWord	Bit	Description
	4	Reserved : MBZ
	3:0	**NOT USED** (was Per Thread Scratch Space)
R0.2	31:5	**Interface Descriptor Pointer.** Specifies the 32-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[31:5]
	4:0	**Reserved :** MBZ
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. This may be used if child threads and/or synchronized root threads are present in IS mode.
R1.7	31:16	**Motion Vectors – Field 1, Backward, Vertical Component.** Each vector component is a 16-bit two's-complement value. The vector is relative to the current macroblock location. According to ISO/IEC 13818-2 Table 7-8, the valid range of each vector component is [-2048, +2047.5], implying a format of s11.1. However, it should be noted that motion vector values are sign extended to 16 bits.
	15:0	**Motion Vectors – Field 1, Backward, Horizontal Component**
R1.6	31:16	**Motion Vectors – Field 1, Forward, Vertical Component**
	15:0	**Motion Vectors – Field 1, Forward, Horizontal Component**
R1.5	31:16	**Motion Vectors – Field 0, Backward, Vertical Component**
	15:0	**Motion Vectors – Field 0, Backward, Horizontal Component**
R1.4	31:16	**Motion Vectors – Field 0, Forward, Vertical Component**
	15:0	**Motion Vectors – Field 0, Forward, Horizontal Component**
R1.3	31:24	**Subblock Coding for Block Cr5**
	23:16	**Subblock Coding for Block Cb4**
	15:8	**Subblock Coding for Block Y3**
	7:0	**Subblock Coding for Block Y2**
R1.2	31:24	**Subblock Coding for Block Y1**
	23:16	**Subblock Coding for Block Y0.** This field specifies the subblock partition and subblock coding pattern for the block. The definition of the 8 bits of this field is listed below. Bits [7:6]: reserved Bits [5:2]: Subblock present Bits [1:0]: Subblock partitioning
	15:12	**Reserved.**

DWord	Bit	Description
	11:6	**Coded Block Pattern.** This field specifies whether blocks are present or not. Format = 6-bit mask. Bit 11: Y0 Bit 10: Y1 Bit 9: Y2 Bit 8: Y3 Bit 7: Cb5 Bit 6: Cr5
	5:0	**Reserved.**
R1.1	31:24	**Reserved.** (Skip Macroblocks)
	23:0	**Reserved.** (Offset into error data)
R1.0	31:28	**Motion Vertical Field Select.** A bit-wise representation of a long [2][2] array as defined in §6.3.17.2 of the *ISO/IEC 13818-2* (see also §7.6.4). Bit | MVector [r] | MVector [s] | MotionVerticalFieldSelect Index 28 | 0 | 0 | 0 29 | 0 | 1 | 1 30 | 1 | 0 | 2 31 | 1 | 1 | 3 Format = MC_MotionVerticalFieldSelect. 0 = The prediction is taken from the <u>top</u> reference field. 1 = The prediction is taken from the <u>bottom</u> reference field.
	27	**Second Field.** This bit indicates that this is the second field in the current frame. The prediction for this macroblock, if it belongs to a field P-picture, should use this bit to determine which frame contains the reference field as described in §7.6.2.1 of the *ISO/IEC 13818-2*. When the picture type is not P or the prediction type is not field, this bit is set to 0. Format = MC_SecondPField 0 = This is not the second field. 1 = This is the second field.
	26	**Reserved.** (HWMC mode)
	25:24	**Motion Type.** When combined with the destination picture type (field or frame) this Motion Type field indicates the type of motion to be applied to the macroblock. See *ISO/IEC 13818-2* §6.3.17.1, Tables 6-17, 6-18. In particular, the device supports dual-prime motion prediction (11) in both frame and field picture type. Format = MC_MotionType Value | Destination = Frame Picture_Structure = 11 | Destination = Field Picture_Structure != 11 '00' | Reserved | Reserved '01' | Field | Field '10' | Frame | 16x8 '11' | Dual-Prime | Dual-Prime

DWord	Bit	Description
	23:22	**Reserved.** (Scan method)
	21	**DCT Type.** This field specifies the DCT type of the current macroblock. The kernel should ignore this field when processing Cb/Cr data. See *ISO/IEC 13818-2* §6.3.17.1. This field is zero if Coded Block Pattern is also zero (no coded blocks present). 0 = MC_FRAME_DCT (Macroblock is frame DCT coded). 1 = MC_FIELD_DCT (Macroblock is field DCT coded).
	20	**Reserved.** (H261 Loop Filter)
	19	**Reserved.** (H263)
	18	**Macroblock Motion Backward.** This field specifies if the backward motion vector is active. See *ISO/IEC 13818-2* Tables B-2 through B-4. 0 = No backward motion vector. 1 = Use backward motion vector(s).
	17	**Macroblock Motion Forward.** This field specifies if the forward motion vector is active. See *ISO/IEC 13818-2* Tables B-2 through B-4. 0 = No forward motion vector. 1 = Use forward motion vector(s).
	16	**Macroblock Intra Type.** 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). See *ISO/IEC 13818-2* Tables B-2 through B-4. 0 = Non-intra macroblock. 1 = Intra macroblock.
	15:0	**Reserved.**
R2.7	31:0	**Reserved.**
R2.6	31:0	**Reserved.**
R2.5	31:0	**Reserved.**
R2.4	31:0	**Reserved.**
R2.3	31:0	**Reserved.**
R2.2	31:0	**Reserved.**
R2.1	31:27	**Reserved.**

	26:20	**Vertical Origin.** Set the vertical origin of the next macroblock in the destination picture in units of macroblocks. (Valid range is 0 to 120).
		Format = U7 in macroblock units.
		Range = [0, 120]

	19:11	**Reserved:** MBZ

	10:4	**Horizontal Origin.** Set the horizontal origin of the next macroblock in the destination picture in units of macroblocks.
		Format = U7 in macroblock units.
		Range = [0, 127]

	3:0	**Reserved.**
R2.0	31:30	**Reserved.**

DWord	Bit	Description
	29	**May need this for WMV.** (Interpolation Rounder Control)
	28	**May need this for WMV.** (Bidirectional Averaging Control)
	27:20	**Reserved.**
	19:18	**Picture Coding Type???.** This field identifies whether the picture is an intra-coded picture (I), predictive-coded picture (P) or bi-directionally predictive-coded picture (B). See *ISO/IEC 13818-2* §6.3.9 for details. Format = MPEG_PICTURE_CODING_TYPE 00 = Reserved 01 = MPEG_I_PICTURE 10 = MPEG_P_PICTURE 11 = MPEG_B_PICTURE
	17:16	**Picture Structure???.** This field specifies whether the picture is encoded in the form of a frame picture or one field (top or bottom) picture. See *ISO/IEC 13818-2* §6.3.10 for details. Format = MPEG_PICTURE_STRUCTURE 00 = Reserved 01 = MPEG_TOP_FIELD 10 = MPEG_BOTTOM_FIELD 11 = MPEG_FRAME
	15	**Reserved.** (8-bit Intra)
	14:13	**Reserved.** (Intra DC Precision)
	12:0	**Reserved.**
None (0 blocks coded) or R3-R[2+4x] where x = number of coded blocks		**DCT Coefficients.** These are the DCT values of the coefficients for the macroblock. Only coded blocks have coefficients present in the array. Beginning in R3, the order of the coefficients for the coded blocks is Y0, Y1, Y2, Y3, Cb4, and Cr5. For each coded block, the 8x8 DCT coefficients, with 1 word each coefficient, are organized in row-major order, occupying four GRF registers. This is shown in Table 1-1, where the index-pair for a DCT coefficient is (Column_Index, Row_Index).

1.8.1.4 VLD-Mode Root Thread

The following table shows the root thread payload messages when VFE is in VLD mode and URB push constant is not enabled. When URB push constant is enabled, it will start at R1. Subsequently, macroblock data starting with motion vectors will be put in GRF registers after the URB push constants.

DWord	Bit	Description
R0.7	31	**Reserved**
	27:24	**Reserved**
	23:0	**Reserved** : MBZ.
R0.6	31:24	**Reserved**
	23:0	**Reserved**
R0.5	31:10	**NOT USED (was Scratch Space Pointer).**
	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. Note: Nothing to free up in this case.
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	**NOT USED (was Sampler State Pointer).**
	4	Reserved : MBZ
	3:0	**NOT USED (was Per Thread Scratch Space)**
R0.2	31:5	**Interface Descriptor Pointer.** Specifies the 32-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[31:5]
	4:0	**Reserved :** MBZ
R0.1	31:0	**Reserved :** MBZ
R0.0	31:16	**Reserved :** MBZ
	15:0	**NOT USED (was URB Handle)**
R1.7	31:16	**Motion Vectors – Field 1, Backward, Vertical Component.** Each vector component is a 16-bit two's-complement value. The vector is relative to the current macroblock location. According to ISO/IEC 13818-2 Table 7-8, the valid range of each vector component is [-2048, +2047.5], implying a format of s11.1. However, it should be noted that motion vector values are sign extended to 16 bits.
	15:0	**Motion Vectors – Field 1, Backward, Horizontal Component**
R1.6	31:16	**Motion Vectors – Field 1, Forward, Vertical Component**

DWord	Bit	Description
	15:0	**Motion Vectors – Field 1, Forward, Horizontal Component**
R1.5	31:16	**Motion Vectors – Field 0, Backward, Vertical Component**
	15:0	**Motion Vectors – Field 0, Backward, Horizontal Component**
R1.4	31:16	**Motion Vectors – Field 0, Forward, Vertical Component**
	15:0	**Motion Vectors – Field 0, Forward, Horizontal Component**
R1.3	31:27	**Reserved.**

	26:20	**Vertical Origin.** Set the vertical origin of the next macroblock in the destination picture in units of macroblocks. (Valid range is 0 to 120).
		Format = U7 in macroblock units.
		Range = [0, 120]

	19:11	**Reserved:** MBZ

	10:4	**Horizontal Origin.** Set the horizontal origin of the next macroblock in the destination picture in units of macroblocks.
		Format = U7 in macroblock units.
		Range = [0, 127]

	3:0	**Reserved.**
R1.2	31:30	**Reserved.**
	29	**Reserved.** (Interpolation Rounder Control)
	28	**Reserved.** (Bidirectional Averaging Control)
	27:20	**Reserved.**
	19:18	**Picture Coding Type.** This field identifies whether the picture is an intra-coded picture (I), predictive-coded picture (P) or bi-directionally predictive-coded picture (B). See *ISO/IEC 13818-2* §6.3.9 for details. Format = MPEG_PICTURE_CODING_TYPE 00 = Reserved 01 = MPEG_I_PICTURE 10 = MPEG_P_PICTURE 11 = MPEG_B_PICTURE
	17:16	**Picture Structure.** This field specifies whether the picture is encoded in the form of a frame picture or one field (top or bottom) picture. See *ISO/IEC 13818-2* §6.3.10 for details. Format = MPEG_PICTURE_STRUCTURE 00 = Reserved 01 = MPEG_TOP_FIELD 10 = MPEG_BOTTOM_FIELD 11 = MPEG_FRAME
	15	**Reserved.** (8-bit Intra)
	14:13	**Intra DC Precision.** See ISO/IEC 13818-2 §6.3.10 for details.
	12	**Disable Mismatch.** This bit is used to disable the mismatch control performed after the inverse quantization operation, as described in ISO/IEC 13818-2 §7.4.4

DWord	Bit	Description
	11:6	**Coded Block Pattern.** This field specifies whether blocks are present or not. Format = 6-bit mask. Bit 11: Y0 Bit 10: Y1 Bit 9: Y2 Bit 8: Y3 Bit 7: Cb5 Bit 6: Cr5
	5	**Quantizer Scale Type**: This field specifies the quantizer scaling type. Format = MPEG_Q_SCALE_TYPE 0: MPEG_QSCALE_LINEAR 1: MPEG_QSCALE_NONLINEAR
	4:0	**Quantization Scale Code.** Combined with the quantization scale type, this value selects the quantizer scale table according to ISO/IEC 13818-2 Table 7-6
R1.1	31:24	**Reserved.** (Skip Macroblocks)
	23:0	**Reserved.** (Offset into error data)
R1.0	31:28	**Motion Vertical Field Select.** A bit-wise representation of a long [2][2] array as defined in §6.3.17.2 of the *ISO/IEC 13818-2* (see also §7.6.4). (see table below) Format = MC_MotionVerticalFieldSelect. 0 = The prediction is taken from the top reference field. 1 = The prediction is taken from the bottom reference field.
	27	**Second Field.** This bit indicates that this is the second field in the current frame. The prediction for this macroblock, if it belongs to a field P-picture, should use this bit to determine which frame contains the reference field as described in §7.6.2.1 of the *ISO/IEC 13818-2*. When the picture type is not P or the prediction type is not field, this bit is set to 0. Format = MC_SecondPField 0 = This is not the second field. 1 = This is the second field.
	26	**Reserved.** (HWMC mode)

Bit	MVector [r]	MVector [s]	MotionVerticalFieldSelect Index
28	0	0	0
29	0	1	1
30	1	0	2
31	1	1	3

DWord	Bit	Description
	25:24	**Motion Type.** When combined with the destination picture type (field or frame) this Motion Type field indicates the type of motion to be applied to the macroblock. See *ISO/IEC 13818-2* §6.3.17.1, Tables 6-17, 6-18. In particular, the device supports dual-prime motion prediction (11) in both frame and field picture type. Format = MC_MotionType

Value	Destination = Frame Picture_Structure = 11	Destination = Field Picture_Structure != 11
'00'	Reserved	Reserved
'01'	Field	Field
'10'	Frame	16x8
'11'	Dual-Prime	Dual-Prime

DWord	Bit	Description
	23:22	**Reserved.** (Scan method)
	21	**DCT Type.** This field specifies the DCT type of the current macroblock. The kernel should ignore this field when processing Cb/Cr data. See *ISO/IEC 13818-2* §6.3.17.1. This field is zero if Coded Block Pattern is also zero (no coded blocks present). 0 = MC_FRAME_DCT (Macroblock is frame DCT coded). 1 = MC_FIELD_DCT (Macroblock is field DCT coded).
	20	**Reserved.** (H261 Loop Filter)
	19	**Reserved.** (H263)
	18	**Macroblock Motion Backward.** This field specifies if the backward motion vector is active. See *ISO/IEC 13818-2* Tables B-2 through B-4. 0 = No backward motion vector. 1 = Use backward motion vector(s).
	17	**Macroblock Motion Forward.** This field specifies if the forward motion vector is active. See *ISO/IEC 13818-2* Tables B-2 through B-4. 0 = No forward motion vector. 1 = Use forward motion vector(s).
	16	**Macroblock Intra Type.** 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). See *ISO/IEC 13818-2* Tables B-2 through B-4. 0 = Non-intra macroblock. 1 = Intra macroblock.
	15:0	**Reserved.**

DWord	Bit	Description
None (0 block coded) or R2-R[1+4x] where x = number of coded blocks		**DCT Coefficients.** These are the DCT values of the coefficients for the macroblock. Only coded blocks have coefficients present in the array. Beginning in R2, the order of the coefficients for the coded blocks is Y0, Y1, Y2, Y3, Cb4, and Cr5. For each coded block, the 8x8 DCT coefficients, with 1 word each coefficient, are organized in row-major order, occupying four GRF registers. This is shown in Table 1-1, where the index-pair for a DCT coefficient is (Column_Index, Row_Index).

Table 1-1. Format of a block of DCT coefficients in GRF registers

Reg. / Words	W15	W14	W13	W12	W11	W10	W9	W8	W7	W6	W5	W4	W3	W2	W1	W0
R[n]	(7,1)	(6,1)	(5,1)	(4,1)	(3,1)	(2,1)	(1,1)	(0,1)	(7,0)	(6,0)	…	…	…	…	(1,0)	(0,0)
R[n+1]	(7,3)	…	…	…	…	…	…	(0,3)	(7,2)	(6,2)	…	…	…	…	(1,2)	(0,2)
R[n+2]	(7,5)	…	…	…	…	…	…	(0,5)	(7,4)	(6,4)	…	…	…	…	(1,4)	(0,4)
R[n+3]	(7,7)	(7,7)	(5,7)	(4,7)	(3,7)	(2,7)	(1,7)	(0,7)	(7,6)	(6,6)	…	…	…	…	(1,6)	(0,6)

* W# (# from 0 to 15) represents WORD location # within an 8-DW register.

1.8.1.5 AVC-IT Mode Root Thread [DevCTG, DevILK]

The following table shows the root thread payload messages when VFE is in AVC-IT mode. One example GRF register location is given for the condition that: CURBE is disabled, Weight-Offset Offset (p) = 4 and Residual Offset (q) = 6.

AVC-IT Mode root thread payload layout

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. If m is 0, this field doesn't exist.
R(m+1)	R1	**In-line Data block**.
R(m+2) – R(m+2+n-1)	R2 – R5	**Indirect Data Motion Vector blocks** If maximum MvSize is 32, the number of GRF space occupied by motion vector, n, is 4. Otherwise (if maximum MvSize is 16), n is 2.
R(m+p+2) – R(m+p+3)	R6 – R7	**Indirect Data Weight-Offset blocks** Weight-Offset Offset (p) must be greater than or equal to n. When p is greater than n, the gap, which is R(m+2+n) to R(m+p+1), contains undefined values.
R(m+q+2) – R(m+q+13)	R8 – R31	**Indirect Data Residual blocks** Residual Offset (q) must be greater than or equal to p+2. When q is greater than p+2, the gap, which is R(m+p+4) to R(m+q+1), contains undefined values.

The R0 header field is the same as in other modes.

AVC-IT Mode R0 Header

DWord	Bit	Description
R0.0	31:24	**Reserved :** MBZ
	23:16	[Dev**ILK]: Thread Dependency Identifier (TDID)**. This field is assigned by TS to be used for hardware scoreboard. Format = U8 [DevCTG] 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. This may be used if child threads and/or synchronized root threads are present in IS mode.
R0.1	31:28	[DevILK] **Scoreboard Mask 4-7** (only with MEDIA_OBJECT_EX)**:** Each bit indicates the corresponding dependency scoreboard is dependent on. This field is AND'd with the corresponding Scoreboard Mask field in the VFE_STATE_EX. **Bit n (for n = 4…7):** Scoreboard n is dependent, where bit 28 maps to n = 4. Format = TRUE/FALSE [Pre-DevILK] Reserved : MBZ

DWord	Bit	Description
	27:26	[DevILK] **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 = U2 [Pre-DevILK] Reserved : MBZ
	25	**Reserved.** MBZ
	24:16	[DevILK] **Scoreboard Y**(only with MEDIA_OBJECT_EX) This field provides the Y term of the scoreboard value of the current thread. Format = U9 [Pre-DevILK] Reserved : MBZ
	15:12	[DevILK] **Scoreboard Mask 0-3** (only with MEDIA_OBJECT_EX)**:** Each bit indicates the corresponding dependency scoreboard is dependent on. This field is AND'd with the corresponding Scoreboard Mask field in the VFE_STATE_EX. **Bit n (for n = 0…3):** Scoreboard n is dependent, where bit 12 maps to n = 0. Format = TRUE/FALSE [Pre-DevILK] Reserved : MBZ
	11:9	**Reserved.** MBZ
	8:0	[DevILK] **Scoreboard X** (only with MEDIA_OBJECT_EX) This field provides the X term of the scoreboard value of the current thread. Format = U9 [Pre-DevILK] Reserved : MBZ
R0.2	31:5	**Interface Descriptor Pointer.** Specifies the 32-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[31:5]
	4:0	**Reserved :** MBZ
R0.3	31:5	**NOT USED** (was Sampler State Pointer).
	4	Reserved : MBZ
	3:0	**NOT USED** (was Per Thread Scratch Space)
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
R0.5	31:10	**NOT USED** (was Scratch Space Pointer).
	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. Note: Nothing to free up in this case.
R0.6	31:24	**Reserved**
	23:0	**Reserved**

DWord	Bit	Description
R0.7	31	**Reserved**
	27:24	**Reserved**
	23:0	**Reserved : MBZ.**

The in-line data block differs from the in-line data in the MEDIA_OBJECT_EX command by one field – QpPrimeY is replaced by the DerivedMbInfo field.

AVC-IT In-line data block

DWord	Bit	Description
R+0.0	31:28	**Reserved.** MBZ
	26:25	**MbAffFieldFlag** This field indicates that the current macroblock is a field macroblock within a MbAff frame picture. It is provided as **Flag = MbaffFrame** & **FieldMbFlag.** 00 = if (Flag == 0) 11 = if (Flag == 1) Other encodings are reserved
	24	**FieldMbPolarityFlag** This field indicates the field polarity of the current macroblock. Within a MbAff frame picture, this field may be different per macroblock and is set to 1 only for the second macroblock in a MbAff pair if FieldMbFlag is set. Otherwise, it is set to 0. Within a field picture, this field is set to 1 if the current picture is the bottom field picture. Otherwise, it is set to 0. It is a constant for the whole field picture. This field is reserved and MBZ for a progressive frame picture. 0 = Current macroblock is a field macroblock from the **top** field 1 = Current macroblock is a field macroblock from the **bottom** field *Programming Note: Here bits [26:24] (MbAffFieldFlag and FiedlMbPolarityFlag) match with bits [10:8] of the Media Block Read message descriptor, simplifying the programming for message generation, as when MbAffFieldFlag is "1", kernels need to override the original "frame" surface state set for MBAFF frame picture.*
	23	**Reserved**: MBZ
	22:17	**Reserved**: MBZ (Forced to zero by VFE to simplify use of this word for data port message)
	16	**Reserved.** MBZ
	15	**Transform8x8Flag**
	14	**FieldMbFlag (Field Macroblock Flag).** This field specifies whether current macroblock is field macroblock. 0 = Frame macroblock. 1 = Field macroblock.
	13	**IntraMacroblock (Intra Macroblock Indicator)**
	12:8	**MbType (Macroblock Type).** This field, along with "IntraMacroblock" specifies the macroblock types.

DWord	Bit	Description
	7:6	**WeightedBiPredFlag (Weighted Bidirectional Prediction Flag)** (from Picture State). Valid only for macroblock in inter mode. Otherwise (intra macroblock), this field is reserved.
	5	**WeightedPredFlag.** (from Picture State). Valid only for macroblock in inter mode. Otherwise (intra macroblock), this field is reserved.
	4	**Reserved.** MBZ
	3:2	**ChromaFormatIdc (Chroma Format Indicator).** This field specifies the chroma format for the decoding process as defined below. This field is constant within a picture. 00 = Luma only (monochrome) 01 = YUV420 sampling 10 is reserved (for YUV422 sampling) 11 is reserved (for YUV444 sampling)
	1	**MbaffFrame.** (from Picture State).
	0	**FieldPicFlag.** (from Picture State)
R+0.1	31:16	**CbpY (Coded Block Pattern Y)** *Not expected to be used by Kernel.*
	15:8	**VertOrigin (Vertical Origin).** This field specifies the vertical origin of current macroblock in the destination picture in units of macroblocks. For field macroblock pair in MBAFF frame, the vertical origins for both macroblocks should be set as if they were located in corresponding field pictures. For example, for field macroblock pair originated at (16, 64) pixel location in an MBAFF frame picture, the Vertical Origin for both macroblocks should be set as 2 (macroblocks). Format = U8 in unit of macroblock.
	7:0	**HorzOrigin (Horizontal Origin).** This field specifies the horizontal origin of current macroblock in the destination picture in units of macroblocks. Format = U8 in unit of macroblock.
R+0.2	31:14	**Reserved.** MBZ (Forced to Zero by VFE)
	13:8	**Coded Block Pattern** (VFE Derived). This field specifies whether blocks (8x8) are present or not. Each bit corresponds to one block. “0” indicates error block isn’t present, “1” indicates error block is present. Bit 13: Y0 Bit 12: Y1 Bit 11: Y2 Bit 10: Y3 Bit 9: Cb Bit 8: Cr
	7:4	**CbpCr** *Not expected to be used by Kernel.*
	3:0	**CbpCb** *Not expected to be used by Kernel.*

DWord	Bit	Description
R+0.3 – R+0.5	31:0 each	**InterIntraSpecificFields** See the tables below for intra macroblocks and inter macroblocks
R+0.6	31:0	**Reserved.** MBZ
R+0.7	31:0	**Reserved.** MBZ

(R+0.3 to R+0.5) for an Intra Macroblock

Dword	Bit	Description
R+0.3	31:8	Reserved
	7:0	**MbIntraStruct (Macroblock Intra Structure)** Bits | MotionVerticalFieldSelect Index 7:6 | **ChromaIntraPredMode** 5 | Reserved 4 | **IntraPredAvailFlagE – E (Left First Half)** 3 | **IntraPredAvailFlagD – D** 2 | **IntraPredAvailFlagC – C** 1 | **IntraPredAvailFlagB – B** 0 | **IntraPredAvailFlagA – A (Left Second Half)**
R+0.4	31:16	**LumaIndraPredModes[1]**
	15:0	**LumaIndraPredModes[0]**
R+0.5	31:16	**LumaIndraPredModes[3]**
	15:0	**LumaIndraPredModes[2]**

(R+0.3 to R+0.5) for an Inter Macroblock

DWord	Bit	Description
R+0.3	31:24	**Log2WeightDenomChroma**
	23:16	**Log2WeightDenomLuma**
	15:8	**SubMbPredMode**
	7:0	**SubMbShape**
R+0.4	31:16	**LumaIndraPredModes[1]**
	15:0	**LumaIndraPredModes[0]**
R+0.5	31:16	**LumaIndraPredModes[3]**
	15:0	**LumaIndraPredModes[2]**

The indirect data Motion Vector blocks occupies either 2 or 4 GRF spaces (n = 2 or 4) depending on maximum MvSize value. Motion vector blocks are copied from the indirect data buffer without modification, including the case with MvSize = 16 where only Motion Vectors of List 0 is present. VFE is responsible of zero-filling the remainder if the indirect data do not fill up the GRF space n. As noted by (*) in the table below, when n = 4, R+2.0 to R+3.7 are zero-filled for MvSize = 0 to 16. If n = 2, R+2.0 to R+3.7 are not present.

Table 1-2. AVC-IT indirect data Motion Vector blocks (with n = 4)

DWord	Bit	MvSize = 0	MvSize = 2	MvSize = 8	MvSize = 16	MvSize = 32
R+0.0	31:16	MBZ	**MVVert_L0**	**MVVert_Y0_L0**	**MVVert_Y0_L0**	**MVVert_Y0_L0**
	15:0	MBZ	**MVHorz_L0**	**MVHorz_Y0_L0**	**MVHorz_Y0_L0**	**MVHorz_Y0_L0**
R+0.1	31:16	MBZ	**MVVert_L1**	**MVVert_Y0_L1**	**MVVert_Y1_L0**	**MVVert_Y0_L1**
	15:0	MBZ	**MVHorz_L1**	**MVHorz_Y0_L1**	**MVHorz_Y1_L0**	**MVHorz_Y0_L1**
R+0.2	31:0	MBZ	MBZ	**MV_Y1_L0**	**MV_Y2_L0**	**MV_Y1_L0**
R+0.3	31:0	MBZ	MBZ	**MV_Y1_L1**	**MV_Y3_L0**	**MV_Y1_L1**
R+0.4	31:0	MBZ	MBZ	**MV_Y2_L0**	**MV_Y4_L0**	**MV_Y2_L0**
R+0.5	31:0	MBZ	MBZ	**MV_Y2_L1**	**MV_Y5_L0**	**MV_Y2_L1**
R+0.6	31:0	MBZ	MBZ	**MV_Y3_L0**	**MV_Y6_L0**	**MV_Y3_L0**
R+0.7	31:0	MBZ	MBZ	**MV_Y3_L1**	**MV_Y7_L0**	**MV_Y3_L1**
R+1.0	31:0	MBZ	MBZ	MBZ	**MV_Y8_L0**	**MV_Y4_L0**
R+1.1	31:0	MBZ	MBZ	MBZ	**MV_Y9_L0**	**MV_Y4_L1**
R+1.2	31:0	MBZ	MBZ	MBZ	**MV_Y10_L0**	**MV_Y5_L0**
R+1.3	31:0	MBZ	MBZ	MBZ	**MV_Y11_L0**	**MV_Y5_L1**
R+1.4	31:0	MBZ	MBZ	MBZ	**MV_Y12_L0**	**MV_Y6_L0**
R+1.5	31:0	MBZ	MBZ	MBZ	**MV_Y13_L0**	**MV_Y6_L1**
R+1.6	31:0	MBZ	MBZ	MBZ	**MV_Y14_L0**	**MV_Y7_L0**
R+1.7	31:0	MBZ	MBZ	MBZ	**MV_Y15_L0**	**MV_Y7_L1**
R+2.0	31:0	MBZ (*)	MBZ (*)	MBZ (*)	MBZ (*)	**MV_Y8_L0**
R+2.1	31:0	MBZ (*)	MBZ (*)	MBZ (*)	MBZ (*)	**MV_Y8_L1**
R+2.2	31:0	MBZ (*)	MBZ (*)	MBZ (*)	MBZ (*)	**MV_Y9_L0**
R+2.3	31:0	MBZ (*)	MBZ (*)	MBZ (*)	MBZ (*)	**MV_Y9_L1**
R+2.4	31:0	MBZ (*)	MBZ (*)	MBZ (*)	MBZ (*)	**MV_Y10_L0**
R+2.5	31:0	MBZ (*)	MBZ (*)	MBZ (*)	MBZ (*)	**MV_Y10_L1**
R+2.6	31:0	MBZ (*)	MBZ (*)	MBZ (*)	MBZ (*)	**MV_Y11_L0**
R+2.7	31:0	MBZ (*)	MBZ (*)	MBZ (*)	MBZ (*)	**MV_Y11_L1**
R+3.0	31:0	MBZ (*)	MBZ (*)	MBZ (*)	MBZ (*)	**MV_Y12_L0**
R+3.1	31:0	MBZ (*)	MBZ (*)	MBZ (*)	MBZ (*)	**MV_Y12_L1**
R+3.2	31:0	MBZ (*)	MBZ (*)	MBZ (*)	MBZ (*)	**MV_Y13_L0**

DWord	Bit	MvSize = 0	MvSize = 2	MvSize = 8	MvSize = 16	MvSize = 32
R+3.3	31:0	MBZ (*)	MBZ (*)	MBZ (*)	MBZ (*)	**MV_Y13_L1**
R+3.4	31:0	MBZ (*)	MBZ (*)	MBZ (*)	MBZ (*)	**MV_Y14_L0**
R+3.5	31:0	MBZ (*)	MBZ (*)	MBZ (*)	MBZ (*)	**MV_Y14_L1**
R+3.6	31:0	MBZ (*)	MBZ (*)	MBZ (*)	MBZ (*)	**MV_Y15_L0**
R+3.7	31:0	MBZ (*)	MBZ (*)	MBZ (*)	MBZ (*)	**MV_Y15_L1**

The indirect data Weight-Offset blocks occupies 2 GRF spaces. Data from the indirect data buffer are copied without modification. VFE is responsible of zero-filling the remainder if the indirect data do not fill up the whole 2 GRF space (MvSize is 0 or 2).

AVC-IT indirect data Weight-Offset blocks

Dword	Bit	MvSize		
		0	2	8, 16, 32
R+0.0	31:24	MBZ	**Offset_Y_L1**	**Offset_Y_Block0_L1**
	23:16	MBZ	**Weight_Y_L1**	**Weight_Y_Block0_L1**
	15:8	MBZ	**Offset_Y_L0**	**Offset_Y_Block0_L0**
	7:0	MBZ	**Weight_Y_L0**	**Weight_Y_Block0_L0**
R+0.1	31:16	MBZ	**WO_Cb_L1**	**WO_Cb_Block0_L1**
	15:0	MBZ	**WO_Cb_L0**	**WO_Cb_Block0_L0**
R+0.2	31:16	MBZ	**WO_Cr_L1**	**WO_Cr_Block0_L1**
	15:0	MBZ	**WO_Cr_L0**	**WO_Cr_Block0_L0**
R+0.3	31:0	MBZ	MBZ	MBZ
R+0.4	31:16	MBZ	MBZ	**WO_Y_Block1_L1**
	15:0	MBZ	MBZ	**WO_Y_Block1_L0**
R+0.5	31:16	MBZ	MBZ	**WO_Cb_Block1_L1**
	15:0	MBZ	MBZ	**WO_Cb_Block1_L0**
R+0.6	31:16	MBZ	MBZ	**WO_Cr_Block1_L1**
	15:0	MBZ	MBZ	**WO_Cr_Block1_L0**
R+0.7	31:0	MBZ	MBZ	MBZ
R+1.0	31:16	MBZ	MBZ	**WO_Y_Block2_L1**
	15:0	MBZ	MBZ	**WO_Y_Block2_L0**
R+1.1	31:16	MBZ	MBZ	**WO_Cb_Block2_L1**
	15:0	MBZ	MBZ	**WO_Cb_Block2_L0**
R+1.2	31:16	MBZ	MBZ	**WO_Cr_Block2_L1**
	15:0	MBZ	MBZ	**WO_Cr_Block2_L0**

Dword	Bit	MvSize		
		0	2	8, 16, 32
R+1.3	31:0	MBZ	MBZ	MBZ
R+1.4	31:16	MBZ	MBZ	**WO_Y_Block3_L1**
	15:0	MBZ	MBZ	**WO_Y_Block3_L0**
R+1.5	31:16	MBZ	MBZ	**WO_Cb_Block3_L1**
	15:0	MBZ	MBZ	**WO_Cb_Block3_L0**
R+1.6	31:16	MBZ	MBZ	**WO_Cr_Block3_L1**
	15:0	MBZ	MBZ	**WO_Cr_Block3_L0**
R+1.7	31:0	MBZ	MBZ	MBZ

If MbType != I_PCM, the residual data are fully expanded with zero-fill to take up 24 GRF registers. Each block of 8x8 residual data are stored in GRF in raster-scan order with 16-bit signed integer samples in 2's compliment form.

If MbType = I_PCM, the residual data take up 12 GRF registers. Each block of 8x8 residual data are stored in GRF in raster-scan order with 8-bit unsigned integer samples. The remaining 12 GRF contains undefined values.

AVC-IT Mode indirect data residual data blocks (MbType != I_PCM)

GRF Registers	Description
R+0 to R+3	**Block Y0** (Zero fill if skipped)
R+4 to R+7	**Block Y1** (Zero fill if skipped)
R+8 to R+11	**Block Y2** (Zero fill if skipped)
R+12 to R+15	**Block Y3** (Zero fill if skipped)
R+16 to R+19	**Block Cb** (Zero fill if skipped)
R+20 to R+23	**Block Cr** (Zero fill if skipped)

AVC-IT Mode indirect data residual data blocks (MbType == I_PCM)

GRF Registers	Description
R+0 to R+1	**Block Y0**
R+2 to R+3	**Block Y1**
R+4 to R+5	**Block Y2**
R+6 to R+7	**Block Y3**
R+8 to R+9	**Block Cb**
R+10 to R+11	**Block Cr**
R+12 to R+23	Undefined

1.8.1.6 AVC-MC Mode Root Thread [DevCTG, DevILK]

The root thread payload messages in AVC-MC mode are identical to that in AVC-IT mode, except the first dword. The first dword is the same as that in AVC-IT mode as shown in the table below

Dword 0 of R+0.0 of AVC-MC In-line data block

Dword	Bit	Description
R+0.0	31:28	**Reserved.** MBZ
	26:25	**MbAffFieldFlag** This field indicates that the current macroblock is a field macroblock within a MbAff frame picture. It is provided as **Flag = MbaffFrame** & **FieldMbFlag.** 00 = if (Flag == 0) 11 = if (Flag == 1) Other encodings are reserved
	24	**FieldMbPolarityFlag** This field indicates the field polarity of the current macroblock. Within a MbAff frame picture, this field may be different per macroblock and is set to 1 only for the second macroblock in a MbAff pair if FieldMbFlag is set. Otherwise, it is set to 0. Within a field picture, this field is set to 1 if the current picture is the bottom field picture. Otherwise, it is set to 0. It is a constant for the whole field picture. This field is reserved and MBZ for a progressive frame picture. 0 = Current macroblock is a field macroblock from the **top** field 1 = Current macroblock is a field macroblock from the **bottom** field *Programming Note: Here bits [26:24] (MbAffFieldFlag and FiedlMbPolarityFlag) match with bits [10:8] of the Media Block Read message descriptor, simplifying the programming for message generation, as when MbAffFieldFlag is "1", kernels need to override the original "frame" surface state set for MBAFF frame picture.*
	23	**Reserved.** MBZ
	22:17	**Reserved.** MBZ (Forced to Zero by VFE)
	16	**Reserved.** MBZ
	15	**Transform8x8Flag**
	14	**FieldMbFlag (Field Macroblock Flag)** This field specifies whether current macroblock is field macroblock. 0 = Frame macroblock. 1 = Field macroblock.
	13	**IntraMacroblock (Intra Macroblock Indicator)**
	12:8	**MbType (Macroblock Type)** This field, along with "IntraMacroblock" specifies the macroblock types.
	7:6	**WeightedBiPredFlag (Weighted Bidirectional Prediction Flag)** (from Picture State). Valid only for macroblock in inter mode. Otherwise (intra macroblock), this field is reserved.
	5	**WeightedPredFlag.** (from Picture State). Valid only for macroblock in inter mode. Otherwise (intra macroblock), this field is reserved.
	4	**Reserved.** MBZ

Dword	Bit	Description
	3:2	**ChromaFormatIdc (Chroma Format Indicator)** This field specifies the chroma format for the decoding process as defined below. This field is constant within a picture. 00 = Luma only (monochrome) 01 = YUV420 sampling 10 is reserved (for YUV422 sampling) 11 is reserved (for YUV444 sampling)
	1	**MbaffFrame.** (from Picture State).
	0	**FieldPicFlag.** (from Picture State)

.

1.8.1.7 VC1-IT Mode Root Thread [DevCTG,, DevILK]

The following table shows the root thread payload messages when VFE is in VC1-IT mode. One example GRF register location is given for the condition that: CURBE is disabled.

VC1-IT Mode root thread payload layout

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. If m is 0, this field doesn't exist.
R(m+1)	R1	**In-line Data block**.
R(m+2) – R(m+13)	R2 – R13	**Indirect Data Residual blocks**

The R0 header field is the same as in other modes.

The in-line data block field is the same as in the MEDIA_OBJECT_EX command.

1.8.1.8 Child Thread

The tread initiation for the child thread is determined by the data stored in the URB by the parent that spawns it. No hardware-defined header is generated. Software should follow the header field definition similar to that for a root thread, when the same fields are used, to be consistent and to reduce message header assemble overhead.

The Parent Thread Count field should be the Thread Count field of the parent thread itself (e.g. copying R0.6[23:0] to R0.7[23:0]). The Thread Count field should have a unique value for each child thread and the unique value should not be dependent on the execution order. This is mostly important for the cases when the child thread generation order may vary depending on the thread completion order. For example, when generating child threads for macroblock-based processing, the Thread Count field for a child thread should be deterministic for a macroblock position.
The following table shows the R0 register contents for a child thread, which is generated by its parent thread. The remaining payloads are application dependent.

DWord	Bit	Description
R0.7	31	**Reserved**
	27:24	**Reserved**
	23:0	**Reserved**
R0.6	31:24	**Reserved**
	23:0	**Reserved**
R0.5-R0.0	31:0	**Software defined**

1.8.1.9 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

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.

***[Dev*ILK*]:** 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.*

***[DevILK]:** 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).

1.8.1.10 Message Descriptor

The following table shows the lower 16 bits of the message descriptor (lower 20 bits for **[DevILK+]**) 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. **[Pre-DevILK]:** this bit is not part of the shared function specific message descriptor**.**
18:5	Reserved: MBZ **[Pre-DevILK]:** Bits 18:16 are not part of the shared function specific message descriptor.
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 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”.

Bit	Description
	0: dereference resource (also used for end of thread) 1: spawn thread

1.8.1.11 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
		Range = [1,63] for root threads

	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

DWord	Bit	Description
M0.2	31:4	**Interface Descriptor Pointer.** Specifies the 16-byte aligned pointer to *the child thread's* interface descriptor. This pointer is used by TS to fetch the interface descriptor for the child thread, and it is also passed to the child thread in its R0 header. This field is ignored if the **Opcode** is "dereference resource" or "spawn a root thread". Format = GeneralStateOffset[31:4]
	3:0	Ignored
	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	Ignored
	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
M0.1	31:0	Ignored
M0.0	31:28	Ignored
	27:24	[Ivb+]Shared Local Memory Index: Indicates the starting index for the shared local memory for the thread group. Each index points to the start of a 4k memory block, 16 possibilities cover the entire 64k shared memory per half-slice. Format = U4 [pre-Ivb] Reserved: MBZ
	23:16	[DevIL+] **Thread Dependency Identifier (TDID)**. This field is assigned by TS to be used for hardware scoreboard. Format = U8 [Pre-DevIL] Reserved : MBZ

DWord	Bit	Description
	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.

2. AVC Bit-Serial Decoder [DevCTG/DevILK]

2.1 Design Assumptions

MbaffFrameFlag can't have different values in different slices of the same picture

512 bits memory access interface, cache line size

No interruptability within a slice decoding. In addition, according to the AVC specification, there is no dependency across Slice boundary with respect to bit stream decoding. Hence, context switching mechanism is much simplified. Currently, a video stream context is consisted of mainly the starting MB number of a Slice in that stream. Context switching mechanism is needed for handling concurrent multiple stream decoding and error handling.

There is no pre-emption for the BCS pipeline (CPU is not able to interrupt the BCS-BSD pipe), hence every command buffer is required to contain all the states setup (preamble), and therefore no need to save and restore context.

If different H.264 applications are running, each will have its own command buffer (responsible for decoding at least one Slice). The driver is required to collect all the command buffers of a picture of a video sequence. Each command buffer will run to completion serially.

If an H.264 application is playbacking multiple video sequences, each sequence has its own command buffer list.

The switching across different video streams of different applications can only be done on a picture boundary.

A command buffer can contain multiple Slices to be decoded, but will not cross the picture boundary. But there is no driver interruption in between Slices. A command buffer must run to its completion. The worst case latency is one picture time, since a Slice can be as big as a picture.

2.2 Bit-Serial Decoding (BSD) Unit

A standalone H/W unit uses for performing the AVC CABAC/CAVLD decoding function. It is operating concurrently with the GEN4 GPE, with its own Command Streamer. The communications and synchronization between BSD and GEN4 GPE is primarily through memory. The Host will parse all header information from an AVC bit-stream, and only send down the raw Slice Data in memory to the BSD. The decoded data are written back to memory for the GEN4 GPE to consume.

Starting code detection is done in the Host driver, only the Slice Data layer is passed down to the HW for processing.

The BSD gets its commands, states and parameters from the CS unit. AI unit now fetches/pre-fetches the indirect slice payload through AM/CS/CI units and takes care of start/end of slice before sending the indirect data (slice payload) to AC unit. AI unit also needs to take care of emulation prevention byte for AC unit.

2.2.1.1 4x4 Spatial Luma, 16x16 Spatial Luma, 8x8 Spatial Chroma

Intra_16x16 uses one intra prediction scheme for the whole macroblock. Pixels may be filled from surrounding macroblocks at the left and the upper edge using one of four possible prediction modes. Intra prediction is also performed for the chroma planes using the same range of prediction modes. However, different modes may be selected for luma and chroma.

Intra_4x4 subdivides the macroblock into 16 subblocks and assigns one of nine prediction modes to each of these 4x4 blocks. The prediction modes offered in Intra_4x4 blocks support gradients or other smooth structures that run in one of eight distinct directions. One additional mode fills a whole subblock with a single value and is used if no other mode fits the actual pixel data inside a block. Intra chroma prediction is performed in an identical way to the one used in Intra_16x16 prediction. Thus, the chroma predictions are not split into subblocks as it is done for luma.

2.2.1.2 PCM

Transmit picture samples (Luma and Chroma) directly without any prediction, transformation and compression. It sets the upper limit (maximum number of bits) for each coded MB. The I_PCM mode is a type of "fail safe" mode that bounds the size of a macroblock in the compressed video bitstream. In I_PCM, coefficient data is replaced by actual 8-bit pixel sample values. Hence, instead of 16-bit per coefficient data, a MB in I_PCM mode will contain packed 8-bit values of Luma and Chroma, and each MB will be exactly 384 bytes (luma + chroma) in size.

According to the AVC Spec., I_PCM mode is enlisted amongst Intra-Prediction modes. And internal parameter settings (listed below) in I_PCM mode are similar to those of Intra16x16 mode.

For a macroblock coded with mb_type equal to I_PCM, the Intra macroblock prediction mode shall be inferred.

Hence, there will be no separation of DC terms from the AC terms as in inter-prediction coding

So, all the Dcblock flags should be either ignored by the H/W in the I_PCM mode (as in intra-prediction modes) or set to 0 (to indicate their absence)

For macroblocks in I_PCM mode, there is no coded_block_pattern syntax element present in the bitstream to derive the variables CodedBlockPatternLuma and CodedBlockPatternChroma.

According to Table 7-11,when in I_PCM mode, the parameters - Intra16x16PredMode, CodedBlockPatternChroma and CodeBlockPattern Luma are don’t care.

Hence, in I_PCM mode, we treat each MB as non-skipped and all 16x16 Luma coefficients and 8x8 Chroma (Cr and Cb) coefficients are present and no separation of DC terms. So, the cbp bits for all Luma and Cr/Cb 8x8 blocks should be set to indicate their presence. In monochrome mode, Cr/Cb should be set to absent. And the separated DC block flags should be 0 to indicate their absence.

When chroma_format_idc is equal to 0 (i.e., monochrome), CodedBlockPatternChroma shall be equal to 0.

DcBlockCodedYFlag [Used by VFE]	Always zero?
DcBlockCodedCbFlag [Used by VFE]	Always zero?
DcBlockCodedCrFlag [Used by VFE]	Always zero?
CbpY (Coded Block Pattern Y)	Always 0xFF?
CbpCr [Used by VFE]	Always 0xF in non-monochrome mode, 0x0 in monochrome mode?
CbpCb [Used by VFE]	Always 0xF in non-monochrome mode, 0x0 in monochrome mode?

2.3 H/W Asynchronous Reset

Global H/W Reset is signalled during system power-up to reset the entire BSD pipeline and to initialize it to a known state. It is also used in situation of errors to clear the pipeline. When the Command Stream Parser is stalled, pipeline reset implemented as CS command would not get through. Hence, the asynchronous H/W Reset implemented through MMIO register and issued by the Host S/W is the only option to get out. Host S/W WatchDog timers may be implemented to detect stalled conditions. After H/W reset, the ring buffer will be lost, and driver re-programming is required to continue. The reset immmediately takes effect regardless the current operation.

2.4 H/W Pipeline Flush

A flush is implicitly carried out at the end of decoding a Slice. An explicit flush can also be issued to the BCS at the picture boundary (at the end of processing a picture). BCS unit will then perform the flush by monitoring the "done" signals from AI, AC and AM units, and issue a subsequent "store Dword".

2.5 MMIO Interface

MMIO is involved in interrupt generation during error handling. MMIO address 4400h is defined for the BSD registers, and 4000h for the BCS MMIO. These are read and write registers.

Currently, there are two set of MMIO registers

- AVC BSD Error Register (16-bit)
- AVC BSD Error Count Register (12-bit each) – do not wrap around when maximum count has reached.

AVC BSD Error Register : MMIO Address : 4400h

The only MMIO write operation is to reset each individual bit of this register to a 0 value, after the error information has been read and/or processed. The driver may choose to read this register in between pictures and video sequence and upon video stream switching. This register is set to 0 at powerup.

16-bit Read/Write Register (32-bit aligned)

Bit	Description
15:4	Reserved. MBZ
3	BSD Premature Completion Error Status Flag When a BSD Premature Completion error has occurred and the BSDPrematureComplete Error Handling bit in the inline data of the AVC_BSD_OBJECT command is set, this error status flag is set until being cleared by a subsequent MMIO write to this register.
2	MPR Error Status Flag When a MPR error has occurred and the MPR Error Handling bit in the inline data of the AVC_BSD_OBJECT command is set, this error status flag is set until being cleared by a subsequent MMIO write to this register.
1	VLD Error Status Flag When a VLD error has occurred and the VLD Error Handling bit in the inline data of the AVC_BSD_OBJECT command is set, this error status flag is set until being cleared by a subsequent MMIO write to this register.
0	BSD Error Status Flag When a BSD error has occurred and the BSD Error Handling bit in the inline data of the AVC_BSD_OBJECT command is set, this error status flag is set until being cleared by a subsequent MMIO write to this register.

AVC BSD Error Count Register : MMIO Address : 4404h

The only MMIO write operation is to reset this register to a 0 value. The driver may choose to read this register in between pictures and video sequence and upon video stream switching. This register is set to 0 at powerup.

16-bit Read/Write Register (32-bit aligned)

Bit	Description
15:12	Reserved. MBZ
11:0	**BSD Error Count** Increment by 1 when any of the recognized errors (BSD, VLD, MPR and PrematureCompletion) has occurred. Do not wrap around when the maximum count has reached.

2.6 Programming the BSD Unit

2.6.1 Example Command Sequence for Decoding a Single Video Stream

BSD Unit uses a simple programming model that does not support pipelined state changes. All state information required to decode a Slice are always sent along (and prior to) with the compressed video data. This eliminates the need to reset the internal state registers for decoding a new Slice; they will be over-written anyway.

The basic steps in programming the BSD Unit are listed below. There is no explicit initialization step for the BSD engine, the H/W unit will be properly initialized itself at power-up and at reset. Some of the steps are optional and some are repetitive. The recommended order should be followed strictly. Some usage restrictions are highlighted for illustration purpose. For details, reader should refer to the respective chapters for these commands.

- Step 1: Issue the AVC_IND_OBJ_BASE_ADDR command for system configuration
    - 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.
    - The fields **AVC Indirect Object Base Address** and **AVC Indirect Object Access Upper Bound** are used to control indirect object load.
- Step 2: Issue AVC_BSD_State commands to program the AVC decoding pipeline states
    - These commands are mandatory, since Slice Data decoding cannot rely on states set by the previous Slice.
    - Multiple such commands in this step are allowed. When the same command is issued multiple times, the last one overwrites previous ones.
    - There are 4 state commands to be sent in a strict order, Except the IMG_STATE that should only be changed on a per picture basis, the others are re-sent for decoding a new Slice whenever there is a change :
        - AVC_BSD_IMG_STATE – sent in for every new picture (first slice of each picture)
        - AVC_BSD_QM_STATE – sent in for every Slice to be decoded
        - AVC_BSD_SLICE_STATE – sent in for every Slice to be decoded
        - AVC_BSD_BUF_BASE_STATE – sent in every Slice to be decoded
- Step 3: Issue the media/primitive AVC_BSD_OBJECT command
    - Multiple AVC_BSD_OBJECT commands can be issued to continue processing BSD media primitives, only if all the state information is unchanged.
    - An implicit pipeline flush is executed at the end of decoding a Slice. The BSD Command Stream Parser will be halted until it has received a complete signal from the BSD unit.
- Step 4: Repeat Step 2 and 3 as many times as needed for processing an entire picture
- Step 5: Send a phantom slice at the end of each picture
    - To make sure that the automatic error concealment in the H/W is triggered to the end of the picture when needed.
- Step 6 : BCS flush sent in at the end of each picture, and perform a subsequent store Dword command
- Step 7: Repeat Step 2 through 5 as many times as needed for processing an entire video sequence

2.7 AVC_BSD Commands

2.7.1 BSD_IND_OBJ_BASE_ADDR Command

The BSD_IND_OBJ_BASE_ADDR command sets the memory base address pointers for the subsequent Indirect Data Start Address specified in the AVC_BSD_OBJECT commands.   This command is shared by VC1 BSD pipe.

While the use of this base addresses is unconditional, the indirection can be effectively disabled by setting the base addresses to zero.

The Command Streamer (BCS) will perform the memory access bound check automatically using the AVC Indirect Object Access Upper Bound specification. If any access is at or beyond this bound, zero value is returned. The request to memory still being sent, but the BSD H/W will detect and perform the zeroing. If the Upper Bound is turned off, the beyond bound request will return whatever on the bus (invalid data).

Notation	Definition
PhysicalAddress[n:m]	Corresponding bits of a physical graphics memory byte address (not mapped by a GTT)
GraphicsAddress[n:m]	Corresponding bits of an absolute, virtual graphics memory byte address (mapped by a GTT)

Dword	Bit	Description
0	31:29	**Command Type =** PARALLEL_VIDEO_PIPE = 3h
	28:16	**AVC Command Opcode =** BSD_IND_OBJ_BASE_ADDR Pipeline[28:27] = BSD = 2h; Opcode[26:24] = AVC = 4h; Sub Opcode[23:16] = 4h
	15:0	**DWord Length** (Excludes DWords 0, 1) = 0001h
1	31:12	**AVC Indirect Object Base Address.** Specifies the 4K-byte aligned memory base address for indirect object load in AVC_BSD_OBJECT command. Format = GraphicsAddress[31:12]
	11:0	Reserved : MBZ
2	31:12	**AVC Indirect Object Access Upper Bound.** This field specifies the 4K-byte aligned (exclusive) maximum Graphics Memory address access by an indirect object load in a AVC_BSD_OBJECT command. Indirect data accessed at this address and beyond will appear to be 0. Setting this field to 0 will cause this range check to be ignored. If non-zero, this address must be greater than the **AVC Indirect Object Base Address**. Hardware ignores this field if indirect data is not present. Format = GraphicsAddress[31:12]
	11:0	Reserved: MBZ

2.7.2 AVC_BSD_STATE Commands

The AVC_BSD_STATE commands are used to load various state information and parameters into the BSD unit to govern its operations. They include the decoding parameters extracted from different AVC syntax layers (e.g. NAL, PPS, SPS, Slice Header, SEI, etc.) that are to be consumed within the BSD (and MPR) unit, as well as decoding information that are used to drive and to be passed along to other processing stages (IT, MC and ILDB) that follow the BSD unit. Some of these state parameters are directly set to the corresponding AVC syntax elements' values; and others (with similar name as the corresponding AVC syntax element) are derived from them as indicated. There are also additional decoding parameters that are not defined as part of the AVC specification, but are needed for proper functioning (e.g. error handling state parameters).

Each state command is of different size, and is designed to be Dword (32-bit) aligned. Some of these parameters are defined in according to the AVC specification, and some are only related to the current implementation of the BSD unit and Intel architecture.

These commands are issued prior to a set of BSD Object commands, so that all the required state variables are set appropriately prior to the actual decoding of the corresponding Slice Data portion of the bitstream. The BSD unit is a stateless machine, all state information and parameters must be set prior to the decoding. However, unless a reset is triggered, the current parameter values (programmed from a previous state command) remain until another corresponding AVC_BSD_STATE command comes in to re-program them.

There is an inherent order of setting the state variables, as dedicated by the H/W interface, and therefore the order of issuing the corresponding AVC_BSD_STATE commands. In general, we will follow the order listed below:

1. AVC_BSD_IMG_STATE (fixed size)
2. AVC_BSD_QM_STATE (variable size)
3. AVC_BSD_SLICE_STATE (variable size with fixed sized data structures)
4. AVC_BSD_BUF_BASE_STATE (fixed size)

All state information are sent to the BSD unit through AVC_BSD_STATE commands as inline data (some are fixed size, and some are variable in length but are predetermined by the driver before issuing the corresponding STATE Command).

The current active SPS can be chosen from an array of SPS with an index in the driver. Likewise, the current active PPS can be chosen from an array of PPS with an index in the driver.

2.7.2.1 AVC_BSD_IMG_STATE Command

AVC_BSD_IMG_STATE command encapsulates the decoding parameters that are read or derived from bitstream syntax elements of a NAL unit, a PPS unit, a SPS unit and a Slice Header (only includes those that are invariant within a picture). It includes all the relevant settings for the Frame Parameter and the Image Parameter Interface of the BSD Unit. These parameters are not changes from slice to slice of the same picture, but may change from picture to picture. Hence, this command is only issued at the beginning of processing a new picture and prior to the AVC_BSD_OBJECT command. It has a fixed size of 8 DWs (including the Command Opcode DW0).

AVC_BSD_IMG_STATE command must be sent in order for the H/W to sync to the beginning of a picture.

The values set for these state variables are retained internally across Slices, until they are reset by H/W Asynchronous Reset or changed by the next AVC_BSD_IMG_STATE command.

Dword	Bit	Description
0	31:29	**Command Type =** PARALLEL_VIDEO_PIPE = 3h
	28:16	**AVC Command Opcode =** AVC_BSD_IMG_STATE Pipeline[28:27] = BSD = 2h; Opcode[26:24] = AVC = 4h; Sub Opcode[23:16] = 0h
	15:0	**DWord Length** (Excludes DWords 0,1) = 0004h
1	31:16	Reserved. MBZ.
	15:0	**Frame size in MB unit, FrameSizeInMBs[15:0]** Note: bit 15 must be 0 and is ignored by the current H/W implementation, i.e. only bits[14:0] are significant. Unsigned integer value. The value for FrameSizeInMBs must match the product of FrameWidthInMBs and FrameHeightInMBs. Max. screen resolution is therefore limited to 2K x 2K in MB unit. e.g for 1920x1080, FrameSizeInMBs[15:0] = 8160 (1920/16 * 1088/16; rounded up 1080) This parameter is specified for Intel interface only.
2	31:24	Reserved. MBZ.
	23:16	**Frame height in MB unit, FrameHeightInMBs[7:0]** Unsigned integer value. It is set to the value of (FrameHeightInMBsMinus1+ 1). Since the max value for FrameHeightInMBs is 255, the max allowed value for FrameHeightInMBsMinus1 is only 254. The min value for FrameHeightInMBs is 1. Although the max. value that can be specified for FrameHeightInMBs is 255 (in the current implementation), FrameWidthInMBs * FrameHeightInMBs must not exceed the max value of FrameSizeInMBs[14:0]. e.g., for 1920x1080, FrameHeightInMBs[7:0] is equal to 68 (1080 divided by 16, and rounded up, i.e. effectively specified as 1088 instead). It is derived from FrameHeightInMbs = (2 – frame_mbs_only_flag) * PicHeightInMapUnits (firmware ???), or PicHeightInMbs = FrameHeightInMbs / (1 + field_pic_flag) internally done (???)
	15:8	Reserved. MBZ.
	7:0	**Frame width in MB unit, FrameWidthInMBs[7:0]** Unsigned integer value. It is set to the value of (FrameWidthInMBsMinus1 + 1). FrameWidthInMBsMinus1 is equal to the value of the syntax element in the current active SPS. Since the max value for FrameWidthInMBs is 255, the max value allowed for FrameWidthInMBsMinus1 is only 254. The min value for FrameWidthInMBs is 1. Although the max. value that can be specified for FrameWidthInMBs is 255 (in the current implementation), FrameWidthInMBs * FrameHeightInMBs must not exceed the max value of FrameSizeInMBs[14:0]. e.g., for 1920x1080, FrameWidthInMBs[7:0] is equal to 120 (1920 divided by 16).
3	31:29	Reserved. MBZ

Dword	Bit	Description
	28:24	**Second Chroma QP Offset, second_chroma_qp_offset[4:0]** Signed integer value. It should be in the range of -12 to +12 (according to AVC spec). It specifies the offset for determining QP Cr from QP Y. It is set to the upper 5 bits of the value of the syntax element (Chroma_qp_offset[9:0]) read from the current active PPS. Chroma_qp_offset [4:0] – chroma_qp_offset_bits (from the current active PPS) Chroma_qp_offset [9:5] – second_chroma_qp_offset_bits
	32:21	Reserved. MBZ
	20:16	**Chroma QP Offset, chroma_qp_offset[4:0]** Signed integer value. It should be in the range of -12 to +12 (according to AVC spec). It specifies the offset for determining QP Cb from QP Y. It is set to the lower 5 bits of the value of the syntax element (Chroma_qp_offset[9:0]) read from the current active PPS. Chroma_qp_offset [4:0] – chroma_qp_offset_bits (from the current active PPS) Chroma_qp_offset [9:5] – second_chroma_qp_offset_bits
	15	**Special Kernel MB Scan Order for AVC ILDB Data Writes** It controls the way of writing the generated ILDB Data package to memory. 0 – MB are arranged in Raster Scan Order 1 – MB are arranged in the Special Scan Order This parameter is specified for Intel interface only.
	14	**Special Kernel MB Scan Order for AVC-IT Command Writes** It controls the way of writing the generated AVC-IT Command package to memory. 0 – MB are arranged in Raster Scan Order 1 – MB are arranged in the Special Scan Order This parameter is specified for Intel interface only.
	13	**Special Kernel MB Scan Order for AVC-IT Data Writes** It controls the way of writing the generated AVC-IT Data package to memory. 0 – MB are arranged in Raster Scan Order 1 – MB are arranged in the Special Scan Order This parameter is specified for Intel interface only.
	12	**Monochrome Prediction Weighting Table Decoding Mode Flag, monochrome_pwt_flag** Added to cover unapproved H.264 Spec. modification. Should always be written as “1”. This parameter is specified for Intel interface only.
	11	Reserved. MBZ

Dword	Bit	Description
	10	**QM Present Flag, qm_present_flag** 0 – no Scaling Matrix is pressent nor sent to the BSD unit; use the flat4x4 (when transform_8x8_mode_flag == 0) or flat8x8 (when transform_8x8_mode_flag == 1) built-in QM matrices. That is, no QM_State command should follow, or the BSD unit should ignore it if it is issued (the BCS will continue to parse QM_State command, but the BSD unit will ignore and not to receive the incoming QM matrices data). 1 – Scaling Matrix may be present,depending on the use_default and list_present flags present in the inline data of a subsequent QM_STATE command. A QM_STATE command must follow to complete the specification of the QM matrices for the current picture decoding. It is derived from the pic_scaling_matrix_present_flag in the current active PPS and the seq_scaling_matrix_present_flag in the current active SPS. This parameter is specified for Intel interface only.
	9:8	**Image Structure, img_structure[1:0]** The current decoding picture structure can only takes on 3 possible values : 00 – Frame Picture 01 – Top Field Picture 11 – Bottom Field Picture 10 – Invalid, not allowed. img_structure[0] can be used as a flag to distinguish between frame and field structure. It must be consistent with the field_pic_flag setting in the Slice Header. This parameter is specified for Intel interface only.

Dword	Bit	Description
	7:0	**Current Decoded Image Frame Store ID, img_dec_fs_idc** Unsigned integer value. It specifies the destination (currently decoded) picture by its binding table index (Intel implementation) of the current DPB. This parameter is programmed in frame. **img_dec_fs_idc**[4:0] can only be equal to 0 to 16. Bit[7:5] must be 0. When it is in the range of 0 to 15, hardware uses the selected address from **direct_mv_rd0** to **direct_mv_rd31 (**based on **img_structure** and **img_dec_fs_idc**) as the **Direct MV Write Base Address** for the Current Picture. When it is 16, hardware uses the **direct_mv_wr0_top** or **direct_mv_wr0_bottom (**depending on **img_structure**) as the **Direct MV Write Base Address** for the Current Picture. **direct_mv_wr0_top** is selected for a frame picture or a top field picturem, and **direct_mv_wr0_bottom** for a bottom field picture. The typical usage is to program this field to 16. And specifically, in order to support 16 reference frames, this field must be programmed as 16. For picture referencing/indexing, Frame Store ID Bit[6:0] = 7F (FF ???) is used to designate a non-existing picture; otherwise, Bit[7:5] should always be 0, For img_dec_fs_idc[7:0] (current decoding picture), it should never be a non-existing picture. (It was used to write the POC of the current frame to the POC list. Since we are now writing the entire POC list at a time this is no longer used for this purpose) In the current implementation, img_dec_fs_idc[4:0] can only be equal to 0 to 31. Bit[7:5] must be 0. For picture referencing/indexing, Frame Store ID Bit[6:0] = 7F (FF ???) is used to designate a non-existing picture; otherwise, Bit[7:5] should always be 0, For img_dec_fs_idc[7:0] (current decoding picture), it should never be a non-existing picture. (It was used to write the POC of the current frame to the POC list. Since we are now writing the entire POC list at a time this is no longer used) It is used to read the POC of the current frame for temporal direct motion vector weighting calculations. We could get the current POC with POCList[RefList[0]], but instead use the existing img_dec_fs_idc to determine the POC from the POCList[img_dec_fs_idc].
4	31:24	**Residual Data Offset** This gives the offset of the residual data in dwords. It should be 8 dword aligned. This should be programmed to the same value as the **Residual Data Source Offset** field in Vfe State Ex.
	23:18	**Reserved: MBZ**
	17	**Thread Synchronization Overwrite.** This field indicates whether hardware overwrites the **Thread Synchronization** field in the MEDIA_OBJECT_EX command of each output macroblock. When this field is set, the **Thread Synchronization** field is set only for intra macroblocks (and at least one of the neighbors A, B, C, D, Left-Bottom is available). Otherwise (if a macroblock is a P, B or I_PCM), the field is not set. 0 = Not Overwrite 1 = Overwrite

Dword	Bit	Description
	16	**Thread Synchronization.** This field indicates whether the thread is a Synchronized Root Thread (SRT). When **Thread Synchronization Overwrite** is not set, this field is forwarded to the corresponding field in the MEDIA_OBJECT_EX command of each output macroblock. When **Thread Synchronization Overwrite** is set, hardware ignores this field. 0 = Not a SRT 1 = SRT
	15:13	Reserved. MBZ.
	12	**Enable16MV** This field, when set, enables the 16MV motion vector format in the AVC-IT output data buffer. If this field is set to 0, all MvSize = 16MV are mapped to 32MV. 0 – no MvSize of 16MV 1 – MvSize of 16MV enabled
	11:10	**Chroma Format IDC, ChromaFormatIdc[1:0]** It specifies the sampling of chroma component (Cb, Cr) in the current picture as follows : 00 – monochrome picture 01 – 4:2:0 picture 10 – 4:2:2 picture (not supported) 11 – 4:4:4 picture (not supported) It is set to the value of the syntax element read from the current active SPS. The corresponding Monochrome Flag (monochrome_flag) can be derived from this field as follows : 0 – Color Picture 1 – Monochrome picture (i.e. no Chroma components)
	9	Reserved. MBZ.
	8	**Enable the In-Loop Deblocking Filter information write, EnableILDBWrite** 1 – enable ILDB writing output 0 – disable the ILDB writing output This bit controls ILDB information write to memory regardless of whether In-Loop Deblocking Filter is enabled or not for a given slice. If this bit is zero, even if a slice has in-loop deblocking enabled, no ILDB data is written to memory. On the other hand, if this bit is set, and if ILDB is disabled for a slice, ILDB data (default to no filtering) are written to memory for each macroblock in the slice. This allows ILDB to be performed at picture level regardless of the mixer of different slices (some has deblocking enabled and some disabled) within the picture. Default behavior when some slices of a picture has deblocking disabled. So, this will control the filling of those slices to the final output. This parameter is specified for Intel interface only.
	7	**Entropy Coding Flag, entropy_coding_flag** 0 – CAVLD bit-serial decoding mode. 1 – CABAC bit-serial decoding mdoe. It specifies one of the two possible bit stream decoding modes in the AVC. It is set to the value of the syntax element read from the current active PPS.

Dword	Bit	Description
	6	**Current Img Disposable Flag or Non-Reference Picture Flag, ImgDisposableFlag** 0 – the current decoding picture may be used as a reference picture for others. 1 – the current decoding picture is not used as a reference picture (e.g. a B-picture cannot be a reference picture for any subsequent decoding) It is derived from ImgDisposableFlag = (nal_ref_idc == 0). nal_ref_idc is a syntax element from a NAL unit.
	5	**Constrained Intra Prediction Flag, constrained_ipred_flag** 0 – allows both intra and inter neighbouring MB to be used in the intra-prediction decoding of the current MB. 1 – allows only to use neighboring Intra MBs in the intra-prediction decoding of the current MB. If the neighbor is an inter MB, it is considered as not available. It is set to the value of the syntax element in the current active PPS.
	4	**Direct 8x8 Inference Flag, direct_8x8_infer_flag** 0 – allows subpartitioning to go below 8x8 block size (i.e. 4x4, 8x4 or 4x8) 1 – allows processing only at 8x8 block size. MB Info is stored for 8x8 block size. It is set to the value of the syntax element in the current active SPS. It specifies the derivation process for luma motion vectors in the Direct MV coding modes (B_Skip, B_Direct_16x16 and B_Direct_8x8). When frame_mbs_only_flag is equal to 0, direct_8x8_inference_flag shall be equal to 1. It must be consistent with the frame_mbs_only_flag and transform_8x8_mode_flag settings.
	3	**8x8 IDCT Transform Mode Flag, trans8x8_mode_flag** Specifies 8x8 IDCT transform may be used in this picture 0 – no 8x8 IDCT Transform, only 4x4 IDCT transform blocks are present 1 – 8x8 Transform is allowed It is set to the value of the syntax element in the current active PPS.
	2	**Frame MB only flag, frame_mbs_only_flag** 0 – not true ; effectively enables the possibility of MBAFF mode. 1 – true, only frame MBs can occur in this sequence, hence disallows the MBAFF mode and field picture. It is set to the value of the syntax element in the current active SPS.
	1	**MBAFF mode is active, mbaff_frame_flag**. This bit is valid only when the img_structure[1:0] indicates the current picture is a frame. 0 – not in MBAFF mode 1 – in MBAFF mode It is derived from MbaffFrameFlag = (mb_adaptive_frame_field_flag && ! field_pic_flag). mb_adaptive_frame_field_flag is a syntax element in the current active SPS and field_pic_flag is a syntax element in the current Slice Header. They both are present only if frame_mbs_only_flag is 0. Although mbaff_frame_flag is a Slice Header parameter, its value is expected to be the same for all the slices of a picture. It must be consistent with the mb_adaptive_frame_field_flag, the field_pic_flag and the frame_mbs_only_flag settings.

Dword	Bit	Description
	0	**Field picture flag, field_pic_flag**, specifies the current slice is a coded field or not. 0 – a slice of a coded frame 1 – a slice of a coded field It is set to the same value as the syntax element in the Slice Header. It must be consistent (??? Expand out) with the img_structure[1:0] and the frame_mbs_only_flag settings. Although field_pic_flag is a Slice Header parameter, its value is expected to be the same for all the slices of a picture.
5	31:0	**AVC-IT Command Header, AvcItCmdHeader** It allows the driver to directly specify the DW0 of the AVC-IT Command to be generated by the BSD unit that will pass down to drive the VFE unit. This parameter is specified for Intel interface only.

2.7.2.2 AVC_BSD_QM_STATE Command

This command is only issued when qm_present_flag is set in the IMG_STATE command. If qm_present_flag is not set in the IMG_State command, and AVC_BSD_QM_STATE command is issued, the BCS will still process the command, but the BSD unit will ignore the data being forwarded.

Quantization matrices are loaded only if

1) the current Slice is the first of a picture,

2) active SPS or active PPS has read in,

3) active PPS has changed between pictures

Only the latest scaling matrices read from the bitstream in processing the current active SPS and PPS are sent down to the BSD unit, as needed. Maximum there can be 8 matrices. If transform_8x8_mode_flag is set, maximum two 8x8 scaling matrices and six 4x4 scaling matrices may be present; otherwise only maximum six 4x4 scaling matrices may be present. Hence, this state descriptor has a variable length; the Pressent Flag, the List_Present flags and the Use_Default flags together can determine which and how many scaling matrices to follow. If the qm_present_flag is set in the IMG_State command, but a use_default flag of QM_State command is set, the corresponding scaling matrix is not sent. That is, all default scaling matrices are generated inside the BSD unit.

Each entry of a scaling matrix occupies 1 byte. Hence, the minimum size of the state descriptor is 1 Dword, and the maximum is 57 Dwords.

The Scaling Matrices, if present, must follow a strict ordering in the state descriptor. They are packed in the ascending order of the i index. Hence, the Dword 1 is refering to the Scaling Matrix with the lowest i. This state descriptor may be sent prior to decoding each Slice. The values set for these state variables are retained internally across Slices (QM_State is actually separated out from the IMG_State), until they are reset by H/W Asychronous Reset or changed by the next AVC_BSD_QM_STATE Command.

Dword	Bit	Description
0	31:29	**Command Type =** PARALLEL_VIDEO_PIPE = 3h
	28:16	**AVC Command Opcode =** AVC_BSD_QM_STATE Pipeline[28:27] = BSD = 2h; Opcode[26:24] = AVC = 4h; Sub Opcode[23:16] = 1h
	15:0	**DWord Length** (Excludes DWords 0,1) = 0000h to max 0038h
1	31:16	Reserved. MBZ

	15:8	Use built-in Default QM Flags for the current Slice, use_default_flags[i], i = 0 to 7
		i=0 – 4x4 Intra Y
		i=1 – 4x4 Intra Cr
		i=2 – 4x4 Intra Cb
		i=3 – 4x4 Inter Y
		i=4 – 4x4 Inter Cr
		i=5 – 4x4 Inter Cb
		i=6 – 8x8 Intra Y
		i=7 – 8x8 Inter Y
		use_default_flags[i] = 1, and if the corresponding qm_list_flags[i] and qm_present_flag are also set, then use the built-in default scaling matrix
		use_default_flags[i] = 0, indicates the built-in default QM matrix is not used, and the scaling matrix read from the bitstream is used.
		qm_present_flag (in IMG_STATE), qm_list_flags[i], use_default_flags[i] and LevelScale matrices must be set in consistent and coherent.
		This parameter is specified for Intel interface only.

	7:0	QM List Present Flags for the current Slice, qm_list_flags[i], i = 0 to 7
		i=0 – 4x4 Intra Y
		i=1 – 4x4 Intra Cb
		i=2 – 4x4 Intra Cr
		i=3 – 4x4 Inter Y
		i=4 – 4x4 Inter Cb
		i=5 – 4x4 Inter Cr
		i=6 – 8x8 Intra Y
		i=7 – 8x8 Inter Y
		qm_list_flags[i] = 1 and qm_present_flag is set, indicates either using the scaling matrix read from the bit stream, or use the built-in default matrix if the Use_Default is also set. The following LevelScale field is present if and only if the corresponding bit in this field is set.
		qm_list_flag[i] = 0, use the fallback rule specified in the AVC spec.
		qm_present_flag (in IMG_STATE), qm_list_flags[i], use_default_flags[i] and LevelScale matrices must be set in consistent and coherent.
		This parameter is specified for Intel interface only.

0 or 4 Dwords	16 bytes	Luma4x4 Intra WeightScale (i=0) Unsigned integer value, ranging from 0 to 255. An array of 4x4 scaling values (one-to-one correspondence with the coefficient position) for an Intra Luma block that is stored in raster order. It is set to the values derived from the syntax elements in the current active PPS and active SPS.

Dword	Bit	Description
0 or 4 Dwords	16 bytes	Cb4x4 Intra WeightScale (i=1) Unsigned integer value, ranging from 0 to 255. An array of 4x4 scaling values (one-to-one correspondence with the coefficient position) for an Intra Chroma Cb block that is stored in row major order (i.e. raster scan order). It is set to the values derived from the syntax elements in the current active PPS and active SPS.
0 or 4 Dwords	16 bytes	Cr4x4 Intra WeightScale (i=2) Unsigned integer value, ranging from 0 to 255. An array of 4x4 scaling values (one-to-one correspondence with the coefficient position) for an Intra Chroma Cr block that is stored in row major order (i.e. raster scan order). It is set to the values derived from the syntax elements in the current active PPS and active SPS.
0 or 4 Dwords	16 bytes	Luma4x4 Inter WeightScale (i=3) Unsigned integer value, ranging from 0 to 255. An array of 4x4 scaling values (one-to-one correspondence with the coefficient position) for an Inter Luma block that is stored in row major order (i.e. raster scan order). It is set to the values derived from the syntax elements in the current active PPS and active SPS.
0 or 4 Dwords	16 bytes	Cb4x4 Inter WeightScale (i=4) Unsigned integer value, ranging from 0 to 255. An array of 4x4 scaling values (one-to-one correspondence with the coefficient position) for an Inter Chroma Cb block that is stored in row major order (i.e. raster scan order). It is set to the values derived from the syntax elements in the current active PPS and active SPS.
0 or 4 Dwords	16 bytes	Cr4x4 Inter WeightScale (i=5) Unsigned integer value, ranging from 0 to 255. An array of 4x4 scaling values (one-to-one correspondence with the coefficient position) for an Inter Chroma Cr block that is stored in row major order (i.e. raster scan order). It is set to the values derived from the syntax elements in the current active PPS and active SPS.
0 or 16 Dwords	64 bytes	Luma8x8 Intra WeightScale (i=6) Unsigned integer value, ranging from 0 to 255. An array of 8x8 scaling values (one-to-one correspondence with the coefficient position) for an Intra Luma block that is stored in row major order (i.e. raster scan order). It is set to the values derived from the syntax elements in the current active PPS and active SPS.
0 or 16 Dwords	64 bytes	Luma8x8 Inter WeightScale (i=7) Unsigned integer value, ranging from 0 to 255. An array of 8x8 scaling values (one-to-one correspondence with the coefficient position) for an Inter Luma block that is stored in row major order (i.e. raster scan order). It is set to the values derived from the syntax elements in the current active PPS and active SPS.

2.7.2.3 AVC_BSD_SLICE_STATE Command

This state descriptor includes all the information for

1. Inter-Prediction Reference Lists L0 and L1
2. Weights and Offset for Inter-Prediction

They are all syntax elements read from or derived from the Slice Header and may, therefore, be changed on a Slice boundary.

The state descriptor of the Slice_State command is of variable size. However, all the constitute components, when present, are of fixed size structure. The Slice_State command should not be issued at all, if there is no inter-prediction decoding or none of its components are present. If the present flag is set to 0, and the corresponding data is still sent, unpredicted result. If

The state information for motion vector prediction specifies the frame indices of the reference list list_0 and list_1. It is sent to the MPR unit for determining which frames to fetch for Temporal Direct Motion Vectors. These MPR states may be changed on a Slice boundary, the entire reference lists are sent, and are packed in a consecutive order (and there is no bubble – i.e. no empty/invalid/unspecified entry in between).

The very first entry of each reference list is always started with index 0, and is named as the top of the reference list.

For the weights and offsets specification in the SLICE_STATE state descriptor, it specifies the weights and offsets for inter-prediction using the reference list list_0 and list_1. It is outputted from the BSD unit to the AVC-IT interface. These Weights and Offsets states are set to the values of syntax elements read from the Slice Header, which may be changed on a Slice boundary. They are packed in a consecutive order (and there is no bubble – i.e. no empty/invalid/unspecified entry in between) . As a result, the weight and offset state descriptor has a fixed length, if it is present.

Only when WeightedPredFlag = 1 (enable weighted prediction), will there be a L0 weight+ offset table (and there is no L1 stuff in P slice) being sent to the BSD unit through the Slice_State command. If WeightedBiPredIdc[1:0] != 1 and WeightedPredFlag = 0, no Slice_State command should be issued for this table.

Only when WeightedBiPredIdc[1:0] = 1 (explicit weighted prediction), will there be a L1 and/or a L0 weight+offset tables being sent to the BSD unit through the Slice_State command. If WeightedBiPredIdc[1:0] != 1 and WeightedPredFlag = 0, no Slice_State command should be issued for this table.

As a result, this state descriptor has a variable length, and is predetermined by the driver. The minimum size is 2 Dwords if this command is ever issued, since at minimum even there is no L1 and L0 and no weight and offset, there is still DW0 and DW1. The maximum size is 113 Dwords.

Weight-Offset programming model[Errata – Dev CTG] Driver needs to look ahead all slices for the frame and check if any weight value is 128. This can only happen when luma_weight_l0/l1/chroma_weight_l0/l1 flags are 0. If it finds a 128 then it needs to remap this value to a non used value within a 16 bit twos compliment range and then give the remapped number to the kernel for identifying this case.

Dword	Bit	Description
0	31:29	**Command Type =** PARALLEL_VIDEO_PIPE = 3h
	28:16	**AVC Command Opcode =** AVC_BSD_SLICE_STATE Pipeline[28:27] = BSD = 2h; Opcode[26:24] = AVC = 4h; Sub Opcode[23:16] = 2h
	15:0	**DWord Length** (Excludes DWords 0,1) = 0000h (min) to 00D0h (max)
1	31:4	Reserved. MBZ.
	3	Weight+Offset L1Table Present Flag If Weight+OffsetL1 Table Present Flag = 1, indicates the full 32-entry L1 Table for Y, Cb and Cr is followed; otherwise it is not present at all (no further inline data). It must be set in consistent with the WeightedPredFlag and WeightedBiPredIdc in the Img_State command. This parameter is specified for Intel interface only.
	2	Weight+OffsetL0 Table Present Flag If Weight+OffsetL0 Table Present Flag = 1, indicates the full 32-entry L0 Table for Y, Cb and Cr is followed; otherwise it is not present at all (no further inline data). It must be set in consistent with the WeightedPredFlag and WeightedBiPredIdc in the Img_State command. This parameter is specified for Intel interface only.
	1	RefPicListL1 Present Flag If RefPicListL1 Present Flag = 1, indicates the full 32-entry L1 list is followed; otherwise it is not present at all (no further inline data). It must be set in consistent with the Num_Ref_Idx_L1 derived from the syntax element in the Slice Header, and if it is a B Slice. This parameter is specified for Intel interface only.
	0	RefPicListL0 Present Flag If RefPicListL0 Present Flag = 1, indicates the full 32-entry L0 list is followed; otherwise, it is not present at all (no further inline data). It must be set in consistent with the Num_Ref_Idx_l0 derived from the syntax element in the Slice Header, and if it is a P or B Slice. This parameter is specified for Intel interface only.

Dword	Bit	Description
2-9 or none	31:0 each	Reference List0 Write This field is present only if the RefPicListL0 Present Flag = 1; otherwise, it should not be present. If present, it always specifies 32 reference pictures in list0, regardless they are “existing picture” or not. If a picture is non-existing, the corresponding entry should be set to all ones. Each list entry is 1 byte. A 32-bit DW can hold 4 list entries in the following format 31:24 entry X+3 (e.g. list0_3) 23:16 entry X+2 (e.g. list0_2) 15:8 entry X+1 (e.g. list0_1) 7:0 entry X (e.g. list0_0) X is replaced by the paddr[2:0] * 4 ; paddr[5:0] with 0x30 and 0x37 The byte definition for a reference picture : Bit 7 : Non-Existing – indicates that frame store index that should have been at this entry did not exist and was replaced by an index for error concealment Bit 6 : Long term bit – set this reference picture to be used as long term reference Bit 5 : Field picture flag – indicates frame/field Bit 4:0 : Frame store index or Frame Store ID (Bit 4:1 is used to form the binding table index in Intel implementation) This is the final Reference List L0 after any reordering specified in the Slice Header as well as modified by the driver, and its indices values are all translated to the Intel specification. If the reference picture is a frame (Bit5 = 1), frame store ID is always an even number. The frame store ID = 0 and 1 are assigned to the current decoding picture; hence reference picture ID will start from 2 onward. This list is used in MV information output of the BSD unit (but non-existing picture are mapped to 00 instead to smilifiy the kernel). DMV also read and write Mvlist0 using this frame store ID.

Dword	Bit	Description
10-17 or none	31:0 each	Reference List1 Write This field is present only if the RefPicListL1 Present Flag = 1; otherwise, it should not be present. If present, it always specifies 32 reference pictures in list1, regardless they are existing or not. If a picture is non-existing, the corresponding entry should be set to all ones. Each list entry is 1 byte. A 32-bit DW can hold 4 list entries in the following format 31:24 entry X+3 (e.g. list1_7) 23:16 entry X+2 (e.g. list1_6) 15:8 entry X+1 (e.g. list1_5) 7:0 entry X (e.g. list1_4) X is replaced by the paddr[2:0] * 4 ; paddr[5:0] with 0x38 and 0x3F The byte definition for a reference picture: Bit 7 : Non-Existing – indicates that frame store index that should have been at this entry did not exist and was replaced by an index to for error concealment Bit 6 : Long term bit – set this reference picture to be used as long term reference Bit 5 : Field picture flag – indicates frame/field Bit 4:0 : Frame store index (or also known as binding table index in Intel implementation) This is the final Reference List L1 after any reordering specified in the Slice Header as well as modified by the driver, and its indices values are all translated to the Intel specification. This list is used in MV information output of the BSD unit. DMV also read and write Mvlist1 using this frame store ID.

Dword	Bit	Description
18-65 or none	31:0 each	WeightOffset[L0=0][i=0 to 31][Y/Cb/Cr][weight/offset] WeightOffset[0][i=0][Y=0][Offset=0] WeightOffset[0][i=0][Y=0][Weight=1] WeightOffset[0][i=0][Cb=1][Offset=0] WeightOffset[0][i=0][Cb=1][Weight=1] WeightOffset[0][i=0][Cr=2][Offset=0] WeightOffset[0][i=0][Cr=2][Weight=1] : WeightOffset[0][i=31][Y=0][Offset=0] WeightOffset[0][i=31][Y=0][Weight=1] WeightOffset[0][i=31][Cb=1][Offset=0] WeightOffset[0][i=31][Cb=1][Weight=1] WeightOffset[0][i=31][Cr=2][Offset=0] WeightOffset[0][i=31][Cr=2][Weight=1] Signed integer values, ranging from -128 to 127. This field is present only if the Weight+OffsetL0 Table Present Flag= 1; otherwise, it should not be present. If present, the full table is always specified. Any reference list L0[i] that does not exist, the corresponding weight and offset are set to 0. Weight and Offset are 1 byte each. Weight is the high byte and Offset is the lower byte. Hence, we are packing 2 sets of weight and offset into 1 DW. Bits[31:16] will store the set of weight and offset will higher values of i or Y/Cr/Cb (represented as 0/1/2 in the indexing). WeightOffset[L0=0][i=0 to 31][Y=0] (i.e. luma_weight_l0[i]) are specified for the weighting and offset factors applied to the luma prediction value for list 0 prediction using RefPicList0[i] (one-to-one correspondence in i). When luma_weight_l0_flag (Slice Header syntax element) is equal to 1, the value of luma_weight_l0[i] shall be in the range of –128 to 127. When luma_weight_l0_flag is equal to 0, luma_weight_l0[i] shall be inferred to be equal to $2^{luma\_log2\_weight\_denom}$ for RefPicList0[i]. luma_log2_weight_denom is a Slice Header syntax element. WeightOffset[L0=0][i=0 to 31][Cb=1] (i.e. chromaCb_weight_l0[i]) are specified for the weighting and offset factors applied to the chroma Cb prediction values for list 0 prediction using RefPicList0[i] (one-to-one correspondence in i). When chroma_weight_l0_flag (Slice Header syntax element) is equal to 1, the value of chromaCb_weight_l0[i] shall be in the range of –128 to 127. When chroma_weight_l0_flag is equal to 0, chromaCb_weight_l0[i] shall be inferred to be equal to $2^{chroma\_log2\_weight\_denom}$ for RefPicList0[i]. chroma_log2_weight_denom is a Slice Header syntax element. WeightOffset[L0=0][i=0 to 31][Cr=2] (i.e. chromaCr_weight_l0[i]) are specified for the weighting and offset factors applied to the chroma Cr prediction values for list 0 prediction using RefPicList0[i] (one-to-one correspondence in i). When chroma_weight_l0_flag (Slice Header syntax element) is equal to 1, the value of chromaCr_weight_l0[i] shall be in the range of –128 to 127. When chroma_weight_l0_flag is equal to 0, chromaCr_weight_l0[i] shall be inferred to be equal to $2^{chroma\_log2\_weight\_denom}$ for RefPicList0[i]. The table is set to the values derived from the syntax elements read in the Slice Header.

Dword	Bit	Description

66-113
or none
	31:0
	each
		WeightOffset[L1=1][32][Y/Cb/Cr][weight/offset]
		WeightOffset[1][0][Y=0][Offset=0]
		WeightOffset[1][0][Y=0][Weight=1]
		WeightOffset[1][0][Cb=1][Offset=0]
		WeightOffset[1][0][Cb=1][Weight=1]
		WeightOffset[1][0][Cr=2][Offset=0]
		WeightOffset[1][0][Cr=2][Weight=1]
		:
		WeightOffset[1][31][Y=0][Offset=0]
		WeightOffset[1][31][Y=0][Weight=1]
		WeightOffset[1][31][Cb=1][Offset=0]
		WeightOffset[1][31][Cb=1][Weight=1]
		WeightOffset[1][31][Cr=2][Offset=0]
		WeightOffset[1][31][Cr=2][Weight=1]
		Signed integer values, ranging from -128 to 127.
		This field is present only if the Weight+OffsetL1 Table Present Flag= 1; otherwise, it should not be present. If present, the full table is always specified. Any reference list L1[i] that does not exist, the corresponding weight and offset are set to 0.
		Weight and Offset are 1 byte each. Weight is the high byte and Offset is the lower byte. Hence, we are packing 2 sets of weight and offset into 1 DW. Bits[31:16] will store the set of weight and offset will higher values of i or Y/Cr/Cb (represented as 0/1/2 in the indexing).
		
		WeightOffset[L1=1][i=0 to 31][Y=0] (i.e. luma_weight_l1[i]) are specified for the weighting and offset factors applied to the luma prediction value for list 1 prediction using RefPicList1[i] (one-to-one correspondence in i). When luma_weight_l1_flag (Slice Header syntax element) is equal to 1, the value of luma_weight_l1[i] shall be in the range of –128 to 127. When luma_weight_l1_flag is equal to 0, luma_weight_l1[i] shall be inferred to be equal to $2^{luma\_log2\_weight\_denom}$ for RefPicList1[i]. luma_log2_weight_denom is a Slice Header syntax element.
		WeightOffset[L1=0][i=0 to 31][Cb=1] (i.e. chromaCb_weight_l1[i]) are specified for the weighting and offset factors applied to the chroma Cb prediction values for list 1 prediction using RefPicList1[i] (one-to-one correspondence in i). When chroma_weight_l1_flag (Slice Header syntax element) is equal to 1, the value of chromaCb_weight_l1[i] shall be in the range of –128 to 127. When chroma_weight_l1_flag is equal to 0, chromaCb_weight_l1[i] shall be inferred to be equal to $2^{chroma\_log2\_weight\_denom}$ for RefPicList1[i]. chroma_log2_weight_denom is a Slice Header syntax element.
		WeightOffset[L1=0][i=0 to 31][Cr=2] (i.e. chromaCr_weight_l1[i]) are specified for the weighting and offset factors applied to the chroma Cr prediction values for list 1 prediction using RefPicList1[i] (one-to-one correspondence in i). When chroma_weight_l1_flag (Slice Header syntax element) is equal to 1, the value of chromaCr_weight_l1[i] shall be in the range of –128 to 127. When chroma_weight_l1_flag is equal to 0, chromaCr_weight_l1[i] shall be inferred to be equal to $2^{chroma\_log2\_weight\_denom}$ for RefPicList1[i].
		The table is set to the values derived from the syntax elements read in the Slice Header.

2.7.2.4 AVC_BSD_BUF_BASE_STATE Command

This command contains the base addresses for memory buffers allocated for BSD operations. It does not include the byte stream read address, which is to be specified in the AVC_BSD_OBJECT Command.

There are six base addresses being defined:

1. BSD Row Store memory base address for R/W,
2. MPR Row Store memory base address for R/W,
3. MB Info memory base address for Write-only,
4. Decoded Uncompressed Coefficients, IPCM Coefficients or MV array memory base address for Write-only,
5. Direct MV memory base address for Write-only, and
6. 31 Direct MV memory base addresses for Read-only.
    a. Not including the current frame, which is represented by the Write Base address
    b. Although the number of Direct MV base addresses should be the same as the number of reference frames in the reference list (specified in the Slice_STATE), since it is possible to have the same frame presents in both list0 and list1 and there is no logics to cross checking, it is decided to send the complete addressess list always.

The state descriptor provides the state and information required to decode a Slice, and their values may subject to change on a Slice boundary. It should be sent prior to decoding each Slice. The values set for these state variables are retained internally across Slices, until they are reset by H/W Asychronous Reset or changed by the next AVC_BSD_BUF_BASE_STATE Command.

When switching video stream, these base address must be reprogrammed.

Xxx

Dword	Bit	Description
0	31:29	**Command Type =** PARALLEL_VIDEO_PIPE = 3h
	28:16	**AVC Command Opcode =** AVC_BSD_BUF_BASE_STATE Pipeline[28:27] = BSD = 2h; Opcode[26:24] = AVC = 4h; Sub Opcode[23:16] = 3h
	15:0	**DWord Length (Excludes DWords 0,1) = 0048h**
1	31:6	**BSD Row Store Base Address** This field provides the base address of the scratch buffer used by BSD unit to store BSD information for the previous row to assist BSD decoding of the macroblocks in the current row. The BSD Row Store buffer must be 64-byte cacheline aligned. 1.5 cacheline (CL) per MB when in MBAFF mode (row of MB pair); 0.75 CL per MB for non-MBAFF. So, to support 120 MBs per row (1920 screen resolution), 1.5 * 120 * 64 bytes = 11,520 bytes are required. Half cacheline alignment should be followed. Hardware uses the horizontal address of each macroblock to address the BSD Row Store.
	5:0	Reserved : MBZ

Dword	Bit	Description
2	31:6	**MPR Row Store Base Address** This field provides the base address of the scratch buffer used by BSD unit to store MPR information for the previous row to assist MPR decoding of the macroblocks in the current row. 1 cacheline (CL) per MB when in MBAFF mode (row of MB pair); 0.5 CL per MB for non-MBAFF, So, ot support 120 MBs per row (1920 screen resolution), 1 * 120 * 64 bytes = 7,680 bytes are required. Half cacheline alignment should be followed. Hardware uses the horizontal address of each macroblock to address the MPR Row Store.
	5:0	Reserved : MBZ
3	31:6	**AVC-IT Command MB Info Write Base Address (defined for streamout mode)** This field provides the base address of the AVC-IT command output buffer that contains the MB Info needed for inverse transform and motion compensation. This buffer must be 64-byte cacheline aligned. Hardware uses the MB number within the picture to address this buffer.. [Also provide the data structure for each macroblock. This is important as the interface is shared between this pipe and the 3DPIPE.] [Addressing in raster order and also the wave-front order. Providing a reference to the description of these two orders. Also discuss how to handle MBAFF.] 1 cacheline (CL) for each MB, 8160 MBs (1920x1088 screen resolution) per frame, hence 1*8160*64 bytes per frame = 522,240 bytes are required. Cacheline alignment should be followed.
	5:0	Reserved : MBZ
4	31:12	**AVC-IT Data Write Base Address** This is the 4KB aligned portion of the base address for the AVC-IT data write buffer. Together with the **AVC-IT Data Write Offset**, it forms the base pointer where the AVC-IT data (including for example motion vectors, weight-offsets, non-zero residual data), where data for the first macroblock will be written to. This field affect the final memory address computed for the AVC-IT Data output. However, it doesn't affect the **Indirect Data Start Address** computed by BSD hardware that is in the AVC-IT Command output (DW3 of the output AVC-IT Command). Each macroblock within the AVC-IT data buffer has a fixed size even though only valid data are written to memory. The remaining bytes for each macroblock are undefined. Assuming 32 cachelines are needed per macroblock, an HD buffer (with a resolution of 1920x1088) is 16,711,680 bytes. Format = GraphicsAddress[31:12] *Programming Note: As the AVC-IT data buffer will be used by GPE as the indirect object buffer for MEDIA_OBJECT_EX command, this field should be programmed the same as the **Indirect Object Base Address** field of the STATE_BASE_ADDRESS command.*
	11:6	**AVC-IT Data Write Offset** This offset is relative to the **AVC-IT Data Write Buffer Base Address**. This field, together with the AVC-IT Data Write Base Address, forms the 64-byte cacheline aligned address of the AVC-IT data write buffer. This field is used to compute the **Indirect Data Start Address** in the AVC-IT Command output (DW3 of the output AVC-IT Command).

Dword	Bit	Description
	5:0	Reserved : MBZ
5	31:6	**ILDB Data Write Base Address** This field provides the base address of the ILDB output data buffer. This buffer must be 64-byte cacheline aligned. 2 cachelines (CL) for each MB, 8160 MBs (1920x1088 screen resolution) per frame, hence 2*8160*64 bytes per frame = 1,044,480 bytes are required. Real data only in 1 CL, extra CL added due to driver/kernel requirement for data expansion. Cacheline alignment should be followed.
	5:0	Reserved : MBZ
6	31:6	**Direct MV Read Base Address for Reference Frame 0, 64-bytes cache-line aligned – direct_mv_rd0[31:6]** This is a ROW Store private buffer space. The read buffer size is 557,056 bytes for 1 frame (the selected colPic). It is scalable with frame height, but does not scale with frame width as H/W uses a fixed with of 128 MBs for row store buffers which is the smallest power of 2 value larger than 120 – 1920x1088 screen resolution. This field may also be used as the **Direct MV Write Base Address** for the Current Picture if **img_dec_fs_idc** is set to 0 to 15. See **img_dec_fs_idc** for more details.
	5:0	Reserved : MBZ
	5:0	Reserved : MBZ
7-37	31:6	**Direct MV Read Base Address for Reference Frame 1 to Reference Frame 31** Definition follows that of direct_mv_rd0[31:6]
	5:0	Reserved : MBZ
38	31:6	**Direct MV Write Base Address for the Current Picture (default to be Reference Frame 0 TOP field), 64-bytes cache-line aligned – direct_mv_wr0_top[31:6]** This is a ROW Store private buffer space The write buffer size is 557,056 bytes for 1 frame (the selected colPic). It is scalable with frame height, but does not scale with frame width as H/W uses a fixed with of 128 MBs for row store buffers which is the smallest power of 2 value larger than 120 – 1920x1088 screen resolution. There is only one write buffer for capturing the DMVs for the current picture. This field is used if **img_dec_fs_idc** is 16 and the current picture is a frame picture or a top field picture.
	5:0	Reserved : MBZ

Dword	Bit	Description
39	31:6	**Direct MV Write Base Address for the Current Picture (default to be Reference Frame 0 BOTTOM field), 64-bytes cache-line aligned – direct_mv_wr0_bottom[31:6]** This is a ROW Store private buffer space The write buffer size is 557,056 bytes for 1 frame (the selected colPic). It is scalable with frame height, but does not scale with frame width as H/W uses a fixed with of 128 MBs for row store buffers which is the smallest power of 2 value larger than 120 – 1920x1088 screen resolution. There is only one write buffer for capturing the DMVs for the current picture. This field is used if **img_dec_fs_idc** is 16 and the current picture is a bottom field picture.
	5:0	Reserved : MBZ
40-73	31:0	**POC List, POCList[0…33][31:0]** Each POC value is a signed 32-bit number. One-to-one correspondence with the 34 Direct MV Read/Write Address for Reference Frames. There are 34 POC entries in the list, indexed by the frame_store_ID. POCList[0] is the CurrPic POC. If it is in a field mode, POCList[0] is the CurrFOC for the top field and the POCLIst[1] is the CurrFOC for the bottom field. For frame mode, every other entry is skipped. Frame_Store_ID[5:1]+T/B bit[0] is used to index into the POCList.

2.7.3 AVC_BSD_OBJECT Command

The AVC_BSD_OBJECT command is the only primitive command for the AVC_BSD Unit. The same command is used for both CABAC and CAVLD modes. The Slice Data portion of the bitstream is loaded as indirect data object.

Before issuing an AVC_BSD_OBJECT command, all BSD states need to be valid. Therefore the commands used to set these states need to have been issued at some point prior to the issue of an AVC_BSD_OBJECT command.

[DevILK] supports transport packet based encryption. Bit 30 of DW1 indicates such packet based encryption mode. Bit 29 of DW 1 specifies the packet size. For packet-based encrypted bitstream, this command has valid information in three additional dwords (DW 9 through 11).

To handle encrypted bitstream decoding, host software is required to align the 16byte chunk containing this first byte of actual bit-stream slice, to a naturally aligned 16byte boundary, i.e. that 16byte chunk should start on an aligned 16byte address boundary. Host software should ensure that after the last valid byte of bitstream data and before encryption, enough padding is added up to the end of a 16byte chunk. In addition, for a protected context, all kernel source and destination data surface addresses must be in PCM space.

Dword	Bits	Description
0	31:29	**Command Type =** PARALLEL_VIDEO_PIPE = 3h
	28:16	**AVC Command Opcode =** AVC_BSD_OBJECT Pipeline[28:27] = BSD = 2h; Opcode[26:24] = AVC = 4h; Sub Opcode[23:16] = 8h
	15:0	**DWord Length** (Excludes DWords 0,1) [**DevCTG**] = 0006h without encryption (bit 31 of DW1 is set to 0), where inline data are in dwords 3-7 = 0007h where inline data are in dwords 3-8 [**DevILK**] = 000Ah for all cases where inline data are in dwords 3-15
1	31	**Reserved**
	30	**[DevCTG] Reserved – MBZ** **[DevILK] Packet-Based Bit Stream.** The input bitstream data is in transport packet format with potential gaps within each 192-Byte packet. ‘0’ – Not packet-based (bitstream data is a contiguous sequence of bits) ‘1’ - Packet-based This bit is a don’t care if the bit stream itself is not encrypted, i.e. bit_31 = ‘0’.
	29	**[DevCTG, Reserved – MBZ** **[DevILK] Packet Format.** Valid information, only for a packet-based bitstream (bit[30]) is set to 1. Else, this bit is a don’t care. ‘0’ – Reserved (was 2KB packet format). ‘1’ – 192B packet format. This information is used to determine the **packet size** of the bitstream generated. Header information is at the beginning of each packet.
	28:22	**Reserved – MBZ**

Dword	Bits	Description
	21:0	**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 is the length in bytes of the bitstream data for the current slice. It includes the first byte of the first macroblock and the last byte of the last macroblock in the slice. Specifically, the zero-padding bytes (if present) and the next start-code are excluded. Hardware ignores the contents after the last non-zero byte. This field is sized to support AVC High Profile 4.1 Level bitstream. Interpreted from the AVC Spec, the maximum number of bits per macroblock for 4:2:0 is 3072 (bounded by IPCM (16x16+64x2)x8). So the maximum slice size (e.g. for 1080i) is 3072 x120 x 68 / 8 = 3133440 bytes, which requires 22 bits. It includes the byte that contains the First MB Bit Offset In implementing a phantom slice at the end of a picture for automatic error concealment, this field should set to 0. For packet-based decryption, this byte length does not include the length for header bytes, but only the byte length for actually valid bitstream slice information. Format = U22 in bytes
2	31:29	Reserved : MBZ
	28:0	**Indirect Data Start Address.** This field specifies the Graphics Memory starting address of the data to be fetched into BSD Unit for processing. This pointer is relative to the **AVC Indirect Object Base Address**. Hardware ignores this field if indirect data is not present. It is a byte-aligned address for the AVC bitstream data in both CABAC/CAVLD Modes. Note however for encrypted bit stream, the 16byte chunk containing this first byte of actual bitstream slice, should be naturally aligned, i.e. that 16byte chunk should start on an aligned 16byte address boundary. In other words the first byte of that 16byte chunk should be aligned to a 16byte boundary. In implementing a phantom slice at the end of a picture for automatic error concealment, this field should set to 0.Range = [0 - 512MB]
3-variable	31:0 Each	**Inline Data** All the required Slice Header parameters and error handling settings are captured as inline data of the AVC_BSD_OBJECT command. The length is device dependent. Its definition is described in the next section.

Note that scattered raw slice data format is not supported by hardware. The size of a compression buffer for a slice needs to be allocated, which can be big, according to the minimal compression ratio required by a given profile/level. Hardware doesn't support the option for the driver to allocate 'nominally sized raw slice buffers' and use multiple such buffers for the worst case slice data.

2.7.3.1 Inline Data Description

The Inline Data includes all the required slice decoding states, extracted primarily from the Slice Header and its derivatives.   It provides information for the following operations:

1. CABAC/CAVLD decoding
2. Internal error handling at the Slice boundary
    a. H/W Automatic Error Concealment

3. Motion vector prediction decoding (MPR)

4. BSD output compositing (feeding the subsequent IT/MC/ILDB operations)

The only H/W error detection and concealment mechanism is comparing the Slice_Start_MB_Num of decoding the new current slice with the Current_MB_Num resulted from decoding the previous slice. If they do not match, an error is assumed. There are two possible cases, either “greater than” or “less than”. If the Slice_Start_MB_Num is less than the Current_MB_Num, the Current_MB_Num is adjusted internally to be the same as the Slice_Start_MB_Num. If the Slice_Start_MB_Num is greater than, there is a gap between the Current_MB_Num and the Slice_Start_MB_Num that needs to be filled. The filling is started from [Current_MB_Num – Rewind_Num], and the method of filling can be either intra or inter as specified in the state descriptor.

These state/parameter values may subject to change on a Slice boundary, and must be provided in each AVC_BSD_OBJECT command. The values set for these variables are retained internally, until they are reset by H/W Asynchronous Reset or changed by the next AVC_BSD_OBJECT command.

Dwords 3-7 are the common in-line data for **[DevCTG and DevILK]**

Dword	Bit	Description
3	31	**Concealment Method** This field specifies the method used for concealment when error is detected. If set, a copy from collocated macroblock location is performed from the concealment reference indicated by the **ConCeal_Pic_Id** field. If it is not set, a copy from the current picture is performed using Intra 16x16 Prediction method. 0 – Intra 16x16 Prediction 1 – Inter P Copy
	30	**Init Current_MB_Number** When set, the current Slice_Start_MB_Num, Slice_MB_Start_Hor_Pos and Slice_MB_Start_Vert_Pos fields will be used to initialize the Current_MB_Number register. This effectively disables the concealment capability.
	29:28	Reserved : MBZ
	27:24	**Rewind_Num** This field provides the number of MBs or MBAFF pairs to rewind when performing concealment. This is ignored if Slice_Start_MB_Num is smaller than or equal to the Current_MB_Num. Error Concealment MB start position = Current_MB_NUMBER – MIN { Decoded_MB_NUMBER, (MBAFF ? 2 : 1)*Force_Skip_Rewind } Decoded_MB_NUMBER = Number of MBs decoded in the previous slice.
	23:22	Reserved : MBZ
	21:16	**Conceal_Pic_Id (Concealment Picture ID)** This field identifies the picture in the reference list to be used for concealment. This field is only valid if **Concealment Method** is Inter P Copy. Bit 21 = 0 – Frame Picture = 1 – Field Picture Bit 20:16 – Frame Store Index[4:0]
	15	Reserved : MBZ

Dword	Bit	Description
	14	**BSDPrematureComplete Error Handling** 1 – Set the interrupt to the driver (provide MMIO registers for MB address R/W) 0 – Ignore the error and continue (masked the interrupt), assume the H/W automatically perform the error handling It occurs in situation where the Slice decode is completed but there are still data in the bitstream.
	13	Reserved : MBZ
	12	MPR Error (MV out of range) Handling– what to do when the specific error has occurred 1 – Set the interrupt to the driver (provide MMIO registers for MB address R/W) 0 – Ignore the error and continue (masked the interrupt), assume the H/W automatically perform the error handling
	11	Reserved : MBZ
	10	Entropy Error Handling – what to do when the specific error has occurred 1 – Set the interrupt to the driver (provide MMIO registers for MB address R/W) 0 – Ignore the error and continue (masked the interrupt), assume the H/W automatically perform the error handling
	9	Reserved : MBZ
	8	MB Header Error Handling – what to do when the specific error has occurred 1 – Set the interrupt to the driver (provide MMIO registers for MB address R/W) 0 – Ignore the error and continue (masked the interrupt), assume the H/W automatically perform the error concealment.
	7:4	Reserved : MBZ
	3:0	**Slice_Type (Slice Type)** 0000 – P Slice 0001 – B Slice 0010 – I Slice Other encodings are reserved (note that bits[3:2] must be 0) It is set to the value of the syntax element read from the Slice Header.
4	31:30	Reserved : MBZ
	29:24	**Num_Ref_Idx_L1 (Number of Reference Pictures in Inter-prediction List 1)** This field is valid only for decoding a B Slice, for which it is expected to have at least one entry in the reference list L1; otherwise (if Slice Type is not a B Slice), this field must be set to 0. This field can be derived for a B Slice from the Slice Header syntax element NumRefIdxActiveMinus1 as, Num_Ref_Idx_L1 = NumRefIdxActiveMinus1[1] + 1. Format U6 with valid range of [0, 32].
	23:22	Reserved : MBZ

Dword	Bit	Description
	21:16	**Num_Ref_Idx_L0 (Number of Reference Pictures in Inter-prediction List 0)** This field is valid for decoding a P or B Slice, for which it is expected to have at least one entry in the reference list L0; otherwise (if Slice Type is not a P or B Slice), this field must be set to 0. This field can be derived for a P or B Slice from the Slice Header syntax element NumRefIdxActiveMinus1 as, Num_Ref_Idx_L0 = NumRefIdxActiveMinus1[0] + 1. Format: U6 with valid range of [0, 32]
	15:11	Reserved : MBZ
	10:8	**Log2WeightDenomChroma** It is the base 2 logarithm of the denominator for all Chroma (Cb and Cr) weighting factors. The value of chroma_log2_weight_denom should be in the range of 0 to 7. It is set to the value of the syntax element read from the Slice Header Pred_Weight_Table(). Format: U3 with valid range of [0, 7].
	7:3	Reserved : MBZ
	2:0	**Log2WeightDenomLuma** It is the base 2 logarithm of the denominator for all Luma weighting factors. It is set to the value of the syntax element read from the Slice Header Pred_Weight_Table(). Format: U3 with valid range of [0, 7].
5	31:30	**Weighted_Pred_Idc (Weighted Prediction Indicator)** This field indicates the Weighted Prediction mode for a P or B Slice. It is a combined field corresponding to the syntax element WeightedBiPredIdc or WeightedPredFlag read from the current active PPS. If it is a B-Slice, these 2 bits is interpreted as : 00 – specifies the default weighted inter-prediction to be applied 01 – specifies the explicit weighted inter-prediction to be applied 10 – specifies the implicit weighted inter-prediction to be applied 11 – Reserved (not allowed) If it is a P Slice, Weighted_Pred_Idc is interpreted as : 00 – disables weighted inter-prediction (default weighted) 01 – enables weighted inter-prediction (explicit weighted) 10-11 – Reserved Only when in B Slice with Weighted_Pred_Idc = 1 (explicit weighted prediction), will there be a L1 and/or a L0 weight+offset tables being sent to the BSD unit through the Slice_State command. Only when in P Slice with Weighted_Pred_Idc = 1, will there be a L0 weight+offset table being sent to the BSD. If Weighted_Pred_Idc != 1 for B Slice or Weighted_Pred_Idc =0 for P Slice, no Slice_State command should be issued to send these tables. If still being issued, the data is read but ignored.

Dword	Bit	Description
	29	**Direct_Pred_Type (Direct Prediction Type)** Type of direct prediction used for B Slices. This field is valid only for Slice_Type = B Slice; otherwise, it must be set to 0. 0 – Temporal 1 – Spatial
	28:27	**Disable_Deblocking_Filter_Idc (Disable Deblocking Filter Indicator)** 00 - filterInternalEdgesFlag is set equal to 1 01 – disable all deblocking operation, no deblocking parameter syntax element is read; filterInternalEdgesFlag is set equal to 0 10 - macroblocks in different slices are considered not available; filterInternalEdgesFlag is set equal to 1 11 – Reserved (not defined in AVC)
	26	Reserved : MBZ
	25:24	**Cabac_Init_Idc** Specifies the index for determining the initialization table used in the context variable initialization process.
	23:22	Reserved : MBZ
	21:16	**Slice_Qp (Slice Quantization Parameter)** Quantization Parameter for current slice. Derived from PPS and slice_delta_qp syntax element in Slice Header.
	15:12	Reserved : MBZ
	11:8	**Slice_Beta_Offset_Div2** Specifies the offset used in accessing the deblocking filter strength tables. It is a sign 2's complement number, ranging from -6 to +6 inclusive. Format: S4 with valid range [-6, 6]
	7:4	Reserved. MBZ.
	3:0	**Slice_Alpha_C0_Offset_Div2** Specifies the offset used in accessing the deblocking filter strength tables. It is a sign 2's complement number, range from -6 to +6 inclusive. Format: S4 with valid range [-6, 6]
6	31:24	**Slice_MB_Start_Vert_Pos (Slice Vertical Position)** This field specifies the position in y-direction of the first macroblock in the Slice in unit of macroblocks.
	23:16	**Slice_MB_Start_Hor_Pos (Slice Horizontal Position)** This field specifies the position in x-direction of the first macroblock in the Slice in unit of macroblocks.
	15	Reserved : MBZ
	14:0	**Slice_Start_Mb_Num** The MB number (linear MB address in a picture) at the start of a Slice, it must match with the Slice Horizontal Position (Slice_MB_Start_Hor_Pos) and Vertical Position (Slice_MB_Start_Vert_Pos) in the picture. In creating the Phantom Slice for error concealment, this field should set to the total number of MB in the current picture + 1.

Dword	Bit	Description
7	31:8	Reserved : MBZ
	7	**Fix_Prev_Mb_Skipped** Enables a method for decoding mb_skipped.
	6:3	Reserved : MBZ
	2:0	**First_MB_Bit_Offset (First Macroblock Bit Offset)** This field provides the bit offset of the first macroblock of the Slice in the first byte of the input compressed bitstream. This field is the number of valid bits minus 1 of the first byte. For example when this field is set to 7, it indicates that the whole first byte (all 8 bits) are valid; If this field is set to 0, it indicates that only the 1 LSB of the first byte is valid data for the first macroblock. ***Programming notes: Byte alignment of CABAC.*** *According to AVC Specification, if a slice is coded as CABAC (entropy_coding_flag = CABAC), the first macroblock of a slice is always byte aligned, i.e., the slice header is bit padded to align on byte boundary. So for CABAC case, this field must be set to 7 indicating that the first macroblock is byte aligned. Host software must take care of slice header bit padding to find the first bit of the first macroblock. It should be noted that hardware can take care of the slice header bit padding for most cases, but it falls to deal with one corner condition – when the first byte where the slice header ends is the last byte of a cache line.* Format: U3

Dwords 8 is always present for [DevILK].

Dword	Bit	Description
8	**31:0**	Reserved

Dwords 9-15 are only supported by [DevILK]

Dword	Bit	Description
9	31:23	Reserved. MBZ
	22:0	Reserved
10	31:29	Packet-based function. Starting bit position of the Bit Vector in the 1st byte of the fetched Bit Vector surface.
	28:16	Reserved : MBZ
	15:0	Total Packet Count. This is a total count of all the packets in the packet-based bitstream, including those that are fetched for processing and, also those packets which are skipped (because they belonged to a different stream). Format = U23 in bytes If the bitstream is not packet-based, all the fields in this DWord are ignored and MBZ.
11	31:29	Reserved : MBZ
	28:0	Packet-based Bitsream Bit Vector Surface starting Byte address. This field specifies the Graphics Memory starting address of the Bit Vector to be used for skipping/fetching bitstream slice packets. This virtual address points to a 4K page and is therefore 4KByte aligned. Each data bit in this surface refers to a bitstream packet and indicates: '0' – The packet is used in the bitstream decode for the current slice, and therefore needs to be fetched. '1' – The packet is not used in the bitstream decode for the current slice, and therefore need not be fetched. If the bitstream is not packet-based, this field is ignored and MBZ.
12	31:0	Weight128LumaL0 – One bit for each of the l0/l1 entries for luma and chroma components. This bit should be set by driver whenever the 16 bit weight value from the application is equal to 128. this will be passed on as inline data in the media object command. Each of the bits is for one entry in the L0 list with the LSB representing if index 0 is 128 or not for the luma component
13	31:0	Weight128LumaL1 – One bit for each of the l0/l1 entries for luma and chroma components. This bit should be set by driver whenever the 16 bit weight value from the application is equal to 128. this will be passed on as inline data in the media object command. Each of the bits is for one entry in the L1 list with the LSB representing if index 0 is 128 or not for the luma component.
14	31:0	Weight128ChromaL0 – One bit for each of the l0/l1 entries for luma and chroma components. This bit should be set by driver whenever the 16 bit weight value from the application is equal to 128. this will be passed on as inline data in the media object command. Each of the bits is for one entry in the L0 list with the LSB representing if index 0 is 128 or not for the chroma component

Dword	Bit	Description
15	31:0	Weight128ChromaL1– One bit for each of the l0/l1 entries for luma and chroma components. This bit should be set by driver whenever the 16 bit weight value from the application is equal to 128. this will be passed on as inline data in the media object command. Each of the bits is for one entry in the L1 list with the LSB representing if index 0 is 128 or not for the chroma component.

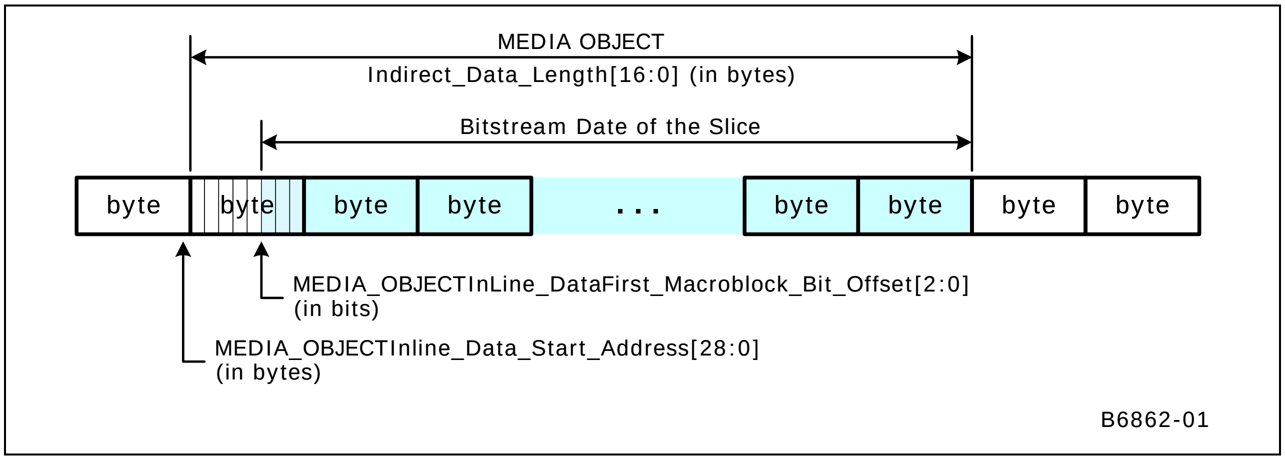
2.7.3.2 Indirect Data Format

Each AVC_BSD_OBJECT command corresponds to the processing of one slice of a picture (a picture can have only one slice). All syntax and derived parameters above Slice Data Layer are passed in either using State commands or as Inline data of the AVC_BSD_OBJECT command, and the Slice Data Layer and below are passed in as indirect data.

The indirect data start address in AVC_BSD_OBJECT specifies the starting Graphics Memory address of the bitsream data that follows the slice header. It provides the byte address for the first macroblock of the slice. Together with the First MB Bit Offset field, it provides the starting bit location of the MB within the compressed bitstream.

The indirect data length in AVC_BSD_OBJECT provides the length in bytes of the bitstream data for this slice. It includes the first byte of the first MB (includes the one that contains the FirstMBBiteOffset) and the last non-zero byte of the last MB in the slice. H/W ignores the contents after the last non-zero byte.

Figure 2-1. Indirect data buffer for a slice


MEDIA OBJECT
Indirect_Data_Length[16:0] (in bytes)
Bitstream Date of the Slice
byte
byte
byte
byte
. . .
byte
byte
byte
byte
MEDIA_OBJECTInLine_DataFirst_Macroblock_Bit_Offset[2:0] (in bits)
MEDIA_OBJECTInline_Data_Start_Address[28:0] (in bytes)
B6862-01

2.8 AVC_BSD Output Definitions

Three data buffers are output from the BSD engine:

- AVC-IT Command Buffer – MB info signals from AC to AM (mb_avail, cbp) + AM (avail, cbp)
- AVC-IT Data Buffer – IPCM or Coeff + MV/Weight+Offset
- AVC-ILDB Data Buffer

The format of the AVC-IT Command Buffer can be found in the *Media* Chapter, under section *MEDIA_OBJECT_EX Command* as well as its subsection *In-line Data Format in AVC-IT Mode*.

The format of the AVC-IT Data Buffer can be found in the *Media* Chapter, under section *MEDIA_OBJECT_EX Command* and subsection *Indirect Data Format in AVC-IT Mode*.

The format of the AVC-LF Data Buffer can be found in the *Data Port* Chapter, under section *AVC Loop Filter Read*.

2.8.1 AVC-IT Command Buffer Description

2.8.2 AVC-IT Data Buffer Description

2.8.2.1 Motion Vectors

Motion vectors (x and y components) are sent to the MC as indirect data in memory together with residual data. In non-I_PCM mode, they are the packed computed MVs (Mvpredicted + MVDelta) of a MB; in I_PCM mode, there is no MV information need to be sent. MV can be generated for DMV-spatial, DMV-temporal or motion-search MV. The two components of MV (MVx, Mvy) are signed extended into a 16-bit signed number each and are packed into a single Dword. MVs of a MB are packed in 512-bit cache lines (or in half-cacheline ???). Based on the MB_type, we can determine the number of MVs being present in the corresponding MB. There can be up to 2, 4, 8, 16 and 32 MVs for a MB in both directions. A MV can associate with a 4x4,4x8,8x4 sub-block, a 8x8, 8x16, 16x8 block or 16x16 MB, and for a direction (L0 and L1). The MV for sub-blocks are sent in a raster order within a block, and in raster order in respect to the blocks within a MB. Zero-value MVs are sent as is. When L1 is not in use, no MV are sent for L1.

DWord	Bit	Description
For each Dword	31:16	**MV_y** Signed extended 2's complement
	15:0	**MV_x** Signed extended 2's complement

2.8.2.1.1 MV Organization for 16x16 Block Size

The MVs are specified in a fixed data structure, which is organized as if there are always four 8x8 blocks. Hence, the real MV(s) is (are) replicated across the remaining entries.

2.8.2.1.2 MV Organization for 16x8 Block

A MB that is 16x8 partitioned consists of 2 parts – top and bottom partitions. However, its MVs are specified as if the MB is broken down into four 8x8 blocks. That is, each 16x8 partition is treated as two identical 8x8 blocks.

2.8.2.1.3 MV Organization for 8x16 Block

A MB that is 8x16 partitioned consists of 2 parts – left and right partitions. However, its MVs are specified as if the MB is broken down into four 8x8 blocks. That is, each 8x16 partition is treated as two identical 8x8 blocks.The MV are specified as if the MB is broken down into four 8x8 blocks.

2.8.2.1.4 MV Organization for 8x8 Block Without Subpartition

A MB that is 8x8 partitioned consists of 4 parts – top-left (0), top-right(1), bottom-left(2) and bottom-right(3) partitions. The MVs are specified for each 8x8 blocks.

2.8.2.1.5 MV Organization for 8x8 Block with Subpartition

A MB that has at least one sub-partitioning (4x4, 8x4 or 4x8), the entire MB is broken down into 16 4x4 units for the specification of MV. That is, the MVs are specified for each 4x4 blocks.

The 4x4 block are scanned in the following order :

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

Replication Rules:

- For an 8x8 partition, the 8x8 MVs are 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 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 are replicated into both 4x4 blocks.
- For a 4x4 subpartition within an 8x8 partition, each 4x4 has its own MVs.
- For L0 MV
    - If L0 interprediction exists, the corresponding L0 MV is used
    - Else if L1 interprediction exits (of the same block), set to the same as L1 MV
    - Else set to 0
- For L1 MV
    - If L1 interprediction exists, the corresponding L1 MV is used
    - Else if L0 interprediction exits (of the same block), set to the same as L0 MV
    - Else set to 0

MBLK data (containing the packed MV/Weight,Offset/Coeff) offset in the indirect buffer can be dword aligned. MV/Weight,offset/Coeff are packed in memory as dword aligned data structure (constraint to Dword align for a buffer), VFE has to derive the size for each based on bMSize. BSD output is half_cacheline aligned.

Block address (right to left)

MV write out first, then coeff

Weight offset does not go down to 4x4.

Replicate Weight/Offset, decomp 16x8 to 8x8, with the same Weight/Offset.

2.8.2.2 Inter-prediction Weight and Offset Block

Weight and offset are packed together next to each other. A weight and an offset are each 1 byte. The high byte is always the weight and the lower byte is always the offset. The smallest block size for Weight and Offset specifications is 8x8 (no subpartitioning, since interprediction mode does not go below 8x8).

2.8.2.2.1 Weight+Offset Organization for 16x16 Block Size

The Weight+Offset are specified in a fixed data structure. Hence, the real Weight+Offsets are replicated across the remaining entries.

	DWord	Description
	DW0 [15:0]	**L0 Weight+Offset_Y0** The high byte is weight and the low byte is offset If L0 interprediction exists, it is equal to L0 Weight+Offset_Y0 Else if L1 interprediction exists, it is a replication of L1 Weight+Offset_Y0 Else set to 0
	DW0 [31:16]	**L1 Weight+Offset_Y0** The high byte is weight and the low byte is offset If L1 interprediction exists, it is equal to L1 Weight+Offset_Y0 Else if L0 interprediction exists, it is a replication of L0 Weight+Offset_Y0 Else set to 0
	DW1 [15:0]	**L0 Weight+Offset_Cb0** The high byte is weight and the low byte is offset If L0 interprediction exists, it is equal to L0 Weight+Offset_Cb0 Else if L1 interprediction exists, it is a replication of L1 Weight+Offset_Cb0 Else set to 0
	DW1 [31:16]	**L1 Weight+Offset_Cb0** The high byte is weight and the low byte is offset If L1 interprediction exists, it is equal to L1 Weight+Offset_Cb0 Else if L0 interprediction exists, it is a replication of L0 Weight+Offset_Cb0 Else set to 0
	DW2 [15:0]	**L0 Weight+Offset_Cr0** The high byte is weight and the low byte is offset If L0 interprediction exists, it is equal to L0 Weight+Offset_Cr0 Else if L1 interprediction exists, it is a replication of L1 Weight+Offset_Cr0 Else set to 0
	DW2 [31:16]	**L1 Weight+Offset_Cr0** The high byte is weight and the low byte is offset If L1 interprediction exists, it is equal to L1 Weight+Offset_Cr0 Else if L0 interprediction exists, it is a replication of L0 Weight+Offset_Cr0 Else set to 0

	DWord	Description
	DW3	Reserved.
	DW4	Replicated DW0 (L0 Weight+Offset_Y0 and L1 Weight+Offset_Y0)
	DW5	Replicated DW1 (L0 Weight+Offset_Cb0 and L1 Weight+Offset_Cb0)
	DW6	Replicated DW2 (L0 Weight+Offset_Cr0 and L1 Weight+Offset_Cr0)
	DW7	Reserved.

2.8.2.2.2 Weight+Offset Organization for non-16x16 Block Size

The Weight+Offset are specified in a fixed data structure. It is organized as if the MB is broken into four 8x8 blocks. Hence, the real Weight+Offsets are replicated across the remaining entries.

Replication Rules:

- For 8x16 partition, each 8x16 is broken down into 2 8x8 stacking vertically. The 8x16 Weight+Offset are replicated into the two 8x8 blocks.
- For 16x8 partition, each 16x8 is broken down into 2 8x8 stacking horizontally. The 16x8 Weight+Offset are replicated into the two 8x8 blocks.
- For 8x8 partition, the MB is broken down into 4 8x8 blocks. Each has its own set of Weight and Offset.

	DWord	Description
	DW0 [15:0]	**L0 Weight+Offset_Y0** The high byte is weight and the low byte is offset If L0 interprediction exists, it is equal to L0 Weight+Offset_Y0 Else if L1 interprediction exists, it is a replication of L1 Weight+Offset_Y0 Else set to 0
	DW0 [31:16]	**L1 Weight+Offset_Y0** The high byte is weight and the low byte is offset If L1 interprediction exists, it is equal to L1 Weight+Offset_Y0 Else if L0 interprediction exists, it is a replication of L0 Weight+Offset_Y0 Else set to 0
	DW1 [15:0]	**L0 Weight+Offset_Cb0** The high byte is weight and the low byte is offset If L0 interprediction exists, it is equal to L0 Weight+Offset_Cb0 Else if L1 interprediction exists, it is a replication of L1 Weight+Offset_Cb0 Else set to 0

	DWord	Description
	DW1 [31:16]	**L1 Weight+Offset_Cb0** The high byte is weight and the low byte is offset If L1 interprediction exists, it is equal to L1 Weight+Offset_Cb0 Else if L0 interprediction exists, it is a replication of L0 Weight+Offset_Cb0 Else set to 0
	DW2 [15:0]	**L0 Weight+Offset_Cr0** The high byte is weight and the low byte is offset If L0 interprediction exists, it is equal to L0 Weight+Offset_Cr0 Else if L1 interprediction exists, it is a replication of L1 Weight+Offset_Cr0 Else set to 0
	DW2 [31:16]	**L1 Weight+Offset_Cr0** The high byte is weight and the low byte is offset If L1 interprediction exists, it is equal to L1 Weight+Offset_Cr0 Else if L0 interprediction exists, it is a replication of L0 Weight+Offset_Cr0 Else set to 0
	DW3	Reserved.
	DW4 [15:0]	L0 Weight+Offset_Y1 The high byte is weight and the low byte is offset If L0 interprediction exists, it is equal to L0 Weight+Offset_Y1 Else if L1 interprediction exists, it is a replication of L1 Weight+Offset_Y1 Else set to 0
	DW4 [31:16]	L1 Weight+Offset_Y1 The high byte is weight and the low byte is offset If L1 interprediction exists, it is equal to L1 Weight+Offset_Y1 Else if L0 interprediction exists, it is a replication of L0 Weight+Offset_Y1 Else set to 0
	DW5 [15:0]	L0 Weight+Offset_Cb1 The high byte is weight and the low byte is offset If L0 interprediction exists, it is equal to L0 Weight+Offset_Cb1 Else if L1 interprediction exists, it is a replication of L1 Weight+Offset_Cb1 Else set to 0
	DW5 [31:16]	L1 Weight+Offset_Cb1 The high byte is weight and the low byte is offset If L1 interprediction exists, it is equal to L1 Weight+Offset_Cb1 Else if L0 interprediction exists, it is a replication of L0 Weight+Offset_Cb1 Else set to 0
	DW6 [15:0]	L0 Weight+Offset_Cr1 The high byte is weight and the low byte is offset If L0 interprediction exists, it is equal to L0 Weight+Offset_Cr1 Else if L1 interprediction exists, it is a replication of L1 Weight+Offset_Cr1 Else set to 0

	DWord	Description
	DW6 [31:16]	L1 Weight+Offset_Cr1 The high byte is weight and the low byte is offset If L1 interprediction exists, it is equal to L1 Weight+Offset_Cr1 Else if L0 interprediction exists, it is a replication of L0 Weight+Offset_Cr1 Else set to 0
	DW7	Reserved.
	DW8 [15:0]	L0 Weight+Offset_Y2 The high byte is weight and the low byte is offset If L0 interprediction exists, it is equal to L0 Weight+Offset_Y2 Else if L1 interprediction exists, it is a replication of L1 Weight+Offset_Y2 Else set to 0
	DW8 [31:16]	L1 Weight+Offset_Y2 The high byte is weight and the low byte is offset If L1 interprediction exists, it is equal to L1 Weight+Offset_Y2 Else if L0 interprediction exists, it is a replication of L0 Weight+Offset_Y2 Else set to 0
	DW9 [15:0]	L0 Weight+Offset_Cb2 The high byte is weight and the low byte is offset If L0 interprediction exists, it is equal to L0 Weight+Offset_Cb2 Else if L1 interprediction exists, it is a replication of L1 Weight+Offset_Cb2 Else set to 0
	DW9 [31:16]	L1 Weight+Offset_Cb2 The high byte is weight and the low byte is offset If L1 interprediction exists, it is equal to L1 Weight+Offset_Cb2 Else if L0 interprediction exists, it is a replication of L0 Weight+Offset_Cb2 Else set to 0
	DW10 [15:0]	L0 Weight+Offset_Cr2 The high byte is weight and the low byte is offset If L0 interprediction exists, it is equal to L0 Weight+Offset_Cr2 Else if L1 interprediction exists, it is a replication of L1 Weight+Offset_Cr2 Else set to 0
	DW10 [31:16]	L1 Weight+Offset_Cr2 The high byte is weight and the low byte is offset If L1 interprediction exists, it is equal to L1 Weight+Offset_Cr2 Else if L0 interprediction exists, it is a replication of L0 Weight+Offset_Cr2 Else set to 0
	DW11	Reserved.

	DWord	Description
	DW12 [15:0]	L0 Weight+Offset_Y3 The high byte is weight and the low byte is offset If L0 interprediction exists, it is equal to L0 Weight+Offset_Y3 Else if L1 interprediction exists, it is a replication of L1 Weight+Offset_Y3 Else set to 0
	DW12 [31:16]	L1 Weight+Offset_Y3 The high byte is weight and the low byte is offset If L1 interprediction exists, it is equal to L1 Weight+Offset_Y3 Else if L0 interprediction exists, it is a replication of L0 Weight+Offset_Y3 Else set to 0
	DW13 [15:0]	L0 Weight+Offset_Cb3 The high byte is weight and the low byte is offset If L0 interprediction exists, it is equal to L0 Weight+Offset_Cb3 Else if L1 interprediction exists, it is a replication of L1 Weight+Offset_Cb3 Else set to 0
	DW13 [31:16]	L1 Weight+Offset_Cb3 The high byte is weight and the low byte is offset If L1 interprediction exists, it is equal to L1 Weight+Offset_Cb3 Else if L0 interprediction exists, it is a replication of L0 Weight+Offset_Cb3 Else set to 0
	DW14 [15:0]	L0 Weight+Offset_Cr3 The high byte is weight and the low byte is offset If L0 interprediction exists, it is equal to L0 Weight+Offset_Cr3 Else if L1 interprediction exists, it is a replication of L1 Weight+Offset_Cr3 Else set to 0
	DW14 [31:16]	L1 Weight+Offset_Cr3 The high byte is weight and the low byte is offset If L1 interprediction exists, it is equal to L1 Weight+Offset_Cr3 Else if L0 interprediction exists, it is a replication of L0 Weight+Offset_Cr3 Else set to 0
	DW15	Reserved.

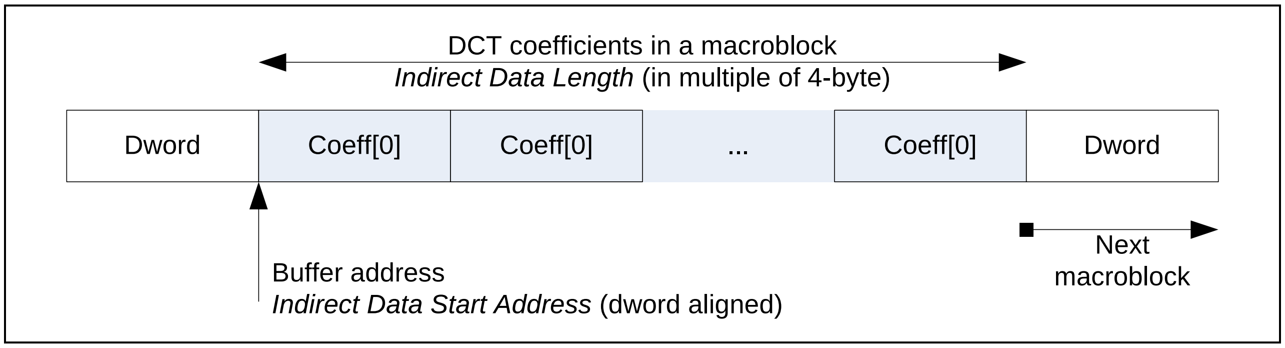
2.8.2.3 Quantized DCT Coefficient Block

In non-I_PCM mode, quantized DCT coefficients are sent to AVC-IT as indirect data in memory (AVC-IT Data Buffer). They are organized on a per MB basis (does it include skip MB and are they zero filled), i.e. all coefficients of a MB are packed together; in I_PCM mode, the AVC-IT buffer is packed with actual 8-bit pixel samples on a per MB basis.

Coefficients are packed in 512-bit cache lines (Note: should be dword aligned, not cache line aligned???).

Only the non-zero quantized DCT coefficients within a MB are sent and they are packed in the order of 4x4DCY (if any), 8x8Y0, 8x8Y1, 8x8Y2, 8x8Y3, 2x2DCCb (if any), 2x2DCCr (if any), 8x8Cb4 and 8x8Cr5. For I16x16Intra mode, the DCY is the very first block; in all other prediction modes, 8x8Y0 is the first block.

Structure of the Quantized DCT Coefficients Data Buffer


DCT coefficients in a macroblock
Indirect Data Length (in multiple of 4-byte)
Dword
Coeff[0]
Coeff[0]
...
Coeff[0]
Dword
Buffer address
Indirect Data Start Address (dword aligned)
Next macroblock

Each non-zero coefficient in the indirect data buffer is contained in a doubleword-size (32-bit) data structure containing the coefficient index, end of block (EOB) flag and the fixed-point coefficient value in 2's compliment form.

index is the row major 'raster' index of the coefficient within an 8x8 block. DCT coefficient is a 16-bit value in 2's complement, which is clamped to a 12-bit signed value by the host.

Structure of a quantized DCT coefficient block

DWord	Bit	Description
For each Dword	31:16	**Qunatized DCT Coefficient Value or Pixel sample value** This field contains the value of the non-zero quantized DCT coefficient in 2's compliment.
	15:7	Reserved: MBZ

	6:1	**Index**
		This field specifies the raster-scan address (raw address) of the quantized DCT coefficient (or pixel sample in I_PCM) within their corresponding block (some of size 8x8, and some are 4x4 or 2x2) . For example, coefficient in an 8x8Y0 at location (row, column) = (0, 0) has an index of 0; that at (2, 3) has an index of 2*8 + 3 = 19. For 4x4 YDC block, index ranges from 0 to 15; for 2x2 CrDC/CbDC, ranges from 0 to 3.
		Format = U6
		Range = [0, 63]

	0	**EOB (End of Block)** This field indicates whether the quantized DCT coefficient is the last one of the current block (8x8, 4x4 or 2x2).

2.8.2.4 I_PCM Pixel Sample Block

DWord	Bit	Description
For each Dword	31:0	**8-bit Packed Pixel sample value** Each Dword contains 4 pixel sample values (zero or not) packed together.

2.8.2.5 Motion Prediction Reference Indices

Motion prediction reference indices for L0 and L1 are sent separately to the MC as inline data in memory. In non-I_PCM mode, they are the packed for a MB and are aligned with the associated MVs; in I_PCM mode, there is no MV information need to be sent. MV can be generated for DMV-spatial, DMV-temporal or motion-search MV. Each reference index is a 4bit unsigned number, except that 1111 (-1) is used to mark for a non-existing reference picture. 8 reference indices can be packed into a single Dword. Reference indices of a MB are packed in 512-bit cache lines (or in half-cacheline ???). Based on the MB_type, we can determine the number of reference indices being present in the corresponding MB. A Reference indices can associate with a a 8x8, 8x16, 16x8 block or 16x16 MB, and for a direction (L0 and L1). The Reference indices for sub-blocks are sent in a raster order within a block, and in raster order in respect to the blocks within a MB. When L1 is not in use, no reference indices are sent for L1.

2.8.2.6 MB Neighbor Availability in Intra-Prediction Modes

mb_nb_avail_intra[4:0] provides the individual MB availability information as follows. [4:3] bit ordering is under discussion. Current MB is labelled as X. For a MB pair, both the top and the bottom have the same picture type.

- mb_nb_avail_intra[0] : MB D (top left neighbor of current MB X) availability
- mb_nb_avail_intra[1] : MB C (top right neighbor of current MB X) availability
- mb_nb_avail_intra[2] : MB B (top neighbor of current MB X) availability
- mb_nb_avail_intra[3] : MB A availability. A is the left neighbor of the current MB X, or A is the top MB of the left neighbor of the current MB pair X.
- mb_nb_avail_intra[4] : MB E availability. E is the bottom MB of the left neighbor of the current MB pair X.

mb_nb_avail_intra[4 :0] is derived from the internal intermediate variable mb_avail_i [0..7] (internally generated by AM unit based on mb_nb_avail[0..3]), which in turns is derived from the initial variable mb_nb_avail[3:0].

mb_nb_avail[3:0] is derived for a MB (a MB unit) if in non-MBAFF mode or for a MB pair (a MB unit) if in MBAFF mode, from the following conditions :

1. A MB is marked as “Not Available” if anyone of the following conditions is true :
   - (MbAddr – Start_MB_addr) < 0 (error condition), for each Slice
   - MbAddr > CurrMBAddr (go beyond the current MB location, i.e. to the right or below)
   - The MB with address MbAddr belongs to a different slice than the MB with address CurrMbAddr
   - Set the mb_nb_avail[i] accordingly for the MB; mb_nb_avail[i] = 0 if “not available”, i = 0 to 3

2. For each non-pair MB with address CurrMbAddr, check the following availability, and set the mb_nb_avail[i] accordingly for the corresponding MB
   - mbAddrA = CurrMbAddr – 1,
     - mbAddrA is marked as not available when CurrMbAddr % PicWidthInMbs is equal to 0 (i.e. the MB with address CurrMbAddr is located along the left edge of the picture)
   - mbAddrB = CurrMbAddr – PicWidthInMbs
   - mbAddrC = CurrMbAddr – PicWidthInMbs + 1
     - mbAddrC is marked as not available when (CurrMbAddr + 1) % PicWidthInMbs is equal to 0 (i.e. the MB with address CurrMbAddr is located along the right edge of the picture)
   - mbAddrD = CurrMbAddr – PicWidthInMbs – 1
     - mbAddrD is marked as not available when CurrMbAddr % PicWidthInMbs is equal to 0 (i.e. the MB with address CurrMbAddr is located along the left edge of the picture)

mbAddrD D (top-left) mb_nb_avail[0] mb_avail_i[0,1]	mbAddrB B (top) mb_nb_avail[2] mb_avail_i[4,5]	mbAddrC C (top-right) mb_nb_avail[1] mb_avail_i[2,3]
mbAddrA A (left) mb_nb_avail[3] mb_avail_i[6,7]	X CurrMbAddrX	N/A
N/A	N/A	N/A

3. For each MB pair (in MBAFF mode) with address CurrMbAddr, check the following availability, and set the mb_nb_avail[i] accordingly for the corresponding neighbor MB pair (Note that in MBAFF mode, the top MB always has an even address and the bottom MB has an odd address; also in MBAFF, both MBs must have the same availability value, hence no need to specify top or bottom MB in the mb_nb_avail[i]. However, the top and the bottom MB may code with different modes - intra or inter).
   - mbAddrA =2 * (CurrMbAddr / 2 – 1)
     - mbAddrA is marked as not available when (CurrMbAddr / 2) % PicWidthInMbs is equal to 0 (i.e. the MB pair with address CurrMbAddr is located along the left edge of the picture)
   - mbAddrB =2 * (CurrMbAddr / 2 – PicWidthInMbs)
   - mbAddrC = 2 * (CurrMbAddr / 2 – PicWidthInMbs + 1)
     - mbAddrC is marked as not available when (CurrMbAddr / 2 + 1) % PicWidthInMbs is equal to 0 (i.e. the MB with address CurrMbAddr is located along the right edge of the picture)
   - mbAddrD = 2 * (CurrMbAddr / 2 – PicWidthInMbs - 1)
     - mbAddrD is marked as not available when (CurrMbAddr / 2) % PicWidthInMbs is equal to 0 (i.e. the MB with address CurrMbAddr is located along the left edge of the picture)
   - mbAddrE = mbAddrA + 1
     - its MB availability should be the same as that of mbAddrA

mbAddrD D0 mb_nb_avail[0] mb_avail_i[0]	mbAddrB B0 mb_nb_avail[2] mb_avail_i[4]	mbAddrC C0 mb_nb_avail[1] mb_avail_i[2]