HSA Runtime Programmer's Reference Manual

0.182

June 3, 2014

Draft

Contents

Chapter 1

Introduction

1.1 Overview

Recent heterogeneous system designs have integrated CPU, GPU, and other accelerator devices into a single platform with a shared high-bandwidth memory system. Specialized accelerators now complement general purpose CPU chips and are used to provide both power and performance benefits. These heterogeneous designs are now widely used in many computing markets including cellphones, tablets, personal computers, and game consoles. The Heterogeneous System Architecture (HSA) builds on the close physical integration of accelerators that is already occurring in the marketplace, and takes the next step by defining standards for uniting the accelerators architecturally. The HSA specifications include requirements for virtual memory, memory coherency, architected dispatch mechanisms, and power-efficient signals. HSA refers to these accelerators as "components".

The system architecture defines a consistent base for building portable applications that access the power and performance benefits of the dedicated HSA components. Many of these components, including GPUs and DSPs, are capable and flexible processors that have been extended with special hardware for accelerating parallel code. Historically these devices have been difficult to program due to a need for specialized or proprietary programming languages. HSA aims to bring the benefits of these components to mainstream programming languages using similar or identical syntax to that which is provided for programming multicore CPUs.

In addition to the system architecture, HSA defines a portable, low-level, compiler intermediate language called “HSAIL”. A high-level compiler generates the HSAIL for the parallel regions of code. A low-level compiler called the “finalizer” translates the intermediate HSAIL to target machine code. Each HSA component provides its own implementation of the finalizer. For more information on HSAIL, refer to the HSA Programmer’s Reference Manual [1].

The final piece of the puzzle is the HSA Runtime API. The runtime is a thin, user-mode API that provides the interfaces necessary for the host to launch compute kernels to the available components. This document describes the architecture and APIs for the HSA Runtime. Key sections of the runtime API include:

- Error Handling
- Runtime initialization and shutdown
- Topology discovery
- Signals and synchronization
- Architected dispatch
- Memory management

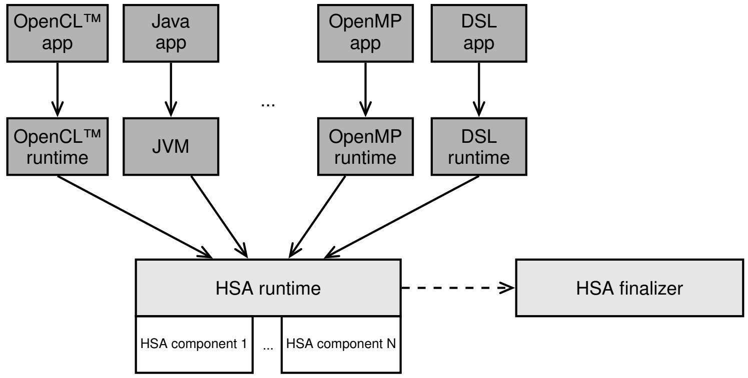

Figure 1.1: HSA Software Architecture

The remainder of this document describes the HSA software architecture and execution model, and includes functional descriptions for all of the HSA APIs and associated data structures.

Figure 1.1 shows how the HSA Runtime fits into a typical software architecture stack. At the top of the stack is a programming model such as OpenCL™, Java, OpenMP, or a domain-specific language (DSL). The programming model must include some way to indicate a parallel region that can be accelerated. For example, OpenCL has calls to `clEnqueueNDRangeKernel` with associated kernels and grid ranges. Java defines stream and lambda APIs, which provide support for both multi-core CPUs and HSA Components. OpenMP contains OMP pragmas that mark loops for parallel computing and that control other aspects of the parallel implementation. Other programming models can also build on this same infrastructure.

The language compiler is responsible for generating HSAIL code for the parallel regions of code. The code can be pre-compiled before runtime or compiled at runtime. A high-level compiler can generate the HSAIL before runtime, in which case when the application loads the finalizer converts the HSAIL to machine code for the target machine. Another option is to run the finalizer when the applications is built, in which case the resulting binary includes the machine code for the target architecture. The HSA Finalizer is an optional component of the HSA Runtime, which can reduce the footprint of the HSA software on systems where the finalization is done before runtime.

Each language also includes a "language runtime" that connects the language implementation to the HSA Runtime. When the language compiler generates code for a parallel region, it will include calls to the HSA Runtime to set up and dispatch the parallel region to the HSA Component. The language runtime is also responsible for initializing HSA, selecting target devices, creating execution queues, managing memory - and may use other HSA Runtime features as well. A runtime implementation may provide optional extensions. Applications can query the runtime to determine which extensions are available. This document describes the extensions for Finalization, Linking and Images.

The API for the HSA Runtime is standard across all HSA vendors, such that languages that use the HSA Runtime can execute on the different vendor's platforms that support the API. Each vendor is responsible for supplying their own HSA Runtime implementation that supports all of the HSA components in the vendor's platform. HSA does not provide a mechanism to combine runtimes from different vendors. The implementation of the HSA Runtime may include kernel-level components (required for some hardware components) or may be entirely user-space (for example, simulators or CPU implementations).

Figure 1.1 shows the "AQL" (Architected Queuing Language) path that application runtimes use to send commands directly to HSA components. For more information on AQL, refer to Section 2.6.

1.2 Execution Model

The HSA runtime exposes several details of the HSA hardware, including architected dispatches. The overall goal of the core runtime design is to provide a high-performance dispatch mechanism that is portable across multiple HSA vendor architectures. Two vendors with the same host ISA but different HSA-compliant GPUs will be able to run the same unmodified application binary file(s), because they support the HSA-architected AQL interface and supply a library that implements the architected core runtime API.

The HSA core runtime takes advantage of architected dispatch. Architected dispatch is the key feature in an HSA system that enables a user-level application to directly issue commands to the HSA Component hardware. Architected dispatch differentiates the HSA runtime from other higher-level runtime systems and programming models: other runtime systems provide software APIs for setting arguments and launching kernels, while HSA architects these at the hardware and specification level. The main advantage of architected dispatch is that the dispatch mechanism is architected at the HSA hardware level which means that an application can use regular memory operations and the wrapper API provided by the runtime to launch a kernel. The application creates user mode queues and AQL packets in memory, and then signals the HSA component to begin executing the packet using lightweight operations.

This section describes various features the HSA runtime provides to support architected dispatch, and the steps an application needs to perform in order to dispatch a kernel.

1.2.1 Initial Setup

One of the first steps an application must perform is agent (device) discovery. Agent discovery is performed after initialization of the runtime and information is made available to the user as data structures. Section 2.3 describes these *topology descriptors*.

The next step in the setup is the creation of the component queues. Queues are an HSA architected mechanism that submits work to the HSA component hardware. For more information about the interfaces for queue creation, refer to Section 2.5. Different components may provide implementation-specific code under the HSA API for these functions.

1.2.2 Kernel Execution

The Systems Architecture Requirements document [2] specifies the structure of the *packets* (i.e. commands) that can be placed on the HSA user mode queues for execution by the component hardware. The Architected Queuing Language (AQL) describes the format of the packets. One type of AQL packet is a Dispatch packet. An application can create an AQL packet and initialize it with the code object that contains a component-specific ISA.

Optimized implementations can cache the result of this step and re-use the AQL packet for subsequent launches. Care must be taken to ensure that the AQL Dispatch packet (and the associated kernel and spill/arg/private memory) is not re-used before the launch completes. For simple cases such as a single-thread, synchronous launch, the AQL Dispatch packet(s) can be declared as a static variable and initialized at the same time the code is finalized. More complex cases might involve creating and tracking several Dispatch packets for a single kernel code object.

HSA hardware defines a packet process for processing these packets and a doorbell mechanism to inform the packet processing hardware that packets have been written into the queue. The HSA runtime defines a structure and update API to inform the hardware that the Dispatch packet has been written to the queue.

For more information on creating queues and the states of queues, refer to Section 2.5. For more information on packet formats and the states of packets, refer to Section 2.6.

After a packet is written and the hardware informed by way of the doorbell, execution can start. Execution of a kernel happens asynchronously. An application can write more packets to launch other kernels in the queue. This activity can overlap the actual execution of the kernel.

1.2.3 Kernel Completion

The architecture specification [2] defines signals as a mechanism for communication between different parts of an HSA system. The HSA core runtime defines signals as opaque objects. The API is defined to send a value to the signal and wait for a value at the signal. For more information on signals, refer to Section 2.4.

The AQL Dispatch packet provides a data field that allows an application to pass an opaque signal. When the HSA Component hardware observes a valid signal in an AQL packet, it sends a value to this signal when execution of the kernel completes (success or error). An application can wait on this signal to determine kernel completion. For more information on errors, refer to Section 2.2.

1.2.4 Example

```
/**
 * Dispatch a kernel in the command queue of a component.
 *
 * The source code has been simplified for readibility. For instance, we do
 * not check the status code return by invocations of the HSA API, and we assume
 * that no asynchronous errors are generated by the runtime while executing the
 * kernel.
 */
#include "assert.h"
#include "stdio.h"
#include "string.h"

#include "hsa.h"

int main() {

  // Initialize the runtime
  hsa_init();

  // Retrieve the first available component and run the kernel on it.
  uint32_t* ids = NULL;
  int* num_ids = NULL;
  hsa_topology_object_ids(HSA_TOPOLOGY_OBJECT_AGENT, &ids, num_ids);
  assert(*num_ids > 0);
  hsa_agent_t component;
  // For simplicity, assume that the first agent listed is also a component.
  hsa_topology_object_descriptor(ids[0], &component);
  assert(component.agent_type & HSA_AGENT_TYPE_COMPONENT);

  // Create a queue in the selected component. The queue can hold up to four
  // packets, and has no callback or service queue associated with it.
  hsa_queue_t *queue;
  hsa_queue_create(&component, 4, HSA_QUEUE_TYPE_SINGLE, NULL, NULL, &queue);

  // Setup the packet encoding the task to execute
  hsa_aql_dispatch_packet_t dispatch_packet;
  const size_t packet_size = sizeof(dispatch_packet);
  memset(&dispatch_packet, 0, packet_size); // reserved fields are zeroed
  dispatch_packet.header.acquire_fence_scope = HSA_FENCE_SCOPE_COMPONENT;
  dispatch_packet.header.release_fence_scope = HSA_FENCE_SCOPE_COMPONENT;
```

```
  dispatch_packet.dimensions = 1;
  dispatch_packet.workgroup_size_x = 256;
  dispatch_packet.grid_size_x = 256;

  // Indicate which ISA to run. The application is expected to have finalized a
  // kernel (for example, using the finalization API). We will assume the object is
  // located at address 0xDEADBEEF
  dispatch_packet.kernel_object_address = 0xDEADBEEF;

  // Assume our kernel receives no arguments, so no need to set the kernarg_address
  // field in the packet.

  // Create a signal with an initial value of one to monitor the task completion
  hsa_signal_handle_t signal;
  hsa_signal_create(1, &signal);
  dispatch_packet.completion_signal = signal;

  // Request a packet ID from the queue
  // Since no packets have been enqueued yet, the expected ID is zero.
  uint64_t write_index = hsa_queue_add_write_index_relaxed(queue, 1);

  // Calculate the virtual address where to place the packet.
  // No need to check if the queue is full or the index might wrap around...
  void* dst = (void*)(queue->base_address + write_index * packet_size);
  memcpy(dst, &dispatch_packet, packet_size);

  // Notify the queue that the packet is ready to be processed
  dispatch_packet.header.format = HSA_AQL_PACKET_FORMAT_DISPATCH;
  hsa_signal_store_release(queue->doorbell_signal, write_index);

  // Wait for the task to finish, which is the same as waiting for the value of the
  // completion signal to be zero.
  hsa_signal_value_t *observed = NULL;
  while (hsa_signal_wait_acquire(signal, HSA_EQ, 0, observed) ==
      HSA_STATUS_ERROR_WAIT_ABANDONED);
  assert(*observed == 0);

  // Done! The kernel has completed. Time to cleanup resources and leave.
  hsa_signal_destroy(signal);
  hsa_queue_destroy(queue);
  hsa_shut_down();
  return 0;
}
```

Chapter 2

HSA Core Programming Guide

This chapter describes the HSA Core runtime APIs, organized by functional area. For information on definitions that are not specific to any functionality, refer to Section 2.9.

Several operating systems allow functions to be executed when a DLL or a shared library is loaded (for example, DLL main in Windows and GCC *constructor/destructor* attributes that allow functions to be executed prior to main in several operating systems). Whether or not the HSA runtime functions are allowed to be invoked in such fashion may be implementation specific and is outside the scope of this specification.

Any header files distributed by the HSA foundation for this specification may contain calling-convention specific prefixes such as __cdecl or __stdcall, which are outside the scope of the API definition.

Unless otherwise stated, functions can be considered thread-safe.

2.1 Initialization and Shut Down

When an application initializes the runtime (**hsa_init**) for the first time in a given process, a runtime instance is created. The instance is internally reference counted such that multiple HSA clients within the same process do not interfere with each other. Invoking the initialization routine n times within a process does not create n runtime instances, but a unique runtime object with an associated reference counter of n. Shutting down the runtime (**hsa_shut_down**) is equivalent to decreasing its reference counter. When the reference counter is less than one, the runtime object ceases to exists and any reference to it (or to any resources created while it was active) results in undefined behavior.

2.1.1 API

2.1.1.1 hsa_init

```
hsa_status_t hsa_init()
```

Initialize the HSA runtime.

Return Values

HSA_STATUS_SUCCESS
    The function has been executed successfully.

HSA_STATUS_ERROR_OUT_OF_RESOURCES

If there is failure to allocate the resources required by the implementation.

HSA_STATUS_ERROR_REFCOUNT_OVERFLOW
If the runtime reference count reaches INT32_MAX.

Description

Initializes the HSA runtime if it is not already initialized, and increases the reference counter associated with the HSA runtime for the current process. Invocation of any HSA function other than **hsa_init** results in undefined behavior if the current HSA runtime reference counter is less than one.

2.1.1.2 hsa_shut_down

```
hsa_status_t hsa_shut_down()
```

Shut down the HSA runtime.

Return Values

HSA_STATUS_SUCCESS
The function has been executed successfully.

HSA_STATUS_ERROR_NOT_INITIALIZED
The runtime has not been initialized.

Description

Decreases the reference count of the runtime instance. When the reference count reaches zero, the runtime is no longer considered valid but the application might call **hsa_init** to initialize the HSA runtime again.

Once the reference count of the runtime reaches zero, all the resources associated with it (queues, signals, topology information, etc.) are considered invalid and any attempt to reference them in subsequent API calls results in undefined behavior. When the reference count reaches zero, the HSA runtime might release resources associated with it.

2.2 Runtime Notifications

The runtime can report notifications (such as errors, and events) synchronously or asynchronously. The runtime uses the return value of functions in the HSA API to pass notifications synchronously. In this case, the notification is a status code of type hsa_status_t that indicates success or error.

The documentation of each function defines what constitutes a successful execution. When a HSA function does not execute successfully, the returned status code might help determining the source of the error. While a few error conditions can be generalized to a certain degree (e.g. failure in allocating resources) many errors can have implementation-specific explanations. For example, operations on signals (see Section 2.4) might fail if the implementation validates the signal object on which the signals operate. Because the representation of a signal is specific to the implementation, the reported error would simply indicate that the signal is invalid.

The hsa_status_t enumeration captures the result of any API function that has been executed, except for accessors and mutators. Success is represented by HSA_STATUS_SUCCESS, which has a value of zero. Error statuses are assigned positive integers and their identifiers start with the HSA_STATUS_ERROR prefix. The application might use **hsa_status_string** to obtain a string describing a status code.

The runtime passes *asynchronous* notifications in a different fashion. When the runtime detects an asynchronous event, the runtime invokes a user-defined callback. For example, queues (see Section 2.5) are a common source of asynchronous events because the tasks queued by an application are asynchronously consumed by the packet processor. Callbacks are associated with queues when they are created, and when invoked they are passed an event object (hsa_event_t) which contains valuable information about the event source and other event-specific details.

The application must use caution when using blocking functions within their callback implementation – a callback that does not return can render the runtime state to be undefined. The application cannot depend on thread local storage within the callbacks implementation and may safely kill the thread that registers the callback. The application is responsible for ensuring that the callback function is thread-safe. The runtime does not implement any default callbacks.

2.2.1 API

2.2.1.1 hsa_status_t

enum **hsa_status_t**

Values

HSA_STATUS_SUCCESS = 0
: The function has been executed successfully.

HSA_EXT_STATUS_INFO_ALREADY_INITIALIZED
: Indicates that initialization attempt failed due to prior initialization.

HSA_EXT_STATUS_INFO_UNRECOGNIZED_OPTIONS
: TODO.

HSA_STATUS_ERROR_WAIT_ABANDONED
: A signal wait has been abandoned before the condition associated with the signal value and the wait is met.

HSA_STATUS_ERROR_INVALID_ARGUMENT
: One of the actual arguments does not meet a precondition stated in the documentation of the corresponding formal argument.

HSA_STATUS_ERROR_INVALID_COMPONENT
The component is invalid.

HSA_STATUS_ERROR_INVALID_SIGNAL
The signal is invalid.

HSA_STATUS_ERROR_INVALID_QUEUE
The queue is invalid.

HSA_STATUS_ERROR_OUT_OF_RESOURCES
The runtime failed to allocate the necessary resources. This error may also occur when the core runtime library needs to spawn threads or create internal OS-specific events.

HSA_STATUS_ERROR_INVALID_PACKET_FORMAT
Indicates that the AQL packet is malformed.

HSA_STATUS_ERROR_SIGNAL_DEPENDENCY
Indicates that a signal we depend on has a negative value.

HSA_STATUS_ERROR_RESOURCE_FREE
An error has been detected while releasing a resource.

HSA_STATUS_ERROR_NOT_REGISTERED
The pointer is not currently registered.

HSA_STATUS_ERROR_NOT_INITIALIZED
An API other than **hsa_init** has been invoked while the reference count of the HSA runtime is zero.

HSA_STATUS_ERROR_REFCOUNT_OVERFLOW
The maximum reference count for the object has been reached.

HSA_EXT_STATUS_ERROR_DIRECTIVE_MISMATCH
TODO.

HSA_EXT_STATUS_ERROR_IMAGE_FORMAT_UNSUPPORTED
Image format is not supported.

HSA_EXT_STATUS_ERROR_IMAGE_SIZE_UNSUPPORTED
Image size is not supported.

2.2.1.2 hsa_status_string

hsa_status_t **hsa_status_string**(
hsa_status_t *status*,
char *const * *status_string*)

Query additional information about a status code.

Parameters

status
(in) Status code that the user is seeking more information on.

status_string
(out) A ISO/IEC 646 encoded English language string that potentially describes the error status. The string terminates in a NUL character.

Return Values

HSA_STATUS_SUCCESS
The function has been executed successfully.

HSA_STATUS_ERROR_NOT_INITIALIZED
The runtime has not been initialized.

HSA_STATUS_ERROR_INVALID_ARGUMENT
If *status_string* is NULL or *status* is not a valid status code.

2.2.1.3 hsa_event_s

```
typedef struct hsa_event_s {
    hsa_status_t status;
    uint64_t data[4];
    char info[64];
    uint64_t timestamp;
} hsa_event_t
```

Event object. Used to pass information from the HSA runtime to the application.

Data Fields

status
Status code associated with the event.

data
Additional information about the event to be interpreted based on *status*.

info
A string containing further information. ISO/IEC 646 character encoding must be used. The string should be NUL terminated.

timestamp
System timestamp to indicate when the event was discovered. If the implementation chooses not to return the current timestamp, then *timestamp* must be zero.

Description

The event object communicates to the application what has happened (*status* field), and might contain event-specific details that can be parsed by the application to further understand the event.

2.2.1.4 hsa_event_callback_t

```
typedef void(* hsa_event_callback_t)(const hsa_event_t *event)
```

Callback for events.

2.3 Topology Discovery

Topology discovery is provided by the runtime so users can programmatically query information about the available agents, memories, caches, etc. This information could be utilized by the user in different ways, including decisions on where to execute a particular task. The runtime specification defines three types of *topology objects*: agents, memories and caches. Every instance of each object is described in a so-called topology descriptor. For example, agent attributes are listed in the hsa_agent_t descriptor.

2.3.1 API

2.3.1.1 hsa_agent_type_t

```
enum hsa_agent_type_t
```

Agent type.

Values

HSA_AGENT_TYPE_HOST = 1
    Host agent (CPU).

HSA_AGENT_TYPE_COMPONENT = 2
    HSA component.

HSA_AGENT_TYPE_AGENT_DISPATCH = 4
    The agent is capable of agent dispatches, and can serve as a target for them.

2.3.1.2 hsa_agent_s

```
typedef struct hsa_agent_s {
    uint32_t node_id;
    uint32_t id;
    hsa_agent_type_t agent_type;
    char vendor[16];
    char name[16];
    uint32_t * memory_descriptors;
    uint32_t number_memory_descriptors;
    uint32_t * cache_descriptors;
    uint32_t number_cache_descriptors;
    uint32_t * subagent_offset_list;
    uint32_t number_subagents;
    uint32_t wavefront_size;
    uint32_t queue_size;
    uint32_t group_memory_size_bytes;
    uint32_t fbarrier_max_count;
    uint8_t is_pic_supported;
} hsa_agent_t
```

HSA agent.

Data Fields

node_id

ID of the node this agent/component belongs to.

id
Unique identifier for an HSA agent.

agent_type
Agent type, bit-field.

vendor
The vendor of the agent/component. ISO/IEC 646 character encoding must be used. If the name is less than 16 characters then remaining characters must be set to 0.

name
The name of this agent/component. ISO/IEC 646 character encoding must be used. If the name is less than 16 characters then remaining characters must be set to 0.

memory_descriptors
Array of memory descriptor offsets. Number of elements in array equals *number_memory_descriptors*.

number_memory_descriptors
Number of the different types of memories available to this agent. Zero indicates that no information is available.

cache_descriptors
Array of cache descriptor offsets. Number of elements in array equals *number_cache_descriptors*.

number_cache_descriptors
Number of caches available to this agent/component. Zero indicates that no information is available.

subagent_offset_list
Subagent list of offsets, points to the offsets in the topology table.

number_subagents
Number of subagents.

wavefront_size
Wave front size, i.e. number of work-items in a wavefront.

queue_size
Maximum size of the user queue in bytes allocatable via the runtime.

group_memory_size_bytes
Size (in bytes) of group memory available to a single work-group.

fbarrier_max_count
Max number of fbarrier that can be used in any kernel and functions it invokes.

is_pic_supported
Indicates if the agent supports position-independent code (the value is not zero). Only applicable when the agent is a component.

2.3.1.3 hsa_segment_s

```
typedef struct hsa_segment_s {
    uint8_t global : 1;
    uint8_t privat : 1;
    uint8_t group : 1;
    uint8_t kernarg : 1;
    uint8_t readonly : 1;
```

```
    uint8_t reserved : 1;
} hsa_segment_t
```

Memory segment.

Data Fields

global
Global segment.

privat
Private segment.

group
Group segment.

kernarg
Kernarg segment.

readonly
Readonly segment.

reserved
Reserved.

2.3.1.4 hsa_memory_descriptor_s

```
typedef struct hsa_memory_descriptor_s {
    uint32_t node_id;
    uint32_t id;
    hsa_segment_t supported_segment_type_mask;
    uint64_t virtual_address_base;
    uint64_t size_in_bytes;
    uint64_t peak_bandwidth_mbps;
} hsa_memory_descriptor_t
```

Memory descriptor.

Data Fields

node_id
ID of the node this memory belongs to.

id
Unique ID for this memory within the system.

supported_segment_type_mask
Information on segments that can use this memory.

virtual_address_base
Base of the virtual address for this memory, if applicable.

size_in_bytes
Size.

peak_bandwidth_mbps
Theoretical peak bandwidth in mega-bits per second to access this memory from the agent/component.

Description

Representation of a physical memory block or region. Implementations may choose not to provide memory bandwidth or latency information, which case zero is returned.

2.3.1.5 hsa_cache_descriptor_s

```
typedef struct hsa_cache_descriptor_s {
    uint32_t node_id;
    uint32_t id;
    uint8_t levels;
    uint8_t * associativity;
    uint64_t * cache_size;
    uint64_t * cache_line_size;
    uint8_t * is_inclusive;
} hsa_cache_descriptor_t
```

Cache descriptor.

Data Fields

node_id
: ID of the node this memory belongs to.

id
: Unique ID for this cache within the system.

levels
: Number of levels of cache (for a multi-level cache).

associativity
: Associativity of this cache. The array has size *levels*. Associativity is expressed as a power of two, where 1 means 'direct mapped', and 255 means 'full associative'. Zero is reserved.

cache_size
: Size at each level. The array has size *levels*.

cache_line_size
: Cache line size at each level. The array has size *levels*.

is_inclusive
: Cache inclusivity with respect to the level above. The array has size *levels*, where *is_inclusive*[*levels* - 1] is always zero.

2.3.1.6 hsa_topology_object_t

```
enum hsa_topology_object_t
```

Topology object type.

Values

HSA_TOPOLOGY_OBJECT_AGENT = 1
: Agent object.

HSA_TOPOLOGY_OBJECT_MEMORY = 2
: Memory object.

HSA_TOPOLOGY_OBJECT_CACHE = 4

Cache object.

2.3.1.7 hsa_topology_object_ids

```
hsa_status_t hsa_topology_object_ids(
    hsa_topology_object_t type,
    uint32_t ** ids,
    int * num_ids)
```

Retrieve the identifiers of all the topology objects.

Parameters

type
(in) Type of object affected by the query.

ids
(out) Pointer to a list containing the identifiers of all the topology objects of type *type*.

num_ids
(out) Pointer to a memory location where the number of elements in *ids* is to be stored.

Return Values

HSA_STATUS_SUCCESS
The function has been executed successfully.

HSA_STATUS_ERROR_NOT_INITIALIZED
The runtime has not been initialized.

HSA_STATUS_ERROR_OUT_OF_RESOURCES
If there is failure to allocate the resources required by the implementation.

HSA_STATUS_ERROR_INVALID_ARGUMENT
If *ids* is NULL or *num_ids* is NULL.

2.3.1.8 hsa_topology_object_descriptor

```
hsa_status_t hsa_topology_object_descriptor(
    uint32_t id,
    void * object_descriptor)
```

Retrieve the topology descriptor associated with a topology object.

Parameters

id
(in) Identifier of the topology object being queried.

object_descriptor
(inout) User-allocated buffer where the descriptor of the object will be copied to. The buffer pointed by *object_descriptor* must be large enough to hold the descriptor for the object.

Return Values

HSA_STATUS_SUCCESS
The function has been executed successfully.

HSA_STATUS_ERROR_NOT_INITIALIZED

The runtime has not been initialized.

HSA_STATUS_ERROR_INVALID_ARGUMENT
If *object_descriptor* is NULL.

2.4 Signals

HSA agents can communicate with each other by using coherent global memory, or by using signals. In an HSA system, a signal can be more power efficient than coherent global memory for communication.

A signal allows multiple producers and consumers. The runtime API uses opaque signal handlers of type hsa_signal_handle_t to represent signals. A signal carries a value that can be updated or conditionally waited upon through an API call or HSAIL instruction. The value occupies four or eight bytes depending on the machine model being used.

Updating the value of a signal is equivalent to sending the signal. In addition to the regular update (store) of a signal value, an application can perform atomic operations such as add, subtract, or compare-and-swap. Each API operation also has specific memory order semantics associated with it. For example, store-release (**hsa_signal_store_release** function) is equivalent to storing a value on the signal handle with release memory ordering. The combinations of actions and memory orders available in the API match the corresponding HSAIL instructions [1].

The application may wait on a signal, with a condition specifying the terms of wait. The wait can be done either in the HSA Component by using an HSAIL **wait** instruction or by using a runtime API call. It is the responsibility of the application to detect that an error has occurred during the wait by checking the output status.

Waiting implies reading the current signal value (which is returned to the application) using an acquire (**hsa_signal_wait_acquire**) or a relaxed (**hsa_signal_wait_relaxed**) memory order. The synchronization should only assume to have been applied if the status returned by the wait API is HSA_STATUS_SUCCESS. HSA implementations might establish a maximum waiting time.

As explained above, an application can modify or wait on signals by using the runtime API. However, signals also allow expressing complex tasks dependencies that are automatically handled by the packet processors. For example, if task y executing in one component consumes the result of task x executing in a different component, then y *depends* on x. In HSA, this dependency can be enforced across components by creating a signal that will be simultaneously used as a) the completion signal of a Dispatch packet px corresponding to x b) the dependency signal in a Barrier packet that precedes the Dispatch packet py corresponding to y. The packet processor enforces the task dependency by not launching py until px has completed.

For more information on HSA interfaces related to adding packets to a queue, refer to Section 2.5. For more information on AQL packets, refer to Section 2.6.

2.4.1 API

2.4.1.1 hsa_signal_handle_t

typedef uint64_t **hsa_signal_handle_t**

Signal handle.

2.4.1.2 hsa_signal_value_t

typedef uintptr_t **hsa_signal_value_t**

Signal value. The value occupies 32 bits in small machine mode, and 64 bits in large machine mode.

2.4.1.3 hsa_signal_create

hsa_status_t **hsa_signal_create**(
    hsa_signal_value_t *initial_value*,
    hsa_signal_handle_t * *signal_handle*)

Create a signal.

Parameters

initial_value
    (in) Initial value of the signal.

signal_handle
    (out) Signal handle.

Return Values

HSA_STATUS_SUCCESS
    The function has been executed successfully.

HSA_STATUS_ERROR_NOT_INITIALIZED
    The runtime has not been initialized.

HSA_STATUS_ERROR_OUT_OF_RESOURCES
    If there is failure to allocate the resources required by the implementation.

HSA_STATUS_ERROR_INVALID_ARGUMENT
    If *signal_handle* is NULL.

2.4.1.4 hsa_signal_destroy

hsa_status_t **hsa_signal_destroy**(
    hsa_signal_handle_t *signal_handle*)

Destroy a signal previous created by **hsa_signal_create**.

Parameters

signal_handle
    (in) Signal handle.

Return Values

HSA_STATUS_SUCCESS
    The function has been executed successfully.

HSA_STATUS_ERROR_NOT_INITIALIZED
    The runtime has not been initialized.

HSA_STATUS_ERROR_INVALID_SIGNAL
    If *signal_handle* is invalid.

2.4.1.5 hsa_signal_load_acquire

```
hsa_status_t hsa_signal_load_acquire(
    hsa_signal_handle_t signal_handle,
    hsa_signal_value_t * value)
```

Atomically read the current signal value.

Parameters

signal_handle
(in) Signal handle.

value
(out) Pointer to memory location where to store the signal value.

Return Values

HSA_STATUS_SUCCESS
The function has been executed successfully.

HSA_STATUS_ERROR_NOT_INITIALIZED
The runtime has not been initialized.

HSA_STATUS_ERROR_INVALID_SIGNAL
If *signal_handle* is invalid.

HSA_STATUS_ERROR_INVALID_ARGUMENT
If *value* is NULL.

2.4.1.6 hsa_signal_load_relaxed

```
hsa_status_t hsa_signal_load_relaxed(
    hsa_signal_handle_t signal_handle,
    hsa_signal_value_t * value)
```

Atomically read the current signal value.

Parameters

signal_handle
(in) Signal handle.

value
(out) Pointer to memory location where to store the signal value.

Return Values

HSA_STATUS_SUCCESS
The function has been executed successfully.

HSA_STATUS_ERROR_NOT_INITIALIZED
The runtime has not been initialized.

HSA_STATUS_ERROR_INVALID_SIGNAL
If *signal_handle* is invalid.

HSA_STATUS_ERROR_INVALID_ARGUMENT
If *value* is NULL.

2.4.1.7 hsa_signal_store_relaxed

hsa_status_t **hsa_signal_store_relaxed**(
    hsa_signal_handle_t *signal_handle*,
    hsa_signal_value_t *value*)

Atomically set the value of a signal.

Parameters

signal_handle
(in) Signal handle.

value
(in) Value to be assigned to the signal handle.

Return Values

HSA_STATUS_SUCCESS
The function has been executed successfully.

HSA_STATUS_ERROR_NOT_INITIALIZED
The runtime has not been initialized.

HSA_STATUS_ERROR_INVALID_SIGNAL
If *signal_handle* is invalid.

2.4.1.8 hsa_signal_store_release

hsa_status_t **hsa_signal_store_release**(
    hsa_signal_handle_t *signal_handle*,
    hsa_signal_value_t *value*)

Atomically set the value of a signal.

Parameters

signal_handle
(in) Signal handle.

value
(in) Value to be assigned to the signal handle.

Return Values

HSA_STATUS_SUCCESS
The function has been executed successfully.

HSA_STATUS_ERROR_NOT_INITIALIZED
The runtime has not been initialized.

HSA_STATUS_ERROR_INVALID_SIGNAL
If *signal_handle* is invalid.

2.4.1.9 hsa_signal_exchange_release

```
hsa_status_t hsa_signal_exchange_release(
    hsa_signal_handle_t signal_handle,
    hsa_signal_value_t value,
    hsa_signal_value_t * prev_value)
```

Atomically set the value of a signal and return its previous value.

Parameters

signal_handle
    (in) Signal handle.

value
    (out) Value to be placed at the signal

prev_value
    (out) Pointer to the value of the signal prior to the exchange.

Return Values

HSA_STATUS_SUCCESS
    The function has been executed successfully.

HSA_STATUS_ERROR_NOT_INITIALIZED
    The runtime has not been initialized.

HSA_STATUS_ERROR_INVALID_SIGNAL
    If *signal_handle* is invalid.

HSA_STATUS_ERROR_INVALID_ARGUMENT
    If *prev_value* is NULL.

2.4.1.10 hsa_signal_exchange_relaxed

```
hsa_status_t hsa_signal_exchange_relaxed(
    hsa_signal_handle_t signal_handle,
    hsa_signal_value_t value,
    hsa_signal_value_t * prev_value)
```

Atomically set the value of a signal and return its previous value.

Parameters

signal_handle
    (in) Signal handle.

value
    (out) Value to be placed at the signal

prev_value
    (out) Pointer to the value of the signal prior to the exchange.

Return Values

HSA_STATUS_SUCCESS
    The function has been executed successfully.

HSA_STATUS_ERROR_NOT_INITIALIZED
    The runtime has not been initialized.

HSA_STATUS_ERROR_INVALID_SIGNAL

If *signal_handle* is invalid.

HSA_STATUS_ERROR_INVALID_ARGUMENT
If *prev_value* is NULL.

2.4.1.11 hsa_signal_cas_release

```
hsa_status_t hsa_signal_cas_release(
    hsa_signal_handle_t signal_handle,
    hsa_signal_value_t expected,
    hsa_signal_value_t value,
    hsa_signal_value_t * observed)
```

Perform a compare and swap on the value of a signal.

Parameters

signal_handle
(in) Signal handle.

expected
(in) The value to compare the handle's value with.

value
(in) The new value of the signal.

observed
(out) Pointer to memory location where to store the observed value of the signal.

Return Values

HSA_STATUS_SUCCESS
The function has been executed successfully.

HSA_STATUS_ERROR_INVALID_SIGNAL
If *signal_handle* is invalid.

HSA_STATUS_ERROR_INVALID_ARGUMENT
If *observed* is NULL.

2.4.1.12 hsa_signal_add_release

```
hsa_status_t hsa_signal_add_release(
    hsa_signal_handle_t signal_handle,
    hsa_signal_value_t value)
```

Atomically increment the value of a signal by a given amount.

Parameters

signal_handle
(in) Signal handle.

value
(in) Value to add to the value of the signal handle.

Return Values

HSA_STATUS_SUCCESS

The function has been executed successfully.

HSA_STATUS_ERROR_NOT_INITIALIZED
The runtime has not been initialized.

HSA_STATUS_ERROR_INVALID_SIGNAL
If *signal_handle* is invalid.

2.4.1.13 hsa_signal_add_relaxed

```
hsa_status_t hsa_signal_add_relaxed(
    hsa_signal_handle_t signal_handle,
    hsa_signal_value_t value)
```

Atomically increment the value of a signal by a given amount.

Parameters

signal_handle
(in) Signal handle.

value
(in) Value to add to the value of the signal handle.

Return Values

HSA_STATUS_SUCCESS
The function has been executed successfully.

HSA_STATUS_ERROR_NOT_INITIALIZED
The runtime has not been initialized.

HSA_STATUS_ERROR_INVALID_SIGNAL
If *signal_handle* is invalid.

2.4.1.14 hsa_signal_subtract_release

```
hsa_status_t hsa_signal_subtract_release(
    hsa_signal_handle_t signal_handle,
    hsa_signal_value_t value)
```

Atomically decrement the value of a signal by a given amount.

Parameters

signal_handle
(in) Signal handle.

value
(in) Value to subtract from the value of the signal handle.

Return Values

HSA_STATUS_SUCCESS
The function has been executed successfully.

HSA_STATUS_ERROR_NOT_INITIALIZED
The runtime has not been initialized.

HSA_STATUS_ERROR_INVALID_SIGNAL
If *signal_handle* is invalid.

2.4.1.15 hsa_signal_subtract_relaxed

```
hsa_status_t hsa_signal_subtract_relaxed(
    hsa_signal_handle_t signal_handle,
    hsa_signal_value_t value)
```

Atomically decrement the value of a signal by a given amount.

Parameters

signal_handle
(in) Signal handle.

value
(in) Value to subtract from the value of the signal handle.

Return Values

HSA_STATUS_SUCCESS
The function has been executed successfully.

HSA_STATUS_ERROR_NOT_INITIALIZED
The runtime has not been initialized.

HSA_STATUS_ERROR_INVALID_SIGNAL
If *signal_handle* is invalid.

2.4.1.16 hsa_signal_and_release

```
hsa_status_t hsa_signal_and_release(
    hsa_signal_handle_t signal_handle,
    hsa_signal_value_t value)
```

Atomically perform a logical AND of the value of a signal and a given value.

Parameters

signal_handle
(in) Signal handle.

value
(in) Value to AND with the value of the signal handle.

Return Values

HSA_STATUS_SUCCESS
The function has been executed successfully.

HSA_STATUS_ERROR_NOT_INITIALIZED
The runtime has not been initialized.

HSA_STATUS_ERROR_INVALID_SIGNAL
If *signal_handle* is invalid.

2.4.1.17 hsa_signal_and_relaxed

```
hsa_status_t hsa_signal_and_relaxed(
    hsa_signal_handle_t signal_handle,
    hsa_signal_value_t value)
```

Atomically perform a logical AND of the value of a signal and a given value.

Parameters

signal_handle
(in) Signal handle.

value
(in) Value to AND with the value of the signal handle.

Return Values

HSA_STATUS_SUCCESS
The function has been executed successfully.

HSA_STATUS_ERROR_NOT_INITIALIZED
The runtime has not been initialized.

HSA_STATUS_ERROR_INVALID_SIGNAL
If *signal_handle* is invalid.

2.4.1.18 hsa_signal_or_release

```
hsa_status_t hsa_signal_or_release(
    hsa_signal_handle_t signal_handle,
    hsa_signal_value_t value)
```

Atomically perform a logical OR of the value of a signal and a given value.

Parameters

signal_handle
(in) Signal handle.

value
(in) Value to OR with the value of the signal handle.

Return Values

HSA_STATUS_SUCCESS
The function has been executed successfully.

HSA_STATUS_ERROR_NOT_INITIALIZED
The runtime has not been initialized.

HSA_STATUS_ERROR_INVALID_SIGNAL
If *signal_handle* is invalid.

2.4.1.19 hsa_signal_or_relaxed

```
hsa_status_t hsa_signal_or_relaxed(
    hsa_signal_handle_t signal_handle,
    hsa_signal_value_t value)
```

Atomically perform a logical OR of the value of a signal and a given value.

Parameters

signal_handle
(in) Signal handle.

value
(in) Value to OR with the value of the signal handle.

Return Values

HSA_STATUS_SUCCESS
The function has been executed successfully.

HSA_STATUS_ERROR_NOT_INITIALIZED
The runtime has not been initialized.

HSA_STATUS_ERROR_INVALID_SIGNAL
If *signal_handle* is invalid.

2.4.1.20 hsa_signal_xor_release

```
hsa_status_t hsa_signal_xor_release(
    hsa_signal_handle_t signal_handle,
    hsa_signal_value_t value)
```

Atomically perform a logical XOR of the value of a signal and a given value.

Parameters

signal_handle
(in) Signal handle.

value
(in) Value to XOR with the value of the signal handle.

Return Values

HSA_STATUS_SUCCESS
The function has been executed successfully.

HSA_STATUS_ERROR_NOT_INITIALIZED
The runtime has not been initialized.

HSA_STATUS_ERROR_INVALID_SIGNAL
If *signal_handle* is invalid.

2.4.1.21 hsa_signal_xor_relaxed

```
hsa_status_t hsa_signal_xor_relaxed(
    hsa_signal_handle_t signal_handle,
```

```
    hsa_signal_value_t value)
```

Atomically perform a logical XOR of the value of a signal and a given value.

Parameters

signal_handle
(in) Signal handle.

value
(in) Value to XOR with the value of the signal handle.

Return Values

HSA_STATUS_SUCCESS
The function has been executed successfully.

HSA_STATUS_ERROR_NOT_INITIALIZED
The runtime has not been initialized.

HSA_STATUS_ERROR_INVALID_SIGNAL
If *signal_handle* is invalid.

2.4.1.22 hsa_signal_condition_t

```
enum hsa_signal_condition_t
```

Wait condition operator.

Values

HSA_EQ
The two operands are equal.

HSA_NE
The two operands are not equal.

HSA_LT
The first operand is less than the second operand.

HSA_GTE
The first operand is greater than or equal to the second operand.

2.4.1.23 hsa_signal_wait_acquire

```
hsa_status_t hsa_signal_wait_acquire(
    hsa_signal_handle_t signal_handle,
    hsa_signal_condition_t condition,
    hsa_signal_value_t compare_value,
    hsa_signal_value_t * return_value)
```

Wait until the value of a signal satisfies a given condition.

Parameters

signal_handle
(in) Signal handle.

condition
(in) Condition used to compare the signal value with *compare_value*.

compare_value
(in) Value to compare with.

return_value
(out) Pointer to a memory location where the observed value of *signal_handle* is written. If the function returns success, the returned value must satisfy the passed condition. If the function returns any other status, the implementation is not required to populate this value.

Return Values

HSA_STATUS_SUCCESS
The function has been executed successfully.

HSA_STATUS_ERROR_NOT_INITIALIZED
The runtime has not been initialized.

HSA_STATUS_ERROR_INVALID_SIGNAL
If *signal_handle* is invalid.

HSA_STATUS_ERROR_WAIT_ABANDONED
If the wait has been abandoned (for example, a spurious wakeup has occurred) before the condition is met.

HSA_STATUS_ERROR_INVALID_ARGUMENT
If *condition* is not a valid condition value, or *return_value* is NULL.

Description

The wait may return before the condition is satisfied due to multiple reasons. It is the user's burden to check the returned status before consuming *return_value*.

When the wait operation atomically loads the value of the passed signal, it uses the memory order indicated in the function name.

2.4.1.24 hsa_signal_wait_relaxed

```
hsa_status_t hsa_signal_wait_relaxed(
    hsa_signal_handle_t signal_handle,
    hsa_signal_condition_t condition,
    hsa_signal_value_t compare_value,
    hsa_signal_value_t * return_value)
```

Wait until the value of a signal satisfies a given condition.

Parameters

signal_handle
(in) Signal handle.

condition
(in) Condition used to compare the signal value with *compare_value*.

compare_value
(in) Value to compare with.

return_value
(out) Pointer to a memory location where the observed value of *signal_handle* is written. If the function returns success, the returned value must satisfy the passed condition. If the function returns any other status, the implementation is not required to populate this value.

Return Values

HSA_STATUS_SUCCESS
The function has been executed successfully.

HSA_STATUS_ERROR_NOT_INITIALIZED
The runtime has not been initialized.

HSA_STATUS_ERROR_INVALID_SIGNAL
If *signal_handle* is invalid.

HSA_STATUS_ERROR_WAIT_ABANDONED
If the wait has been abandoned (for example, a spurious wakeup has occurred) before the condition is met.

HSA_STATUS_ERROR_INVALID_ARGUMENT
If *condition* is not a valid condition value, or *return_value* is NULL.

Description

The wait may return before the condition is satisfied due to multiple reasons. It is the user's burden to check the returned status before consuming *return_value*.

When the wait operation atomically loads the value of the passed signal, it uses the memory order indicated in the function name.

2.4.1.25 hsa_signal_wait_timeout_acquire

```
hsa_status_t hsa_signal_wait_timeout_acquire(
    hsa_signal_handle_t signal_handle,
    uint64_t timeout,
    hsa_signal_condition_t condition,
    hsa_signal_value_t compare_value,
    hsa_signal_value_t * return_value)
```

Wait until the value of a signal satisfies a given condition.

Parameters

signal_handle
(in) Signal handle.

timeout
(in) Maximum wait duration hint. The operation might block for a shorter or longer time even if the condition is not met. Specified in the same unit as the system timestamp. A value of UINT64_MAX indicates no maximum.

condition
(in) Condition used to compare the signal value with *compare_value*.

compare_value
(in) Value to compare with.

return_value
(out) Pointer to a memory location where the observed value of *signal_handle* is written. If the function returns success, the returned value must satisfy the passed condition. If the function returns any other status, the implementation is not required to populate this value.

Return Values

HSA_STATUS_SUCCESS

The function has been executed successfully.

HSA_STATUS_ERROR_NOT_INITIALIZED
The runtime has not been initialized.

HSA_STATUS_ERROR_INVALID_SIGNAL
If *signal_handle* is invalid.

HSA_STATUS_ERROR_WAIT_ABANDONED
If the wait has been abandoned (for example, it timed out or a spurious wakeup has occurred) before the condition is met.

HSA_STATUS_ERROR_INVALID_ARGUMENT
If *condition* is not a valid condition value, or *return_value* is NULL.

Description

The wait may return before the condition is satisfied due to multiple reasons. It is the user's burden to check the returned status before consuming *return_value*.

The application might indicate a preference about the maximum wait duration, which implementations can ignore.

When the wait operation atomically loads the value of the passed signal, it uses the memory order indicated in the function name.

2.4.1.26 hsa_signal_wait_timeout_relaxed

```
hsa_status_t hsa_signal_wait_timeout_relaxed(
    hsa_signal_handle_t signal_handle,
    uint64_t timeout,
    hsa_signal_condition_t condition,
    hsa_signal_value_t compare_value,
    hsa_signal_value_t * return_value)
```

Wait until the value of a signal satisfies a given condition.

Parameters

signal_handle
(in) Signal handle.

timeout
(in) Maximum wait duration hint. The operation might block for a shorter or longer time even if the condition is not met. Specified in the same unit as the system timestamp. A value of UINT64_MAX indicates no maximum.

condition
(in) Condition used to compare the signal value with *compare_value*.

compare_value
(in) Value to compare with.

return_value
(out) Pointer to a memory location where the observed value of *signal_handle* is written. If the function returns success, the returned value must satisfy the passed condition. If the function returns any other status, the implementation is not required to populate this value.

Return Values

HSA_STATUS_SUCCESS

The function has been executed successfully.

HSA_STATUS_ERROR_NOT_INITIALIZED
: The runtime has not been initialized.

HSA_STATUS_ERROR_INVALID_SIGNAL
: If *signal_handle* is invalid.

HSA_STATUS_ERROR_WAIT_ABANDONED
: If the wait has been abandoned (for example, it timed out or a spurious wakeup has occurred) before the condition is met.

HSA_STATUS_ERROR_INVALID_ARGUMENT
: If *condition* is not a valid condition value, or *return_value* is NULL.

Description

The wait may return before the condition is satisfied due to multiple reasons. It is the user's burden to check the returned status before consuming *return_value*.

The application might indicate a preference about the maximum wait duration, which implementations can ignore.

When the wait operation atomically loads the value of the passed signal, it uses the memory order indicated in the function name.

2.5 Queues

HSA hardware supports packet execution through user-level queues. A queue is associated with a specific component, which might have several queues attached to it. Applications launch packets (explained in more detail in the next section) by placing the packets in the user mode queue of a component. The queue memory is then processed by the packet processor as though it were a ring buffer, with separate memory locations defining the current write and read addresses.

The HSA runtime provides an interface (**hsa_queue_create**) to create a user mode queue. When an application creates a queue, the runtime allocates memory for the data structure that represents the queue, as well as the buffer indicated by the *base_address* field.

In the HSA API, queues are of type hsa_queue_t. Queues are read-only data structures. Writing values directly to a queue struct results in undefined behavior.

In addition to the visible fields listed in hsa_queue_t, a queue also contains a read index and a write index. The write index is the number of packets allocated so far in that queue; the read index is the number of packets that have been processed and released by the queue's packet processor (i.e., the identifier of the next packet to be released). Both the write index and the read index are 64-bit unsigned integers that can exceed the maximum number of packets the queue can hold.

The write index and the read index are initialized to a value of 0. The runtime does not directly expose the index values to an application. An application can only access the values by using dedicated APIs. The available index functions differ on the index of interest (read or write), action to be performed (addition, compare and swap, etc.), and memory order (relaxed, release, etc.).

When the application observes that the read index matches the write index, the queue can be considered empty – this does not mean that the kernels have finished execution, just that all packets have been consumed. On the other hand, if the observed write index is greater or equal than the sum of the read index and the size of the queue, then the queue is full.

The *doorbell_signal* field contains a signal that the agent writing packets uses to indicate the packet processor that it has work to do. The value which the doorbell signal must be signaled with corresponds to the identifier of the packet that is ready to be launched. The new task might be consumed by the packet processor even before the doorbell signal has been signaled by the agent. This is because the packet processor might be already processing some other packet and observes that there is new work available, so it processes the new packets. In any case, the agent is required to signal the doorbell for every batch of packets it writes.

The runtime uses agent dispatch packets to specify runtime-defined or user-registered functions that will be executed on the agent (typically, the host CPU). Service queues consume agent dispatch packets. The

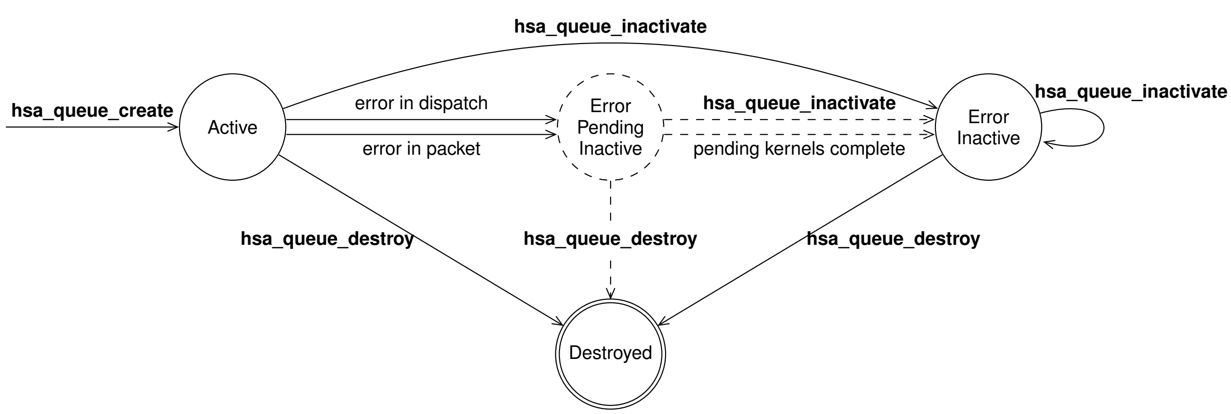

Figure 2.1: Queue state diagram.

service queue is visible to HSA agents through the queue structure *service_queue* field and is serviced by an appropriate HSA agent. The application may chose not to use a service queue.

2.5.1 Queue States

A queue in HSA, once created, can be in one of the following states: *active*, *error pending inactive*, *error inactive* or *destroyed*. A state diagram showing the various states and transitions is shown in Figure 2.1.

Active When a queue is successfully created using the **hsa_queue_create** API, it enters the active state. Packets can be added to the queue and are consumed by the packet processor. The actual initiation of dispatch may depend on the resources available for the dispatch. Writing packets to the queue, updating the write index or ringing the doorbell have an effect only when the queue is in the active state. The queue is not monitored by a packet processor in any other state.

Error pending inactive If an error is encountered during packet processing (invalid packet format, wrong signal, etc.) or dispatch, the packet processor stops. At this point, there might be in-flight kernels and resources (such as segment allocation) that have been setup for a dispatch but have not yet been freed. So the queue is not entirely inactive, but when the asynchronous activity concludes, it will become inactive. A queue in *error pending inactive* state is not considered destroyed. It still needs to be destroyed so that the runtime can reclaim the memory allocated for this queue. If an application provides a callback at the time the application creates the queue, the callback is invoked after the queue is marked inactive.

Inactive If all the asynchronous activity concludes, the queue enters the inactive state. A queue can also enter this state when the user explicitly invokes the **hsa_queue_inactivate** API (note that the callback implementation for the queue error callback can invoke this API). In an inactive state, the queue structure and its packets may be inspected. Only the packets that are between the read index and the write index in the queue structure are considered to be valid for inspection by the user. The packet processor guarantees that all the packets that have been consumed by the packet processor (see Section 2.6.1) will be notified. Inactivating a queue that is already in the inactive state has no effect.

Destroyed The queue has been destroyed by the user. The resources allocated to the queue and the memory for the queue are no longer valid. The queue structure is no longer valid.

2.5.2 API

2.5.2.1 hsa_queue_type_t

enum **hsa_queue_type_t**

Queue type. Intended to be used for dynamic queue protocol determination.

Values

HSA_QUEUE_TYPE_MULTI = 0
  Multiple producers are supported.

HSA_QUEUE_TYPE_SINGLE = 1
  Only a single producer is supported.

2.5.2.2 hsa_queue_feature_t

enum **hsa_queue_feature_t**

Queue features.

Values

HSA_QUEUE_FEATURE_DISPATCH = 1
    Queue supports dispatch packets.

HSA_QUEUE_FEATURE_AGENT_DISPATCH = 2
    Queue supports agent dispatch packets.

2.5.2.3 hsa_queue_s

```
typedef struct hsa_queue_s {
    hsa_queue_type_t type;
    uint32_t features;
    uint64_t base_address;
    hsa_signal_handle_t doorbell_signal;
    uint32_t size;
    uint32_t id;
    uint64_t service_queue;
} hsa_queue_t
```

User mode queue.

Data Fields

type
    Queue type.

features
    Queue features mask. Applications should ignore any unknown set bits.

base_address
    Starting address of the runtime-allocated buffer which is used to store the AQL packets. Aligned to the size of an AQL packet.

doorbell_signal
    Signal object used by the application to indicate the ID of a packet that is ready to be processed.

    The HSA runtime is responsible for the life cycle of the doorbell signal: replacing it with another signal or destroying it is not allowed and results in undefined behavior.

    If *type* is HSA_QUEUE_TYPE_SINGLE, it is the application's responsibility to update the doorbell signal value with monotonically increasing indexes.

size
    Maximum number of packets the queue can hold. Must be a power of two.

id
    Queue identifier which is unique per process.

service_queue
    A pointer to another user mode queue that can be used by the HSAIL kernel to request system services.

Description

Queues are read-only, but HSA agents can directly modify the contents of the buffer pointed by *base_-address*, or use runtime APIs to access the doorbell signal or the service queue.

2.5.2.4 hsa_queue_create

```
hsa_status_t hsa_queue_create(
    const hsa_agent_t * component,
    size_t size,
    hsa_queue_type_t type,
    hsa_event_callback_t event_callback,
    hsa_queue_t * service_queue,
    hsa_queue_t ** queue)
```

Create a user mode queue.

Parameters

component
(in) Pointer to the component on which this queue is to be created.

size
(in) Number of packets the queue is expected to hold. Must be a power of two.

type
(in) Type of the queue.

event_callback
(in) Callback to be invoked for events related with this queue. Can be NULL.

service_queue
(in) Pointer to service queue to be associated with the newly created queue. It might be NULL, or another previously created queue that supports agent dispatch.

queue
(out) The queue structure, filled up and returned by the runtime.

Return Values

HSA_STATUS_SUCCESS
The function has been executed successfully.

HSA_STATUS_ERROR_NOT_INITIALIZED
The runtime has not been initialized.

HSA_STATUS_ERROR_OUT_OF_RESOURCES
If there is failure to allocate the resources required by the implementation.

HSA_STATUS_ERROR_INVALID_COMPONENT
If the component is invalid.

HSA_STATUS_ERROR_INVALID_ARGUMENT
If *component* is NULL, *size* is not a power of two, *type* is not a valid queue type, or *queue* is NULL.

Description

When a queue is created, the runtime also allocates the packet buffer and the completion signal. The application should only rely on the error code returned to determine if the queue is valid.

2.5.2.5 hsa_queue_destroy

```
hsa_status_t hsa_queue_destroy(
    hsa_queue_t * queue)
```

Destroy a user mode queue.

Parameters

queue
(in) Pointer to a queue.

Return Values

HSA_STATUS_SUCCESS
The function has been executed successfully.

HSA_STATUS_ERROR_NOT_INITIALIZED
The runtime has not been initialized.

HSA_STATUS_ERROR_INVALID_QUEUE
If the queue is invalid.

HSA_STATUS_ERROR_INVALID_ARGUMENT
If *queue* is NULL.

Description

A destroyed queue might not be accessed after being destroyed. When a queue is destroyed, the state of the AQL packets that have not been yet fully processed becomes undefined.

2.5.2.6 hsa_queue_inactivate

```
hsa_status_t hsa_queue_inactivate(
    hsa_queue_t * queue)
```

Inactivate a queue.

Parameters

queue
(in) Pointer to a queue.

Return Values

HSA_STATUS_SUCCESS
The function has been executed successfully.

HSA_STATUS_ERROR_NOT_INITIALIZED
The runtime has not been initialized.

HSA_STATUS_ERROR_INVALID_QUEUE
If the queue is invalid.

HSA_STATUS_ERROR_INVALID_ARGUMENT
If *queue* is NULL.

Description

Inactivating the queue aborts any pending executions and prevent any new packets from being processed. Any more packets written to the queue once it is inactivated will be ignored by the packet processor.

2.5.2.7 hsa_queue_load_read_index_relaxed

```
uint64_t hsa_queue_load_read_index_relaxed(
    hsa_queue_t * queue)
```

Atomically load the read index of a queue.

Parameters

queue
(in) Pointer to a queue.

Returns

Read index of the queue pointed by *queue*.

2.5.2.8 hsa_queue_load_read_index_acquire

```
uint64_t hsa_queue_load_read_index_acquire(
    hsa_queue_t * queue)
```

Atomically load the read index of a queue.

Parameters

queue
(in) Pointer to a queue.

Returns

Read index of the queue pointed by *queue*.

2.5.2.9 hsa_queue_load_write_index_relaxed

```
uint64_t hsa_queue_load_write_index_relaxed(
    hsa_queue_t * queue)
```

Atomically load the write index of a queue.

Parameters

queue
(in) Pointer to a queue.

Returns

Write index of the queue pointed by *queue*.

2.5.2.10 hsa_queue_load_write_index_acquire

```
uint64_t hsa_queue_load_write_index_acquire(
    hsa_queue_t * queue)
```

Atomically load the write index of a queue.

Parameters

queue
: (in) Pointer to a queue.

Returns

Write index of the queue pointed by *queue*.

2.5.2.11 hsa_queue_store_write_index_relaxed

```
void hsa_queue_store_write_index_relaxed(
    hsa_queue_t * queue,
    uint64_t value)
```

Atomically set the write index of a queue.

Parameters

queue
: (in) Pointer to a queue.

value
: (in) Value to assign to the write index.

2.5.2.12 hsa_queue_store_write_index_release

```
void hsa_queue_store_write_index_release(
    hsa_queue_t * queue,
    uint64_t value)
```

Atomically set the write index of a queue.

Parameters

queue
: (in) Pointer to a queue.

value
: (in) Value to assign to the write index.

2.5.2.13 hsa_queue_cas_write_index_relaxed

```
uint64_t hsa_queue_cas_write_index_relaxed(
    hsa_queue_t * queue,
    uint64_t expected,
```

```
    uint64_t value)
```

Atomically compare and set the write index of a queue.

Parameters

queue
(in) Pointer to a queue.

expected
(in) The expected index value.

value
(in) Value to assign to the write index if *expected* matches the observed write index.

Returns

Previous value of the write index.

2.5.2.14 hsa_queue_cas_write_index_release

```
uint64_t hsa_queue_cas_write_index_release(
    hsa_queue_t * queue,
    uint64_t expected,
    uint64_t value)
```

Atomically compare and set the write index of a queue.

Parameters

queue
(in) Pointer to a queue.

expected
(in) The expected index value.

value
(in) Value to assign to the write index if *expected* matches the observed write index.

Returns

Previous value of the write index.

2.5.2.15 hsa_queue_cas_write_index_acquire

```
uint64_t hsa_queue_cas_write_index_acquire(
    hsa_queue_t * queue,
    uint64_t expected,
    uint64_t value)
```

Atomically compare and set the write index of a queue.

Parameters

queue

(in) Pointer to a queue.

expected
(in) The expected index value.

value
(in) Value to assign to the write index if *expected* matches the observed write index.

Returns

Previous value of the write index.

2.5.2.16 hsa_queue_cas_write_index_acquire_release

```
uint64_t hsa_queue_cas_write_index_acquire_release(
    hsa_queue_t * queue,
    uint64_t expected,
    uint64_t value)
```

Atomically compare and set the write index of a queue.

Parameters

queue
(in) Pointer to a queue.

expected
(in) The expected index value.

value
(in) Value to assign to the write index if *expected* matches the observed write index.

Returns

Previous value of the write index.

2.5.2.17 hsa_queue_add_write_index_relaxed

```
uint64_t hsa_queue_add_write_index_relaxed(
    hsa_queue_t * queue,
    uint64_t value)
```

Atomically increment the write index of a queue by an offset.

Parameters

queue
(in) Pointer to a queue.

value
(in) Value to add to the write index.

Returns

Previous value of the write index.

2.5.2.18 hsa_queue_add_write_index_acquire

```
uint64_t hsa_queue_add_write_index_acquire(
    hsa_queue_t * queue,
    uint64_t value)
```

Atomically increment the write index of a queue by an offset.

Parameters

queue
: (in) Pointer to a queue.

value
: (in) Value to add to the write index.

Returns

Previous value of the write index.

2.5.2.19 hsa_queue_add_write_index_release

```
uint64_t hsa_queue_add_write_index_release(
    hsa_queue_t * queue,
    uint64_t value)
```

Atomically increment the write index of a queue by an offset.

Parameters

queue
: (in) Pointer to a queue.

value
: (in) Value to add to the write index.

Returns

Previous value of the write index.

2.5.2.20 hsa_queue_add_write_index_acquire_release

```
uint64_t hsa_queue_add_write_index_acquire_release(
    hsa_queue_t * queue,
    uint64_t value)
```

Atomically increment the write index of a queue by an offset.

Parameters

queue
: (in) Pointer to a queue.

value

(in) Value to add to the write index.

Returns
Previous value of the write index.

2.5.2.21 hsa_queue_store_read_index_relaxed

```
void hsa_queue_store_read_index_relaxed(
    hsa_queue_t * queue,
    uint64_t value)
```

Atomically set the read index of a queue.

Parameters

queue
(in) Pointer to a queue.

value
(in) Value to assign to the read index.

2.5.2.22 hsa_queue_store_read_index_release

```
void hsa_queue_store_read_index_release(
    hsa_queue_t * queue,
    uint64_t value)
```

Atomically set the read index of a queue.

Parameters

queue
(in) Pointer to a queue.

value
(in) Value to assign to the read index.

2.6 Architected Queuing Language Packets

The Architected Queuing Language (AQL) is a standard binary interface used to describe commands (such as a dispatch command or a barrier command). An AQL packet is a blob of data with a defined format encoding one command.

This section provides more information about AQL packet types and how the HSA API represents AQL packet types.

2.6.1 Dispatch packet

An application uses a Dispatch packet to submit tasks to an HSA component. The HSA API uses the hsa_-aql_dispatch_packet_t type to represent a dispatch. A Dispatch packet can be in one of the following five states: *queued*, *launch*, *error*, *active* or *complete*. Figure 2.2 shows the state transition diagram.

Queued The queued state means that the format of the packet is not HSA_AQL_PACKET_FORMAT_-ALWAYS_RESERVED nor HSA_AQL_PACKET_FORMAT_INVALID. If the *barrier* bit is set, the transition to launch state occurs after all the preceding kernels have completed execution. If the *barrier* bit is not set, then the transition to launch state occurs after the preceding kernels have completed their launch phase.

Launch The launch state indicates processing of the packet, but not execution of a task. This phase finalizes by applying an acquire memory fence. If an error is detected during launch, the queue is set into an error state and the event callback associated with the queue (if present) is invoked. The following error might be communicated during the launch phase: a) HSA_STATUS_ERROR_INVALID_PACKET_FORMAT, if the AQL packet is malformed.

Active The active state means that the kernel encoded by the packet has started execution. If an error is detected during this phase, a release fence is applied to the packet and its completion signal (if present) is assigned a negative value.

Error The error state means that an error was encountered during the launch or active phases.

Complete The complete state means that a memory release fence has been applied and the completion signal (if present) decremented.

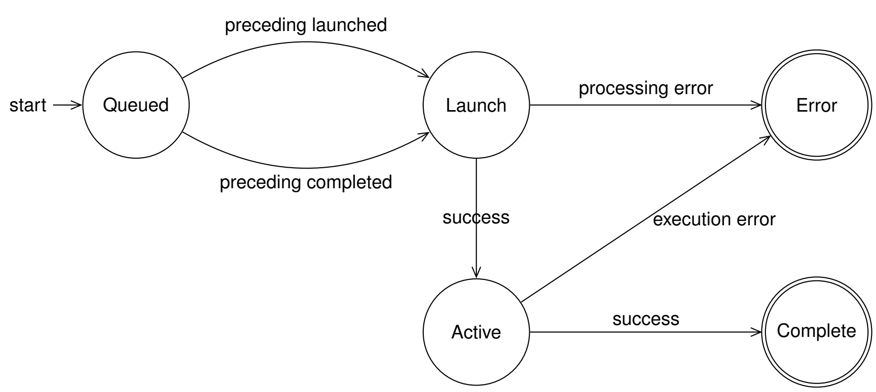

Figure 2.2: Packet State Diagram

2.6.1.1 Segment sizes

If the kernel being dispatched uses private or group segments, the HSA runtime requires the application to specify the sizes of the segments in the Dispatch packet. Manually calculating this information is not feasible and requires visual inspection of the input program, which itself may have been generated by a higher-level compiler. For this reason, the user must rely on the finalizer to get the corresponding segment sizes. For more information on determining segment sizes, refer to Section 3.1.

Another kind of HSA segment is the kernarg segment, used to pass arguments into a kernel. The application populates the *kernarg_address* field of the Dispatch packet with the address of a buffer of kernarg memory previously allocated using **hsa_memory_allocate_kernarg**. The buffer contains the actual parameters passed to the kernel being dispatched.

2.6.2 Agent Dispatch packet

Agent Dispatch packets dispatch jobs to HSA agents. The HSA API defines type hsa_aql_agent_dispatch_-packet_t to represent Agent Dispatch packets. The states and state transitions for Agent Dispatch packets is identical to that of Dispatch packets.

2.6.3 Barrier packet

The Barrier packet hsa_aql_barrier_packet_t allows an application to specify up to five dependencies as hsa_signal_handle_t objects and requires the HSA Packet Processor to resolve those dependencies before proceeding. The Barrier packet is a blocking packet in the sense that the processing of the Barrier packet completes the packet. This is unlike a Dispatch packet whose completion may occur at some future time after the packet has finished processing.

The HSA Packet Processor will not launch any further packets until the Barrier packet is complete. The execution phase for the Barrier packet completes when either of these two conditions have been met:

- Any of the dependent signals have been observed with a negative value after the Barrier packet launched, in which case the HSA Packet Processor assigns an error value to the *completion_signal* .
- All of the dependent signals have been observed with the value 0 after the Barrier packet launched (note that it is not required that all dependent signals are observed to be 0 at the same time).

As stated above, if any of the dependent signals have been assigned a negative value, the Barrier packet will indicate failure in its completion signal. The HSA Packet Processor assigns an error (negative) value to the *completion_signal*. For more information, refer to Section 2.4.

The state diagram of the Barrier packet is identical to that shown for Dispatch packets. However, each state might cause a different set of actions:

- No memory fence is applied in the launch phase.
- In the active phase, instead of executing a kernel, the packet waits for all the conditions to be met. The following errors might be communicated during the active phase: a) HSA_STATUS_ERROR_-SIGNAL_DEPENDENCY, if an error is detected in one of the dependent signals (a negative value is observed).
- No action is performed in the completion phase.

2.6.4 API

2.6.4.1 hsa_aql_packet_format_t

enum **hsa_aql_packet_format_t**

Packet type.

Values

HSA_AQL_PACKET_FORMAT_ALWAYS_RESERVED = 0
: Initial format of a packets when the queue is created. Always reserved packet have never been assigned to the packet processor. From a functional view always reserved packets are equivalent to invalid packets. All queues support this packet format.

HSA_AQL_PACKET_FORMAT_INVALID = 1
: The packet slot has been processed in the past, and has not been reassigned to the packet processor (is available). All queues support this packet format.

HSA_AQL_PACKET_FORMAT_DISPATCH = 2
: Packet used by HSA agents for dispatching jobs to HSA components. Not all queues support packets of this type (see hsa_queue_feature_t).

HSA_AQL_PACKET_FORMAT_BARRIER = 3
: Packet used by HSA agents to delay processing of subsequent packets, and to express complex dependencies between multiple packets. All queues support this packet format.

HSA_AQL_PACKET_FORMAT_AGENT_DISPATCH = 4
: Packet used by HSA agents for dispatching jobs to HSA agents. Not all queues support packets of this type (see hsa_queue_feature_t).

2.6.4.2 hsa_fence_scope_t

enum **hsa_fence_scope_t**

Scope of the memory fence operation associated with a packet.

Values

HSA_FENCE_SCOPE_NONE = 0
: No scope. Only valid for barrier packets.

HSA_FENCE_SCOPE_COMPONENT = 1
: The fence is applied with component scope for the global segment.

HSA_FENCE_SCOPE_SYSTEM = 2
: The fence is applied with system scope for the global segment.

2.6.4.3 hsa_aql_packet_header_s

```
typedef struct hsa_aql_packet_header_s {
    hsa_aql_packet_format_t format : 8;
    uint16_t barrier : 1;
```

```
    hsa_fence_scope_t acquire_fence_scope : 2;
    hsa_fence_scope_t release_fence_scope : 2;
    uint16_t reserved : 3;
} hsa_aql_packet_header_t
```

AQL packet header.

Data Fields

format
Packet type.

barrier
If set, the processing of the current packet only launches when all preceding packets (within the same queue) are complete.

acquire_fence_scope
Determines the scope and type of the memory fence operation applied before the packet enters the active phase.

release_fence_scope
Determines the scope and type of the memory fence operation applied after kernel completion but before the packet is completed.

reserved
Must be a value of 0.

2.6.4.4 hsa_aql_dispatch_packet_s

```
typedef struct hsa_aql_dispatch_packet_s {
    hsa_aql_packet_header_t header;
    uint16_t dimensions : 2;
    uint16_t reserved : 14;
    uint16_t workgroup_size_x;
    uint16_t workgroup_size_y;
    uint16_t workgroup_size_z;
    uint16_t reserved2;
    uint32_t grid_size_x;
    uint32_t grid_size_y;
    uint32_t grid_size_z;
    uint32_t private_segment_size_bytes;
    uint32_t group_segment_size_bytes;
    uint64_t kernel_object_address;
    uint64_t kernarg_address;
    uint64_t reserved3;
    hsa_signal_handle_t completion_signal;
} hsa_aql_dispatch_packet_t
```

AQL dispatch packet.

Data Fields

header
Packet header.

dimensions

Number of dimensions specified in the grid size. Valid values are 1, 2, or 3.

reserved
Reserved, must be a value of 0.

workgroup_size_x
X dimension of work-group (measured in work-items).

workgroup_size_y
Y dimension of work-group (measured in work-items).

workgroup_size_z
Z dimension of work-group (measured in work-items).

reserved2
Reserved. Must be a value of 0.

grid_size_x
X dimension of grid (measured in work-items).

grid_size_y
Y dimension of grid (measured in work-items).

grid_size_z
Z dimension of grid (measured in work-items).

private_segment_size_bytes
Size (in bytes) of private memory allocation request (per work-item).

group_segment_size_bytes
Size (in bytes) of group memory allocation request (per work-group).

kernel_object_address
Address of an object in memory that includes an implementation-defined executable ISA image for the kernel.

kernarg_address
Address of memory containing kernel arguments.

reserved3
Reserved. Must be a value of 0.

completion_signal
Signaling object handle used to indicate completion of the job.

2.6.4.5 hsa_aql_agent_dispatch_packet_s

```
typedef struct hsa_aql_agent_dispatch_packet_s {
    hsa_aql_packet_header_t header;
    uint16_t type;
    uint32_t reserved2;
    uint64_t return_location;
    uint64_t arg[4];
    uint64_t reserved3;
    hsa_signal_handle_t completion_signal;
} hsa_aql_agent_dispatch_packet_t
```

Agent dispatch packet.

Data Fields

header
Packet header.

type
The function to be performed by the destination HSA Agent. The type value is split into the following ranges: 0x0000:0x3FFF (vendor specific), 0x4000:0x7FFF (HSA runtime) 0x8000:0xFFFF (user registered function).

reserved2
Reserved. Must be a value of 0.

return_location
Pointer to location to store the function return value(s) in.

arg
64-bit direct or indirect arguments.

reserved3
Reserved. Must be a value of 0.

completion_signal
Signaling object handle used to indicate completion of the job.

2.6.4.6 hsa_aql_barrier_packet_s

```
typedef struct hsa_aql_barrier_packet_s {
    hsa_aql_packet_header_t header;
    uint16_t reserved2;
    uint32_t reserved3;
    hsa_signal_handle_t dep_signal[5];
    uint64_t reserved4;
    hsa_signal_handle_t completion_signal;
} hsa_aql_barrier_packet_t
```

Barrier packet.

Data Fields

header
Packet header.

reserved2
Reserved. Must be a value of 0.

reserved3
Reserved. Must be a value of 0.

dep_signal
Array of dependent signal objects.

reserved4
Reserved. Must be a value of 0.

completion_signal
Signaling object handle used to indicate completion of the job.

2.7 Memory

One of the key features of HSA is its ability to share global pointers between the host application and code executing on the component. This ability means that an application can directly pass a pointer to memory allocated on the host to a kernel function dispatched to a component without an intermediate copy.

2.7.1 Registration

When a buffer will be accessed by a kernel running on a HSA device, programmers are encouraged to register the corresponding address range beforehand by using the appropriate HSA core API invocation. While kernels running on HSA devices can access any valid system memory pointer allocated by means of standard libraries (for example, malloc in the C language) without resorting to registration, there might be a performance benefit from registering the buffer with the HSA core component. When an HSA program no longer needs to access a registered buffer in a device, the user should deregister that virtual address range by using the appropriate HSA core API invocation.

2.7.2 Global Memory Allocation

While a HSA component is capable of accessing pageable system memory by definition, for scenarios where wants memory allocated that has already been registered (combine the allocation with memory registration), the HSA runtime provides an interface, **hsa_memory_allocate** to allocate memory that is internally registered by the runtime:

2.7.3 Kernarg Memory

The kernarg memory that AQL packet points to (see Section 2.6) holds information about any arguments required to execute AQL dispatch on a HSA component. While any system memory may be used for kernarg memory, implementation/platform specific optimizations are possible if HSA core runtime provided API are utilized for allocating and copying to the allocated kernarg memory.

2.7.4 Component Local Memory

Component local memory is a memory type that is dedicated specifically for a particular HSA component. This memory could provide higher bandwidth for component access (than system memory) with the limitation that the host might not be able to access it directly. HSA runtime provides a host interface to allocate/deallocate and access component local memory. The user should not register or deregister component local memory.

2.7.5 API

2.7.5.1 hsa_memory_register

```
hsa_status_t hsa_memory_register(
    void * address,
    size_t size)
```

Register memory.

Parameters

address
(in) A pointer to the base of the memory region to be registered. If a null pointer is passed, no operation is performed.

size
(in) Requested registration size in bytes. If a size of zero is passed, no operation is performed.

Return Values

HSA_STATUS_SUCCESS
The function has been executed successfully.

HSA_STATUS_ERROR_NOT_INITIALIZED
The runtime has not been initialized.

HSA_STATUS_ERROR_OUT_OF_RESOURCES
If there is a failure in allocating the necessary resources.

Description

Registering a system memory region for use with all the available devices This is an optional interface that is solely provided as a performance optimization hint to the underlying implementation so it may prepare for the future use of the memory by the devices. The interface is only beneficial for system memory that will be directly accessed by a device.

Overlapping registrations are allowed. This is neither detrimental nor beneficial.

2.7.5.2 hsa_memory_deregister

```
hsa_status_t hsa_memory_deregister(
    void * address)
```

Deregister memory.

Parameters

address
(in) A pointer to the base of the memory region to be deregistered. If a NULL pointer is passed, no operation is performed.

Return Values

HSA_STATUS_SUCCESS
The function has been executed successfully.

HSA_STATUS_ERROR_NOT_REGISTERED
If the pointer has not been registered before.

Description

Used for deregistering a memory region previously registered.

Deregistration must be performed using an address that was previously registered. In the event that deregistration is performed on an address that has been used in multiple registrations, the smallest of the registrations is deregistered.

2.7.5.3 hsa_memory_allocate

```
hsa_status_t hsa_memory_allocate(
    size_t size_bytes,
    void ** address)
```

Allocate system memory.

Parameters

size_bytes
(in) Allocation size.

address
(in) Address pointer allocated by the user. Dereferenced and assigned to the pointer to the memory allocated for this request.

Return Values

HSA_STATUS_SUCCESS
The function has been executed successfully.

HSA_STATUS_ERROR_NOT_INITIALIZED
The runtime has not been initialized.

HSA_STATUS_ERROR_OUT_OF_RESOURCES
If there is a failure in allocation. This error may also occur when the core runtime library needs to spawn threads or create internal OS-specific events.

HSA_STATUS_ERROR_INVALID_ARGUMENT
If the passed address is NULL.

Description

The returned buffer is already registered. Allocation of size 0 is allowed and returns a NULL pointer.

2.7.5.4 hsa_memory_free

```
hsa_status_t hsa_memory_free(
    void * ptr)
```

Free system memory.

Parameters

ptr
(in) Pointer to be released. If NULL, no action is performed

Return Values

HSA_STATUS_SUCCESS
The function has been executed successfully.

HSA_STATUS_ERROR_NOT_INITIALIZED
The runtime has not been initialized.

2.7.5.5 hsa_memory_allocate_kernarg

```
hsa_status_t hsa_memory_allocate_kernarg(
    const hsa_agent_t * component,
    size_t size,
    void ** address)
```

Allocate kernarg memory.

Parameters

component
(in) A valid pointer to the component for which the specified amount of kernarg memory is to be allocated.

size
(in) Requested allocation size in bytes. If size is 0, NULL is returned.

address
(out) A valid pointer to the location of where to return the pointer to the base of the allocated region of memory.

Return Values

HSA_STATUS_SUCCESS
The function has been executed successfully.

HSA_STATUS_ERROR_NOT_INITIALIZED
The runtime has not been initialized.

HSA_STATUS_ERROR_INVALID_ARGUMENT
If the passed address is NULL.

2.7.5.6 hsa_memory_free_kernarg

```
hsa_status_t hsa_memory_free_kernarg(
    void * ptr)
```

Free kernarg memory.

Parameters

ptr
(in) Pointer to be released. If NULL, no action is performed

Return Values

HSA_STATUS_SUCCESS
The function has been executed successfully.

HSA_STATUS_ERROR_NOT_INITIALIZED
The runtime has not been initialized.

2.7.5.7 hsa_memory_copy_kernarg_to_system

```
hsa_status_t hsa_memory_copy_kernarg_to_system(
    void * dst,
    const void * src,
```

```
    size_t size)
```

Copy between the system and kernarg segments.

Parameters

dst
(out) A valid pointer to the destination array where the content is to be copied.

src
(in) A valid pointer to the source of data to be copied.

size
(in) Number of bytes to copy.

Return Values

HSA_STATUS_SUCCESS
The function has been executed successfully.

HSA_STATUS_ERROR_NOT_INITIALIZED
The runtime has not been initialized.

HSA_STATUS_ERROR_INVALID_ARGUMENT
If the source or destination pointers are invalid.

2.7.5.8 hsa_memory_copy_system_to_kernarg

```
hsa_status_t hsa_memory_copy_system_to_kernarg(
    void * dst,
    const void * src,
    size_t size)
```

Copy between the system and kernarg segments.

Parameters

dst
(out) A valid pointer to the destination array where the content is to be copied.

src
(in) A valid pointer to the source of data to be copied.

size
(in) Number of bytes to copy.

Return Values

HSA_STATUS_SUCCESS
The function has been executed successfully.

HSA_STATUS_ERROR_NOT_INITIALIZED
The runtime has not been initialized.

HSA_STATUS_ERROR_INVALID_ARGUMENT
If the source or destination pointers are invalid.

2.7.5.9 hsa_memory_allocate_component_local

```
hsa_status_t hsa_memory_allocate_component_local(
    const hsa_agent_t * component,
    size_t size,
    void ** address)
```

Allocate memory on HSA Device.

Parameters

component
: (in) A valid pointer to the HSA device for which the specified amount of global memory is to be allocated.

size
: (in) Requested allocation size in bytes. If size is 0, NULL is returned.

address
: (out) A valid pointer to the location of where to return the pointer to the base of the allocated region of memory.

Return Values

HSA_STATUS_SUCCESS
: The function has been executed successfully.

HSA_STATUS_ERROR_NOT_INITIALIZED
: The runtime has not been initialized.

HSA_STATUS_ERROR_OUT_OF_RESOURCES
: If there is a failure in allocation of an internal structure required by the core runtime library. This error may also occur when the core runtime library needs to spawn threads or create internal OS-specific events.

HSA_STATUS_ERROR_INVALID_ARGUMENT
: If the passed component is NULL or invalid, or if the passed pointer is NULL.

Description

Allocate global device memory associated with specified device.

2.7.5.10 hsa_memory_free_component_local

```
hsa_status_t hsa_memory_free_component_local(
    void * address)
```

Deallocate memory on HSA component.

Parameters

address
: (in) A pointer to the address to be deallocated. If the pointer is NULL, no operation is performed.

Return Values

HSA_STATUS_SUCCESS
: The function has been executed successfully.

HSA_STATUS_ERROR_NOT_INITIALIZED
: The runtime has not been initialized.

Description

Deallocate component memory that was allocated with **hsa_memory_allocate_component_local**.

2.7.5.11 hsa_memory_copy_component_local_to_system

hsa_status_t **hsa_memory_copy_component_local_to_system**(
  void * *dst*,
  const void * *src*,
  size_t *size*,
  hsa_signal_handle_t *signal*)

Copy between the system and local heaps.

Parameters

dst
  (out) A valid pointer to the destination array where the content is to be copied.

src
  (in) A valid pointer to the source of data to be copied.

size
  (in) Number of bytes to copy.

signal
  (in) The signal that will be incremented by the runtime when the copy is complete.

Return Values

HSA_STATUS_SUCCESS
  The function has been executed successfully.

HSA_STATUS_ERROR_NOT_INITIALIZED
  The runtime has not been initialized.

HSA_STATUS_ERROR_OUT_OF_RESOURCES
  If there is a failure in allocation of an internal structure required by the core runtime library. This error may also occur when the core runtime library needs to spawn threads or create internal OS-specific events.

HSA_STATUS_ERROR_INVALID_ARGUMENT
  If any argument is invalid.

2.7.6 Example - memory registration

A buffer is registered by indicating its starting address and a size. The size does not need to match that of the original allocation. For example:

```
void* ptr = malloc(16);
status = hsa_memory_register(ptr, 8);
if(status == HSA_STATUS_ERROR_INVALID_ARGUMENT)
  handle_error(status);
```

is a valid program. On the other hand:

```
void* ptr = malloc(16);
status = hsa_memory_register(ptr, 20);
if(status == HSA_STATUS_ERROR_INVALID_ARGUMENT)
  handle_error(status);
```

is not a valid program, because we are registering a range that spans several allocations, or might not be entirely allocated.

Registrations can overlap previously registered intervals. A special case of overlapped registrations is multiple registration. If the same interval is registered several times with different sizes, the HSA core component will select the maximum as the size of all the registrations. Therefore, the following program:

```
status = hsa_memory_register(ptr, 8);
if(status == HSA_STATUS_ERROR_INVALID_ARGUMENT)
  handle_error(status);
status = hsa_memory_register(ptr, 16);
if(status == HSA_STATUS_ERROR_INVALID_ARGUMENT)
  handle_error(status);
```

behaves identically to this program:

```
hsa_memory_register(ptr, 16);
if(status == HSA_STATUS_ERROR_INVALID_ARGUMENT)
  handle_error(status);
hsa_memory_register(ptr, 16);
if(status == HSA_STATUS_ERROR_INVALID_ARGUMENT)
  handle_error(status);
```

While the described behavior might seem counter-intuitive, consider the following scenario: A pointer is registered twice with different sizes s1 and s2. When the pointer is deregistered, which interval should be deregistered: (p, s1) or (p, s2)? If all the registrations of the same pointer are considered identical by the core runtime, that problem is eliminated.

Deregistering a pointer that has not been previously registered results in an *info* status indicating the same.

The following code snippet revisits the introductory example. The code is almost identical to the original, except that we register the buffers that will be accessed from the device after allocating them, and we deregister all that memory before releasing it. In some platforms, we expect this version to perform better than the original one.

2.7.7 Example - device memory

Component memory is allocated by indicating the size and the HSA device it corresponds to. For example, the following code allocates 1024 bytes of device local memory:

```
void* component_ptr = NULL;
hsa_memory_allocate_component_local(1024, component, &component_ptr);
```

To access component memory from the host, the user can call **hsa_memory_copy_component_local_-to_system** in similar fashion as in memcpy. This interface allows the user to perform component-to-host memory copy. For example:

```
const size_t DATA_SIZE = 1024;
void* src_ptr = malloc(DATA_SIZE);
void* dest_ptr = NULL;
hsa_memory_allocate_component_local(DATA_SIZE, device, &dest_ptr);
hsa_memory_copy_component_local_to_system(dest_ptr, src_ptr, DATA_SIZE);
```

copies 1024 bytes from system to component local memory.

2.8 Extensions to the Core Runtime API

Extensions to the Core API can be optional (multi-vendor) or vendor specific. The difference is in the naming scheme used for the symbols (defines, structures, functions, etc.) associated with the function:

- Symbols for multi-vendor extensions defined in the global namespace must use the *hsa_ext_* prefix in their identifiers.
- Symbols for single vendor extensions defined in the global namespace must use the *hsa_svext_VENDOR_* prefix in their identifiers. Company names must be registered with the HSA Foundation, must be unique, and may be abbreviated to improve the readability of the symbols.

Any constant definitions in the extension (#define or enumeration values) use the same naming convention, except using all capital letters.

The symbols for all vendor extensions (both single-vendor and multi-vendor) are captured in the file **hsa/vendor_extensions.h**. This file is maintained by the HSA Foundation. This file includes the enumeration hsa_extension_t which defines a unique code for each vendor extension and multi-vendor extension. Vendors can reserve enumeration encodings through the HSA Foundation. Multi-vendor enumerations begin at the value of HSA_EXT_START, while single-vendor enumerations begin at HSA_SVEXT_START

2.8.1 API

2.8.1.1 hsa_extension_t

enum **hsa_extension_t**

HSA extensions.

Values

HSA_EXT_START = 0
: Start of the multi vendor extension range.

HSA_EXT_FINALIZER = HSA_EXT_START
: Finalizer extension. Finalizes the brig to compilation units that represent kernel and function code objects.

HSA_EXT_LINKER = 1
: Linker extension.

HSA_EXT_IMAGES = 2
: Images extension.

HSA_SVEXT_START = 10000
: Start of the single vendor extension range.

2.8.1.2 hsa_vendor_extension_query

hsa_status_t **hsa_vendor_extension_query**(
    hsa_extension_t *extension*,
    void * *extension_structure*,

```
    int * result)
```

Query vendor extensions.

Parameters

extension
(in) The vendor extension that is being queried.

extension_structure
(out) Extension structure.

result
(out) Pointer to memory location where to store the query result.

Return Values

HSA_STATUS_SUCCESS
The function has been executed successfully.

HSA_STATUS_ERROR_NOT_INITIALIZED
The runtime has not been initialized.

HSA_STATUS_ERROR_INVALID_ARGUMENT
If *extension* is not a valid value for a single vendor extension or *result* is NULL.

Description

If successful, the extension information is written with extension-specific information such as version information, function pointers, and data values. If the extension is not supported, the extension information is not modified.

2.8.1.3 hsa_extension_query

```
hsa_status_t hsa_extension_query(
    hsa_extension_t extension,
    int * result)
```

Query HSA extensions.

Parameters

extension
(in) The extension that is being queried.

result
(out) Pointer to memory location where to store the query result.

Return Values

HSA_STATUS_SUCCESS
The function has been executed successfully.

HSA_STATUS_ERROR_NOT_INITIALIZED
The runtime has not been initialized.

HSA_STATUS_ERROR_INVALID_ARGUMENT
If *extension* is not a valid value for a HSA extension or *result* is NULL.

2.8.2 Example

An example that shows a hypothetical single-vendor extension “Foo” registered by company “ACME”. The example includes four defines and two API functions. Note the use of the structure hsa_svext_acme_foo_t and how this interacts with the **hsa_vendor_extension_query** API call.

```
// Sample hsa/vendor_extensions.h
// Company name is "ACME" and extension is "Foo"
#define HSA_EXT_ACME_MYDEFINE1 0x1000
#define HSA_EXT_ACME_MYDEFINE2 0x0100

// The structure which defines the version, functions, and data for
// the extension:
typedef struct hsa_ext_acme_foo_s {
    int major_version;  // major version number of the extension.
    int minor_version;  // minor version number of the extension.
    // Function pointers:
    int (*function1) ( int p1, int *p2, float p3, int p4);
    int (*function2) ( int* p1, int p2);
    // Data:
    unsigned foo_data1;
} hsa_ext_acme_foo_t;

main() {
    struct hsa_ext_acme_foo_t acmeFoo;
    hsa_status_t status = hsa_vendor_extension_query(HSA_EXT_ACME_FOO, &acmeFoo);
    if (status == HSA_STATUS_SUCCESS) {
        (*(acmeFoo.function2))(0, 0);
    }
}
```

2.9 Common Definitions

2.9.1 API

2.9.1.1 hsa_powertwo8_t

```
typedef uint8_t hsa_powertwo8_t
```

Value expressed as a power of two.

2.9.1.2 hsa_powertwo_t

```
enum hsa_powertwo_t
```

Power of two between 1 and 256.

Values

HSA_POWERTWO_1 = 0

HSA_POWERTWO_2 = 1

HSA_POWERTWO_4 = 2

HSA_POWERTWO_8 = 3

HSA_POWERTWO_16 = 4

HSA_POWERTWO_32 = 5

HSA_POWERTWO_64 = 6

HSA_POWERTWO_128 = 7

HSA_POWERTWO_256 = 8

2.9.1.3 hsa_dim3_s

```
typedef struct hsa_dim3_s {
    uint32_t x;
    uint32_t y;
    uint32_t z;
} hsa_dim3_t
```

Three-dimensional coordinate.

Data Fields

x
: X dimension.

y
: Y dimension.

z
: Z dimension.

2.9.1.4 hsa_dim_t

```
enum hsa_dim_t
```

Dimensions in a 3D space.

Values

HSA_DIM_X = 0
X dimension.

HSA_DIM_Y = 1
Y dimension.

HSA_DIM_Z = 2
Z dimension.

2.9.1.5 hsa_runtime_caller_s

```
typedef struct hsa_runtime_caller_s {
    uint64_t caller;
} hsa_runtime_caller_t
```

Opaque pointer that is passed to all runtime functions that use callbacks. The runtime passes this pointer as the first argument to all callbacks made by the function.

Data Fields

caller
Opaque pointer that is passed as the first argument to callback functions invoked by a runtime function.

2.9.1.6 hsa_runtime_alloc_data_callback_t

```
typedef hsa_status_t(* hsa_runtime_alloc_data_callback_t)(hsa_runtime_caller_t caller, size_t byte_size, void **address)
```

Call back function for allocating data.

Chapter 3

HSA Extensions Programming Guide

3.1 HSAIL Finalization

Note: The text in this section will be updated according to the API changes introduced in version 0.180 of the specification.

3.1.1 API

3.1.1.1 hsa_ext_brig_profile8_t

```
typedef uint8_t hsa_ext_brig_profile8_t
```

BrigProfile is used to specify the kind of profile. This controls what features of HSAIL are supported. For more information see HSA Programmer's Reference Manual.

3.1.1.2 hsa_ext_brig_profile_t

```
enum hsa_ext_brig_profile_t
```

BRIG profile values.

Values

HSA_EXT_BRIG_PROFILE_BASE = 0
    Base profile.

HSA_EXT_BRIG_PROFILE_FULL = 1
    Full profile.

3.1.1.3 hsa_ext_brig_machine_model8_t

```
typedef uint8_t hsa_ext_brig_machine_model8_t
```

Machine model type. This controls the size of addresses used for segment and flat addresses. For more information see HSA Programmer's Reference Manual.

3.1.1.4 hsa_ext_brig_machine_model_t

enum **hsa_ext_brig_machine_model_t**

BRIG machine model.

Values

HSA_EXT_BRIG_MACHINE_SMALL = 0
Use 32 bit addresses for global segment and flat addresses.

HSA_EXT_BRIG_MACHINE_LARGE = 1
Use 64 bit addresses for global segment and flat addresses.

3.1.1.5 hsa_ext_brig_section_id32_t

typedef uint32_t **hsa_ext_brig_section_id32_t**

BRIG section id. The index into the array of sections in a BRIG module.

3.1.1.6 hsa_ext_brig_section_id_t

enum **hsa_ext_brig_section_id_t**

The fixed BRIG sections ID of the predefined BRIG sections.

Values

HSA_EXT_BRIG_SECTION_DATA = 0
Data section, containing all character strings and byte data used in the finalization unit.

HSA_EXT_BRIG_SECTION_CODE = 1
All of the executable operations. Most operations contain offsets to the .operand section.

HSA_EXT_BRIG_SECTION_OPERAND = 2
The operands, such as immediate constants, registers, and address expressions, that appear in the operations.

3.1.1.7 hsa_ext_brig_section_header_s

```
typedef struct hsa_ext_brig_section_header_s {
    uint32_t byte_count;
    uint32_t header_byte_count;
    uint32_t name_length;
    uint8_t name[1];
} hsa_ext_brig_section_header_t
```

BRIG section header. The first entry in every section must be this hsa_ext_brig_section_header_t structure.

Data Fields

byte_count
Size in bytes of the section.

header_byte_count
Size of the header in bytes.

name_length
Length of *name*

name
Dynamically sized section name.

3.1.1.8 hsa_ext_brig_module_s

```
typedef struct hsa_ext_brig_module_s {
    uint32_t section_count;
    hsa_ext_brig_section_header_t * section[1];
} hsa_ext_brig_module_t
```

Top level BRIG module.

Data Fields

section_count
Number of sections in this BRIG module.

section
Sections in this BRIG module.

3.1.1.9 hsa_ext_brig_module_handle_s

```
typedef struct hsa_ext_brig_module_handle_s {
    uint64_t handle;
} hsa_ext_brig_module_handle_t
```

An opaque handle to the BRIG module.

Data Fields

handle
HSA component specific handle to the brig module.

3.1.1.10 hsa_ext_brig_code_section_offset32_t

typedef uint32_t **hsa_ext_brig_code_section_offset32_t**

BRIG code section offset.

3.1.1.11 hsa_ext_exception_kind16_t

typedef uint16_t **hsa_ext_exception_kind16_t**

The set of exceptions supported by HSAIL. This is represented as a bit set.

3.1.1.12 hsa_ext_exception_kind_t

enum **hsa_ext_exception_kind_t**

HSAIL exceptions.

Values

HSA_EXT_EXCEPTION_INVALID_OPERATION = 1
Operations are performed on values for which the results are not defined. These are:

- Operations on signaling NaN (sNaN) floating-point values.
- Signalling comparisons: comparisons on quiet NaN (qNaN) floating-point values.
- Multiplication: mul(0.0, infinity) or mul(infinity, 0.0).
- Fused multiply add: fma(0.0, infinity, c) or fma(infinity, 0.0, c) unless c is a quiet NaN, in which case it is implementation-defined if an exception is generated.
- Addition, subtraction, or fused multiply add: magnitude subtraction of infinities, such as: add(positive infinity, negative infinity), sub(positive infinity, positive infinity).
- Division: div(0.0, 0.0) or div(infinity, infinity).
- Square root: sqrt(negative).
- Conversion: A cvt with a floating-point source type, an integer destination type, and a nonsaturating rounding mode, when the source value is a NaN, infinity, or the rounded value, after any flush to zero, cannot be represented precisely in the integer type of the destination.

HSA_EXT_EXCEPTION_DIVIDE_BY_ZERO = 2
A finite non-zero floating-point value is divided by zero. It is implementation defined if integer div or rem operations with a divisor of zero will generate a divide by zero exception.

HSA_EXT_EXCEPTION_OVERFLOW = 4
The floating-point exponent of a value is too large to be represented.

HSA_EXT_EXCEPTION_UNDERFLOW = 8
A non-zero tiny floating-point value is computed and either the ftz modifier is specified, or the ftz modifier was not specified and the value cannot be represented exactly.

HSA_EXT_EXCEPTION_INEXACT = 16
A computed floating-point value is not represented exactly in the destination. This can occur due to rounding. In addition, it is implementation defined if operations with the ftz modifier that cause a value to be flushed to zero generate the inexact exception.

3.1.1.13 hsa_ext_control_directive_present64_t

typedef uint64_t **hsa_ext_control_directive_present64_t**

Bit set of control directives supported in HSAIL. See HSA Programmer's Reference Manual description of control directives with the same name for more information. For control directives that have an associated value, the value is given by the field in hsa_ext_control_directives_t. For control directives that are only present of absent (such as requirenopartialworkgroups) they have no corresponding field as the presence of the bit in this mask is sufficient.

3.1.1.14 hsa_ext_control_directive_present_t

enum **hsa_ext_control_directive_present_t**

HSAIL control directives.

Values

HSA_EXT_CONTROL_DIRECTIVE_ENABLE_BREAK_EXCEPTIONS = 0
If not enabled then must be 0, otherwise must be non-0 and specifies the set of HSAIL exceptions that must have the BREAK policy enabled. If this set is not empty then the generated code may have lower performance than if the set is empty. If the kernel being finalized has any enablebreakexceptions control directives, then the values specified by this argument are unioned with the values in these control directives. If any of the functions the kernel calls have an enablebreakexceptions control directive, then they must be equal or a subset of, this union.

HSA_EXT_CONTROL_DIRECTIVE_ENABLE_DETECT_EXCEPTIONS = 1
If not enabled then must be 0, otherwise must be non-0 and specifies the set of HSAIL exceptions that must have the DETECT policy enabled. If this set is not empty then the generated code may have lower performance than if the set is empty. However, an implementation should endeavour to make the performance impact small. If the kernel being finalized has any enabledetectexceptions control directives, then the values specified by this argument are unioned with the values in these control directives. If any of the functions the kernel calls have an enabledetectexceptions control directive, then they must be equal or a subset of, this union.

HSA_EXT_CONTROL_DIRECTIVE_MAX_DYNAMIC_GROUP_SIZE = 2
If not enabled then must be 0, and any amount of dynamic group segment can be allocated for a dispatch, otherwise the value specifies the maximum number of bytes of dynamic group segment that can be allocated for a dispatch. If the kernel being finalized has any maxdynamicsize control directives, then the values must be the same, and must be the same as this argument if it is enabled. This value can be used by the finalizer to determine the maximum number of bytes of group memory used by each work-group by adding this value to the group memory required for all group segment variables used by the kernel and all functions it calls, and group memory used to implement other HSAIL features such as fbarriers and the detect exception operations. This can allow the finalizer to determine the expected number of work-groups that can be executed by a compute unit and allow more resources to be allocated to the work-items if it is known that fewer work-groups can be executed due to group memory limitations.

HSA_EXT_CONTROL_DIRECTIVE_MAX_FLAT_GRID_SIZE = 4
: If not enabled then must be 0, otherwise must be greater than 0. Specifies the maximum number of work-items that will be in the grid when the kernel is dispatched. For more information see HSA Programmer's Reference Manual.

HSA_EXT_CONTROL_DIRECTIVE_MAX_FLAT_WORKGROUP_SIZE = 8
: If not enabled then must be 0, otherwise must be greater than 0. Specifies the maximum number of work-items that will be in the work-group when the kernel is dispatched. For more information see HSA Programmer's Reference Manual.

HSA_EXT_CONTROL_DIRECTIVE_REQUESTED_WORKGROUPS_PER_CU = 16
: If not enabled then must be 0, and the finalizer is free to generate ISA that may result in any number of work-groups executing on a single compute unit. Otherwise, the finalizer should attempt to generate ISA that will allow the specified number of work-groups to execute on a single compute unit. This is only a hint and can be ignored by the finalizer. If the kernel being finalized, or any of the functions it calls, has a requested control directive, then the values must be the same. This can be used to determine the number of resources that should be allocated to a single work-group and work-item. For example, a low value may allow more resources to be allocated, resulting in higher per work-item performance, as it is known there will never be more than the specified number of work-groups actually executing on the compute unit. Conversely, a high value may allocate fewer resources, resulting in lower per work-item performance, which is offset by the fact it allows more work-groups to actually execute on the compute unit.

HSA_EXT_CONTROL_DIRECTIVE_REQUIRED_GRID_SIZE = 32
: If not enabled then all elements for Dim3 must be 0, otherwise every element must be greater than 0. Specifies the grid size that will be used when the kernel is dispatched. For more information see HSA Programmer's Reference Manual.

HSA_EXT_CONTROL_DIRECTIVE_REQUIRED_WORKGROUP_SIZE = 64
: If not enabled then all elements for Dim3 must be 0, and the produced code can be dispatched with any legal work-group range consistent with the dispatch dimensions. Otherwise, the code produced must always be dispatched with the specified work-group range. No element of the specified range must be 0. It must be consistent with required_dimensions and max_flat_workgroup_size. If the kernel being finalized, or any of the functions it calls, has a requiredworkgroupsize control directive, then the values must be the same. Specifying a value can allow the finalizer to optimize work-group id operations, and if the number of work-items in the work-group is less tha the WAVESIZE then barrier operations can be optimized to just a memory fence.

HSA_EXT_CONTROL_DIRECTIVE_REQUIRED_DIM = 128
: If not enabled then must be 0 and the produced kernel code can be dispatched with 1, 2 or 3 dimensions. If enabled then the value is 1..3 and the code produced must only be dispatched with a dimension that matches. Other values are illegal. If the kernel being finalized, or any of the functions it calls, has a requireddimsize control directive, then the values must be the same. This can be used to optimize the code generated to compute the absolute and flat work-group and work-item id, and the dim HSAIL operations.

HSA_EXT_CONTROL_DIRECTIVE_REQUIRE_NO_PARTIAL_WORKGROUPS = 256
: Specifies that the kernel must be dispatched with no partial work-groups. It can be placed in either a kernel or a function code block. This is only a hint and can be ignored by the finalizer.

  It is undefined if the kernel is dispatched with any dimension of the grid size not being an exact multiple of the corresponding dimension of the work-group size.

  A finalizer might be able to generate better code for currentworkgroupsize if it knows there are no partial work-groups, because the result becomes the same as the workgroupsize operation. An HSA component might be able to dispatch a kernel more efficiently if it knows there are no partial work-groups.

The control directive applies to the whole kernel and all functions it calls. It can appear multiple times in a kernel or function. If it appears in a function (including external functions), then it must also appear in all kernels that call that function (or have been specified when the finalizer was invoked), either directly or indirectly.

If require no partial work-groups is specified when the finalizer is invoked, the kernel behaves as if the requirenopartialworkgroups control directive has been specified.

require_no_partial_work_groups does not have a field since having the bit set in enabledControlDirectives indicates that the cpntrol directive is present.

3.1.1.15 hsa_ext_control_directives_s

```
typedef struct hsa_ext_control_directives_s {
    hsa_ext_control_directive_present64_t enabled_control_directives;
    hsa_ext_exception_kind16_t enable_break_exceptions;
    hsa_ext_exception_kind16_t enable_detect_exceptions;
    uint32_t max_dynamic_group_size;
    uint32_t max_flat_grid_size;
    uint32_t max_flat_workgroup_size;
    uint32_t requested_workgroups_per_cu;
    hsa_dim3_t required_grid_size;
    hsa_dim3_t required_workgroup_size;
    uint8_t required_dim;
    uint8_t reserved[75];
} hsa_ext_control_directives_t
```

The hsa_ext_control_directives_t specifies the values for the HSAIL control directives. These control how the finalizer generates code. This struct is used both as an argument to hsaFinalizeKernel to specify values for the control directives, and is used in HsaKernelCode to record the values of the control directives that the finalize used when generating the code which either came from the finalizer argument or explicit HSAIL control directives. See the definition of the control directives in HSA Programmer's Reference Manual which also defines how the values specified as finalizer arguments have to agree with the control directives in the HSAIL code.

Data Fields

enabled_control_directives
This is a bit set indicating which control directives have been specified. If the value is 0 then there are no control directives specified and the rest of the fields can be ignored. The bits are accessed using the hsa_ext_control_directives_present_mask_t. Any control directive that is not enabled in this bit set must have the value of all 0s.

enable_break_exceptions
If enableBreakExceptions is not enabled then must be 0, otherwise must be non-0 and specifies the set of HSAIL exceptions that must have the BREAK policy enabled. If this set is not empty then the generated code may have lower performance than if the set is empty. If the kernel being finalized has any enablebreakexceptions control directives, then the values specified by this argument are unioned with the values in these control directives. If any of the functions the kernel calls have an enablebreakexceptions control directive, then they must be equal or a subset of, this union.

enable_detect_exceptions

If enableDetectExceptions is not enabled then must be 0, otherwise must be non-0 and specifies the set of HSAIL exceptions that must have the DETECT policy enabled. If this set is not empty then the generated code may have lower performance than if the set is empty. However, an implementation should endeavour to make the performance impact small. If the kernel being finalized has any enabledetectexceptions control directives, then the values specified by this argument are unioned with the values in these control directives. If any of the functions the kernel calls have an enabledetectexceptions control directive, then they must be equal or a subset of, this union.

max_dynamic_group_size

If maxDynamicGroupSize is not enabled then must be 0, and any amount of dynamic group segment can be allocated for a dispatch, otherwise the value specifies the maximum number of bytes of dynamic group segment that can be allocated for a dispatch. If the kernel being finalized has any maxdynamicsize control directives, then the values must be the same, and must be the same as this argument if it is enabled. This value can be used by the finalizer to determine the maximum number of bytes of group memory used by each work-group by adding this value to the group memory required for all group segment variables used by the kernel and all functions it calls, and group memory used to implement other HSAIL features such as fbarriers and the detect exception operations. This can allow the finalizer to determine the expected number of work-groups that can be executed by a compute unit and allow more resources to be allocated to the work-items if it is known that fewer work-groups can be executed due to group memory limitations.

max_flat_grid_size

If maxFlatGridSize is not enabled then must be 0, otherwise must be greater than 0. See HSA Programmer's Reference Manual description of maxflatgridsize control directive.

max_flat_workgroup_size

If maxFlatWorkgroupSize is not enabled then must be 0, otherwise must be greater than 0. See HSA Programmer's Reference Manual description of maxflatworkgroupsize control directive.

requested_workgroups_per_cu

If requestedWorkgroupsPerCu is not enabled then must be 0, and the finalizer is free to generate ISA that may result in any number of work-groups executing on a single compute unit. Otherwise, the finalizer should attempt to generate ISA that will allow the specified number of work-groups to execute on a single compute unit. This is only a hint and can be ignored by the finalizer. If the kernel being finalized, or any of the functions it calls, has a requested control directive, then the values must be the same. This can be used to determine the number of resources that should be allocated to a single work-group and work-item. For example, a low value may allow more resources to be allocated, resulting in higher per work-item performance, as it is known there will never be more than the specified number of work-groups actually executing on the compute unit. Conversely, a high value may allocate fewer resources, resulting in lower per work-item performance, which is offset by the fact it allows more work-groups to actually execute on the compute unit.

required_grid_size

If not enabled then all elements for Dim3 must be 0, otherwise every element must be greater than 0. See HSA Programmer's Reference Manual description of requiredgridsize control directive.

required_workgroup_size

If requiredWorkgroupSize is not enabled then all elements for Dim3 must be 0, and the produced code can be dispatched with any legal work-group range consistent with the dispatch dimensions. Otherwise, the code produced must always be dispatched with the specified work-group range. No element of the specified range must be 0. It must be consistent with required_dimensions and max_flat_workgroup_size. If the kernel being finalized, or any of the functions it calls, has a requiredworkgroupsize control directive, then the values must be the same. Specifying a value can allow the finalizer to optimize work-group id operations, and if the number of work-items in the work-group is less than the WAVESIZE then barrier operations can be optimized to just a memory fence.

required_dim

If requiredDim is not enabled then must be 0 and the produced kernel code can be dispatched with 1, 2 or 3 dimensions. If enabled then the value is 1..3 and the code produced must only be dispatched with a dimension that matches. Other values are illegal. If the kernel being finalized, or any of the functions it calls, has a requireddimsize control directive, then the values must be the same. This can be used to optimize the code generated to compute the absolute and flat work-group and work-item id, and the dim HSAIL operations.

reserved
Reserved. Must be 0.

3.1.1.16 hsa_ext_code_kind32_t

typedef uint32_t **hsa_ext_code_kind32_t**

The kinds of code objects that can be contained in hsa_ext_code_descriptor_t.

3.1.1.17 hsa_ext_code_kind_t

enum **hsa_ext_code_kind_t**

Type of code object.

Values

HSA_EXT_CODE_NONE = 0
Not a code object.

HSA_EXT_CODE_KERNEL = 1
HSAIL kernel that can be used with an AQL dispatch packet.

HSA_EXT_CODE_INDIRECT_FUNCTION = 2
HSAIL indirect function.

HSA_EXT_CODE_RUNTIME_FIRST = 0x40000000
HSA runtime code objects. For example, partially linked code objects.

HSA_EXT_CODE_RUNTIME_LAST = 0x7fffffff

HSA_EXT_CODE_VENDOR_FIRST = 0x80000000
Vendor specific code objects.

HSA_EXT_CODE_VENDOR_LAST = 0xffffffff

3.1.1.18 hsa_ext_program_call_convention_id32_t

typedef uint32_t **hsa_ext_program_call_convention_id32_t**

Program call convention.

3.1.1.19 hsa_ext_program_call_convention_id_t

```
enum hsa_ext_program_call_convention_id_t
```

Types of program call conventions.

Values

HSA_EXT_PROGRAM_CALL_CONVENTION_FINALIZER_DETERMINED = -1

3.1.1.20 hsa_ext_code_handle_s

```
typedef struct hsa_ext_code_handle_s {
    uint64_t handle;
} hsa_ext_code_handle_t
```

An opaque handle to the code object.

Data Fields

handle
HSA component specific handle to the code.

3.1.1.21 hsa_ext_debug_information_handle_s

```
typedef struct hsa_ext_debug_information_handle_s {
    uint64_t handle;
} hsa_ext_debug_information_handle_t
```

An opaque handle to the debug information.

Data Fields

handle
HSA component specific handle to the debug information.

3.1.1.22 hsa_ext_code_descriptor_s

```
typedef struct hsa_ext_code_descriptor_s {
    hsa_ext_code_kind32_t code_type;
    uint32_t workgroup_group_segment_byte_size;
    uint64_t kernarg_segment_byte_size;
    uint32_t workitem_private_segment_byte_size;
    uint32_t workgroup_fbarrier_count;
    hsa_ext_code_handle_t code;
    hsa_powertwo8_t kernarg_segment_alignment;
    hsa_powertwo8_t group_segment_alignment;
    hsa_powertwo8_t private_segment_alignment;
    hsa_powertwo8_t wavefront_size;
    hsa_ext_program_call_convention_id32_t program_call_convention;
    hsa_ext_brig_module_handle_t module;
    hsa_ext_brig_code_section_offset32_t symbol;
```

```
    hsa_ext_brig_profile8_t hsail_profile;
    hsa_ext_brig_machine_model8_t hsail_machine_model;
    uint16_t reserved1;
    hsa_ext_debug_information_handle_t debug_information;
    char agent_vendor[24];
    char agent_name[24];
    uint32_t hsail_version_major;
    uint32_t hsail_version_minor;
    uint64_t reserved2;
    hsa_ext_control_directives_t control_directive;
} hsa_ext_code_descriptor_t
```

hsa_ext_code_descriptor_t is the descriptor for the code object produced by the Finalizer and contains information that applies to all code entities in the program.

Data Fields

code_type
Type of code object.

workgroup_group_segment_byte_size
The amount of group segment memory required by a work-group in bytes. This does not include any dynamically allocated group segment memory that may be added when the kernel is dispatched.

kernarg_segment_byte_size
The size in bytes of the kernarg segment that holds the values of the arguments to the kernel.

workitem_private_segment_byte_size
The amount of memory required for the combined private, spill and arg segments for a work-item in bytes.

workgroup_fbarrier_count
Number of fbarrier's used in the kernel and all functions it calls. If the implementation uses group memory to allocate the fbarriers then that amount must already be included in the workgroupGroupSegmentByteSize total.

code
Opaque handle to code object.

kernarg_segment_alignment
The maximum byte alignment of variables used by the kernel in the kernarg memory segment. Expressed as a power of two. Must be at least HSA_POWERTWO_16

group_segment_alignment
The maximum byte alignment of variables used by the kernel in the group memory segment. Expressed as a power of two. Must be at least HSA_POWERTWO_16

private_segment_alignment
The maximum byte alignment of variables used by the kernel in the private memory segment. Expressed as a power of two. Must be at least HSA_POWERTWO_16

wavefront_size
Wavefront size expressed as a power of two. Must be a power of 2 in range 1..64 inclusive. Used to support runtime query that obtains wavefront size, which may be used by application to allocated dynamic group memory and set the dispatch work-group size.

program_call_convention
Program call convention id this code descriptor holds.

module

BRIG module handle this code descriptor associated with.

symbol
BRIG directive offset this code descriptor associated with.

hsail_profile
The HSAIL profile defines which features are used. This information is from the HSAIL version directive. If this hsa_ext_code_descriptor_t is not generated from an **hsa_ext_finalize** then must still indicate what profile is being used.

hsail_machine_model
The HSAIL machine model gives the address sizes used by the code. This information is from the HSAIL version directive. If this hsa_ext_code_descriptor_t is not generated from an **hsa_ext_finalize** then must still indicate for what machine mode the code is generated.

reserved1
Reserved for BRIG target options if any are defined in the future. Must be 0.

debug_information
Opaque handle to debug information.

agent_vendor
The vendor of the HSA Component on which this Kernel Code object can execute. ISO/IEC 624 character encoding must be used. If the name is less than 24 characters then remaining characters must be set to 0.

agent_name
The vendor's name of the HSA Component on which this Kernel Code object can execute. ISO/IEC 624 character encoding must be used. If the name is less than 24 characters then remaining characters must be set to 0.

hsail_version_major
The HSAIL major version. This information is from the HSAIL version directive. If this hsa_ext_code_descriptor_t is not generated from an **hsa_ext_finalize** then must be 0.

hsail_version_minor
The HSAIL minor version. This information is from the HSAIL version directive. If this hsa_ext_code_descriptor_t is not generated from an **hsa_ext_finalize** then must be 0.

reserved2
Reserved. Must be 0.

control_directive
The values should be the actually values used by the finalizer in generating the code. This may be the union of values specified as finalizer arguments and explicit HSAIL control directives. If the finalizer chooses to ignore a control directive, and not generate constrained code, then the control directive should not be marked as enabled even though it was present in the HSAIL or finalizer argument. The values are intended to reflect the constraints that the code actually requires to correctly execute, not the values that were actually specified at finalize time.

3.1.1.23 hsa_ext_finalization_request_s

```
typedef struct hsa_ext_finalization_request_s {
    hsa_ext_brig_module_handle_t module;
    hsa_ext_brig_code_section_offset32_t symbol;
    hsa_ext_program_call_convention_id32_t program_call_convention;
} hsa_ext_finalization_request_t
```

Finalization request. Contains hsa_ext_brig_module_handle_t which points to the hsa_ext_brig_module_t to be finalized, as well as the desired call convention to use when finalizing given BRIG module.

Data Fields

module
Handle to the hsa_ext_brig_module_t, which needs to be finalized.

symbol
BRIG code section offset.

program_call_convention
Desired program call convention.

3.1.1.24 hsa_ext_finalization_descriptor_s

```
typedef struct hsa_ext_finalization_descriptor_s {
    uint32_t code_descriptor_count;
    uint32_t reserved1;
    hsa_ext_code_descriptor_t code_descriptors[1];
} hsa_ext_finalization_descriptor_t
```

Finalization descriptor is the descriptor for the code object produced by the Finalizer and contains information that applies to all code entities in the program.

Data Fields

code_descriptor_count
Number of code descriptors produced.

reserved1
Reserved. Must be 0.

code_descriptors
Dynamically sized array of code descriptors.

3.1.1.25 hsa_ext_symbol_definition_callback_t

```
typedef hsa_status_t(* hsa_ext_symbol_definition_callback_t)(hsa_runtime_caller_t caller, hsa_ext_brig_module_handle_t module, hsa_ext_brig_code_section_offset32_t symbol, hsa_ext_brig_module_handle_t *definition_module, hsa_ext_brig_module_t *definition_module_brig, hsa_ext_brig_code_section_offset32_t *definition_symbol)
```

Call back function to get the definition of a module scope variable/fbarrier or kernel/function.

3.1.1.26 hsa_ext_symbol_address_callback_t

typedef hsa_status_t(* **hsa_ext_symbol_address_callback_t**)(hsa_runtime_caller_t caller, hsa_ext_brig_module_handle_t module, hsa_ext_brig_code_section_offset32_t symbol, uint64_t *symbol_address)

Call back function to get the address of global segment variables, kernel table variable, indirect function table variable.

3.1.1.27 hsa_ext_error_message_callback_t

typedef hsa_status_t(* **hsa_ext_error_message_callback_t**)(hsa_runtime_caller_t caller, hsa_ext_brig_module_handle_t module, hsa_ext_brig_code_section_offset32_t statement, uint32_t indent_level, const char *message)

Call back function to get the string representation of the error message.

3.1.1.28 hsa_ext_finalize

hsa_status_t **hsa_ext_finalize**(
    hsa_runtime_caller_t *caller*,
    hsa_agent_t * *agent*,
    uint32_t *program_agent_id*,
    uint32_t *program_agent_count*,
    size_t *finalization_request_count*,
    hsa_ext_finalization_request_t * *finalization_request_list*,
    hsa_ext_control_directives_t * *control_directives*,
    hsa_ext_symbol_definition_callback_t *symbol_definition_callback*,
    hsa_ext_symbol_address_callback_t *symbol_address_callback*,
    hsa_ext_error_message_callback_t *error_message_callback*,
    uint8_t *optimization_level*,
    const char * *options*,
    int *debug_information*,
    hsa_ext_finalization_descriptor_t ** *finalization_descriptor*)

Finalizes provided BRIG modules.

Parameters

caller
    (in) Opaque pointer which is passed to all call back functions made by this call of the finalizer.

agent
    (in) The HSA agent for which code must be produced.

program_agent_id
    (in) Program agent id.

program_agent_count
    (in) Number of program agents.

finalization_request_count

(in) The number of kernels and indirect functions that are in HSAIL modules in HSAIL program.

finalization_request_list
(in) List of kernels and indirect functions that are in HSAIL modules in HSAIL program.

control_directives
(in) The control directives that can be specified to influence how the finalizer generates code. If NULL then no control directives are used. If this call is successful and control_directives is not NULL, then the resulting hsa_ext_code_descriptor_t object will have control directives which were used by the finalizer.

symbol_definition_callback
(in) Call back function to get the definition of a module scope variable/fbarrier or kernel/function.

symbol_address_callback
(in) Call back function to get the address of global segment variables, kernel table variables, indirect function table variable.

error_message_callback
(in) Call back function to get the string representation of the error message.

optimization_level
(in) An implementation defined value that control the level of optimization performed by the finalizer.

options
(in) Implementation defined options that can be specified to the finalizer.

debug_information
(in) The flag for including/excluding the debug information for *finalization_descriptor*. 0 - exclude debug information, 1 - include debug information.

finalization_descriptor
(out) the descriptor for the code object produced by the Finalizer and contains information that applies to all code entities in the program.

Return Values

HSA_STATUS_SUCCESS
The function has been executed successfully.

HSA_EXT_STATUS_ERROR_DIRECTIVE_MISMATCH
If the directive in the control directive structure and in the HSAIL kernel mismatch or if the same directive is used with a different value in one of the functions used by this kernel.

HSA_STATUS_ERROR_INVALID_ARGUMENT
If *finalization_request_list* is NULL or invalid.

HSA_STATUS_ERROR_OUT_OF_RESOURCES
If the finalize API cannot allocate memory for *finalization_descriptor*.

HSA_EXT_STATUS_INFO_UNRECOGNIZED_OPTIONS
If the options are not recognized, no error is returned, just an info status is used to indicate invalid options.

Description

Invokes the finalizer on the provided list of kernels and indirect functions that are in HSAIL modules in HSAIL program.

3.1.1.29 hsa_ext_destroy_finalization_descriptor

```
hsa_status_t hsa_ext_destroy_finalization_descriptor(
    hsa_ext_finalization_descriptor_t * finalization_descriptor)
```

Destroys the finalization descriptor.

Parameters

finalization_descriptor
(in) A pointer to the finalization descriptor that needs to be destroyed.

Return Values

HSA_STATUS_SUCCESS
The function has been executed successfully.

HSA_STATUS_ERROR_INVALID_ARGUMENT
If *finalization_descriptor* is NULL or does not point to a valid finalization descriptor object.

HSA_STATUS_ERROR_RESOURCE_FREE
If some of the resources consumed during initialization by the runtime could not be freed.

3.1.1.30 hsa_ext_serialize_finalization_descriptor

```
hsa_status_t hsa_ext_serialize_finalization_descriptor(
    hsa_runtime_caller_t caller,
    hsa_agent_t * agent,
    hsa_ext_finalization_descriptor_t * finalization_descriptor,
    hsa_runtime_alloc_data_callback_t alloc_serialize_data_callback,
    hsa_ext_error_message_callback_t error_message_callback,
    int debug_information,
    void * serialized_object)
```

Serializes the finalization descriptor.

Parameters

caller
(in) Opaque pointer and will be passed to all call back functions made by this call.

agent
(in) The HSA agent for which *finalization_descriptor* must be serialized.

finalization_descriptor
(in) Finalization descriptor to serialize.

alloc_serialize_data_callback
(in) Call back function for allocation.

error_message_callback
(in) Call back function to get the string representation of the error message.

debug_information
(in) The flag for including/excluding the debug information for *finalization_descriptor*. 0 - exclude debug information, 1 - include debug information.

serialized_object
(out) Pointer to the serialized object.

Return Values

HSA_STATUS_SUCCESS

The function has been executed successfully.

HSA_STATUS_ERROR_INVALID_ARGUMENT
If *finalization_descriptor* is either NULL or does not point to a valid finalization descriptor object.

HSA_STATUS_ERROR_OUT_OF_RESOURCES
If no memory can be allocated for *serialized_object*.

Description

Serializes finalization descriptor for specified *agent*. The caller can set *debug_information* to 1 in order to include debug information of this finalization descriptor in the serialized object.

3.1.1.31 hsa_ext_deserialize_finalization_descriptor

```
hsa_status_t hsa_ext_deserialize_finalization_descriptor(
    hsa_runtime_caller_t caller,
    void * serialized_object,
    hsa_agent_t * agent,
    uint32_t program_agent_id,
    uint32_t program_agent_count,
    hsa_ext_symbol_address_callback_t symbol_address_callback,
    hsa_ext_error_message_callback_t error_message_callback,
    int debug_information,
    hsa_ext_finalization_descriptor_t ** finalization_descriptor)
```

Deserializes the finalization descriptor.

Parameters

caller
(in) Opaque pointer and will be passed to all call back functions made by this call.

serialized_object
(in) Serialized object to be deserialized.

agent
(in) The HSA agent for which *finalization_descriptor* must be serialized.

program_agent_id
(in) TODO.

program_agent_count
(in) TODO.

symbol_address_callback
(in) Call back function to get the address of global segment variables, kernel table variables, indirect function table variable.

error_message_callback
(in) Call back function to get the string representation of the error message.

debug_information
(in) The flag for including/excluding the debug information for *finalization_descriptor*. 0 - exclude debug information, 1 - include debug information.

finalization_descriptor
(out) Deserialized finalization descriptor.

Return Values

HSA_STATUS_SUCCESS
The function has been executed successfully.

HSA_STATUS_ERROR_INVALID_ARGUMENT
If *serialized_object* is either NULL, or is not valid, or the size is 0.

HSA_STATUS_ERROR_OUT_OF_RESOURCES
If no memory can be allocated for *finalization_descriptor*.

Description

Deserializes finalization descriptor for specified *agent*. The caller can set *debug_information* to 1 in order to include debug information of this finalization descriptor from the serialized object.

3.2 HSAIL Linking

Note: The text in this section will be updated according to the API changes introduced in version 0.180 of the specification.

3.2.1 API

3.2.1.1 hsa_ext_program_handle_s

```
typedef struct hsa_ext_program_handle_s {
    uint64_t handle;
} hsa_ext_program_handle_t
```

An opaque handle to the HSAIL program. Created by **hsa_ext_program_create**, and destroyed by **hsa_ext_program_destroy**.

Data Fields

handle
HSA component specific handle to the program.

3.2.1.2 hsa_ext_program_create

```
hsa_status_t hsa_ext_program_create(
    hsa_agent_t * agents,
    uint32_t agent_count,
    hsa_ext_brig_machine_model8_t machine_model,
    hsa_ext_brig_profile8_t profile,
    hsa_ext_program_handle_t * program)
```

Creates an HSAIL program.

Parameters

agents
(in) One or more HSA agent for which this HSAIL program is created.

agent_count
(in) Number of HSA agents for which this HSAIL program is created.

machine_model
(in) The kind of machine model this HSAIL program is created for.

profile
(in) The kind of profile this HSAIL program is created for.

program
(out) A valid pointer to a program handle for the HSAIL program created.

Return Values

HSA_STATUS_SUCCESS
The function has been executed successfully.

HSA_STATUS_ERROR_INVALID_ARGUMENT
If *agent* is NULL, or not valid. If *agent_count* is 0. If *machine_model* is not valid. If *profile* is not valid.

HSA_EXT_STATUS_INFO_ALREADY_INITIALIZED
If *program* is already a valid program.

Description

Creates an HSAIL program for specified *agent_count* of *agents*, with specified BRIG machine model *machine_-model* and BRIG profile *profile*. Returns a handle to the created HSAIL program, and hsa_status_t, which describes the status of execution of this function. There should be at least one agent specified, and *machine_model* and *profile* have to be valid hsa_ext_brig_profile_t and hsa_ext_brig_machine_model_t, otherwise returns HSA_STATUS_ERROR_INVALID_ARGUMENT. If the program handle *program* is already a valid program, HSA_EXT_STATUS_INFO_ALREADY_INITIALIZED is returned.

3.2.1.3 hsa_ext_program_destroy

```
hsa_status_t hsa_ext_program_destroy(
    hsa_ext_program_handle_t program)
```

Destroys an HSAIL program.

Parameters

program
(in) Program handle for the HSAIL program to be destroyed.

Return Values

HSA_STATUS_SUCCESS
The function has been executed successfully.

HSA_STATUS_ERROR_INVALID_ARGUMENT
If *program* is not a valid hsa_ext_program_handle_t object.

HSA_STATUS_ERROR_RESOURCE_FREE
If *program* is already destroyed or has never been created.

Description

Destroys an HSAIL program pointed to by program handle *program*. Returns hsa_status_t, which describes the status of execution of this function. HSAIL program handle *program* has to be a valid hsa_ext_program_-handle_t object, otherwise HSA_STATUS_ERROR_INVALID_ARGUMENT is returned. If the program handle *program* is already destroyed or has never been created HSA_STATUS_ERROR_RESOURCE_FREE is returned.

3.2.1.4 hsa_ext_add_module

```
hsa_status_t hsa_ext_add_module(
    hsa_ext_program_handle_t program,
    hsa_ext_brig_module_t * brig_module,
    hsa_ext_brig_module_handle_t * module)
```

Adds an existing BRIG module to an existing HSAIL program.

Parameters

program
(in) Program handle for the HSAIL program.

brig_module

(in) BRIG module to add to the HSAIL program.

module
(out) The handle for the *brig_module*.

Return Values

HSA_STATUS_SUCCESS
The function has been executed successfully.

3.2.1.5 hsa_ext_finalize_program

hsa_status_t **hsa_ext_finalize_program**(
hsa_ext_program_handle_t *program*,
hsa_agent_t * *agent*,
size_t *finalization_request_count*,
hsa_ext_finalization_request_t * *finalization_request_list*,
hsa_ext_control_directives_t * *control_directives*,
hsa_ext_error_message_callback_t *error_message_callback*,
uint8_t *optimization_level*,
const char * *options*,
int *debug_information*)

Finalizes provided BRIG modules.

Parameters

program
(in) Handle to the program.

agent
(in) The HSA agent for which code must be produced.

finalization_request_count
(in) The number of kernels and indirect functions that are in HSAIL modules in HSAIL program.

finalization_request_list
(in) List of kernels and indirect functions that are in HSAIL modules in HSAIL program.

control_directives
(in) The control directives that can be specified to influence how the finalizer generates code. If NULL then no control directives are used. If this call is successful and control_directives is not NULL, then the resulting hsa_ext_code_descriptor_t object will have control directives which were used by the finalizer.

error_message_callback
(in) Call back function to get the string representation of the error message.

optimization_level
(in) An implementation defined value that control the level of optimization performed by the finalizer.

options
(in) Implementation defined options that can be specified to the finalizer.

debug_information
(in) The flag for including/excluding the debug information for *finalization_descriptor*. 0 - exclude debug information, 1 - include debug information.

Return Values

HSA_STATUS_SUCCESS

The function has been executed successfully.

HSA_EXT_STATUS_ERROR_DIRECTIVE_MISMATCH
If the directive in the control directive structure and in the HSAIL kernel mismatch or if the same directive is used with a different value in one of the functions used by this kernel.

HSA_STATUS_ERROR_INVALID_ARGUMENT
If *finalization_request_list* is NULL or invalid.

HSA_EXT_STATUS_INFO_UNRECOGNIZED_OPTIONS
If the options are not recognized, no error is returned, just an info status is used to indicate invalid options.

Description

Provides and services call backs to core Finalizer to manage looking up global segment variable allocation and variable/function/fbarrier definitions. Takes the result of core Finalizer and updates kernel and indirect function table variables. Done as atomic store release to system scope so ldi_acq and ldk_acq can synchronize with the update. Other query operations must be used to get code address of kernels/indirect functions finalized.

3.2.1.6 hsa_ext_query_program_agent_id

```
hsa_status_t hsa_ext_query_program_agent_id(
    hsa_ext_program_handle_t program,
    hsa_agent_t * agent,
    uint32_t * program_agent_id)
```

Queries program agent's id.

Parameters

program
(in) Program to query agent's id from.

agent
(in) Agent to query agent's id from.

program_agent_id
(out) Program agent's id.

Return Values

HSA_STATUS_SUCCESS
The function has been executed successfully.

HSA_STATUS_ERROR_INVALID_ARGUMENT
If provided *program* or *agent* is invalid.

3.2.1.7 hsa_ext_query_program_agent_count

```
hsa_status_t hsa_ext_query_program_agent_count(
    hsa_ext_program_handle_t program,
    uint32_t * program_agent_count)
```

Queries program agent count.

Parameters

program
(in) Program to query agent count from.

program_agent_count
(out) Number of agents in the program.

Return Values

HSA_STATUS_SUCCESS
The function has been executed successfully.

HSA_STATUS_ERROR_INVALID_ARGUMENT
If provided *program* is invalid.

3.2.1.8 hsa_ext_query_program_agents

```
hsa_status_t hsa_ext_query_program_agents(
    hsa_ext_program_handle_t program,
    uint32_t program_agent_count,
    hsa_agent_t * agents)
```

Queries program agents.

Parameters

program
(in) Program to query agents from.

program_agent_count
(in) Number of agents to query.

agents
(out) HSA program agents.

Return Values

HSA_STATUS_SUCCESS
The function has been executed successfully.

HSA_STATUS_ERROR_INVALID_ARGUMENT
If provided *program* is invalid.

Description

Queries *program_agent_count* number of agents.

3.2.1.9 hsa_ext_query_program_module_count

```
hsa_status_t hsa_ext_query_program_module_count(
    hsa_ext_program_handle_t program,
    uint32_t * program_module_count)
```

Queries program module count.

Parameters

program

(in) Program to query module count from.

program_module_count
(out) Number of modules in the program.

Return Values

HSA_STATUS_SUCCESS
The function has been executed successfully.

HSA_STATUS_ERROR_INVALID_ARGUMENT
If provided *program* is invalid.

3.2.1.10 hsa_ext_query_program_modules

```
hsa_status_t hsa_ext_query_program_modules(
    hsa_ext_program_handle_t program,
    uint32_t program_module_count,
    hsa_ext_brig_module_handle_t * modules)
```

Queries program modules.

Parameters

program
(in) Program to query modules from.

program_module_count
(in) Number of module to query.

modules
(out) Queried modules.

Return Values

HSA_STATUS_SUCCESS
The function has been executed successfully.

HSA_STATUS_ERROR_INVALID_ARGUMENT
If provided *program* is invalid.

Description

Queries *program_module_count* number of modules.

3.2.1.11 hsa_ext_query_program_brig_module

```
hsa_status_t hsa_ext_query_program_brig_module(
    hsa_ext_program_handle_t program,
    hsa_ext_brig_module_handle_t module,
    hsa_ext_brig_module_t * brig_module)
```

Queries program brig modules.

Parameters

program
(in) Program to query module from.

module
(in) Module handle.

brig_module
(out) Queried module.

Return Values

HSA_STATUS_SUCCESS
The function has been executed successfully.

HSA_STATUS_ERROR_INVALID_ARGUMENT
If provided *program* is invalid, or *module* is invalid.

Description

Query a program brig module with specified module handle.

3.2.1.12 hsa_ext_query_call_convention

```
hsa_status_t hsa_ext_query_call_convention(
    hsa_ext_program_handle_t program,
    hsa_agent_t * agent,
    hsa_ext_program_call_convention_id32_t * first_call_convention_id,
    uint32_t * call_convention_count)
```

Queries call convention.

Parameters

program
(in) program Program to query module for.

agent
(in) HSA Agent to query call convention for.

first_call_convention_id
(out) First call convention.

call_convention_count
(out) Number of call conventions in the program.

Return Values

HSA_STATUS_SUCCESS
The function has been executed successfully.

HSA_STATUS_ERROR_INVALID_ARGUMENT
If provided *program* is invalid, or *agent* is invalid.

3.2.1.13 hsa_ext_define_program_allocation_global_variable_address

```
hsa_status_t hsa_ext_define_program_allocation_global_variable_address(
    hsa_ext_program_handle_t program,
    hsa_ext_brig_module_handle_t module,
    hsa_ext_brig_code_section_offset32_t symbol,
    hsa_ext_error_message_callback_t error_message_callback,
```

```
    void * address)
```

Defines program's global variable address.

Parameters

program
(in) Program to define global variable address for.

module
(in) Module to define global variable address for.

symbol
(in) Offset.

error_message_callback
(in) Call back function to get the string representation of the error message.

address
(in) Specified address.

Return Values

HSA_STATUS_SUCCESS
The function has been executed successfully.

HSA_STATUS_ERROR_INVALID_ARGUMENT
If provided *program* is invalid, or *module* is invalid.

3.2.1.14 hsa_ext_query_program_allocation_global_variable_address

```
hsa_status_t hsa_ext_query_program_allocation_global_variable_address(
    hsa_ext_program_handle_t program,
    hsa_ext_brig_module_handle_t module,
    hsa_ext_brig_code_section_offset32_t symbol,
    void ** address)
```

Queries program's global variable address.

Parameters

program
(in) Program to query global variable address for.

module
(in) Module to query global variable address for.

symbol
(in) Offset.

address
(out) Queried address.

Return Values

HSA_STATUS_SUCCESS
The function has been executed successfully.

HSA_STATUS_ERROR_INVALID_ARGUMENT
If provided *program* is invalid, or *module* is invalid.

3.2.1.15 hsa_ext_define_agent_allocation_global_variable_address

```
hsa_status_t hsa_ext_define_agent_allocation_global_variable_address(
    hsa_ext_program_handle_t program,
    hsa_agent_t * agent,
    hsa_ext_brig_module_handle_t module,
    hsa_ext_brig_code_section_offset32_t symbol,
    hsa_ext_error_message_callback_t error_message_callback,
    void * address)
```

Defines agent's global variable address.

Parameters

program
(in) Program to define global variable address for.

agent
(in) HSA Agent to define global variable address for.

module
(in) Module to define global variable address for.

symbol
(in) Offset.

error_message_callback
(in) Call back function to get the string representation of the error message.

address
(in) Specified address.

Return Values

HSA_STATUS_SUCCESS
The function has been executed successfully.

HSA_STATUS_ERROR_INVALID_ARGUMENT
If provided *program* is invalid, or *module* is invalid, or *agent* is invalid.

3.2.1.16 hsa_ext_query_agent_global_variable_address

```
hsa_status_t hsa_ext_query_agent_global_variable_address(
    hsa_ext_program_handle_t program,
    hsa_agent_t * agent,
    hsa_ext_brig_module_handle_t module,
    hsa_ext_brig_code_section_offset32_t symbol,
    void ** address)
```

Queries agent's global variable address.

Parameters

program
(in) Program to query global variable address for.

agent
(in) HSA Agent to query global variable address for.

module
(in) Module to query global variable address for.

symbol
(in) Offset.

address
(out) Queried address.

Return Values

HSA_STATUS_SUCCESS
The function has been executed successfully.

HSA_STATUS_ERROR_INVALID_ARGUMENT
If provided *program* is invalid, or *module* is invalid, or *agent* is invalid.

3.2.1.17 hsa_ext_define_readonly_variable_address

```
hsa_status_t hsa_ext_define_readonly_variable_address(
    hsa_ext_program_handle_t program,
    hsa_agent_t * agent,
    hsa_ext_brig_module_handle_t module,
    hsa_ext_brig_code_section_offset32_t symbol,
    hsa_ext_error_message_callback_t error_message_callback,
    void * address)
```

Defines agent's read-only variable address.

Parameters

program
(in) Program to define read-only variable address for.

agent
(in) HSA Agent to define read-only variable address for.

module
(in) Module to define read-only variable address for.

symbol
(in) Offset.

error_message_callback
(in) Call back function to get the string representation of the error message.

address
(in) Specified address.

Return Values

HSA_STATUS_SUCCESS
The function has been executed successfully.

HSA_STATUS_ERROR_INVALID_ARGUMENT
If provided *program* is invalid, or *module* is invalid, or *agent* is invalid.

3.2.1.18 hsa_ext_query_readonly_variable_address

```
hsa_status_t hsa_ext_query_readonly_variable_address(
    hsa_ext_program_handle_t program,
    hsa_agent_t * agent,
    hsa_ext_brig_module_handle_t module,
    hsa_ext_brig_code_section_offset32_t symbol,
    void ** address)
```

Queries agent's read-only variable address.

Parameters

program
(in) Program to query read-only variable address for.

agent
(in) HSA Agent to query read-only variable address for.

module
(in) Module to query read-only variable address for.

symbol
(in) Offset.

address
(out) Queried address.

Return Values

HSA_STATUS_SUCCESS
The function has been executed successfully.

HSA_STATUS_ERROR_INVALID_ARGUMENT
If provided *program* is invalid, or *module* is invalid, or *agent* is invalid.

3.2.1.19 hsa_ext_query_kernel_descriptor_address

```
hsa_status_t hsa_ext_query_kernel_descriptor_address(
    hsa_ext_program_handle_t program,
    hsa_ext_brig_module_handle_t module,
    hsa_ext_brig_code_section_offset32_t symbol,
    void ** address)
```

Queries kernel descriptor address.

Parameters

program
(in) Program to query kernel descriptor address from.

module
(in) BRIG module handle.

symbol
(in) Offset.

address
(out) The address of kernel descriptor.

Return Values

HSA_STATUS_SUCCESS

The function has been executed successfully.

HSA_STATUS_ERROR_INVALID_ARGUMENT
If *program* or *module* are not valid HSAIL program or BRIG module respectively.

Description

Queries kernel descriptor address. Needed to create the dispatch packet.

3.2.1.20 hsa_ext_query_indirect_function_descriptor_address

```
hsa_status_t hsa_ext_query_indirect_function_descriptor_address(
    hsa_ext_program_handle_t program,
    hsa_ext_brig_module_handle_t module,
    hsa_ext_brig_code_section_offset32_t symbol,
    void ** address)
```

Queries indirect function descriptor address.

Parameters

program
(in) Program to query indirect function descriptor address from.

module
(in) BRIG module handle.

symbol
(in) Offset.

address
(out) The address of indirect function descriptor.

Return Values

HSA_STATUS_SUCCESS
The function has been executed successfully.

HSA_STATUS_ERROR_INVALID_ARGUMENT
If *program* or *module* are not valid HSAIL program or BRIG module respectively.

Description

Queries indirect function descriptor address, which allows host program to perform indirect function table variable initialization.

3.2.1.21 hsa_ext_validate_program

```
hsa_status_t hsa_ext_validate_program(
    hsa_ext_program_handle_t program,
    hsa_ext_error_message_callback_t error_message_callback)
```

Validates HSAIL program.

Parameters

program
(in) Handle to the HSAIL program to validate.

error_message_callback
(in) Call back function to get the string representation of the error message.

Return Values

HSA_STATUS_SUCCESS
The function has been executed successfully.

If the program is not valid, refer to the error call back function to get string representation of the failure.

Description

Validates HSAIL program with specified program handle. Returns either HSA_STATUS_SUCCESS or ::HSA_STATUS_ERROR if the validation is successful or not. Refer to the *error_message* call back to get the string representation of the failure.

3.2.1.22 hsa_ext_validate_program_module

```
hsa_status_t hsa_ext_validate_program_module(
    hsa_ext_program_handle_t program,
    hsa_ext_brig_module_handle_t module,
    hsa_ext_error_message_callback_t error_message_callback)
```

Validates program module.

Parameters

program
(in) Handle to the HSAIL program.

module
(in) Handle to the module to validate.

error_message_callback
(in) Call back function to get the string representation of the error message.

Return Values

HSA_STATUS_SUCCESS
The function has been executed successfully.

If the module is not valid, refer to the error call back function to get string representation of the failure.

Description

Validates program module with specified module handle. Returns either HSA_STATUS_SUCCESS or ::HSA_STATUS_ERROR if the validation is successful or not. Refer to the *error_message_callback* call back to get the string representation of the failure.

3.2.1.23 hsa_ext_serialize_program

```
hsa_status_t hsa_ext_serialize_program(
    hsa_runtime_caller_t caller,
    hsa_ext_program_handle_t program,
    hsa_runtime_alloc_data_callback_t alloc_serialize_data_callback,
```

```
hsa_ext_error_message_callback_t error_message_callback,
int debug_information,
void * serialized_object)
```

Serializes the HSAIL program.

Parameters

caller
(in) Opaque pointer and will be passed to all call back functions made by this call.

program
(in) HSAIL program to be serialized.

alloc_serialize_data_callback
(in) Call back function for allocation.

error_message_callback
(in) Call back function to get the string representation of the error message.

debug_information
(in) The flag for including/excluding the debug information for *finalization_descriptor*. 0 - exclude debug information, 1

- include debug information.

serialized_object
(in) Pointer to the serialized object.

Return Values

HSA_STATUS_SUCCESS
The function has been executed successfully.

HSA_STATUS_ERROR_INVALID_ARGUMENT
If *program* is not a valid program.

HSA_STATUS_ERROR_OUT_OF_RESOURCES
If no memory can be allocated for *serialized_object*.

Description

Serializes the *program* to *serialized_object*. Used for offline compilation.

3.2.1.24 hsa_ext_program_allocation_symbol_address_t

```
typedef hsa_status_t(* hsa_ext_program_allocation_symbol_address_t)(hsa_runtime_caller_t caller, const char *name, uint64_t *symbol_adress)
```

Call back function to get program's address of global segment variables, kernel table variable, indirect function table variable based on the symbolic name.

3.2.1.25 hsa_ext_agent_allocation_symbol_address_t

```
typedef hsa_status_t(* hsa_ext_agent_allocation_symbol_address_t)(hsa_runtime_caller_t caller,
hsa_agent_t *agent, const char *name, uint64_t *symbol_adress)
```

Call back function to get agents's address of global segment variables, kernel table variable, indirect function table variable based on the symbolic name.

3.2.1.26 hsa_ext_deserialize_program

```
hsa_status_t hsa_ext_deserialize_program(
    hsa_runtime_caller_t caller,
    void * serialized_object,
    hsa_ext_program_allocation_symbol_address_t program_allocation_symbol_address,
    hsa_ext_agent_allocation_symbol_address_t agent_allocation_symbol_address,
    hsa_ext_error_message_callback_t error_message_callback,
    int debug_information,
    hsa_ext_program_handle_t ** program)
```

Deserializes the HSAIL program.

Parameters

caller
: (in) Opaque pointer and will be passed to all call back functions made by this call.

serialized_object
: (in) Serialized object to be deserialized.

program_allocation_symbol_address
: (in) Call back function to get program's address of global segment variables, kernel table variable, indirect function table variable based on the symbolic name. Allows symbols defined by application to be relocated.

agent_allocation_symbol_address
: (in) Call back function to get agent's address of global segment variables, kernel table variable, indirect function table variable based on the symbolic name. Allows symbols defined by application to be relocated.

error_message_callback
: (in) Call back function to get the string representation of the error message.

debug_information
: (in) The flag for including/excluding the debug information for *finalization_descriptor*. 0 - exclude debug information, 1 - include debug information.

program
: (out) Deserialized program.

Return Values

HSA_STATUS_SUCCESS
: The function has been executed successfully.

HSA_STATUS_ERROR_INVALID_ARGUMENT
: If *serialized_object* is either NULL, or is not valid, or the size is 0.

HSA_STATUS_ERROR_OUT_OF_RESOURCES

If no memory can be allocated for *finalization_descriptor*.

Description

Deserializes the program from *serialized_object*. Used for offline compilation. Includes call back functions hsa_program, agent_allocation_symbol_address_t, where call back functions take symbolic name, this allows symbols defined by application to be relocated.

3.3 Images and Samplers

An HSA runtime uses an image handle hsa_ext_image_handle_t to access images. The image handle references the image data in memory and records information about resource layout and other properties. HSA decouples the storage of the image data and the description of how the device interprets that data. This allows the application developer to control the location of the image data storage and manage memory more efficiently.

The HSA image format is specified using a format descriptor (hsa_ext_image_format_t) that contains information about the image channel type and the channel order. The image channel type describes how the data is to be interpreted along with the bit size, and image channel order describes the number and the order. Not all image channel types and channel order combinations are valid on a HSA agent. All HSA agents have to support a required minimum set of image formats. For more information, refer to the HSA Programmer's Reference Manual[1]. An application can use **hsa_ext_image_get_format_capability** to query a runtime to obtain image format capabilities.

An implementation-independent image format descriptor (hsa_ext_image_descriptor_t) is composed of geometry along with the image format. The image descriptor is used to inquire the runtime for the HSA component-specific image data size and alignment details by calling **hsa_ext_image_get_info** for the purpose of determining the implementation's storage requirements.

The memory requirements (hsa_ext_image_info_t) include the size of the memory needed as well as any alignment constraints. An application can either allocate new memory for the image data, or sub-allocate a memory block from an existing memory if the memory size allows. Before the image data is used, an HSA agent-specific image handle must be created using it and if necessary, cleared and prepared according to the intended use.

A HSA agent-specific image handle (hsa_ext_image_handle_t) is used by the HSAIL language for reading or writing using HSAIL **rd image**, **ldimage** and **stimage** operations. **hsa_ext_image_create_handle** creates an image handle from a implementation-independent image format descriptor and independently allocated image data that conforms to the requirements provided by **hsa_ext_image_get_info**.

It must be noted that while the image data technically accessible from its pointer in the raw form, the data layout and organization is agent-specific and should be treated as opaque. The internal implementation of an optimal image data organization could vary depending on the attributes of the image format descriptor. As a result, there are no guarantees on the data layout when accessed from another HSA agent. The only reliable way to import or export image data from optimally organized images is to copy their data to and from a linearly organized data layout in memory, as specified by the image's format attributes.

The HSA Runtime provides interfaces to allow operations on images. Image data transfer to and from memory with a linear layout can be performed using **hsa_ext_image_export** and **hsa_ext_image_import** respectively. A portion of an image could be copied to another image using **hsa_ext_image_copy**. An image can be cleared using **hsa_ext_image_clear**. It is the application's responsibility to ensure proper synchronization and preparation of images on accesses from other image operations. See HSA System Architecture spec 2.13 for the HSA Image memory model.

A HSA agent-specific sampler handle (hsa_ext_sampler_handle_t) is used by the HSAIL language to describe how images are processed by the **rdimage** HSAIL operation. **hsa_ext_sampler_create_handle** creates a sampler handle from an agent independent sampler descriptor (hsa_ext_sampler_descriptor_t).

3.3.1 API

3.3.1.1 hsa_ext_image_handle_s

```
typedef struct hsa_ext_image_handle_s {
    uint64_t handle;
} hsa_ext_image_handle_t
```

Image handle, populated by **hsa_ext_image_create_handle**. Images handles are only unique within an agent, not across agents.

Data Fields

handle
HSA component specific handle to the image.

3.3.1.2 hsa_ext_image_format_capability_t

```
enum hsa_ext_image_format_capability_t
```

Image format capability returned by **hsa_ext_image_get_format_capability**.

Values

HSA_EXT_IMAGE_FORMAT_NOT_SUPPORTED = 0x0
Images of this format are not supported.

HSA_EXT_IMAGE_FORMAT_READ_ONLY = 0x1
Images of this format can be accessed for read operations.

HSA_EXT_IMAGE_FORMAT_WRITE_ONLY = 0x2
Images of this format can be accessed for write operations.

HSA_EXT_IMAGE_FORMAT_READ_WRITE = 0x4
Images of this format can be accessed for read and write operations.

HSA_EXT_IMAGE_FORMAT_READ_MODIFY_WRITE = 0x8
Images of this format can be accessed for read-modify-write operations.

HSA_EXT_IMAGE_FORMAT_ACCESS_INVARIANT_IMAGE_DATA = 0x10
Images of this format are guaranteed to have consistent data layout regardless of the how it is accessed by the HSA agent.

3.3.1.3 hsa_ext_image_info_s

```
typedef struct hsa_ext_image_info_s {
    size_t image_size;
    size_t image_alignment;
} hsa_ext_image_info_t
```

Agent-specific image size and alignment requirements. This structure stores the agent-dependent image data sizes and alignment, and populated by **hsa_ext_image_get_info**.

Data Fields

image_size
Component specific image data size in bytes.

image_alignment
Component specific image data alignment in bytes.

3.3.1.4 hsa_ext_image_access_permission_t

enum **hsa_ext_image_access_permission_t**

Defines how the HSA device expects to access the image. The access pattern used by the HSA agent specified in **hsa_ext_image_create_handle**.

Values

HSA_EXT_IMAGE_ACCESS_PERMISSION_READ_ONLY
Image handle is to be used by the HSA agent as read-only using an HSAIL roimg type.

HSA_EXT_IMAGE_ACCESS_PERMISSION_WRITE_ONLY
Image handle is to be used by the HSA agent as write-only using an HSAIL woimg type.

HSA_EXT_IMAGE_ACCESS_PERMISSION_READ_WRITE
Image handle is to be used by the HSA agent as read and/or write using an HSAIL rwimg type.

3.3.1.5 hsa_ext_image_geometry_t

enum **hsa_ext_image_geometry_t**

Geometry associated with the HSA image (image dimensions allowed in HSA). The enumeration values match the HSAIL BRIG type BrigImageGeometry.

Values

HSA_EXT_IMAGE_GEOMETRY_1D = 0
One-dimensional image addressed by width coordinate.

HSA_EXT_IMAGE_GEOMETRY_2D = 1
Two-dimensional image addressed by width and height coordinates.

HSA_EXT_IMAGE_GEOMETRY_3D = 2
Three-dimensional image addressed by width, height, and depth coordinates.

HSA_EXT_IMAGE_GEOMETRY_1DA = 3
Array of one-dimensional images with the same size and format. 1D arrays are addressed by index and width coordinate.

HSA_EXT_IMAGE_GEOMETRY_2DA = 4
Array of two-dimensional images with the same size and format. 2D arrays are addressed by index and width and height coordinates.

HSA_EXT_IMAGE_GEOMETRY_1DB = 5
One-dimensional image interpreted as a buffer with specific restrictions.

HSA_EXT_IMAGE_GEOMETRY_2DDEPTH = 6
Two-dimensional depth image addressed by width and height coordinates.

HSA_EXT_IMAGE_GEOMETRY_2DADEPTH = 7
Array of two-dimensional depth images with the same size and format. 2D arrays are addressed by index and width and height coordinates.

3.3.1.6 hsa_ext_image_channel_type_t

enum **hsa_ext_image_channel_type_t**

Component type associated with the image. See Image section in HSA Programming Reference Manual for definitions on each component type. The enumeration values match the HSAIL BRIG type BrigImageChannelType.

Values

HSA_EXT_IMAGE_CHANNEL_TYPE_SNORM_INT8 = 0

HSA_EXT_IMAGE_CHANNEL_TYPE_SNORM_INT16 = 1

HSA_EXT_IMAGE_CHANNEL_TYPE_UNORM_INT8 = 2

HSA_EXT_IMAGE_CHANNEL_TYPE_UNORM_INT16 = 3

HSA_EXT_IMAGE_CHANNEL_TYPE_UNORM_INT24 = 4

HSA_EXT_IMAGE_CHANNEL_TYPE_UNORM_SHORT_555 = 5

HSA_EXT_IMAGE_CHANNEL_TYPE_UNORM_SHORT_565 = 6

HSA_EXT_IMAGE_CHANNEL_TYPE_UNORM_SHORT_101010 = 7

HSA_EXT_IMAGE_CHANNEL_TYPE_SIGNED_INT8 = 8

HSA_EXT_IMAGE_CHANNEL_TYPE_SIGNED_INT16 = 9

HSA_EXT_IMAGE_CHANNEL_TYPE_SIGNED_INT32 = 10

HSA_EXT_IMAGE_CHANNEL_TYPE_UNSIGNED_INT8 = 11

HSA_EXT_IMAGE_CHANNEL_TYPE_UNSIGNED_INT16 = 12

HSA_EXT_IMAGE_CHANNEL_TYPE_UNSIGNED_INT32 = 13

HSA_EXT_IMAGE_CHANNEL_TYPE_HALF_FLOAT = 14

HSA_EXT_IMAGE_CHANNEL_TYPE_FLOAT = 15

3.3.1.7 hsa_ext_image_channel_order_t

enum **hsa_ext_image_channel_order_t**

Image component order associated with the image. See Image section in HSA Programming Reference Manual for definitions on each component order. The enumeration values match the HSAIL BRIG type BrigImageChannelOrder.

Values

HSA_EXT_IMAGE_CHANNEL_ORDER_A = 0

HSA_EXT_IMAGE_CHANNEL_ORDER_R = 1

HSA_EXT_IMAGE_CHANNEL_ORDER_RX = 2

HSA_EXT_IMAGE_CHANNEL_ORDER_RG = 3

HSA_EXT_IMAGE_CHANNEL_ORDER_RGX = 4

HSA_EXT_IMAGE_CHANNEL_ORDER_RA = 5

HSA_EXT_IMAGE_CHANNEL_ORDER_RGB = 6

HSA_EXT_IMAGE_CHANNEL_ORDER_RGBX = 7

HSA_EXT_IMAGE_CHANNEL_ORDER_RGBA = 8

HSA_EXT_IMAGE_CHANNEL_ORDER_BGRA = 9

HSA_EXT_IMAGE_CHANNEL_ORDER_ARGB = 10

HSA_EXT_IMAGE_CHANNEL_ORDER_ABGR = 11

HSA_EXT_IMAGE_CHANNEL_ORDER_SRGB = 12

HSA_EXT_IMAGE_CHANNEL_ORDER_SRGBX = 13

HSA_EXT_IMAGE_CHANNEL_ORDER_SRGBA = 14

HSA_EXT_IMAGE_CHANNEL_ORDER_SBGRA = 15

HSA_EXT_IMAGE_CHANNEL_ORDER_INTENSITY = 16

HSA_EXT_IMAGE_CHANNEL_ORDER_LUMINANCE = 17

HSA_EXT_IMAGE_CHANNEL_ORDER_DEPTH = 18

HSA_EXT_IMAGE_CHANNEL_ORDER_DEPTH_STENCIL = 19

3.3.1.8 hsa_ext_image_format_s

```
typedef struct hsa_ext_image_format_s {
    hsa_ext_image_channel_type_t channel_type;
    hsa_ext_image_channel_order_t channel_order;
} hsa_ext_image_format_t
```

Image format descriptor (attributes of the image format).

Data Fields

channel_type
Channel type of the image.

channel_order
Channel order of the image.

3.3.1.9 hsa_ext_image_descriptor_s

```
typedef struct hsa_ext_image_descriptor_s {
    hsa_ext_image_geometry_t geometry;
    size_t width;
    size_t height;
    size_t depth;
    size_t array_size;
    hsa_ext_image_format_t format;
} hsa_ext_image_descriptor_t
```

Implementation-independent HSA Image descriptor.

Data Fields

geometry
Geometry of the image.

width
Width of the image in components.

height
Height of the image in components, only used if geometry is 2D or higher.

depth
Depth of the image in slices, only used if geometry is 3D depth = 0 is same as depth = 1.

array_size
Number of images in the image array, only used if geometry is 1DArray and 2DArray.

format
Format of the image.

3.3.1.10 hsa_ext_image_range_s

```
typedef struct hsa_ext_image_range_s {
    uint32_t width;
    uint32_t height;
    uint32_t depth;
} hsa_ext_image_range_t
```

Three-dimensional image range description.

Data Fields

width
The width for an image range (in coordinates).

height
The height for an image range (in coordinates).

depth
The depth for an image range (in coordinates).

3.3.1.11 hsa_ext_image_region_s

```
typedef struct hsa_ext_image_region_s {
    hsa_dim3_t image_offset;
    hsa_ext_image_range_t image_range;
} hsa_ext_image_region_t
```

Image region description. Used by image operations such as import, export, copy, and clear.

Data Fields

image_offset
Offset in the image (in coordinates).

image_range
Dimensions of the image range (in coordinates).

3.3.1.12 hsa_ext_sampler_handle_s

```
typedef struct hsa_ext_sampler_handle_s {
    uint64_t handle;
} hsa_ext_sampler_handle_t
```

Sampler handle. Samplers are populated by **hsa_ext_sampler_create_handle**. Sampler handles are only unique within an agent, not across agents.

Data Fields

handle
Component-specific HSA sampler.

3.3.1.13 hsa_ext_sampler_addressing_mode_t

```
enum hsa_ext_sampler_addressing_mode_t
```

Sampler address modes. The sampler address mode describes the processing of out-of-range image coordinates. The values match the HSAIL BRIG type BrigSamplerAddressing.

Values

HSA_EXT_SAMPLER_ADDRESSING_UNDEFINED = 0
Out-of-range coordinates are not handled.

HSA_EXT_SAMPLER_ADDRESSING_CLAMP_TO_EDGE = 1
Clamp out-of-range coordinates to the image edge.

HSA_EXT_SAMPLER_ADDRESSING_CLAMP_TO_BORDER = 2
Clamp out-of-range coordinates to the image border.

HSA_EXT_SAMPLER_ADDRESSING_REPEAT = 3
Wrap out-of-range coordinates back into the valid coordinate range.

HSA_EXT_SAMPLER_ADDRESSING_MIRRORED_REPEAT = 4
Mirror out-of-range coordinates back into the valid coordinate range.

3.3.1.14 hsa_ext_sampler_coordinate_mode_t

```
enum hsa_ext_sampler_coordinate_mode_t
```

Sampler coordinate modes. The enumeration values match the HSAIL BRIG BRIG_SAMPLER_COORD bit in the type BrigSamplerModifier.

Values

HSA_EXT_SAMPLER_COORD_NORMALIZED = 0
Coordinates are all in the range of 0.0 to 1.0.

HSA_EXT_SAMPLER_COORD_UNNORMALIZED = 1
Coordinates are all in the range of 0 to (dimension-1).

3.3.1.15 hsa_ext_sampler_filter_mode_t

```
enum hsa_ext_sampler_filter_mode_t
```

Sampler filter modes. The enumeration values match the HSAIL BRIG type BrigSamplerFilter.

Values

HSA_EXT_SAMPLER_FILTER_NEAREST = 0
Filter to the image element nearest (in Manhattan distance) to the specified coordinate.

HSA_EXT_SAMPLER_FILTER_LINEAR = 1
Filter to the image element calculated by combining the elements in a 2x2 square block or 2x2x2 cube block around the specified coordinate. The elements are combined using linear interpolation.

3.3.1.16 hsa_ext_sampler_descriptor_s

```
typedef struct hsa_ext_sampler_descriptor_s {
    hsa_ext_sampler_coordinate_mode_t coordinate_mode;
    hsa_ext_sampler_filter_mode_t filter_mode;
    hsa_ext_sampler_addressing_mode_t address_mode;
} hsa_ext_sampler_descriptor_t
```

Implementation-independent sampler descriptor.

Data Fields

coordinate_mode
Sampler coordinate mode describes the normalization of image coordinates.

filter_mode
Sampler filter type describes the type of sampling performed.

address_mode
Sampler address mode describes the processing of out-of-range image coordinates.

3.3.1.17 hsa_ext_image_get_format_capability

```
hsa_status_t hsa_ext_image_get_format_capability(
    const hsa_agent_t * agent,
    const hsa_ext_image_format_t * image_format,
    hsa_ext_image_geometry_t image_geometry,
    uint32_t * capability_mask)
```

Retrieve image format capabilities for the specified image format on the specified HSA component.

Parameters

agent
(in) HSA agent to be associated with the image.

image_format
(in) Image format.

image_geometry
(in) Geometry of the image.

capability_mask

(out) Image format capability bit-mask.

Return Values

HSA_STATUS_SUCCESS
The function has been executed successfully.

HSA_STATUS_ERROR_NOT_INITIALIZED
The runtime has not been initialized.

HSA_STATUS_ERROR_INVALID_ARGUMENT
If *agent*, *image_format*, or *capability_mask* are NULL.

Description

If successful, the queried image format's capabilities bit-mask is written to the location specified by *capability_-mask*. See hsa_ext_image_format_capability_t to determine all possible capabilities that can be reported in the bit mask.

3.3.1.18 hsa_ext_image_get_info

```
hsa_status_t hsa_ext_image_get_info(
    const hsa_agent_t * agent,
    const hsa_ext_image_descriptor_t * image_descriptor,
    hsa_ext_image_access_permission_t access_permission,
    hsa_ext_image_info_t * image_info)
```

Inquires the required HSA component-specific image data details from a implementation independent image descriptor.

Parameters

agent
(in) HSA agent to be associated with the image.

image_descriptor
(in) Implementation-independent image descriptor describing the image.

access_permission
(in) Access permission of the image by the HSA agent.

image_info
(out) Image info size and alignment requirements that the HSA agent requires.

Return Values

HSA_STATUS_SUCCESS
The function has been executed successfully.

HSA_STATUS_ERROR_NOT_INITIALIZED
The runtime has not been initialized.

HSA_STATUS_ERROR_INVALID_ARGUMENT
If any of the arguments is NULL.

HSA_EXT_STATUS_ERROR_IMAGE_FORMAT_UNSUPPORTED
If the HSA agent does not support the image format specified by the descriptor.

HSA_EXT_STATUS_ERROR_IMAGE_SIZE_UNSUPPORTED
If the HSA agent does not support the image dimensions specified by the format descriptor.

Description

If successful, the queried HSA agent-specific image data info is written to the location specified by *image_-info*. Based on the implementation the optimal image data size and alignment requirements could vary depending on the image attributes specified in *image_descriptor*.

The implementation must return the same image info requirements for different access permissions with exactly the same image descriptor as long as **hsa_ext_image_get_format_capability** reports HSA_EXT_-IMAGE_FORMAT_ACCESS_INVARIANT_IMAGE_DATA for the image format specified in the image descriptor.

3.3.1.19 hsa_ext_image_create_handle

```
hsa_status_t hsa_ext_image_create_handle(
    const hsa_agent_t * agent,
    const hsa_ext_image_descriptor_t * image_descriptor,
    const void * image_data,
    hsa_ext_image_access_permission_t access_permission,
    hsa_ext_image_handle_t * image_handle)
```

Creates a agent-defined image handle from an implementation-independent image descriptor and a agent-specific image data. The image access defines how the HSA agent expects to use the image and must match the HSAIL image handle type used by the agent.

Parameters

agent
(in) HSA agent to be associated with the image.

image_descriptor
(in) Implementation-independent image descriptor describing the image.

image_data
(in) Address of the component-specific image data.

access_permission
(in) Access permission of the image by the HSA agent.

image_handle
(out) Agent-specific image handle.

Return Values

HSA_STATUS_SUCCESS
The function has been executed successfully.

HSA_STATUS_ERROR_NOT_INITIALIZED
The runtime has not been initialized.

HSA_STATUS_ERROR_INVALID_ARGUMENT
If any of the arguments is NULL.

HSA_EXT_STATUS_ERROR_IMAGE_FORMAT_UNSUPPORTED
If the HSA agent does not have the capability to support the image format using the specified *agent_-access*.

HSA_STATUS_ERROR_OUT_OF_RESOURCES
If the HSA agent cannot create the specified handle because it is out of resources.

Description

If successful, the image handle is written to the location specified by *image_handle*. The image data memory must be allocated using the previously queried **hsa_ext_image_get_info** memory requirements with the same HSA agent and implementation-independent image descriptor.

The image data is not initialized and any previous memory contents is preserved. The memory management of image data is the application's responsibility and can only be freed until the memory is no longer needed and any image handles using it are destroyed.

access_permission defines how the HSA agent expects to use the image handle. The image format specified in the image descriptor must be capable by the HSA agent for the intended permission.

Image handles with different permissions can be created using the same image data with exactly the same image descriptor as long as HSA_EXT_IMAGE_FORMAT_ACCESS_INVARIANT_IMAGE_DATA is reported by **hsa_ext_image_get_format_capability** for the image format specified in the image descriptor. Images of non-linear s-form channel order can share the same image data with its equivalent linear non-s form channel order, provided the rest of the image descriptor parameters are identical.

If necessary, an application can use image operations (import, export, copy, clear) to prepare the image for the intended use regardless of the access permissions.

3.3.1.20 hsa_ext_image_import

```
hsa_status_t hsa_ext_image_import(
    const hsa_agent_t * agent,
    const void * src_memory,
    size_t src_row_pitch,
    size_t src_slice_pitch,
    hsa_ext_image_handle_t dst_image_handle,
    const hsa_ext_image_region_t * image_region,
    const hsa_signal_handle_t * completion_signal)
```

Imports a linearly organized image data from memory directly to an image handle.

Parameters

agent
(in) HSA agent to be associated with the image.

src_memory
(in) Source memory.

src_row_pitch
(in) Number of bytes in one row of the source memory.

src_slice_pitch
(in) Number of bytes in one slice of the source memory.

dst_image_handle
(in) Destination Image handle.

image_region
(in) Image region to be updated.

completion_signal
(in) Signal to set when the operation is completed.

Return Values

HSA_STATUS_SUCCESS
The function has been executed successfully.

HSA_STATUS_ERROR_NOT_INITIALIZED
The runtime has not been initialized.

HSA_STATUS_ERROR_INVALID_ARGUMENT
    If *agent*, *src_memory* or *image_region* are NULL.

Description

This operation updates the image data referenced by the image handle from the source memory. The size of the data imported from memory is implicitly derived from the image region.

If *completion_signal* is NULL, the operation occurs synchronously. Otherwise the function returns immediately and the completion signal is signaled when the operation completes.

If *src_row_pitch* is smaller than the destination region width (in bytes), then *src_row_pitch* = region width.

If *src_slice_pitch* is smaller than the destination region width * region height (in bytes), then *src_slice_pitch* = region width * region height.

It is the application's responsibility to avoid out of bounds memory access.

None of the source memory or image data memory in the previously created **hsa_ext_image_create_handle** image handle can overlap. Overlapping of any of the source and destination memory within the import operation produces undefined results.

3.3.1.21 hsa_ext_image_export

```
hsa_status_t hsa_ext_image_export(
    const hsa_agent_t * agent,
    hsa_ext_image_handle_t src_image_handle,
    void * dst_memory,
    size_t dst_row_pitch,
    size_t dst_slice_pitch,
    const hsa_ext_image_region_t * image_region,
    const hsa_signal_handle_t * completion_signal)
```

Export image data from the image handle directly to memory organized linearly.

Parameters

agent
    (in) HSA agent to be associated with the image.

src_image_handle
    (in) Source image handle.

dst_memory
    (in) Destination memory.

dst_row_pitch
    (in) Number of bytes in one row of the destination memory.

dst_slice_pitch
    (in) Number of bytes in one slice of the destination memory.

image_region
    (in) Image region to be exported.

completion_signal
    (in) Signal to set when the operation is completed.

Return Values

HSA_STATUS_SUCCESS
    The function has been executed successfully.

HSA_STATUS_ERROR_NOT_INITIALIZED
The runtime has not been initialized.

HSA_STATUS_ERROR_INVALID_ARGUMENT
If *agent*, *dst_memory* or *image_region* are NULL.

Description

The operation updates the destination memory with the image data in the image handle. The size of the data exported to memory is implicitly derived from the image region.

If *completion_signal* is NULL, the operation occurs synchronously. Otherwise the function returns immediately and the completion signal is signaled when the operation completes.

If *dst_row_pitch* is smaller than the source region width (in bytes), then *dst_row_pitch* = region width.

If *dst_slice_pitch* is smaller than the source region width * region height (in bytes), then *dst_slice_pitch* = region width * region height.

It is the application's responsibility to avoid out of bounds memory access.

None of the destination memory or image data memory in the previously created **hsa_ext_image_create_handle** image handle can overlap. Overlapping of any of the source and destination memory within the export operation produces undefined results.

3.3.1.22 hsa_ext_image_copy

```
hsa_status_t hsa_ext_image_copy(
    const hsa_agent_t * agent,
    hsa_ext_image_handle_t src_image_handle,
    hsa_ext_image_handle_t dst_image_handle,
    const hsa_ext_image_region_t * image_region,
    const hsa_signal_handle_t * completion_signal)
```

Copies a region from one image to another.

Parameters

agent
(in) HSA agent to be associated with the image.

src_image_handle
(in) Source image handle.

dst_image_handle
(in) Destination image handle.

image_region
(in) Image region to be copied.

completion_signal
(in) Signal to set when the operation is completed.

Return Values

HSA_STATUS_SUCCESS
The function has been executed successfully.

HSA_STATUS_ERROR_NOT_INITIALIZED
The runtime has not been initialized.

HSA_STATUS_ERROR_INVALID_ARGUMENT

If *agent* or *image_region* are NULL.

Description

The operation copies the image data from the source image handle to the destination image handle. The size of the image data copied is implicitly derived from the image region.

If *completion_signal* is NULL, the operation occurs synchronously. Otherwise the function returns immediately and the completion signal is signaled when the operation completes.

It is the application's responsibility to avoid out of bounds memory access.

The source and destination handles must have been previously created using **hsa_ext_image_create_handle**. The source and destination image data memory are not allowed to be the same. Overlapping any of the source and destination memory produces undefined results.

The source and destination image formats don't have to match; appropriate format conversion is performed automatically. The source and destination images must be of the same geometry.

3.3.1.23 hsa_ext_image_clear

```
hsa_status_t hsa_ext_image_clear(
    const hsa_agent_t * agent,
    hsa_ext_image_handle_t image_handle,
    const float data[4],
    const hsa_ext_image_region_t * image_region,
    const hsa_signal_handle_t * completion_signal)
```

Clears the image to a specified 4-component floating point data.

Parameters

agent
(in) HSA agent to be associated with the image.

image_handle
(in) Image to be cleared.

data
(in) 4-component clear value in floating point format.

image_region
(in) Image region to clear.

completion_signal
(in) Signal to set when the operation is completed.

Return Values

HSA_STATUS_SUCCESS
The function has been executed successfully.

HSA_STATUS_ERROR_NOT_INITIALIZED
The runtime has not been initialized.

HSA_STATUS_ERROR_INVALID_ARGUMENT
If *agent* or *image_region* are NULL.

Description

The operation clears the elements of the image with the data specified. The lowest bits of the data (number of bits depending on the image component type) are stored in the cleared image are based on the image component order. The size of the image data cleared is implicitly derived from the image region.

If *completion_signal* is NULL, the operation occurs synchronously. Otherwise the function returns immediately and the completion signal is signaled when the operation completes.

It is the application's responsibility to avoid out of bounds memory access.

Clearing an image automatically performs value conversion on the provided floating point values as is appropriate for the image format used.

For images of UNORM types, the floating point values must be in the [0..1] range. For images of SNORM types, the floating point values must be in the [-1..1] range. For images of UINT types, the floating point values are rounded down to an integer value. For images of SRGB types, the clear data is specified in a linear space, which is appropriately converted by the Runtime to sRGB color space.

Specifying clear value outside of the range representable by an image format produces undefined results.

3.3.1.24 hsa_ext_image_destroy_handle

```
hsa_status_t hsa_ext_image_destroy_handle(
    const hsa_agent_t * agent,
    hsa_ext_image_handle_t * image_handle)
```

Destroys the specified image handle.

Parameters

agent
(in) HSA agent to be associated with the image.

image_handle
(in) Image handle.

Return Values

HSA_STATUS_SUCCESS
The function has been executed successfully.

HSA_STATUS_ERROR_NOT_INITIALIZED
The runtime has not been initialized.

HSA_STATUS_ERROR_INVALID_ARGUMENT
If *agent* or *image_handle* is NULL.

Description

If successful, the image handle previously created using **hsa_ext_image_create_handle** is destroyed.

Destroying the image handle does not free the associated image data.

The image handle should not be destroyed while there are references to it queued for execution or currently being used in a dispatch. Failure to properly track image data lifetime causes undefined results due to premature image handle deletion.

3.3.1.25 hsa_ext_sampler_create_handle

```
hsa_status_t hsa_ext_sampler_create_handle(
    const hsa_agent_t * agent,
    const hsa_ext_sampler_descriptor_t * sampler_descriptor,
    hsa_ext_sampler_handle_t * sampler_handle)
```

Create an HSA component-defined sampler handle from a component-independent sampler descriptor.

Parameters

agent
(in) HSA agent to be associated with the image.

sampler_descriptor
(in) Implementation-independent sampler descriptor.

sampler_handle
(out) Component-specific sampler handle.

Return Values

HSA_STATUS_SUCCESS
The function has been executed successfully.

HSA_STATUS_ERROR_NOT_INITIALIZED
The runtime has not been initialized.

HSA_STATUS_ERROR_INVALID_ARGUMENT
If any of the arguments is NULL.

HSA_STATUS_ERROR_OUT_OF_RESOURCES
If the HSA agent cannot create the specified handle because it is out of resources.

Description

If successful, the sampler handle is written to the location specified by the sampler handle.

3.3.1.26 hsa_ext_sampler_destroy_handle

```
hsa_status_t hsa_ext_sampler_destroy_handle(
    const hsa_agent_t * agent,
    hsa_ext_sampler_handle_t * sampler_handle)
```

Destroys the specified sampler handle.

Parameters

agent
(in) HSA agent to be associated with the image.

sampler_handle
(in) Sampler handle.

Return Values

HSA_STATUS_SUCCESS
The function has been executed successfully.

HSA_STATUS_ERROR_NOT_INITIALIZED
The runtime has not been initialized.

HSA_STATUS_ERROR_INVALID_ARGUMENT
If any of the arguments is NULL.

Description

If successful, the sampler handle previously created using **hsa_ext_sampler_create_handle** is destroyed.

The sampler handle should not be destroyed while there are references to it queued for execution or currently being used in a dispatch.

3.4 Component Initiated Dispatches

Due to architected support for a queue and design of AQL, HSA supports component-initiated dispatch, which is the ability for a kernel to dispatch a new kernel by writing an AQL packet directly to a user queue. In simple use cases, the AQL packet can be created on the host and passed as a parameter to the kernel. This eliminates the need to do dynamic memory allocation on the component, but has the limitation that the problem fanout must be known at the time the first kernel is launched (so that the AQL packets can be preallocated). HSA also supports more advanced use cases where the AQL packet is dynamically allocated (including the memory space for kernel arguments and spill/arg/private space) on the component. This usage model obviously requires dynamic component-side memory allocation, for both host and component memory.

Some requirements to do component-initiated dispatch:

- Ability to dynamically choose a kernel to dispatch: Let us assume for example that there are three kernels (A, B and C). If the host launches A, then the user has the choice of launching B or C, or even A in case of recursion. So, the user should be able to get the ISA and segment size (HsaAqlKernel) from the corresponding BRIG dynamically. [caveat: The code sample here does not show how we can do this. It assumes that the HsaAqlKernel is being passed as an argument to the parent kernel (A in this case)]
- Ability to dynamically allocate memory from the shader: We need to allocate memory for AQLPacket, different kernel segments in the AQLPacket, kernel arguments, and so forth.
- Ability for a finalizer to identify a default HSA queue to write AQLPacket: The HSA queue information resides in the runtime layer of the stack. This needs to be exchanged with the compiler so it can be stored in the global space. This way, when the compiler sees the queue, it knows where to pick the HSA queue information to write the AQL-Packet.
- Ability to notify the completion of all the component-initiated dispatches on the host:
  - The beginning of execution of the child kernel may or may not wait for the parent kernel's completion. This is determined by the user and could be algorithm dependent.
  - If the parent (initiated from host) kernel finishes successfully, it means all kernels it initiated also finished successfully.
  - To implement this, we need to track the list of kernels launched from the parent. Change the status of parent to complete, only if parent and all its child kernels have completed successfully.

Implementations that support component initiated dispatches will need to support these requirements. If the implementation supports the stated requirements, the following actions will allow a component to initiate a dispatch:

- The queue and hsa_ext_code_descriptor_t (describing the kernel to launch) can be passed as arguments to the parent (the one launched from the host) kernel. If the dispatch is to the same queue, it is accessible via an HSAIL instruction.
- If not, get the HsaAqlKernel from the BRIG for the kernel that is chosen to be dynamically dispatched.
- When new work is to be created, the HSAIL code would:
  - Use the kernel dynamic memory allocator to allocate a new AQLPacket.
  - Use inline HSAIL to replicate the functionality of the HsaInitAQLPacket function. We could perhaps provide an HSAIL library to implement this functionality. Recall this function:
    * Copies some fields from the HsaAqlKernel structure (for example, the kernel ISA) to the AQLPacket

    * Uses a host allocator to allocate memory for the kernel arguments
    * Uses a component allocator to allocate memory for spill, private, and arg segments

- The HSAIL knows the signature of the called function and can fill in the AQL packet with regular HSAIL global store instructions.
- The HSA queue is architected, so the HSAIL can use memory store instructions to dispatch the kernel for dispatch. Depending how the user queues are configured, atomic accesses might be necessary to handle contention with other writers. Note that, if the queue information is not passed in as an argument, the default queue can be chosen by the finalizer as it was exchanged earlier from the runtime layer.
- We also need to handle deallocation of the kernel arguments and spill/private/arg space after the kernel completes.
- On the host, check if the parent has finished. If the parent has finished successfully, then it means that all the child kernels have finished successfully too. If the parent or any of the child kernels failed, an error code will be returned.

Index - Core APIs

Index - Extension APIs

Bibliography

[1] HSA Foundation. The HSA Programmer's Reference Manual. Technical report, HSA Foundation, 2013. 1, 18, 97

[2] HSA Foundation. The HSA Platform System Architecture Specification. Technical report, HSA Foundation, 2014. 3, 4