# Micro Engine Scheduler Specification

Date: April 2024

# Specification Agreement

This Specification Agreement (this "Agreement") is a legal agreement between Advanced Micro Devices, Inc. ("AMD") and "You" as the recipient of the attached AMD Specification (the "Specification"). If you are accessing the Specification as part of your performance of work for another party, you acknowledge that you have authority to bind such party to the terms and conditions of this Agreement. If you accessed the Specification by any means or otherwise use or provide Feedback (defined below) on the Specification, You agree to the terms and conditions set forth in this Agreement. If You do not agree to the terms and conditions set forth in this Agreement, you are not licensed to use the Specification; do not use, access or provide Feedback about the Specification. In consideration of Your use or access of the Specification (in whole or in part), the receipt and sufficiency of which are acknowledged, You agree as follows:

1. You may review the Specification only (a) as a reference to assist You in planning and designing Your product, service or technology ("Product") to interface with an AMD product in compliance with the requirements as set forth in the Specification and (b) to provide Feedback about the information disclosed in the Specification to AMD.

2. Except as expressly set forth in Paragraph 1, all rights in and to the Specification are retained by AMD. This Agreement does not give You any rights under any AMD patents, copyrights, trademarks or other intellectual property rights. You may not (i) duplicate any part of the Specification; (ii) remove this Agreement or any notices from the Specification, or (iii) give any part of the Specification, or assign or otherwise provide Your rights under this Agreement, to anyone else.

3. The Specification may contain preliminary information, errors, or inaccuracies, or may not include certain necessary information. Additionally, AMD reserves the right to discontinue or make changes to the Specification and its products at any time without notice. The Specification is provided entirely "AS IS." AMD MAKES NO WARRANTY OF ANY KIND AND DISCLAIMS ALL EXPRESS, IMPLIED AND STATUTORY WARRANTIES, INCLUDING BUT NOT LIMITED TO IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, NONINFRINGEMENT, TITLE OR THOSE WARRANTIES ARISING AS A COURSE OF DEALING OR CUSTOM OF TRADE. AMD SHALL NOT BE LIABLE FOR DIRECT, INDIRECT, CONSEQUENTIAL, SPECIAL, INCIDENTAL, PUNITIVE OR EXEMPLARY DAMAGES OF ANY KIND (INCLUDING LOSS OF BUSINESS, LOSS OF INFORMATION OR DATA, LOST PROFITS, LOSS OF CAPITAL, LOSS OF GOODWILL) REGARDLESS OF THE FORM OF ACTION WHETHER IN CONTRACT, TORT (INCLUDING NEGLIGENCE) AND STRICT PRODUCT LIABILITY OR OTHERWISE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

4. Furthermore, AMD’s products are not designed, intended, authorized or warranted for use as components in systems intended for surgical implant into the body, or in other applications intended to support or sustain life, or in any other application in which the failure of AMD’s product could create a situation where personal injury, death, or severe property or environmental damage may occur.

5. You have no obligation to give AMD any suggestions, comments or feedback ("Feedback") relating to the Specification. However, any Feedback You voluntarily

provide may be used by AMD without restriction, fee or obligation of confidentiality. Accordingly, if You do give AMD Feedback on any version of the Specification, You agree AMD may freely use, reproduce, license, distribute, and otherwise commercialize Your Feedback in any product, as well as has the right to sublicense third parties to do the same. Further, You will not give AMD any Feedback that You may have reason to believe is (i) subject to any patent, copyright or other intellectual property claim or right of any third party; or (ii) subject to license terms which seek to require any product or intellectual property incorporating or derived from Feedback or any Product or other AMD intellectual property to be licensed to or otherwise provided to any third party.

6. You shall adhere to all applicable U.S., European, and other export laws, including but not limited to the U.S. Export Administration Regulations ("EAR"), (15 C.F.R. Sections 730 through 774), and E.U. Council Regulation (EC) No 428/2009 of 5 May 2009. Further, pursuant to Section 740.6 of the EAR, You hereby certifies that, except pursuant to a license granted by the United States Department of Commerce Bureau of Industry and Security or as otherwise permitted pursuant to a License Exception under the U.S. Export Administration Regulations ("EAR"), You will not (1) export, re-export or release to a national of a country in Country Groups D:1, E:1 or E:2 any restricted technology, software, or source code You receive hereunder, or (2) export to Country Groups D:1, E:1 or E:2 the direct product of such technology or software, if such foreign produced direct product is subject to national security controls as identified on the Commerce Control List (currently found in Supplement 1 to Part 774 of EAR). For the most current Country Group listings, or for additional information about the EAR or Your obligations under those regulations, please refer to the U.S. Bureau of Industry and Security's website at http://www.bis.doc.gov/.

7. If You are a part of the U.S. Government, then the Specification is provided with "RESTRICTED RIGHTS" as set forth in subparagraphs (c) (1) and (2) of the Commercial Computer Software-Restricted Rights clause at FAR 52.227-14 or subparagraph (c) (1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.277-7013, as applicable.

8. This Agreement is governed by the laws of the State of California without regard to its choice of law principles. Any dispute involving it must be brought in a court having jurisdiction of such dispute in Santa Clara County, California, and You waive any defenses and rights allowing the dispute to be litigated elsewhere. If any part of this agreement is unenforceable, it will be considered modified to the extent necessary to make it enforceable, and the remainder shall continue in effect. The failure of AMD to enforce any rights granted hereunder or to take action against You in the event of any breach hereunder shall not be deemed a waiver by AMD as to subsequent enforcement of rights or subsequent actions in the event of future breaches. This Agreement is the entire agreement between You and AMD concerning the Specification; it may be changed only by a written document signed by both You and an authorized representative of AMD.

# Contents

# Scheduling requirements

At a high-level, the scheduling requirements can be summarized as:

- Fair and efficient scheduling of the application's work on the GPU
- Implementation of multiple priority levels for a variety of user scenarios

These high-level requirements can also be described from a user scenario perspective:

- Applications with the same priority level should get the equal amount of the GPU execution time
- Applications with the user focus (for e.g. compositor) should receive larger GPU time, but not infinitely starve the Normal priority level
- Real time work such VR, Super-Wet ink or True audio should run immediately and can infinitely starve work in the lower priority levels
- Low-priority work such as OneDrive, photo enhancement, compression or Folding@home should only run when all higher priority levels are idle

Scheduler implements the above stated requirements via 4 levels of queue prioritization.

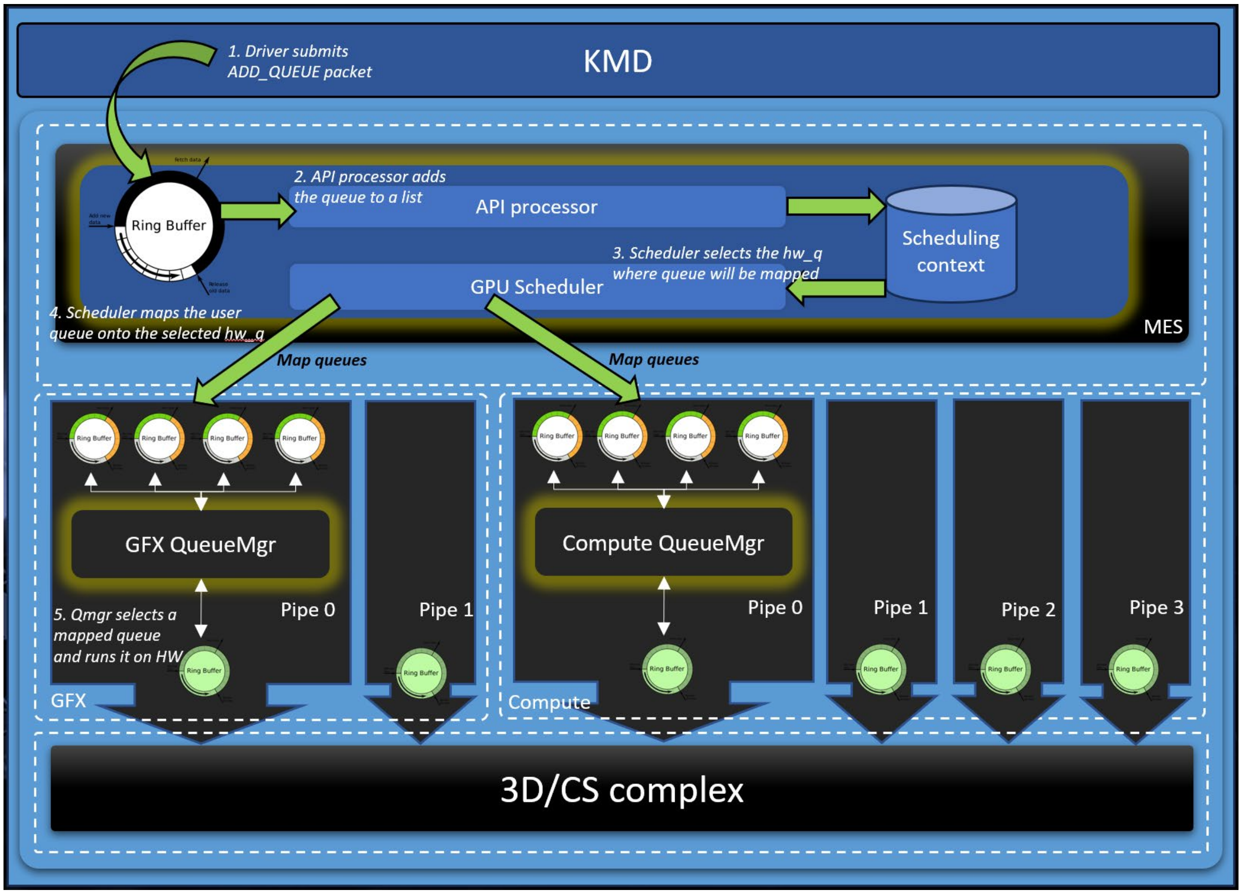
| **Level** | **Scheduling expectation** | **What runs here** |
|---|---|---|
| Real time | Lowest possible launch latency. | VR compositor, Super wet ink, True audio next. |
| Focus | Provides no forward progress guarantee for the lower levels. | Desktop compositor, Video post processing, foreground app's work. |
| Normal | Gets majority of GPU execution time in the absence of Real time work. | Typical work from the application that does not have the user focus |
| Low | Ensures forward progress for the Normal level work. | All background work with no strict deadline requirements for e.g. file compression, encryption etc. |

This scheduling behavior mirrors Microsoft specifications for GPU scheduling. The requirements are captured in the Microsoft GPU scheduling specification and are not explained further.

# HW architecture overview

The scheduler firmware's main role is to map the scheduling requirement on to the HW architecture. Therefore, it is required to understand the HW architecture to understand how scheduling firmware achieves the scheduling requirements on the AMD GPUs.

The following diagram describes the high-level HW architecture and execution flow to schedule/run an application queue.



Key highlights of HW architecture can be summarized as follows.

- The GPU frontend has three micro-processors meant to execute scheduling, compute and gfx firmware
- There are multiple GFX and Compute pipes where each pipe contains a queue mgr that arbitrates a certain number of HW queues attached to that pipe
- There are two levels of scheduling:
    - First level of scheduling is at firmware, where firmware decides the applications queues that should be mapped onto the available hardware queues on various pipes

  - Second level of scheduling is in the Queue Manager HW where it selects one of the ready hardware queue and runs it on the shader complex. Although the second level of scheduling is done by Queue manager hardware, scheduler FW is able to influence the Queue manager's hardware queue selection and execution via various knobs such as hardware queue priority, quantum etc.

- Queue manager's arbitration logic selects a HW queue and runs it on the shader complex. The mapped hardware queue selected for execution is called a "connected queue"

- Each pipe provides an independent path to launch a queue's work inside 3D/CS complex. So potentially there could be #pipes worth of "connected queues" running in parallel

- There is a shared pool of ALUs for GFX and compute work

Refer to RDNA3 Instruction Set Architecture Reference Guide for additional information.

# Scheduler FW architecture

The scheduler firmware architecture can be decomposed into following key components:

1. **Scheduler APIs**
   These are the commands sent by the driver to inform scheduler of the events such as queue creation, destruction, suspension, or any changes to its priority. Each API is described later under APIs section.

2. **Scheduler context**
   Data structures where scheduler maintains application, queue state or any other scheduling state or configuration.

   Scheduler context is the state that API processor and Core scheduler thread works on. The scheduler context consists of:

   HW resource state

   - HQD State – Current Queue mapped, queue type, scheduled time.
   - VMID State  - Current process mapped
   - GDS State – Current process using the GDS partition.

   Process scheduling state

   - Scheduling level state - process list, grace period, normalband percentage, has_ready_queues
   - Process state – Gang list for each context priority(-7/+7), processquantum, running time carryover
   - Per Gang state – Queuelist, running time carryover, gang quantum.

3. **API processor**
   Processes the APIs submitted by the driver and modifies the scheduler state if required.

4. **Core Scheduler**
   Looks at the scheduler state, decide next set of scheduling actions and applies them.
   For example, mapping a queue when it is created, or suspending as required. The scheduling algorithm is described in a dedicated section later in this document.

5. **Interrupt Handler**
   Handles interrupts from various internal HW blocks.
   For example, interrupt handlers reads the API data from the fetcher or collects the busy, idle state of various hardware queues.

These are the main types of interrupts that RS64 processer will receive:

| Interrupt source | Description |
|---|---|
| ME0 Pipe0 | Gfx pipe |
| ME1 Pipe0/1/2/3 | First 4 compute pipes |
| ME2 Pipe0/1/2/3 | Other 4 compute pipes |
| MES packet fifo | Indicates new data in the MES queues |
| Hardware queue Message interrupt | QueueManager interrupts |
| Software interrupt | Caused by MES fw itself |
| Timer interrupt | Used for Timer expiration |
| Unprivileged access | Unprivileged access of MES registers |
| External interrupt | From Non-gfx blocks |

# Scheduling algorithm

Here are queue terminologies with descriptions to assist in understanding the queue state transitions, before describing the scheduling algorithm.

- **User queue**
  Represents a linear command stream of draws or dispatches from an application. It would be analogous to a thread in the CPU world. There we few memory resources allocated for user queue such as ring buffer where command packets are submitted by the application and a memory to save the HW execution state of the queue when it is preempted. A user queue does not execute on its own. It needs to be mapped onto a HW queue for it to execute.

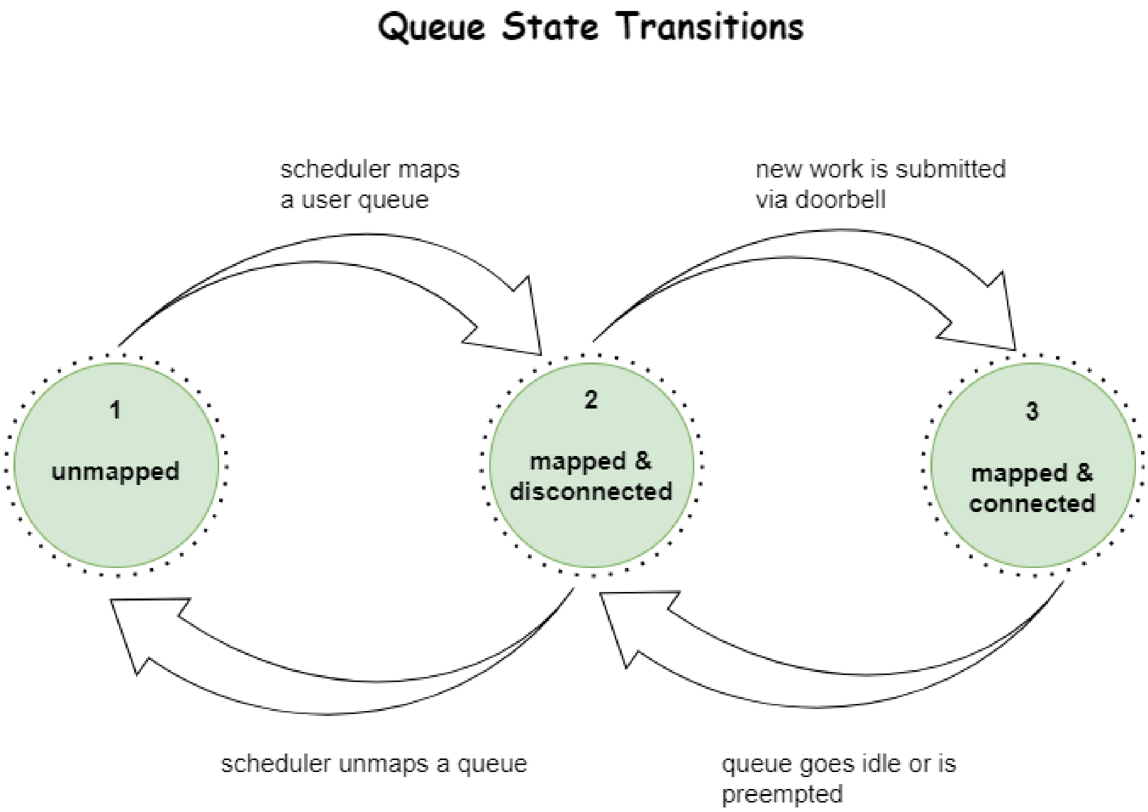
- **Hardware queue**
  A hardware descriptor that holds the user queue state (for e.g. ring buffer address, read, write pointers etc). A hardware queue could be in a mapped or unmapped state. And a mapped queue could be in a connected or a disconnected state.

- **Queue mapping/un-mapping**
  Mapping is an act of loading a user queue state onto a hardware queue. And un mapping is an act of moving the queue state from a hardware queue descriptor to memory. A hardware queue can only be unmapped after preemption.

- **Connected queue**
  Hardware queue that is selected by queue manager to run on the 3d/CS complex.

## Queue state transitions

This diagram describes the possible queue states and triggers for the transitions.



Based on this illustration, a queue could be in one of the following states:

- **Unmapped**
  The user queue has not been initialized into a hardware queue and it solely exists in memory.

- **Mapped & disconnected**
  The user queue has been initialized into a hardware queue but is currently not connected to the shader subsystem so is not able to execute.

- **Mapped and connected**
  The user queue has been initialized into a hardware queue and is connected to the shader subsystem. Only connected queues are able to request and launch their work on the shader resources. Only queues with pending work are allowed to connect.

The GPUSCH implementation can be explained in two steps where first we go into the round robin scheduling and secondly we look at how different levels of queue priority are implemented.

## Round robin scheduling

Round robin scheduling refers to the vanilla round robin scheduling where queues from all applications have the same priority, and the scheduler is expected to provide an equal amount of gpu time to each application.

Schedules achieves this by:

- maintaining a database of queues from all applications
- mapping them on to available hardware queues based on their scheduling turn. The database referred above is the scheduling context that contains queue list for each unique pair of queue type(GFX, Compute, DMA) and priority level.

There are 12 queue lists in total maintained inside the scheduler context.

Scheduler context also contains queue or process specific information such as MQD pointers, VMIDs or any special resources allocated to the queue or the process. Various APIs from the driver result in queue and process information to be updated inside the scheduler context.

Any updates to the scheduler context are then acted upon by the scheduler by performing certain scheduling actions such as queue map or unmap.

AMD GPU has certain number of pipes, and each pipe has a fixed set of hardware queues. The user queues must be mapped onto the hardware queues to execute their work. Since there are limited number of hardware queues, the scheduler will attempt to map as many user queues on the hardware queues as possible.

When a user queue is mapped on the hardware queue, the scheduler configures a quantum that the queue must run. Once the quantum has expired, the queue manager will connect the next hardware queue on the same pipe.

When the hardware queues are not over-subscribed (#user queues <= #hardware queues), the scheduler will map all user queues on the hardware queues and configure equal quantum for all queues.

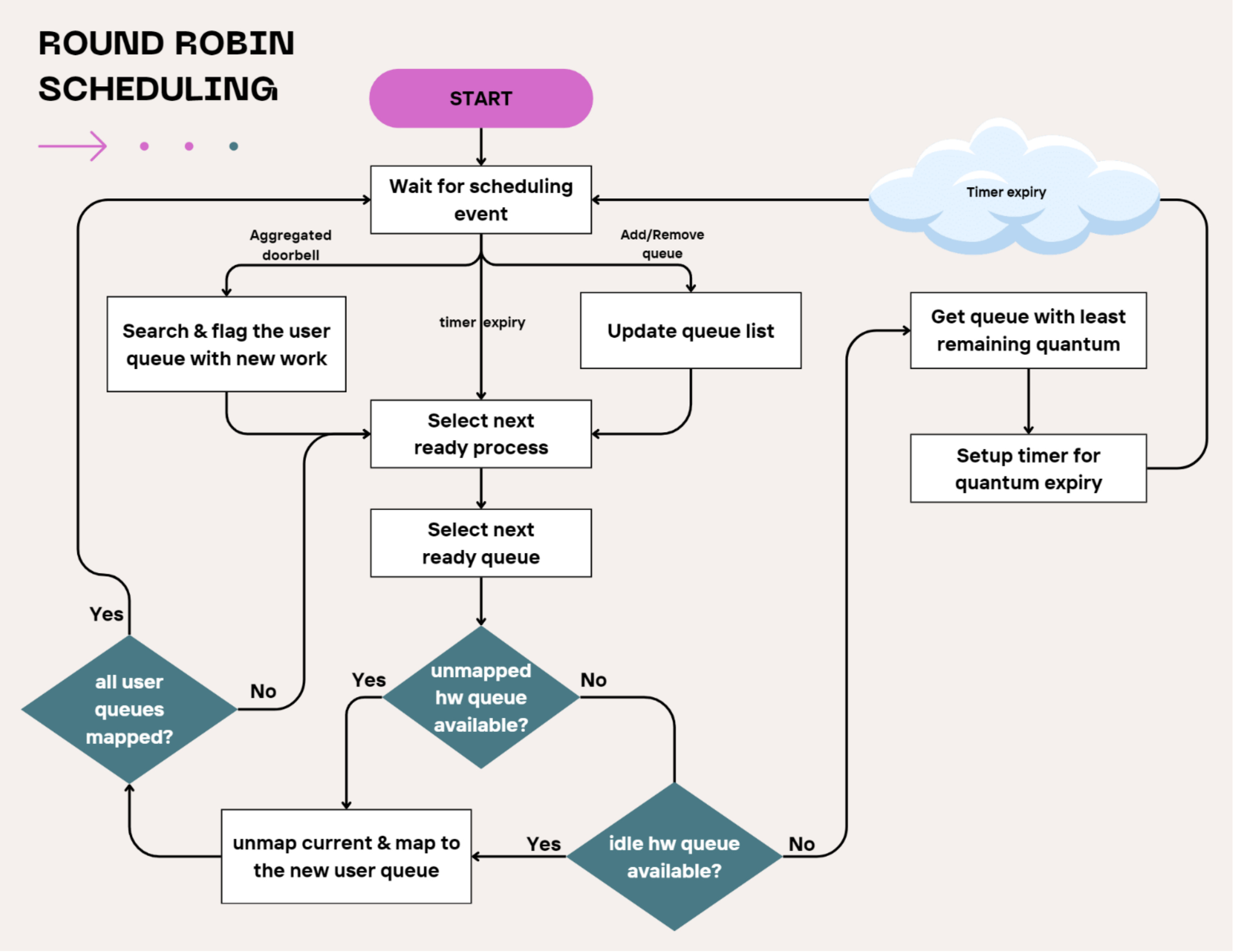
This allows the queue manager to “connect” each hardware queue for an equal amount of configure time. It is possible that a “connected queue” may go idle before its quantum has expired, in which case the queue manager will connect the next hardware queue that has ready work to execute.

When the hardware queues are over-subscribed (#user queues > #hardware queues), the scheduler will map as many queues possible on the available HW queues and will unmap them gradually upon quantum expiry or when they go idle to map the queues from the next process.

To ensure that the limited number of hardware queues are used in best way possible, the scheduler only maps user queues with outstanding work to execute. This requires the scheduler to be informed when an unmapped queue has new work.

This is achieved using aggregated doorbells. Aggregated doorbells are special doorbells that are written by SW when it submits work to an unmapped queue. Write to an aggregated doorbell informs the scheduler of new work to an unmapped queue. The scheduler then uses this notification to map the queue as soon as possible, based on the queue's priority relative to the other work. When aggregated doorbells are not available or used by the SW, scheduler start to periodically polls the write pointer memory of the unmapped queues to discover if they have new work. This is only done when there is a queue-over subscription as all user queues could not be mapped on to the limited hardware queues.

This flowchart shows the event driven scheduling design and how scheduler handles these events to implement a basic round robin scheduling of the user queues.



## Queue prioritization

The scheduler maps as many user queues as possible to the available HW queues. Once the HW queues are over-subscribed, the scheduler starts to round robin the user queues onto the available HW queues.

This basic round robin scheme falls short when it comes to executing work of varying priority levels. The scheduler uses a combination of various hardware prioritization features to implement the desired scheduling behavior for each priority level.

Before discussing the scheduler's usage of these prioritization features, it's useful to discuss the various hardware prioritization features available for scheduler's use:

- **Mid command buffer preemption**
  Queue preemption is the most fundamental feature that is employed in various prioritization scenarios to achieve the desired quality of service. Preemption can be issued at several different work boundaries that affects the latency and the amount of state that gets saved or restored. For example, compute work can be preempted at a submission, dispatch, thread group or at a shader instruction boundary. The preemption latency and amount of saved or restored states will vary based on the preemption granularity.

- **Wave limiting**
  This method reduces the workload from other queues by limiting the number of waves that can be issued. "Wave" represents a group of shader threads.

- **Pipe priority**
  Connected queues on each pipe asserts a pipe priority to the shader HW. The shader HW uses this priority to select and launch upcoming work based on pipe priority.

- **Dispatch tunneling**
  The method immediately disables the work from other queues when a dispatch from a high-priority queue is executed. The ability to tunnel dispatches is configured as a queue-property.

- **Queue quantum**
  Quantum is implemented by both queue manager hardware and scheduler firmware. The queue manager connects and disconnects queues based on the quantum configured in the hardware queue by the scheduler firmware.
  During queue oversubscription, the scheduler firmware un-maps the queue once its quantum has expired to allow mapping of other unmapped user queues on the hardware queues.

- **Queue connection priority**
  The queue connection priority is specified for each hardware queue and is used by the queue manager hardware to select the next hardware queue that will be connected to the pipe.

- **Compute unit reservation**
  This method allows a certain number of compute units to be carved out and only made available for a particular queue. This method is used in scenarios where the machine utilization launch latency is critical.

The scheduler uses a combination of the described methods to achieve the desired prioritization in the presence of workload from queues with different priorities.

The following table lists how various methods are employed in different scenarios:

| Ready work to run | Expected scheduling behavior | How scheduler achieves it |
|---|---|---|
| 1. Real time compute queue<br>2. Focus gfx queue<br>3. Normal priority compute queues<br>4. Idle Compute queue | 1. *Real time* priority queue runs without any delays<br>2. Once *Real time* queue is idle, *Focus* queue will start to execute.<br>3. Once *Focus* queue has executed for a configured amount of time, the Normal queue will execute for a certain period of time.<br>4. Once all *Real time*, *Focus* and *Normal* queues have nothing else to execute, only then the Idle queue will execute | **Real time prioritization**<br>1. Real time queue once created stays mapped(max 4 RT queues allowed i.e. max 1 RT queue/pipe)<br>2. A certain # of Compute units are reserved for the Real time queue. Certain Real time queues will use Wave limiting instead of Compute unit reservation to quickly get their work to execute.<br>3. Highest queue connection priority<br>4. Highest shader type priority<br>**Focus and Normal prioritization**<br>1. Focus queue is mapped as the same connection priority as Normal queue.<br>2. Focus queue has a larger quantum relative to the Normal queue.<br>3. Focus queues have higher pipe priority.<br>4. Scheduler firmware may also unmap Normal queues on other pipes when they have long running shaders that prevent the Focus work from being able to launch on the compute units.<br>5. Normal queues get preempted with a higher level of preemption than the Focus queues.<br>**Idle prioritization**<br>Executes when all queue in the higher priority levels have been idle for some time. |

# MES API

This section describes MES API usage. The kernel mode driver (KMD) communicates with the Micro Engine Scheduler (MES) firmware by submitting API commands to the MES queue ring buffer.

- Some API's fields are for debug purposes which are not enabled by default. These fields have **Debug Only** in their descriptions

## MES API format

- MES scheduler APIs are defined in mes_api_def.h
- Each API has length 64 DWORDS as defined in `enum {API_FRAME_SIZE_IN_DWORDS = 64}`

The following format is applicable to all APIs:

```
union MESAPI__APINAME
{
    struct
    {
        union MES_API_HEADER      header;
        //API specific info

        struct MES_API_STATUS     api_status;

        uint64_t                  timestamp;
    };

    uint32_t max_dwords_in_api[API_FRAME_SIZE_IN_DWORDS];
};
```

Each API contains its specific information and three common fields: header, timestamp and api_status:

```
union MES_API_HEADER

{

    struct

    {

        uint32_t type     : 4;    /* 0 - Invalid; 1 - Scheduling; 2-15 - Reserved*/

        uint32_t opcode   : 8;  /* API command defined in MES_SCH_API_OPCODE enum */

        uint32_t dwsize   : 8;  /* Size in DWORD of the API command including header */

        uint32_t reserved : 12;

    };

    uint32_t u32All;

};
```

Opcode defines all supported MES APIs:

```
enum MES_SCH_API_OPCODE

{

    MES_SCH_API_SET_HW_RSRC = 0,

    MES_SCH_API_SET_SCHEDULING_CONFIG = 1,

    MES_SCH_API_ADD_QUEUE = 2,

    MES_SCH_API_REMOVE_QUEUE = 3,

    MES_SCH_API_PERFORM_YIELD = 4,

    MES_SCH_API_SET_GANG_PRIORITY_LEVEL = 5,

    MES_SCH_API_SUSPEND = 6,

    MES_SCH_API_RESUME = 7,

    MES_SCH_API_RESET = 8,

    MES_SCH_API_SET_LOG_BUFFER = 9,

    MES_SCH_API_CHANGE_GANG_PRORITY = 10,

    MES_SCH_API_QUERY_SCHEDULER_STATUS = 11,

    MES_SCH_API_PROGRAM_GDS = 12,

    MES_SCH_API_SET_DEBUG_VMID = 13,

    MES_SCH_API_MISC = 14,

    MES_SCH_API_UPDATE_ROOT_PAGE_TABLE = 15,

    MES_SCH_API_AMD_LOG = 16,

    MES_SCH_API_SET_SE_MODE = 17,

    MES_SCH_API_SET_GANG_SUBMIT = 18,

    MES_SCH_API_MAX = 0xFF

};
```

The api_status in each API command contains fence address and fence value that the KMD inserts. MES firmware writes the fence value to the given address to notify the KMD that the API has been processed by scheduler.

```
struct MES_API_STATUS

{

    uint64_t api_completion_fence_addr;

    uint64_t api_completion_fence_value;

};
```

## MES_SCH_API_SET_HW_RSRC

This is the first API that KMD submits to MES during initialization.

It provides list of hardware resources (hardware queues, virtual memory ID (VMID), etc.) to be managed by the scheduler and configuration flags (OS dependent features, workaround, etc.).

```
enum { MAX_COMPUTE_PIPES = 8 };
enum { MAX_GFX_PIPES     = 2 };
enum { MAX_SDMA_PIPES    = 2 };

enum MES_AMD_PRIORITY_LEVEL
{
    AMD_PRIORITY_LEVEL_LOW       = 0,
    AMD_PRIORITY_LEVEL_NORMAL    = 1,
    AMD_PRIORITY_LEVEL_MEDIUM    = 2,
    AMD_PRIORITY_LEVEL_HIGH      = 3,
    AMD_PRIORITY_LEVEL_REALTIME  = 4,


    AMD_PRIORITY_NUM_LEVELS
};

union MESAPI_SET_HW_RESOURCES
{
    struct
    {
        union MES_API_HEADER   header;
        uint32_t                           vmid_mask_mmhub;
        uint32_t                           vmid_mask_gfxhub;
        uint32_t                           gds_size;
        uint32_t                           paging_vmid;
        uint32_t                           compute_hqd_mask[MAX_COMPUTE_PIPES];
        uint32_t                           gfx_hqd_mask[MAX_GFX_PIPES];
        uint32_t                           sdma_hqd_mask[MAX_SDMA_PIPES];
        uint32_t                           aggregated_doorbells[AMD_PRIORITY_NUM_LEVELS];
        uint64_t                           g_sch_ctx_gpu_mc_ptr;
        uint64_t                           query_status_fence_gpu_mc_ptr;
        uint32_t                           gc_base[MES_MAX_HWIP_SEGMENT];
```

```
        uint32_t                           mmhub_base[MES_MAX_HWIP_SEGMENT];
        uint32_t                           osssys_base[MES_MAX_HWIP_SEGMENT];
        struct MES_API_STATUS    api_status;
        union
        {
            struct
            {
                uint32_t disable_reset : 1;
                uint32_t use_different_vmid_compute : 1;
                uint32_t disable_mes_log   : 1;
                uint32_t apply_mmhub_pgvm_invalidate_ack_loss_wa : 1;
                uint32_t apply_grbm_remote_register_dummy_read_wa : 1;
                uint32_t second_gfx_pipe_enabled : 1;
                uint32_t enable_level_process_quantum_check : 1;
                uint32_t legacy_sch_mode : 1;
                uint32_t disable_add_queue_wptr_mc_addr : 1;
                uint32_t enable_mes_event_int_logging : 1;
                uint32_t enable_reg_active_poll : 1;
                uint32_t use_disable_queue_in_legacy_uq_preemption : 1;
                uint32_t send_write_data : 1;
                uint32_t os_tdr_timeout_override : 1;
                uint32_t use_rs64mem_for_proc_gang_ctx : 1;
                uint32_t use_add_queue_unmap_flag_addr : 1;
                uint32_t enable_mes_sch_stb_log : 1;
                uint32_t reserved : 15;
            };
            uint32_t uint32_all;
        };
        uint32_t                            oversubscription_timer;
        uint64_t                            doorbell_info;
        uint64_t                            event_intr_history_gpu_mc_ptr;
        uint64_t                            timestamp;
        uint32_t                            os_tdr_timeout_in_sec;
    };
    uint32_t max_dwords_in_api[API_FRAME_SIZE_IN_DWORDS];
};
```

- vmid_mask_gfxhub – Bit mask of VMIDs in GC hub that are available for scheduler to manage. Each bit position indicates the availability of the corresponding VMID, e.g., 0x6 means VMID 1 and 2 are available

- vmid_mask_mmhub – Obsolete
- gds_size – Size of the global data storage (GDS) on the chip
- paging_vmid – VMID that driver assigns to paging process (excluded from vmid_mask_gfxhub)
- compute_hqd_mask – Per pipe bit mask of compute hardware queue descriptors (HQD) that are managed by scheduler. Each bit position indicates the availability of corresponding compute HQD on the particular pipe, e.g., 0x3 means compute HQD 0 and 1 of the pipe are available
- gfx_hqd_mask - Per pipe bit mask of graphics (GFX) HQDs that are managed by scheduler. Each bit position indicates the availability of corresponding GFX HQD on the particular pipe, e.g., 0x3 means GFX queue 0 and 1 of the pipe are available
- sdma_hqd_mask – Per pipe bit mask of SDMA HQDs that are managed by scheduler. Each bit position indicates the availability of corresponding SDMA HQD on the particular pipe, e.g., 0x3 means SDMA queue 0 and 1 of the pipe are available
- aggregated_doorbells – Offsets of aggregated doorbells. Value of 0XFFFFFFFF indicates invalid offset
- g_sch_ctx_gpu_mc_ptr – Obsolete
- query_status_fence_gpu_mc_ptr – MC address of query_status packet fence memory.
- gc_base – HWIP base for GC block
- mmhub_base – HWIP base for MM block
- ossys_base – HWIP base for OSSYS block
- oversubscription_timer – Duration in micro-second of timer when oversubscription happens. Scheduler wakes up to check if any unmapped queue has new work when timer is up
- doorbell_info – Debug only. Memory to hold aggregated doorbell counter
- event_intr_history_gpu_mc_ptr – Debug only. MC address to hold MES event/interrupt/API history log
- os_tdr_timeout_in_sec – Unmap timeout value in seconds. The driver is able to use this to overwrite the default unmap time out value of 2 seconds. Only valid when os_tdr_timeout_override is set

## *Flags*

- disable_reset – Disable MES automatic hang detection
- use_different_vmid_compute – Scheduler assigns different VMIDs for GFX and compute of the same process
- disable_mes_log– Disables MSFT GPU hardware scheduling log
- apply_mmhub_pgvm_invalidate_ack_loss_wa – Obsolete
- apply_grbm_remote_register_dummy_read_wa – Obsolete
- second_gfx_pipe_enabled – Enables 2nd GFX pipe
- enable_level_process_quantum_check – Enable an optimization that jumps out of the scheduling loop to handle an API event
- legacy_sch_mode – Set to 1 on the older OSes that do not understand or support the GPU hardware scheduling.
- disable_add_queue_wptr_mc_addr – If set to 1, the scheduler uses part of memory queue descriptor (MQD) memory for wptr poll memory. Otherwise, scheduler use the address passed in ADD_QUEUE API (see MES_SCH_API_ADD_QUEUE for details)
- enable_mes_event_int_logging – Debug only. Enables MES internal event/interrupt/API logging
- enable_reg_active_poll – Controls how the scheduler polls queue's active bit. 1: poll HQD register; 0: poll MQD memory
- use_disable_queue_in_legacy_uq_preemption – Set to 1 to allow the scheduler to use disable_queue bit in MQD for OS preemption
- send_write_data – Set to 1 for the scheduler to send a write_date packet to write a fence following each KIQ packet
- os_tdr_timeout_override – Enables unmap timeout overwrite
- use_rs64mem_for_proc_gang_ctx – Enables scheduler optimization that puts the process context and gang context into the MES scheduler local memory
- use_add_queue_unmap_flag_addr – If set to 1, the scheduler uses MC address passed in MES_SCH_API_ADD_QUEUE for queue unmap status. Else, scheduler will use the MQD memory
- enable_mes_sch_stb_log – Enables MES to log into Smart Trace Buffer

# MES_SCH_API_ADD_QUEUE

The KMD uses this API to add a use queue into the scheduler's internal structure to schedule it on GPU hardware.

```
union MESAPI__ADD_QUEUE
{
    struct
    {
        union MES_API_HEADER         header;
        uint32_t                     process_id;
        uint64_t                     page_table_base_addr;
        uint64_t                     process_va_start;
        uint64_t                     process_va_end;
        uint64_t                     process_quantum;
        uint64_t                     process_context_addr;
        uint64_t                     gang_quantum;
        uint64_t                     gang_context_addr;
        uint32_t                     inprocess_gang_priority;
        enum MES_AMD_PRIORITY_LEVEL gang_global_priority_level;
        uint32_t                     doorbell_offset;
        uint64_t                     mqd_addr;
        uint64_t                     wptr_addr;  //From MES_API_VERSION 2, mc addr is
expected for wptr_addr
        uint64_t                     h_context;
        uint64_t                     h_queue;
        enum MES_QUEUE_TYPE          queue_type;
        uint32_t                     gds_base;
        uint32_t                     gds_size;
        uint32_t                     gws_base;
        uint32_t                     gws_size;
        uint32_t                     oa_mask;
        uint64_t                     trap_handler_addr;
        uint32_t                     vm_context_cntl;
        struct
        {
            uint32_t paging      : 1;
            uint32_t debug_vmid  : 4;
            uint32_t program_gds : 1;
            uint32_t is_gang_suspended : 1;
            uint32_t is_tmz_queue : 1;
```

```
            uint32_t map_kiq_utility_queue : 1;
            uint32_t is_kfd_process : 1;
            uint32_t trap_en : 1;
            uint32_t is_aql_queue : 1;
            uint32_t skip_process_ctx_clear : 1;
            uint32_t map_legacy_kq : 1;
            uint32_t exclusively_scheduled : 1;
            uint32_t is_long_running : 1;
            uint32_t is_dwm_queue : 1;
            uint32_t is_video_blit_queue : 1;
            uint32_t reserved    : 14;
        };
        struct MES_API_STATUS       api_status;
        uint64_t                    tma_addr;
        uint32_t                    sch_id;
        uint64_t                    timestamp;
        uint32_t                    process_context_array_index;
        uint32_t                    gang_context_array_index;
        uint32_t                    pipe_id;    //used for mapping legacy kernel queue
        uint32_t                    queue_id;
        uint32_t                    alignment_mode_setting;
        uint64_t                    unmap_flag_addr; //Used for letting driver know queue
is unmapped, mc addr is expected
    };
    uint32_t max_dwords_in_api[API_FRAME_SIZE_IN_DWORDS];
};
```

- process_id – Process ID that appears in the IH Cookie as pasid. The KMD assigns unique process ID to each process

- page_table_base_addr – Page table base address of the process, and is programmed in VM_CONTEXTx_PAGE_TABLE_BASE_LO/HI registers

- process_va_start – Starting VA that's covered by the process's page table. Programmed in VM_CONTEXTx_PAGE_TABLE_START_LO/HI

- process_va_end – End VA that's covered by the process's page table. Programmed in VM_CONTEXTx_PAGE_TABLE_END_LO/HI

- process_quantum – Measured in 100ns units. Indicates the minimum time a process is allowed to run on the GPU

- process_context_addr – The memory where process specific information is saved. The scheduler owns the format of content saved in this memory. The size of the process context is defined in mes_api_def.h
- gang_quantum – Measured in 100ns units. Indicates the minimum amount of time a gang runs on the GPU
- gang_context_addr – memory where gang specific information is saved. Scheduler owns the format of content saved in this memory. The size of this memory is defined in the mes_api_def.h
- inprocess_gang_priority – The priority number assigned to the gang relative to other gangs within the same process
- gang_global_priority_level – The global priority level assigned to the gang. All queues within a gang share this priority level
- doorbell_offset – The doorbell offset (DWORD offset, i.e bits[27:2]) assigned to the queue
- mqd_addr – The MC address of queue's MQD memory
- wptr_addr – If MES_SCH_API_SET_HW_RSRC.disable_add_queue_wptr_mc_addr is set, GPUVA of wptr poll memory. Else, it’s the MC address of wptr poll memory
- h_context – OS handle of the context
- h_queue – OS handle of the queue
- queue_type – GFX/compute/SDMA
- gds_base/size – GDS base/size
- gws_base/size – GWS base/size
- oa_mask – OA mask
- trap_handler_addr – CWSR trap handler GPU VA
- tma_addr – CWSR TMA GPU VA
- vm_context_cntl – Programmed in VM_CONTEXTx_CNTL
- sch_id – The scheduler ID of the engine node belonging to the queue
- timestamp – The CPU time stamp of when driver submits this packet to the ring. Used for debugging only.
- process_context_array_index – The index of the process context array in scheduler's

local memory; valid only when MES_SCH_API_SET_HW_RSRC.use_rs64mem_for_proc_gang_ctx is **True**

- gang_context_array_index – The index of the gang context array in scheduler's local memory; valid only when MES_SCH_API_SET_HW_RSRC.use_rs64mem_for_proc_gang_ctx is **True**
- pipe_id – Used to map a kernel queue; the Pipe ID of the kernel queue
- queue_id – Used to map a kernel queue; the Queue ID of the kernel queue
- alignment_mode_setting – The shader alignment mode to be programmed in SH_MEM_CONFIG
- unmap_flag_addr – The MC address for queue unmap status memory. Only valid when MES_SCH_API_SET_HW_RSRC. use_add_queue_unmap_flag_addr is set

## *Flags*

- paging – The queue belonging to the paging process
- debug_vmid – Process requires the debug vmid (used by RGP (Radeon GFX Profiling) tool
- program_gds – Process uses GDS
- is_gang_suspended – A queue's context in suspended state to prevent scheduling of a queue
- is_tmz_queue – Obsolete
- map_kiq_utility_queue – Obsolete
- is_kfd_process – Queue belonging to the KFD process
- trap_en – Enables trap for shader debugger
- is_aql_queue – The AQL queue
- map_legacy_kq – The kernel queue
- exclusively_scheduled – Supports cooperative launch
- is_long_running – Indicates that the queue has a long running compute job
- is_dwm_queue – Indicates that the queue belongs to the DWM process

- is_video_blit_queue - Indicates the queue is a video blit queue

## MES_SCH_API_AMD_LOG

Copy MES_SCH_CONTEXT to AMGLOG specified memory location for TDR analysis.

```
union MESAPI_AMD_LOG
{
    struct
    {
        union MES_API_HEADER        header;
        uint64_t                    p_buffer_memory;
        uint64_t                    p_buffer_size_used;
        struct MES_API_STATUS       api_status;
        uint64_t                    timestamp;
    };
    uint32_t max_dwords_in_api[API_FRAME_SIZE_IN_DWORDS];
};
```

- p_buffer_memory - Pointer to amdlog buffer
- p_buffer_size_used - Not used, buffer size is equal to sizeof(struct MES_SCH_CONTEXT)

## MES_SCH_API_REMOVE_QUEUE

The KMD uses this API to remove a user queue from the scheduler's internal structure.

If the queue being removed is the last queue in the gang, all information related to the gang is removed from the scheduler context.

If the removed gang is the last in the process, the process information is removed from the scheduler context.

```
union MESAPI__REMOVE_QUEUE
{
    struct
    {
        union MES_API_HEADER        header;
        uint32_t                    doorbell_offset;
        uint64_t                    gang_context_addr;
        struct
        {
```

```
            uint32_t reserved01                : 1;
            uint32_t unmap_kiq_utility_queue   : 1;
            uint32_t preempt_legacy_gfx_queue  : 1;
            uint32_t unmap_legacy_queue        : 1;
            uint32_t reserved                  : 28;
        };
        struct MES_API_STATUS       api_status;
        uint32_t                    pipe_id;
        uint32_t                    queue_id;
        uint64_t                    tf_addr;
        uint32_t                    tf_data;
        enum MES_QUEUE_TYPE         queue_type;
        uint64_t                    timestamp;
        uint32_t                    gang_context_array_index;
    };
    uint32_t max_dwords_in_api[API_FRAME_SIZE_IN_DWORDS];
};
```

- doorbell_offset – Doorbell offset [DWORD offset, bits [27:2]] of the queue to be removed
- gang_context_addr – The gang's context address that maintains the info of all queues belonging to that gang
- pipe/queue_id – Used to remove a kernel queue (i.e., queues are managed by KMD); pipe ID/queue ID of the kernel queue being removed
- tf_addr/data – Trailing fence address and value for OS preemption
- queue_type – Gfx/compute/SDMA
- gang_context_array_index – Index of the gang context array in scheduler's local memory; valid only when MES_SCH_API_SET_HW_RSRC.use_rs64mem_for_proc_gang_ctx is true

### *Flags*

- unmap_kiq_utility_queue – Obsolete
- preempt_legacy_gfx_queue – Indicates that this is for OS preemption
- unmap_legacy_queue – Indicates that this is for kernel queue

## MES_SCH_API_SET_SCHEDULING_CONFIG

Corresponds to Windows DDI DxgkDdiSetProrityBands.

Sets up process quantum and other related information during bootup for each priority band. The MES scheduler uses this information for scheduling decisions.

```
union MESAPI__SET_SCHEDULING_CONFIG
{
    struct
    {
        union MES_API_HEADER        header;
        uint64_t                    grace_period_other_levels[AMD_PRIORITY_NUM_LEVELS];
        /* Default quantum for scheduling across processes within a priority band. */
        uint64_t                    process_quantum_for_level[AMD_PRIORITY_NUM_LEVELS];

        /* Default grace period for processes that preempt each other within a priority
band.*/
        uint64_t
process_grace_period_same_level[AMD_PRIORITY_NUM_LEVELS];

        /* For normal level this field specifies the target GPU percentage in situations
when it's starved by the high level.
           Valid values are between 0 and 50, with the default being 10.*/
        uint32_t                    normal_yield_percent;

        struct MES_API_STATUS       api_status;
        uint64_t                    timestamp;
    };

    uint32_t max_dwords_in_api[API_FRAME_SIZE_IN_DWORDS];
};
```

- grace_period_other_levels - Grace period when preempting another priority band for this priority band. The value for idle priority band is ignored, as it never preempts other bands
- process_quantum_for_level - Default quantum for scheduling across processes within a priority band
- process_grace_period_same_level - Default grace period for processes that preempt each other within a priority band
- normal_yield_percent - For normal level this field specifies the target GPU percentage in situations when it's starved by the high level. Valid values are between 0 and 50, with the default being 10

**Note**: In current fw, only relevant quantum is process_quantum_for_level, other fields are not used in scheduling/

## MES_SCH_API_PERFORM_YIELD

This API is not currently supported.

## MES_SCH_API_SET_GANG_PRIORITY_LEVEL

This API is not currently supported.

## MES_SCH_API_SUSPEND

When MES_SCH_API_SET_HW_RSRC.legacy_sch_mode is set, the KMD uses this API to suspend a single queue to prevent it from being scheduled for a single engine in Windows OS preemption.

(Used in the following DDIs in Windows: DxgkDdiSuspendContext, DxgkDdiPreemptCommand.)

```
union MESAPI__SUSPEND
{
    struct
    {
        union MES_API_HEADER        header;
        /* false – suspend all gangs; true – specific gang */
        struct
        {
            uint32_t                  suspend_all_gangs : 1;
            uint32_t                  reserved : 31;
        };
        /* gang_context_addr is valid only if suspend_all = false */

        uint64_t gang_context_addr;

        uint64_t                      suspend_fence_addr;
        uint32_t                      suspend_fence_value;

        struct MES_API_STATUS         api_status;

        union
        {
            uint32_t return_value; // to be removed
            uint32_t sch_id;       //keep the old return_value temporarily for
compatibility
        };
        uint32_t                    doorbell_offset;
        uint64_t                    timestamp;
        enum MES_QUEUE_TYPE         legacy_uq_type;
        enum MES_AMD_PRIORITY_LEVEL legacy_uq_priority_level;
        uint32_t                    gang_context_array_index;
```

```
    };

    uint32_t max_dwords_in_api[API_FRAME_SIZE_IN_DWORDS];
};
```

- gang_context_addr - Gang context address for target queue to be suspended
- suspend_fence_addr – MC address for suspend completion fence
- suspend_fence_value - Suspend fence ID
- doorbell_offset - Doorbell offset for target queue to be suspended. Only used if no flag is set
- gang_context_array_index - Gang context array index for target queue to be suspended. Valid only when MES_SCH_API_SET_HW_RSRC.use_rs64mem_for_proc_gang_ctx is set

The following fields are only valid for Windows OS preemption.

- return_value - Obsolete
- sch_id –Scheduler ID for target engine to be suspended
- legacy_uq_type – Queue type for target engine to be suspended (GFX/compute/SDMA)
- legacy_uq_priority_level – Priority level to be suspended

### *Flags*

- suspend_all_gangs – Not currently supported

## MES_SCH_API_RESUME

The KMD uses this API to resume a single queue suspended by MES_SCH_API_SUSPEND, or resume scheduling after reset.

(Used in the following DDIs in Windows OS: DxgkDdiResumeContext, DxgkDdiResumeHwEngine.)

```
union MESAPI__RESUME
{
    struct
    {
        union MES_API_HEADER        header;
        /* false - resume all gangs; true - specified gang */
        struct
        {
            uint32_t                resume_all_gangs : 1;
            uint32_t                reserved : 31;
        };
```

```
        /* valid only if resume_all_gangs = false */
        uint64_t                    gang_context_addr;

        struct MES_API_STATUS       api_status;
        uint32_t                    doorbell_offset;
        uint64_t                    timestamp;
        uint32_t                    gang_context_array_index;
    };

    uint32_t max_dwords_in_api[API_FRAME_SIZE_IN_DWORDS];
};
```

- gang_context_addr - Gang context address for target queue to be resumed. Valid only if resume_all_gangs = 0
- gang_context_array_index – Gang context array index for target queue to be resumed. Valid only if resume_all_gangs = 0 and MES_SCH_API_SET_HW_RSRC.use_rs64mem_for_proc_gang_ctx = 1

### *Flags*

- resume_all_gangs - Resume all scheduling. Meant to be called after an engine reset

## MES_SCH_API_RESET

In Windows, the KMD uses this API for hang detection and reset. The MES scheduler returns a list of doorbell offsets of hung queues. If the list is empty, no hangs are detected.

Used in the following Windows DDIs; DxgkDdiQueryEngineStatus, DxgkDdiResetEngine, DxgkDdiResetHwEngine

The KMD can also use this API to reset kernel queues by setting reset_legacy_gfx flag.

```
union MESAPI__RESET
{
    struct
    {
        union MES_API_HEADER        header;
        struct
        {
            uint32_t                reset_queue_only : 1; // Only reset the queue given
by doorbell_offset (not entire gang)
            uint32_t                hang_detect_then_reset : 1; // Hang detection first
then reset any queues that are hung
            uint32_t                hang_detect_only : 1; // Only do hang detection (no
reset)
```

```
            uint32_t                reset_legacy_gfx : 1; // Reset HP and LP kernel
queues not managed by MES
            uint32_t                use_connected_queue_index : 1; // Fallback to use
conneceted queue index when CP_CNTX_STAT method fails (gfx pipe 0)
            uint32_t                use_connected_queue_index_p1 : 1; // For gfx pipe 1
            uint32_t                reserved : 26;
        };
        uint64_t                    gang_context_addr;
        /* valid only if reset_queue_only = true */
        uint32_t                    doorbell_offset;
        /* valid only if hang_detect_then_reset = true */
        uint64_t                    doorbell_offset_addr;
        enum MES_QUEUE_TYPE         queue_type;
        //valid only if reset_legacy_gfx = true
        uint32_t pipe_id_lp;
        uint32_t queue_id_lp;
        uint32_t vmid_id_lp;
        uint64_t mqd_mc_addr_lp;
        uint32_t doorbell_offset_lp;
        uint64_t wptr_addr_lp;
        uint32_t pipe_id_hp;
        uint32_t queue_id_hp;
        uint32_t vmid_id_hp;
        uint64_t mqd_mc_addr_hp;
        uint32_t doorbell_offset_hp;
        uint64_t wptr_addr_hp;
        struct MES_API_STATUS       api_status;
        uint32_t                    active_vmids;
        uint64_t                    timestamp;
        uint32_t                    gang_context_array_index;

        uint32_t                    connected_queue_index;
        uint32_t                    connected_queue_index_p1;
    };
    uint32_t max_dwords_in_api[API_FRAME_SIZE_IN_DWORDS];
};
```

- gang_context_addr - Obsolete
- doorbell_offset – Doorbell offset of the queue. Only valid when reset_queue_only = 1

- doorbell_offset_addr – MC address of memory that holds doorbell offset array. MES scheduler populates this array with offsets for queues that are hung
- queue_type - Indicates which engine MES should reset/hang detect (GFX/compute/SDMA)
- active_vmids - Workaround to indicate which VMIDs are currently active for CP_CNTX_STAT hang detect method
- gang_context_array_index - Obsolete
- connected_queue_index - Workaround to indicate which queue is currently connected on GFX Pipe 0. Valid only when use_connected_queue_index = 1
- connected_queue_index_p1 - Workaround to indicate which queue is currently connected on GFX Pipe 1. Valid only when use_connected_queue_index_p1 = 1

The following fields are only valid when reset_legacy_gfx is set and are used in Windows:

- pipe_id_lp - Pipe ID for low priority GFX Kernel queue
- queue_id_lp - Queue ID for low priority GFX Kernel queue
- vmid_id_lp - VMID for low priority GFX Kernel queue
- mqd_mc_addr_lp - MQD MC address for low priority GFX Kernel queue
- doorbell_offset_lp - Doorbell offset for low priority GFX Kernel queue
- wptr_addr_lp - Write pointer poll memory address for low priority GFX Kernel queue
- pipe_id_hp - Pipe ID for high priority GFX Kernel queue
- queue_id_hp - Queue ID for high priority GFX Kernel queue
- vmid_id_hp - VMID for high priority GFX Kernel queue
- mqd_mc_addr_hp - MQD MC address for high priority GFX Kernel queue
- doorbell_offset_hp - Doorbell offset for high priority GFX Kernel queue
- wptr_addr_hp - Write pointer poll memory address for high priority GFX Kernel queue

### *Flags*

- reset_queue_only - Reset single queue with no hang detection
- hang_detect_then_reset - Performs hang detection, and reset all hung queues. Return doorbell offsets of all hung queues
- hang_detect_only - Perform hang detection only. Returns doorbell offsets of all hung queues
- reset_legacy_gfx – Resets legacy GFX queue
- No flag set - Obsolete. The driver is expected to set one of the above flags

## MES_SCH_API_SET_LOG_BUFFER

The KMD uses this API to save log buffer information passed from Windows OS DDI DxgkDdiSetSchedulingLogBuffer.

```
union MESAPI__SET_LOGGING_BUFFER

{

    struct

    {

        union MES_API_HEADER        header;

        /* There are separate log buffers for each queue type */

        enum MES_QUEUE_TYPE         log_type;

        /* Log buffer GPU Address */

        uint64_t                    logging_buffer_addr;

        /* number of entries in the log buffer */

        uint32_t                    number_of_entries;

        /* Entry index at which CPU interrupt needs to be signalled */

        uint32_t                    interrupt_entry;


        struct MES_API_STATUS       api_status;

        uint64_t                    timestamp;

        uint32_t                    vmid;

    };


    uint32_t max_dwords_in_api[API_FRAME_SIZE_IN_DWORDS];

};
```

- log_type - Target engine type for this log buffer update (each engine has its own log buffer)
- logging_buffer_addr – GPU virtual address of log buffer
- number_of_entries - Log buffer size
- interrupt_entry - When number of entries logged in the log buffer reaches this log entry index, it raises an interrupt to KMD/OS. The interrupt is meant to give OS advanced warning of when the existing log buffer is going to be filled up so that it can allocate a new log buffer

## MES_SCH_API_CHANGE_GANG_PRORITY

In the Windows use-case, this API corresponds to DDI DxgkDDiSetContextSchedulingProperties. The Windows OS changes user queue quantum to reflect changes in the owning process's status. For example, when a user’s mouse focus changes from one process to another.

```
union MESAPI__CHANGE_GANG_PRIORITY_LEVEL
{
    struct
    {
        union MES_API_HEADER        header;
        uint32_t                    inprocess_gang_priority;
        enum MES_AMD_PRIORITY_LEVEL gang_global_priority_level;
        uint64_t                    gang_quantum;
        uint64_t                    gang_context_addr;
        struct MES_API_STATUS       api_status;
        uint32_t                    doorbell_offset;
        uint64_t                    timestamp;
        uint32_t                    gang_context_array_index;
        struct
        {
            uint32_t                queue_quantum_scale         : 2;
            uint32_t                queue_quantum_duration      : 8;
            uint32_t                apply_quantum_all_processes : 1;
            uint32_t                reserved                    : 21;
        };
    };

    uint32_t max_dwords_in_api[API_FRAME_SIZE_IN_DWORDS];
};
```

- inprocess_gang_priority - Gang priority within a process, not used in current FW
- gang_global_priority_level - Overall gang priority level, lower priority gangs tend to get preempted for high priority gangs during scheduling
- gang_quantum - Quantum provided by Windows OS, usually 2ms, queue is considered "expired" after its quantum runs out
- doorbell_offset – Obsolete
- gang_context_array_index – index of the gang context array in scheduler's local memory; valid only when MES_SCH_API_SET_HW_RSRC.use_rs64mem_for_proc_gang_ctx is set
- queue_quantum_scale – Used by Windows OS
- queue_quantum_duration – Used by Windows OS
- apply_quantum_all_processes – Used by Windows OS

## MES_SCH_API_QUERY_SCHEDULER_STATUS

The KMD uses this API to query status/info from MES firmware.

```
enum MES_API_QUERY_MES_OPCODE
{
    MES_API_QUERY_MES__GET_CTX_ARRAY_SIZE,
    MES_API_QUERY_MES__CHECK_HEALTHY,
    MES_API_QUERY_MES__MAX,
};

union MESAPI__QUERY_MES_STATUS
{
    struct
    {
        union MES_API_HEADER            header;
        enum MES_API_QUERY_MES_OPCODE   subopcode;
        struct MES_API_STATUS           api_status;
        uint64_t                        timestamp;
        union
        {
            struct MES_API_QUERY_MES__CTX_ARRAY_SIZE    ctx_array_size;
            struct MES_API_QUERY_MES__HEALTHY_CHECK     healthy_check;
            uint32_t data[QUERY_MES_MAX_SIZE_IN_DWORDS];
        };
    };

    uint32_t max_dwords_in_api[API_FRAME_SIZE_IN_DWORDS];
};
```

- subopcode - Changes functionality based on what MES_API_QUERY_MES_OPCODE is used

## MES_API_QUERY_MES__GET_CTX_ARRAY_SIZE

The KMD uses this to query MES internal structure size.

```
struct MES_API_QUERY_MES__CTX_ARRAY_SIZE
{
    uint64_t    proc_ctx_array_size_addr;
    uint64_t    gang_ctx_array_size_addr;
};
```

- proc_ctx_array_size_addr - Memory address where MES will write process context array size
- gang_ctx_array_size_addr - Memory address where MES will write gang context array size

## MES_API_QUERY_MES__HEALTHY_CHECK

The KMD uses this API to check if MES is running and responding.

```
struct MES_API_QUERY_MES__HEALTHY_CHECK
{
    uint64_t    healthy_addr;
};
```

- healthy_addr – Not used. Currently, MES firmware writes fence to the memory to notify KMD that MES is not hang

## MES_SCH_API_PROGRAM_GDS

The KMD uses this API to request MES for GDS programming for the target process. GDS registers are programmed when VMID is allocated. If VMID is already allocated, registers will be programmed before API returns.

```
union MESAPI__PROGRAM_GDS
{
    struct
    {
        union MES_API_HEADER           header;
        uint64_t                       process_context_addr;
        uint32_t                       gds_base;
        uint32_t                       gds_size;
        uint32_t                       gws_base;
        uint32_t                       gws_size;
        uint32_t                       oa_mask;
        struct MES_API_STATUS          api_status;
        uint64_t                       timestamp;
        uint32_t                       process_context_array_index;
    };

    uint32_t max_dwords_in_api[API_FRAME_SIZE_IN_DWORDS];
};
```

- process_context_addr - Memory where process specific information is saved. Scheduler owns the format of this memory. The size of the process context is defined in the mes_api_def.h

- gds_base - GDS base address. Programming for GDS_VMIDx_BASE register
- gds_size - GDS aperture size. Programming for GDS_VMIDx_SIZE register
- gws_base - GWS base. Programming for BASE field in GDS_GWS_VMIDx register
- gws_size - GWS size. Programming for SIZE field in GDS_GWS_VMIDx register
- oa_mask - Bit mask representing the alloc counters allocated VMID can use. Programming for GDS_OA_VMIDx register
- process_context_array_index - Processes context array index for target process. Valid only when MES_SCH_API_SET_HW_RSRC.use_rs64mem_for_proc_gang_ctx is set

## MES_SCH_API_SET_DEBUG_VMID

The KMD uses this API to set up the page table for a process that requests debug VMID for tools like Radeon GPU Profiler (RGP).

The user mode driver can request debug VMID, and KMD/MES will allocate a VMID for this process. The page table base registers for this allocated debug VMID will be programed to this process's page table base.

```
union MESAPI__SET_DEBUG_VMID
{
    struct
    {
        union MES_API_HEADER        header;
        struct MES_API_STATUS       api_status;
        union
        {
            struct
            {
                uint32_t use_gds   : 1;
                uint32_t operation : 2;
                uint32_t reserved  : 29;
            }flags;
            uint32_t u32All;
        };
        uint32_t                      reserved;
        uint32_t                      debug_vmid;
        uint64_t                      process_context_addr;
        uint64_t                      page_table_base_addr;
        uint64_t                      process_va_start;
        uint64_t                      process_va_end;
```

```
        uint32_t                      gds_base;
        uint32_t                      gds_size;
        uint32_t                      gws_base;
        uint32_t                      gws_size;
        uint32_t                      oa_mask;
        uint64_t                      output_addr; // output addr of the acquired vmid 
value
        uint64_t                      timestamp;
        uint32_t                      process_vm_cntl;
        enum MES_QUEUE_TYPE           queue_type;
        uint32_t                      process_context_array_index;
        uint32_t                      alignment_mode_setting;
    };

    uint32_t max_dwords_in_api[API_FRAME_SIZE_IN_DWORDS];
};
```

- debug_vmid - The VMID reserved as the debug VMID, used when operation flag = DEBUG_VMID_OP_RELEASE (2)
- process_context_addr – Memory where process specific context is saved. Scheduler owns the format of this memory. The size of the process context is defined in the mes_api.def.h, this is for the process that requests the debug VMID
- page_table_base_addr – page table base address of the process
- process_va_start - Starting address of the process's VA space
- process_va_end - Ending address of the process's VA space
- gds_base/size – GDS base/size
- gws_base/size – GWS base/size
- oa_mask – OA mask
- output_addr – MES scheduler writes the allocated debug VMID value to this address for driver to read. This is used when operation flag = DEBUG_VMID_OP_ALLOCATE (1)
- process_vm_cntl - Not used
- queue_type – gfx/compute/SDMA
- process_context_array_index – Index of the process context array in scheduler's local memory; valid only when MES_SCH_API_SET_HW_RSRC.use_rs64mem_for_proc_gang_ctx is set
- alignment_mode_setting – alignment mode setting to be programmed in SH_MEM_CONFIG

# MES_SCH_API_UPDATE_ROOT_PAGE_TABLE

The KMD uses this API to change page table base of a process.

```
union MESAPI__UPDATE_ROOT_PAGE_TABLE

{

    struct

    {

        union MES_API_HEADER        header;

        uint64_t                    page_table_base_addr;

        uint64_t                    process_context_addr;

        struct MES_API_STATUS       api_status;

        uint64_t                    timestamp;

        uint32_t                    process_context_array_index;

    };


    uint32_t max_dwords_in_api[API_FRAME_SIZE_IN_DWORDS];

};
```

- page_table_base_addr - Page table base address
- process_context_addr – Memory where process specific context is saved
- process_context_array_index – Index of the process context array in scheduler's local memory; valid only when MES_SCH_API_SET_HW_RSRC.use_rs64mem_for_proc_gang_ctx is set

# MES_SCH_API_SET_SE_MODE

The API allows the driver to turn off the second shader engine.

```
enum MES_SE_MODE
{
    MES_SE_MODE_INVALID = 0,
    MES_SE_MODE_SINGLE_SE = 1,
    MES_SE_MODE_DUAL_SE = 2,
    MES_SE_MODE_LOWER_POWER = 3,
};


union MESAPI__SET_SE_MODE

{

    struct

    {

        union MES_API_HEADER header;

        /* the new SE mode to apply*/

        MES_SE_MODE new_se_mode;

        /* the fence to make sure the ItCpgCtxtSync packet is completed */
```

```
        uint64_t cpg_ctxt_sync_fence_addr;
        uint32_t cpg_ctxt_sync_fence_value;

        /* log_seq_time - Scheduler logs the switch seq start/end ts in the IH cookies */
        union
        {
            struct
            {
                uint32_t log_seq_time : 1;
                uint32_t reserved : 31;
            };
            uint32_t uint32_all;
        };
        struct MES_API_STATUS api_status;
    };

    uint32_t max_dwords_in_api[API_FRAME_SIZE_IN_DWORDS];
};
```

- new_se_mode – New SE mode to be applied

## MES_SCH_API_SET_GANG_SUBMIT

The KMD uses this API to pair two queues together for the purpose of gang submission. MES scheduler will guarantee that the paired queues will always be mapped at the same time.

```
struct SET_GANG_SUBMIT
{
    uint64_t gang_context_addr;
    uint64_t slave_gang_context_addr;
    uint32_t gang_context_array_index;
    uint32_t slave_gang_context_array_index;
};

union MESAPI__SET_GANG_SUBMIT
{
    struct
    {
        union MES_API_HEADER    header;
        struct MES_API_STATUS   api_status;
        struct SET_GANG_SUBMIT  set_gang_submit;
    };
```

```

    uint32_t max_dwords_in_api[API_FRAME_SIZE_IN_DWORDS];
};
```

- gang_context_addr - Gang context address of master queue
- slave_gang_context_addr - Gang context address of slave queue
- gang_context_array_index - Gang context array index of master queue. Valid only when use_rs64mem_for_proc_gang_ctx is set in mes_sch_api_set_hw_rsrc
- slave_gang_context_array_index - Gang context array index of slave queue. Valid only when use_rs64mem_for_proc_gang_ctx is set in mes_sch_api_set_hw_rsrc

## MES_SCH_API_MISC

This API contains miscellaneous non-scheduling functionalities. Each functionality has a sub-opcode and corresponding structures.

```
union MESAPI__MISC
{
    struct
    {
        union MES_API_HEADER    header;
        enum MESAPI_MISC_OPCODE opcode;
        struct MES_API_STATUS   api_status;

        union
        {
            struct WRITE_REG write_reg;
            struct INV_GART inv_gart;
            struct QUERY_STATUS query_status;
            struct READ_REG read_reg;
            struct WAIT_REG_MEM wait_reg_mem;
            struct SET_SHADER_DEBUGGER set_shader_debugger;
            enum MES_AMD_PRIORITY_LEVEL queue_sch_level;
            uint32_t data[MISC_DATA_MAX_SIZE_IN_DWORDS];
        };
        uint64_t                timestamp;
        uint32_t                doorbell_offset;
        uint32_t                os_fence;
    };

    uint32_t max_dwords_in_api[API_FRAME_SIZE_IN_DWORDS];
```

```
};

enum MESAPI_MISC_OPCODE
{
    MESAPI_MISC__WRITE_REG,
    MESAPI_MISC__INV_GART,
    MESAPI_MISC__QUERY_STATUS,
    MESAPI_MISC__READ_REG,
    MESAPI_MISC__WAIT_REG_MEM,
    MESAPI_MISC__SET_SHADER_DEBUGGER,
    MESAPI_MISC__NOTIFY_WORK_ON_UNMAPPED_QUEUE,
    MESAPI_MISC__NOTIFY_TO_UNMAP_PROCESSES,

    MESAPI_MISC__MAX,
};
```

- opcode - Changes functionality based on what MESAPI_MISC_OPCODE is used. See each opcode's section for more details

## MESAPI_MISC__WRITE_REG

Perform register write on request of KMD.

```
struct WRITE_REG
{
    uint32_t                    reg_offset;
    uint32_t                    reg_value;
};
```

- reg_offset - Offset of the register
- reg_value - Value to be written to the register

## MESAPI_MISC__INV_GART

Perform GART invalidation.

```
struct INV_GART
{
    uint64_t                    inv_range_va_start;
    uint64_t                    inv_range_size;
};
```

- inv_range_va_start - starting VA for invalidation range

- inv_range_size - invalidation range size

**Note**: If inv_range_va_start = 0 or inv_range_size = 0, then MES scheduler invalidates entire range.

## MESAPI_MISC__QUERY_STATUS

The KMD uses this to trigger an interrupt from KIQ.

```
struct QUERY_STATUS
{
    uint32_t context_id;
};
```

- context_id - Value is copied to CONTEXT_ID in QueryStatus PM4 packet

## MESAPI_MISC__READ_REG

Perform register read on request of the KMD.

```
struct READ_REG
{
    uint32_t reg_offset;
    uint64_t buffer_addr;
    union
    {
        struct
        {
            uint32_t read64Bits : 1;
            uint32_t reserved : 31;
        }bits;
        uint32_t all;
    }option;
};
```

- reg_offset - Offset of the register
- buffer_addr – MC address to which MES scheduler writes the register value
- read64Bits - Control bit to enable 64-bit register read (0 = 32-bit, 1 = 64-bit)

## MESAPI_MISC__WAIT_REG_MEM

The KMD uses this API to request for the MES to wait on specific register values.

```
enum WRM_OPERATION
{
```

```
    WRM_OPERATION__WAIT_REG_MEM,
    WRM_OPERATION__WR_WAIT_WR_REG,

    WRM_OPERATION__MAX,
};

struct WAIT_REG_MEM
{
    enum WRM_OPERATION op;
    // only function = equal_to_the_reference_value and mem_space = register_space
supported for now
    uint32_t reference;
    uint32_t mask;
    uint32_t reg_offset1;
    uint32_t reg_offset2;
};
```

- op - WRM_OPERATION opcode
- WRM_OPERATION__WAIT_REG_MEM (0) - MES will tight loop on reg_offset1 until it equals reference value
- WRM_OPERATION__WR_WAIT_WR_REG (1) - MES will first write reference to reg_offset1, then it will poll reg_offset2 until it equals reference value
- reference - Reference value to poll (op = 0), or reference value to poll/write (op = 1).
- mask - Mask off comparison bits
- reg_offset1 - Register to poll (op = 0), or target register to write to (op = 1)
- reg_offset2 - Register to poll (op = 1)

## MESAPI_MISC__SET_SHADER_DEBUGGER

This API enables shader debugger register programming.

The MES also clears the process context if the process has not been added.

The shader debugger settings are saved to the process context.

Registers are programmed whenever a compute queue belonging to the process is mapped. Registers are restored to their default settings when process has no compute queues mapped.

```
struct SET_SHADER_DEBUGGER
{
    uint64_t process_context_addr;
    union
    {
```

```
        struct
        {
            uint32_t single_memop : 1;  // SQ_DEBUG.single_memop
            uint32_t single_alu_op : 1; // SQ_DEBUG.single_alu_op
            uint32_t reserved : 30;
        }flags;
        uint32_t u32All;
    };
    uint32_t spi_gdbg_per_vmid_cntl;
    uint32_t tcp_watch_cntl[4]; // TCP_WATCHx_CNTL
    uint32_t trap_en;
};
```

- single_memop - SINGLE_MEMOP setting in SQ_DEBUG register
- single_alu_op - SINGLE_ALU_OP setting in SQ_DEBUG register
- process_context_addr - Memory where process specific context is saved
- spi_gdbg_per_vmid_cntl - Setting for SPI_GDBG_PER_VMID_CNTL register
- tcp_watch_cntl[4] - Setting for TCP_WATCHx_CNTL registers
- trap_en - TRAP_EN setting in SQ_SHADER_TBA_HI register

## MESAPI_MISC__NOTIFY_WORK_ON_UNMAPPED_QUEUE

KMD uses this API as a workaround for aggregate doorbell. Meant to be called when an unmapped queue has a new submission. Notifies MES that target priority level has new work and MES will try to schedule queues of this level.

```
enum MES_AMD_PRIORITY_LEVEL queue_sch_level;
```

- queue_sch_level - Target priority level that has new work

## MESAPI_MISC__NOTIFY_TO_UNMAP_PROCESSES

The KMD uses this API to request the MES to unmap queues for all processes.

# Scheduler log

As described in previous sections, MES scheduler firmware interacts with kernel mode driver and CP block. Events between MES scheduler and KMD, MES scheduler and CP are of interests to understand system state when it comes to debugging MES issues.

To use MES log, KMD needs to allocate log buffer memory and passes GPU address of the log buffer memory to MES scheduler in API MES_SCH_API_SET_HW_RSRC.

MES log format is defined in the following structure.

```
struct MES_EVT_INTR_HIST_LOG

{

    struct MES_SCH_INTR_HIST_INFO   interrupt_history[MES_SCH_MAX_NUM_MES_INTR_ENTRY];

    struct MES_SCH_EVT_LOG_HIST_INFO
event_log_history[MES_SCH_MAX_NUM_MES_EVT_LOG_ENTRY];

    struct MES_SCH_API_HIST_INFO    api_history[MES_SCH_MAX_NUM_API_CALL_ENTRY];

    uint32                          interrupt_history_index;

    uint32                          event_log_history_index;

    uint32                          api_history_index;

};
```

It contains three arrays, api_history is for events from KMD to MES scheduler, event_log_history is for events from MES scheduler to CP and interrupt_history is for interrupt events from CP to MES scheduler. These arrays are updated in a circular buffer fashion and each array has an index which always points to the entry in the array that will be updated next.

## API history

Each entry in api_history array has the following format:

```
struct MES_SCH_API_HIST_INFO

{

    enum MES_SCH_API_CALL_ID api_id;

    uint64_t                time_before_call;

    uint64_t                time_after_call;

    uint32_t                error_code;

    struct

    {

        uint32 status : 1;

        uint32 reserved : 31;
```

```
    };
};
```

- **api_id** – indicates which API command of this entry
- **time_before_call** – GPU timestamp when MES scheduler starts processing this API command
- **timer_after_call** – GPU timestamp when MES scheduler finishes processing this API command
- **error_code** – error code for certain APIs if API processing encounters error. Error code is defined in mes_sch_context.h
- **status** – 1: API processing is successful; 0: otherwise

## Event log history

Each entry in event_log_hisotry array has the following format:

```
struct MES_SCH_EVT_LOG_HIST_INFO
{
    enum MES_SCH_EVT_LOG_ID event_log_id;
    uint32_t                doorbell_offset;
    uint64_t                time_before_call;
    uint64_t                time_after_call;
    struct
    {
        uint32 status : 1;
        uint32 queue_type : 2;
        uint32 reserved : 29;
    };
};

enum MES_SCH_EVT_LOG_ID
{
    MES_EVT_LOG_MAP_QUEUE = 0,
    MES_EVT_LOG_UNMAP_QUEUE = 1,
    MES_EVT_LOG_QUERY_STATUS = 2,
    MES_EVT_LOG_UNMAP_RESET_QUEUE = 3
};
```

- **event_log_id** – events that MES scheduler sends to CP; Defined in enum MES_SCH_EVT_LOG_ID
- **doorbell_offset** – doorbell offset of the queue for which the event is sent
- **time_before_call** – GPU timestamp at which scheduler sends event to CP
- **time_after_call** – GPU timestamp at which CP finishes processing the event
- **status** – if CP processing event is successful or not, 1: success, 0: otherwise
- **queue_type** – queue type (gfx/compute/sdma) of the queue for which the event is sent

## Interrupt history

Each entry in interrupt_history array has the following format:

```
struct MES_SCH_INTR_HIST_INFO
{
    enum MES_SCH_INTR_ID intr_id;
    uint64_t               time_trace;
    struct MES_SCH_INTR_CB_DATA intr_callback;
};
enum MES_SCH_INTR_ID
{
    MES_INTR_ME_0 = 0,
    MES_INTR_ME_1 = 1,
    MES_INTR_PACKET = 2,
    MES_INTR_TIMER = 3,
    MES_INTR_AGGREAGATE_DOORBELL = 4
};
```

- **intr_id** – interrupt ID defined in enum MES_SCH_INTR_ID
- **time_trace** – GPU timestamp at which MES scheduler receives the interrupt
- **intr_callback** – Interrupt call back data defined in struct MES_SCH_INTR_CB_DATA below

```
struct MES_SCH_INTR_CB_DATA
{
    union
    {
```

```
        struct
        {
            uint32_t                enc_inter : 5;
            uint32_t                intr_pipe_id : 2;
            uint32_t                intr_queue_id : 3;
            uint32_t                reserved1 : 1;
            uint32_t                action_id : 4;
            uint32_t                enc_inter_valid : 1;
            uint32_t                reserved2 : 12;
            uint32_t                vmid : 4;
        } inter_encode;
        uint32_t inter_enc;
    };
    union
    {
        struct
        {
            uint64_t                intr_data : 62;
            uint64_t                intr_pipe_id : 2;
        } inter_data_pipe;
        struct
        {
            uint64_t                doorbell_offset : 26;
            uint64_t                reserved3 : 6;
            uint64_t                data : 32;
        } fence;
        uint64_t inter_data;
        uint64_t inter_addr;
    };
};
```

## Example of log usage

When KMD reports MES API timeout error message, one may use MES log to understand the failure.

For example, one of the most common MES API timeout error is message 3 timeout. From enum MES_SCH_API_OPCODE defined in mes_api_def.h, 3 is MES_SCH_API_REMOVE_QUEUE. KMD issues this API to request MES scheduler to remove a user queue. There may be multiple reasons of this API failure. From MES log, one can find the most recent entry in api_history array which has api_id MES_API_REMOVE_QUEUE (3). Then, from the error_code (see below), one can check the reason of the error.

```
enum MES_SCH_API_REMOVEQUEUE_ERRCODE

{

    API_REMOVEQUEUE_NOERROR = 0,

    API_REMOVEQUEUE_UNMAP_FAIL = 1,

    API_REMOVEQUEUE_HQDQUEUE_MAP_MISMATCH = 2,

    API_REMOVEQUEUE_CLEANUP_FAIL = 3,

    API_REMOVEQUEUE_QUEUE_NOT_FOUND = 4,

    API_REMOVEQUEUE_NULL_GANG = 5,

};
```

If error_code is 1, it means when MES scheduler requests CP to unmap the queue, CP failed the unmap request. This usually means the queue being unmapped is in a hang state. As the next debugging step, one need to look for the reason why the queue is hang. In this scenario, in the most recent entry in event_log_history array with event_log_id MES_EVT_LOG_UNAMP_QUEUE, one would see the status bit is 0, which means unmap failure and doorbell_offset field tells which queue has triggered this error.

If error_code is not 1, it means error in either MES scheduler firmware or in driver. For example, 3 means when MES scheduler cleans up its internal structure, it encounters some issue; 5 means KMD has passed a null gang in the API command.

# Disclaimer

The information contained herein is for informational purposes only, and is subject to change without notice. This document may contain technical inaccuracies, omissions and typographical errors, and AMD is under no obligation to update or otherwise correct this information. Advanced Micro Devices, Inc. makes no representations or warranties with respect to the accuracy or completeness of the contents of this document, and assumes no liability of any kind, including the implied warranties of noninfringement, merchantability or fitness for particular purposes, with respect to the operation or use of AMD hardware, software or other products described herein.

No license, including implied or arising by estoppel, to any intellectual property rights is granted by this document. Terms and limitations applicable to the purchase or use of AMD's products or technology are as set forth in a signed agreement between the parties or in AMD's Standard Terms and Conditions of Sale.

AMD, the AMD Arrow logo, and combinations thereof are trademarks of Advanced Micro Devices, Inc. OpenCL is a trademark of Apple Inc. used by permission by Khronos Group, Inc. DirectX is a registered trademark of Microsoft Corporation in the US and other jurisdictions. Other product names used in this publication are for identification purposes only and may be trademarks of their respective companies.

© 2018-2024 Advanced Micro Devices, Inc. All rights reserved.

**Advanced Micro Devices, Inc.**
2485 Augustine Drive
Santa Clara, CA, 95054
www.amd.com

# Notices

© 2024 Advanced Micro Devices, Inc. All rights reserved.

The information contained herein is for informational purposes only, and is subject to change without notice. While every precaution has been taken in the preparation of this document, it may contain technical inaccuracies, omissions and typographical errors, and AMD is under no obligation to update or otherwise correct this information. Advanced Micro Devices, Inc. makes no representations or warranties with respect to the accuracy or completeness of the contents of this document, and assumes no liability of any kind, including the implied warranties of noninfringement, merchantability or fitness for particular purposes, with respect to the operation or use of AMD hardware, software or other products described herein. No license, including implied or arising by estoppel, to any intellectual property rights is granted by this document. Terms and limitations applicable to the purchase or use of AMD's products are as set forth in a signed agreement between the parties or in AMD's Standard Terms and Conditions of Sale. Any unauthorized copying, alteration, distribution, transmission, performance, display or other use of this material is prohibited.

# Trademarks

AMD, the AMD Arrow logo, AMD AllDay, AMD Virtualization, AMD-V, PowerPlay, Vari-Bright, and combinations thereof are trademarks of Advanced Micro Devices, Inc. Other product names used in this publication are for identification purposes only and may be trademarks of their respective companies.

Microsoft, Windows, Windows Vista, MSFT, OneDrive and DirectX are registered trademarks of Microsoft Corporation in the US and/or other countries.

USE OF THIS PRODUCT IN ANY MANNER THAT COMPLIES WITH THE MPEG ACTUAL OR DE FACTO VIDEO AND/OR AUDIO STANDARDS IS EXPRESSLY PROHIBITED WITHOUT ALL NECESSARY LICENSES UNDER APPLICABLE PATENTS. SUCH LICENSES MAY BE ACQUIRED FROM VARIOUS THIRD PARTIES INCLUDING, BUT NOT LIMITED TO, IN THE MPEG PATENT PORTFOLIO, WHICH LICENSE IS AVAILABLE FROM MPEG LA, L.L.C., 6312 S. FIDDLERS GREEN CIRCLE, SUITE 400E, GREENWOOD VILLAGE, COLORADO 80111.