HSA Runtime Programmer's Reference Manual

1.00 Provisional

August 7, 2014

Acknowledgments

This specification is the result of the contributions of many people. Here is a partial list of the contributors, including the company that they represented at the time of their contribution:

AMD
Blinzer, Paul
Méndez-Lojo, Mario (spec editor)
Sander, Ben
Thangirala, Hari
Tipparaju, Vinod
Tye, Tony
Zhuravlyov, Konstantin

ARM
Kovacevic, Djordje
Parker, Jason
Persson, Håkan

Codeplay
Potter, Ralph
Richards, Andrew (workgroup chair)

Imagination
Aldis, James
Glew, Andy
Howson, John
McCarthy, James
Meredith, Jason
Rankilor, Mark

Mediatek
Agarwal, Rahul
Bagley, Richard
Hsu, Barz
Huang, Emerson
Ju, Roy
Lin, Jason
Lo, Trent

Qualcomm
Bin, Lihan
Bourd, Alex
Gaster, Ben
Howes, Lee
Rychlik, Bob
Simpson, Robert J.

Samsung Electronics
Kohii, Soma
Llamas, Ignacio
Ryu, Soojung

Shebanow, Michael

Sandia National Laboratories
Hammond, Simon D.
Stark, Dylan

Via Alliance Technologies
Hong, Mike

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 HSA 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 multi-core CPUs. For more information on the system architecture, refer to the HSA Platform System Architecture Specification, version 1.0 'Provisional Ratified' [2].

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. The finalizer can be run at compile-time, install-time, or run-time. Each HSA component provides its own implementation of the finalizer. For more information on HSAIL, refer to the HSA Programmer’s Reference Manual, version 1.0 'Provisional Ratified' [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
- System and Agent information
- Signals and synchronization
- Architected dispatch
- Memory management

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.

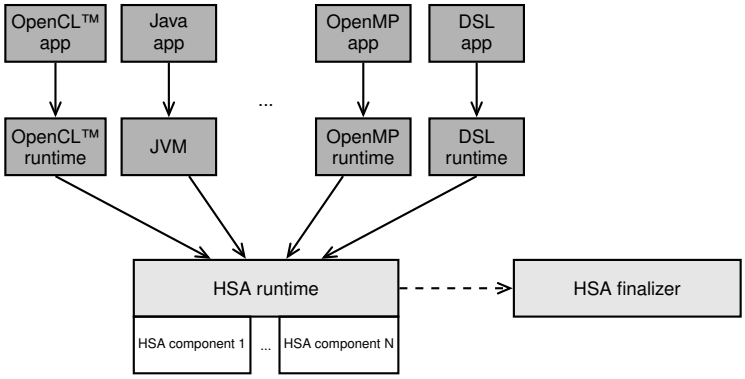

Figure 1.1: HSA Software Architecture

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 application 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 vendors' 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 Programming Model

This section introduces the main concepts behind the HSA programming model by outlining how they are exposed in the runtime API. In this introductory example we show the basic steps that are needed to launch a kernel.

The rest of the sections in this specification provide a more formal and detailed description of the different components of the HSA API, including many not discussed here.

1.2.1 Initialization and Component Discovery

Any HSA application must initialize the runtime before invoking any other API:

```
hsa_init();
```

The next step is to find a device where to launch the kernel. In HSA parlance, a regular device is called an *agent*, and if the agent can run kernels then it is also a *component*. The Glossary at the end of this document contains more precise definitions of these terms. Agents and components are represented in the HSA API using opaque handles of type hsa_agent_t.

The HSA runtime API exposes the set of available agents via **hsa_iterate_agents**. This function receives a callback and a buffer from the application; the callback is invoked once per agent unless it returns a special 'break' value or an error. In this case, the callback queries an agent attribute (HSA_AGENT_INFO_FEATURE) in order to determine whether the agent is also a component. If this is the case, the component is stored in the buffer and the iteration ends:

```
hsa_agent_t component;
hsa_iterate_agents(get_component, &component);
```

, where the application-provided callback *get_component* is:

```
hsa_status_t get_component(hsa_agent_t agent, void* data) {
    uint32_t features = 0;
    hsa_agent_get_info(agent, HSA_AGENT_INFO_FEATURE, &features);
    if (features & HSA_AGENT_FEATURE_DISPATCH) {
        // Store component in the application-provided buffer and return
        hsa_agent_t* ret = (hsa_agent_t*) data;
        *ret = agent;
        return HSA_STATUS_INFO_BREAK;
    }
    // Keep iterating
    return HSA_STATUS_SUCCESS;
}
```

Section 2.3 lists the set of available agent and system-wide attributes, and describes the functions to query them.

1.2.2 Queues and AQL packets

When an HSA application wants to launch a kernel in a component, it does so by placing an *AQL packet* in a *queue* owned by the component. A packet is a memory buffer encoding a single command. There are different types of packets; the one used for dispatching a kernel is named *Dispatch* packet.

The binary structure of the different packet types is defined in the HSA Architecture Specification [2] standard. For example, all the packets types occupy 64 bytes of storage and share a common header, and the

Dispatch packets should specify the address of the ISA to run at offset 32. The packet structure is known to the application (Dispatch packets correspond to the hsa_dispatch_packet_t type in the HSA API), but also to the hardware. This is a key HSA feature that enables applications to launch a packet in a specific agent by simply placing it in one of its *queues*.

A queue is a runtime-allocated resource that contains a packet buffer and a packet processor. The packet processor tracks which packets in the buffer have already been processed. When it has been informed by the application that a new packet has been enqueued, the packet processor is able to process it because the packet format is standard and the packet contents are self-contained – they include all the necessary information to run a command. The *packet processor* is generally a hardware unit that is aware of the different packet formats.

After introducing the basic concepts related to packets and queues, we can go back to our example and create a queue in the component using **hsa_queue_create**. The queue creation can be configured in multiple ways. In the snippet below the application indicates that the queue should be able to hold 256 packets.

```
hsa_queue_t *queue;
hsa_queue_create(component, 256, HSA_QUEUE_TYPE_SINGLE, NULL, NULL, &queue);
```

The next step is to create a packet and push it into the newly created queue. Packets are not created using an HSA runtime function. Instead, the application can directly access the packet buffer of any queue and setup a kernel dispatch by simply filling all the fields mandated by the Agent packet format (type hsa_dispatch_packet_t). The location of the packet buffer is available in the *base_address* field of any queue:

```
hsa_dispatch_packet_t* dispatch_packet = (hsa_dispatch_packet_t*) queue->base_address;

// Configure dispatch dimensions: use a total of 256 work-items
dispatch_packet->dimensions = 1;
dispatch_packet->grid_size_x = 256;

// Configuration of the rest of the Dispatch packet is omitted for simplicity
```

In a real-world scenario, the application needs to exercise more caution when enqueuing a packet – there could be another thread writing a packet to the same memory location. The HSA API exposes several functions that allow the application to determine the buffer index where to write a packet, and when to write it. For more information on queues, refer to Section 2.5. For more information on AQL packets, refer to Section 2.6.

1.2.3 Signals and Packet launch

The Dispatch packet is not launched until the application informs the packet processor that there is new work available. The notification is divided in two parts:

1. The *type* field in the packet header must be atomically set to the appropriate value using a release memory ordering. This ensures that the modifications to the rest of the packet listed before are globally visible before or at the same time the desired packet type is visible. One possible implementation of the atomic storage (in GCC) is:

   ```
   __atomic_store_n((uint8_t*) &dispatch_packet->header, (uint8_t) HSA_PACKET_TYPE_DISPATCH, __ATOMIC_RELEASE);
   ```

   , where the specified address is that of the *header* field, and not *type*, because the C standard disallows taking the address of a bit-field.

2. The buffer index where the packet has been written (in the example, zero) must be stored in the *doorbell signal* of the queue.

A *signal* is a runtime-allocated, opaque object used for communication between agents in an HSA system. Signals are similar to shared memory locations containing an integer. Agents can atomically store a new integer value in a signal, atomically read the current value of the signal, etc. using HSA runtime functions. Signals are the preferred communication mechanism in an HSA system because signal operations usually perform better (in terms of power or speed) than their shared memory counterparts. For more information on signals, refer to Section 2.4.

When the runtime creates a queue, it also automatically creates a “doorbell” signal that must be used by the application to communicate with the packet processor and inform it of the index of the packet ready to be consumed. The doorbell signal is contained in the *doorbell_signal* field of the queue. The value of a signal can be updated using **hsa_signal_store_release**:

```
hsa_signal_store_release(queue->doorbell_signal, 0);
```

After the packet processor has been notified, the execution of the kernel may start asynchronously at any moment. The application could simultaneously write more packets to launch other kernels in the same queue.

In this introductory example, we omitted some important steps in the dispatch process. In particular, we did not discuss how to compile a kernel, indicate which ISA to run in the Dispatch packet, or how to pass arguments to the kernel. However, some relevant differences with other runtime systems and programming models are already evident. Other runtime systems provide software APIs for setting arguments and launching kernels, while HSA architects these at the hardware and specification level. An HSA application can use regular memory operations and a very lightweight set of runtime APIs to launch a kernel or in general submit a packet.

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. The API follows the requirements listed in the HSA Programmer's Reference Manual, version 1.0 'Provisional Ratified' [1], and the HSA Platform System Architecture Specification, version 1.0 'Provisional Ratified' [2].

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 exist 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, agent 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 may release resources associated with it.

2.2 Runtime Notifications

The runtime can report notifications (errors or events) synchronously or asynchronously. The runtime uses the return value of functions in the HSA API to pass synchronous notifications to the application. 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 an HSA function does not execute successfully, the returned status code might help determining the source of the error. While some conditions can be generalized to a certain degree (e.g. failure in allocating resources), others have implementation-specific explanations. For example, certain operations on signals (explained in Section 2.4) might fail if the runtime implementation validates the signal object passed by the application. 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, it invokes an application-defined callback. For example, queues (described in Section 2.5) are a common source of asynchronous events because the tasks queued by an application are asynchronously consumed by the packet processor. When the runtime detects an error in a queue, it invokes the callback associated with that queue and passes it a status code (indicating what happened) and a pointer to the erroneous queue. Callbacks are associated with queues when they are created.

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

```
typedef enum {
    HSA_STATUS_SUCCESS = 0,
    HSA_STATUS_INFO_BREAK = 0x1,
    HSA_EXT_STATUS_INFO_ALREADY_INITIALIZED = 0x4000,
    HSA_EXT_STATUS_INFO_UNRECOGNIZED_OPTIONS = 0x4001,
    HSA_STATUS_ERROR = 0x10000,
    HSA_STATUS_ERROR_INVALID_ARGUMENT = 0x10001,
    HSA_STATUS_ERROR_INVALID_QUEUE_CREATION = 0x10002,
    HSA_STATUS_ERROR_INVALID_ALLOCATION = 0x10003,
    HSA_STATUS_ERROR_INVALID_AGENT = 0x10004,
    HSA_STATUS_ERROR_INVALID_REGION = 0x10005,
    HSA_STATUS_ERROR_INVALID_SIGNAL = 0x10006,
    HSA_STATUS_ERROR_INVALID_QUEUE = 0x10007,
    HSA_STATUS_ERROR_OUT_OF_RESOURCES = 0x10008,
    HSA_STATUS_ERROR_INVALID_PACKET_FORMAT = 0x10009,
    HSA_STATUS_ERROR_RESOURCE_FREE = 0x1000A,
    HSA_STATUS_ERROR_NOT_INITIALIZED = 0x1000B,
    HSA_STATUS_ERROR_REFCOUNT_OVERFLOW = 0x1000C,
    HSA_EXT_STATUS_ERROR_DIRECTIVE_MISMATCH = 0x14000,
```

```
    HSA_EXT_STATUS_ERROR_IMAGE_FORMAT_UNSUPPORTED = 0x14001,
    HSA_EXT_STATUS_ERROR_IMAGE_SIZE_UNSUPPORTED = 0x14002
} hsa_status_t;
```

Values

HSA_STATUS_SUCCESS
: The function has been executed successfully.

HSA_STATUS_INFO_BREAK
: A traversal over a list of elements has been interrupted by the application before completing.

HSA_EXT_STATUS_INFO_ALREADY_INITIALIZED
: An initialization attempt failed due to prior initialization.

HSA_EXT_STATUS_INFO_UNRECOGNIZED_OPTIONS
: The finalization options cannot be recognized.

HSA_STATUS_ERROR
: A generic error has occurred.

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_QUEUE_CREATION
: The requested queue creation is not valid.

HSA_STATUS_ERROR_INVALID_ALLOCATION
: The requested allocation is not valid.

HSA_STATUS_ERROR_INVALID_AGENT
: The agent is invalid.

HSA_STATUS_ERROR_INVALID_REGION
: The memory region 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
: The AQL packet is malformed.

HSA_STATUS_ERROR_RESOURCE_FREE
: An error has been detected while releasing a resource.

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
: Mismatch between a directive in the control directive structure and in the HSAIL kernel.

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,
    const char ** status_string);
```

Query additional information about a status code.

Parameters

status
(in) Status code.

status_string
(out) A NUL-terminated string that describes the error status.

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

2.3 System and Agent Information

The HSA runtime API represents agents using opaque handles of type hsa_agent_t. The application can traverse the list of agents that are available in the system using **hsa_iterate_agents**, and query agent-specific attributes using **hsa_agent_get_info**. Examples of agent attributes include: name, type of backing device (CPU, GPU), and supported queue types.

If an agent supports Dispatch packets, then it is also a component (supports the AQL packet format and the HSAIL instruction set). The application might inspect the HSA_AGENT_INFO_FEATURE attribute in order to determine if the agent is a component. Components expose a rich set of attributes related to kernel dispatches such as wavefront size or maximum number of work-items in the grid.

Implementations of **hsa_iterate_agents** are required to at least report:

- The host CPU agent.
- One component.

The application might also query system-wide attributes using **hsa_system_get_info**. Note that the value of some attributes is not constant: for example, the current timestamp (HSA_SYSTEM_INFO_TIMESTAMP) value returned by the runtime might increase as time progresses. For more information on timestamps, please refer to [2], Section 2.5.

2.3.1 API

2.3.1.1 hsa_system_info_t

```
typedef enum {
    HSA_SYSTEM_INFO_VERSION_MAJOR,
    HSA_SYSTEM_INFO_VERSION_MINOR,
    HSA_SYSTEM_INFO_TIMESTAMP,
    HSA_SYSTEM_INFO_TIMESTAMP_FREQUENCY,
    HSA_SYSTEM_INFO_SIGNAL_MAX_WAIT
} hsa_system_info_t;
```

System attributes.

Values

HSA_SYSTEM_INFO_VERSION_MAJOR
: Major version of the HSA runtime specification supported by the implementation. The type of this attribute is uint16_t.

HSA_SYSTEM_INFO_VERSION_MINOR
: Minor version of the HSA runtime specification supported by the implementation. The type of this attribute is uint16_t.

HSA_SYSTEM_INFO_TIMESTAMP
: Current timestamp. The value of this attribute monotonically increases at a constant rate. The type of this attribute is uint64_t.

HSA_SYSTEM_INFO_TIMESTAMP_FREQUENCY
: Timestamp value increase rate, in MHz. The timestamp (clock) frequency is in the range 1-400MHz. The type of this attribute is uint16_t.

HSA_SYSTEM_INFO_SIGNAL_MAX_WAIT
: Maximum duration of a signal wait operation. Expressed as a count based on the timestamp frequency. The type of this attribute is uint64_t.

2.3.1.2 hsa_system_get_info

```
hsa_status_t hsa_system_get_info(
    hsa_system_info_t attribute,
    void * value);
```

Get the current value of a system attribute.

Parameters

attribute
(in) Attribute to query.

value
(out) Pointer to an application-allocated buffer where to store the value of the attribute. If the buffer passed by the application is not large enough to hold the value of *attribute*, the behavior is undefined.

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 *attribute* is not a valid system attribute, or *value* is NULL.

2.3.1.3 hsa_agent_t

```
typedef uint64_t hsa_agent_t;
```

Opaque handle representing an agent, a device that participates in the HSA memory model. An agent can submit AQL packets for execution, and may also accept AQL packets for execution (Agent Dispatch packets or Dispatch packets launching HSAIL-derived binaries).

2.3.1.4 hsa_agent_feature_t

```
typedef enum {
    HSA_AGENT_FEATURE_DISPATCH = 1,
    HSA_AGENT_FEATURE_AGENT_DISPATCH = 2
} hsa_agent_feature_t;
```

Agent features.

Values

HSA_AGENT_FEATURE_DISPATCH
The agent supports AQL packets of Dispatch type. If this feature is enabled, the agent is also a component.

HSA_AGENT_FEATURE_AGENT_DISPATCH
The agent supports AQL packets of Agent Dispatch type.

2.3.1.5 hsa_device_type_t

```
typedef enum {
    HSA_DEVICE_TYPE_CPU = 0,
    HSA_DEVICE_TYPE_GPU = 1,
    HSA_DEVICE_TYPE_DSP = 2
} hsa_device_type_t;
```

Hardware device type.

Values

HSA_DEVICE_TYPE_CPU
: CPU device.

HSA_DEVICE_TYPE_GPU
: GPU device.

HSA_DEVICE_TYPE_DSP
: DSP device.

2.3.1.6 hsa_agent_info_t

```
typedef enum {
    HSA_AGENT_INFO_NAME,
    HSA_AGENT_INFO_VENDOR_NAME,
    HSA_AGENT_INFO_FEATURE,
    HSA_AGENT_INFO_WAVEFRONT_SIZE,
    HSA_AGENT_INFO_WORKGROUP_MAX_DIM,
    HSA_AGENT_INFO_WORKGROUP_MAX_SIZE,
    HSA_AGENT_INFO_GRID_MAX_DIM,
    HSA_AGENT_INFO_GRID_MAX_SIZE,
    HSA_AGENT_INFO_FBARRIER_MAX_SIZE,
    HSA_AGENT_INFO_QUEUES_MAX,
    HSA_AGENT_INFO_QUEUE_MAX_SIZE,
    HSA_AGENT_INFO_QUEUE_TYPE,
    HSA_AGENT_INFO_NODE,
    HSA_AGENT_INFO_DEVICE,
    HSA_AGENT_INFO_CACHE_SIZE,
    HSA_EXT_AGENT_INFO_IMAGE1D_MAX_DIM,
    HSA_EXT_AGENT_INFO_IMAGE2D_MAX_DIM,
    HSA_EXT_AGENT_INFO_IMAGE3D_MAX_DIM,
    HSA_EXT_AGENT_INFO_IMAGE_ARRAY_MAX_SIZE,
    HSA_EXT_AGENT_INFO_IMAGE_RD_MAX,
    HSA_EXT_AGENT_INFO_IMAGE_RDWR_MAX,
    HSA_EXT_AGENT_INFO_SAMPLER_MAX
} hsa_agent_info_t;
```

Agent attributes.

Values

HSA_AGENT_INFO_NAME
: Agent name. The type of this attribute is a NUL-terminated char[64].

HSA_AGENT_INFO_VENDOR_NAME
: Name of vendor. The type of this attribute is a NUL-terminated char[64].

HSA_AGENT_INFO_FEATURE

Agent capability. The type of this attribute is hsa_agent_feature_t.

HSA_AGENT_INFO_WAVEFRONT_SIZE
Number of work-items in a wavefront. Must be a power of 2 in the range [1,256]. The value of this attribute is undefined if the agent is not a component. The type of this attribute is uint32_t.

HSA_AGENT_INFO_WORKGROUP_MAX_DIM
Maximum number of work-items of each dimension of a work-group. Each maximum must be greater than 0. No maximum can exceed the value of HSA_AGENT_INFO_WORKGROUP_MAX_SIZE. The value of this attribute is undefined if the agent is not a component. The type of this attribute is uint16_t[3].

HSA_AGENT_INFO_WORKGROUP_MAX_SIZE
Maximum total number of work-items in a work-group. The value of this attribute is undefined if the agent is not a component. The type of this attribute is uint32_t.

HSA_AGENT_INFO_GRID_MAX_DIM
Maximum number of work-items of each dimension of a grid. Each maximum must be greater than 0, and must not be smaller than the corresponding value in HSA_AGENT_INFO_WORKGROUP_MAX_-DIM. No maximum can exceed the value of HSA_AGENT_INFO_GRID_MAX_SIZE. The value of this attribute is undefined if the agent is not a component. The type of this attribute is hsa_dim3_t.

HSA_AGENT_INFO_GRID_MAX_SIZE
Maximum total number of work-items in a grid. The value of this attribute is undefined if the agent is not a component. The type of this attribute is uint32_t.

HSA_AGENT_INFO_FBARRIER_MAX_SIZE
Maximum number of fbarriers per work-group. Must be at least 32. The value of this attribute is undefined if the agent is not a component. The type of this attribute is uint32_t.

HSA_AGENT_INFO_QUEUES_MAX
Maximum number of queues that can be active (created but not destroyed) at one time in the agent. The type of this attribute is uint32_t.

HSA_AGENT_INFO_QUEUE_MAX_SIZE
Maximum number of packets that a queue created in the agent can hold. Must be a power of 2 and greater than 0. The type of this attribute is uint32_t.

HSA_AGENT_INFO_QUEUE_TYPE
Type of a queue created in the agent. The type of this attribute is hsa_queue_type_t.

HSA_AGENT_INFO_NODE
Identifier of the NUMA node associated with the agent. The type of this attribute is uint32_t.

HSA_AGENT_INFO_DEVICE
Type of hardware device associated with the agent. The type of this attribute is hsa_device_type_t.

HSA_AGENT_INFO_CACHE_SIZE
Array of data cache sizes (L1..L4). Each size is expressed in bytes. A size of 0 for a particular level indicates that there is no cache information for that level. The type of this attribute is uint32_t[4].

HSA_EXT_AGENT_INFO_IMAGE1D_MAX_DIM
Maximum dimensions (width, height, depth) of one-dimensional images, in image elements. The Y and Z dimensions maximums must be 0. The X maximum must be at most 16384. The value of this attribute is undefined if the agent is not a component, or the implementation does not support images. The type of this attribute is hsa_dim3_t.

HSA_EXT_AGENT_INFO_IMAGE2D_MAX_DIM
Maximum dimensions (width, height, depth) of two-dimensional images, in image elements. The Z dimension maximum must be 0. The X and Y maximums must be at most 16384. The value of this attribute is undefined if the agent is not a component, or the implementation does not support images. The type of this attribute is hsa_dim3_t.

HSA_EXT_AGENT_INFO_IMAGE3D_MAX_DIM
: Maximum dimensions (width, height, depth) of three-dimensional images, in image elements. The maximum along any dimension cannot exceed 2048. The value of this attribute is undefined if the agent is not a component, or the implementation does not support images. The type of this attribute is hsa_dim3_t.

HSA_EXT_AGENT_INFO_IMAGE_ARRAY_MAX_SIZE
: Maximum number of image layers in a image array. Must not exceed 2048. The value of this attribute is undefined if the agent is not a component, or the implementation does not support images. The type of this attribute is uint32_t.

HSA_EXT_AGENT_INFO_IMAGE_RD_MAX
: Maximum number of read-only image handles that can be created at any one time. Must be at least 128. The value of this attribute is undefined if the agent is not a component, or the implementation does not support images. The type of this attribute is uint32_t.

HSA_EXT_AGENT_INFO_IMAGE_RDWR_MAX
: Maximum number of write-only and read-write image handles that can be created at any one time. Must be at least 64. The value of this attribute is undefined if the agent is not a component, or the implementation does not support images. The type of this attribute is uint32_t.

HSA_EXT_AGENT_INFO_SAMPLER_MAX
: Maximum number of sampler handlers that can be created at any one time. Must be at least 16. The value of this attribute is undefined if the agent is not a component, or the implementation does not support images. The type of this attribute is uint32_t.

2.3.1.7 hsa_agent_get_info

```
hsa_status_t hsa_agent_get_info(
    hsa_agent_t agent,
    hsa_agent_info_t attribute,
    void * value);
```

Get the current value of an attribute for a given agent.

Parameters

agent
: (in) A valid agent.

attribute
: (in) Attribute to query.

value
: (out) Pointer to an application-allocated buffer where to store the value of the attribute. If the buffer passed by the application is not large enough to hold the value of *attribute*, the behavior is undefined.

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_AGENT
: If the agent is invalid.

HSA_STATUS_ERROR_INVALID_ARGUMENT
: If *attribute* is not a valid agent attribute, or *value* is NULL.

2.3.1.8 hsa_iterate_agents

```
hsa_status_t hsa_iterate_agents(
    hsa_status_t (*callback)(hsa_agent_t agent, void *data),
    void * data);
```

Iterate over the available agents, and invoke an application-defined callback on every iteration.

Parameters

callback
: (in) Callback to be invoked once per agent. The runtime passes two arguments to the callback, the agent and the application data. If *callback* returns a status other than HSA_STATUS_SUCCESS for a particular iteration, the traversal stops and **hsa_iterate_agents** returns that status value.

data
: (in) Application data that is passed to *callback* on every iteration. Might be NULL.

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 *callback* is NULL.

2.4 Signals

HSA agents can communicate with each other by using coherent shared (global) memory or by using signals. Agents can perform operations on signals similar to those on shared memory locations: atomically store an integer value on them, atomically load their current value, etc. However, signals can only be manipulated using the HSA runtime API or HSAIL instructions. The advantage of signals over shared memory is that signal operations usually perform better in terms of power or speed. For example, a spin loop involving atomic memory operations that waits for a shared memory location to satisfy a condition can be replaced with an HSA signal wait operator such as **hsa_signal_wait_acquire**, which is implemented by the runtime using efficient hardware features.

The runtime API uses opaque signal handlers of type hsa_signal_t to represent signals. A signal carries an integer value of type hsa_signal_value_t that can be accessed or conditionally waited upon through an API call or HSAIL instruction. The value occupies four or eight bytes depending on the machine model (small or large, respectively) being used. The application creates a signal using **hsa_signal_create**.

Modifying 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 read or write signal operation specifies which memory order to use. 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. For more information on memory orders and the HSA memory model, please refer to the other HSA specifications [1, 2].

The application may wait on a signal, with a condition specifying the terms of the wait. The wait can be done either in the HSA component by using an HSAIL **wait** instruction or in the host CPU by using a runtime API call. Waiting for a signal 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 signal value returned by the wait operation is not guaranteed to satisfy the wait condition due to multiple reasons:

- A spurious wakeup interrupts the wait.
- The wait time exceeded the user-specified timeout.
- The wait time exceeded the system timeout HSA_SYSTEM_INFO_SIGNAL_MAX_WAIT.
- The wait has been interrupted because the signal value satisfies the specified condition, but the value is modified before the implementation of the wait operation has the opportunity of reading it.

2.4.1 API

2.4.1.1 hsa_signal_value_t

```
 #ifdef HSA_LARGE_MODEL
     typedef int64_t hsa_signal_value_t
#else
     typedef int32_t hsa_signal_value_t
#endif
```

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

2.4.1.2 hsa_signal_t

```
typedef uint64_t hsa_signal_t;
```

Signal handle. The value 0 is reserved.

2.4.1.3 hsa_signal_create

```
hsa_status_t hsa_signal_create(
    hsa_signal_value_t initial_value,
    uint32_t num_consumers,
    const hsa_agent_t * consumers,
    hsa_signal_t * signal);
```

Create a signal.

Parameters

initial_value
(in) Initial value of the signal.

num_consumers
(in) Size of *consumers*. A value of 0 indicates that any HSA agent might wait on the signal.

consumers
(in) List of agents that might consume (wait on) the signal. If *num_consumers* is 0, this argument is ignored; otherwise, the runtime might use the list to optimize the handling of the signal object. If an agent not listed in *consumers* waits on the returned signal, the behavior is undefined. The memory associated with *consumers* can be reused or freed after the function returns.

signal
(out) Pointer to a memory location where the runtime will store the newly created 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* is NULL or *num_consumers* is greater than 0 but *consumers* is NULL.

2.4.1.4 hsa_signal_destroy

```
hsa_status_t hsa_signal_destroy(
    hsa_signal_t signal);
```

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

Parameters

signal
(in) Signal.

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* is invalid.

HSA_STATUS_ERROR_INVALID_ARGUMENT
If *signal* is 0.

2.4.1.5 hsa_signal_load

```
hsa_signal_value_t hsa_signal_load_acquire(
    hsa_signal_t signal);

hsa_signal_value_t hsa_signal_load_relaxed(
    hsa_signal_t signal);
```

Atomically read the current value of a signal.

Parameters

signal
(in) Signal.

Returns

Value of the signal.

2.4.1.6 hsa_signal_store

```
void hsa_signal_store_relaxed(
    hsa_signal_t signal,
    hsa_signal_value_t value);

void hsa_signal_store_release(
    hsa_signal_t signal,
    hsa_signal_value_t value);
```

Atomically set the value of a signal.

Parameters

signal
(in) Signal.

value
(in) New signal value.

Description

If the value of the signal is modified, all the agents waiting on *signal* for which *value* satisfies their wait condition are awakened.

2.4.1.7 hsa_signal_exchange

```
hsa_signal_value_t hsa_signal_exchange_acq_rel(
    hsa_signal_t signal,
    hsa_signal_value_t value);

hsa_signal_value_t hsa_signal_exchange_acquire(
    hsa_signal_t signal,
    hsa_signal_value_t value);

hsa_signal_value_t hsa_signal_exchange_relaxed(
    hsa_signal_t signal,
    hsa_signal_value_t value);

hsa_signal_value_t hsa_signal_exchange_release(
    hsa_signal_t signal,
    hsa_signal_value_t value);
```

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

Parameters

signal
(in) Signal. If *signal* is a queue doorbell signal, the behavior is undefined.

value
(in) New value.

Returns

Value of the signal prior to the exchange.

Description

If the value of the signal is modified, all the agents waiting on *signal* for which *value* satisfies their wait condition are awakened.

2.4.1.8 hsa_signal_cas

```
hsa_signal_value_t hsa_signal_cas_acq_rel(
    hsa_signal_t signal,
    hsa_signal_value_t expected,
    hsa_signal_value_t value);

hsa_signal_value_t hsa_signal_cas_acquire(
    hsa_signal_t signal,
    hsa_signal_value_t expected,
    hsa_signal_value_t value);

hsa_signal_value_t hsa_signal_cas_relaxed(
    hsa_signal_t signal,
    hsa_signal_value_t expected,
    hsa_signal_value_t value);

hsa_signal_value_t hsa_signal_cas_release(
    hsa_signal_t signal,
    hsa_signal_value_t expected,
    hsa_signal_value_t value);
```

Atomically set the value of a signal if the observed value is equal to the expected value. The observed value is returned regardless of whether the replacement was done.

Parameters

signal
(in) Signal. If *signal* is a queue doorbell signal, the behavior is undefined.

expected
(in) Value to compare with.

value
(in) New value.

Returns

Observed value of the signal.

Description

If the value of the signal is modified, all the agents waiting on *signal* for which *value* satisfies their wait condition are awakened.

2.4.1.9 hsa_signal_add

```
void hsa_signal_add_acq_rel(
    hsa_signal_t signal,
    hsa_signal_value_t value);

void hsa_signal_add_acquire(
    hsa_signal_t signal,
    hsa_signal_value_t value);

void hsa_signal_add_relaxed(
    hsa_signal_t signal,
    hsa_signal_value_t value);

void hsa_signal_add_release(
    hsa_signal_t signal,
    hsa_signal_value_t value);
```

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

Parameters

signal
(in) Signal. If *signal* is a queue doorbell signal, the behavior is undefined.

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

Description

If the value of the signal is modified, all the agents waiting on *signal* for which *value* satisfies their wait condition are awakened.

2.4.1.10 hsa_signal_subtract

```
void hsa_signal_subtract_acq_rel(
    hsa_signal_t signal,
    hsa_signal_value_t value);

void hsa_signal_subtract_acquire(
    hsa_signal_t signal,
    hsa_signal_value_t value);

void hsa_signal_subtract_relaxed(
    hsa_signal_t signal,
    hsa_signal_value_t value);

void hsa_signal_subtract_release(
    hsa_signal_t signal,
    hsa_signal_value_t value);
```

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

Parameters

signal
(in) Signal. If *signal* is a queue doorbell signal, the behavior is undefined.

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

Description

If the value of the signal is modified, all the agents waiting on *signal* for which *value* satisfies their wait condition are awakened.

2.4.1.11 hsa_signal_and

```
void hsa_signal_and_acq_rel(
    hsa_signal_t signal,
    hsa_signal_value_t value);

void hsa_signal_and_acquire(
    hsa_signal_t signal,
    hsa_signal_value_t value);

void hsa_signal_and_relaxed(
    hsa_signal_t signal,
    hsa_signal_value_t value);

void hsa_signal_and_release(
    hsa_signal_t signal,
    hsa_signal_value_t value);
```

Atomically perform a bitwise AND operation between the value of a signal and a given value.

Parameters

signal
(in) Signal. If *signal* is a queue doorbell signal, the behavior is undefined.

value

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

Description

If the value of the signal is modified, all the agents waiting on *signal* for which *value* satisfies their wait condition are awakened.

2.4.1.12 hsa_signal_or

```
void hsa_signal_or_acq_rel(
    hsa_signal_t signal,
    hsa_signal_value_t value);

void hsa_signal_or_acquire(
    hsa_signal_t signal,
    hsa_signal_value_t value);

void hsa_signal_or_relaxed(
    hsa_signal_t signal,
    hsa_signal_value_t value);

void hsa_signal_or_release(
    hsa_signal_t signal,
    hsa_signal_value_t value);
```

Atomically perform a bitwise OR operation between the value of a signal and a given value.

Parameters

signal

(in) Signal. If *signal* is a queue doorbell signal, the behavior is undefined.

value

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

Description

If the value of the signal is modified, all the agents waiting on *signal* for which *value* satisfies their wait condition are awakened.

2.4.1.13 hsa_signal_xor

```
void hsa_signal_xor_acq_rel(
    hsa_signal_t signal,
    hsa_signal_value_t value);

void hsa_signal_xor_acquire(
    hsa_signal_t signal,
    hsa_signal_value_t value);

void hsa_signal_xor_relaxed(
    hsa_signal_t signal,
    hsa_signal_value_t value);
```

```
void hsa_signal_xor_release(
    hsa_signal_t signal,
    hsa_signal_value_t value);
```

Atomically perform a bitwise XOR operation between the value of a signal and a given value.

Parameters

signal
(in) Signal. If *signal* is a queue doorbell signal, the behavior is undefined.

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

Description

If the value of the signal is modified, all the agents waiting on *signal* for which *value* satisfies their wait condition are awakened.

2.4.1.14 hsa_signal_condition_t

```
typedef enum {
    HSA_EQ,
    HSA_NE,
    HSA_LT,
    HSA_GTE
} 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.

2.4.1.15 hsa_wait_expectancy_t

```
typedef enum {
    HSA_WAIT_EXPECTANCY_SHORT,
    HSA_WAIT_EXPECTANCY_LONG,
    HSA_WAIT_EXPECTANCY_UNKNOWN
} hsa_wait_expectancy_t;
```

Expected duration of a wait call.

Values

HSA_WAIT_EXPECTANCY_SHORT
The signal value is expected to meet the specified condition in a short period of time.

HSA_WAIT_EXPECTANCY_LONG
: The signal value is expected to meet the specified condition after a longer period of time.

HSA_WAIT_EXPECTANCY_UNKNOWN
: The expected duration of the wait call is not known.

2.4.1.16 hsa_signal_wait

```
hsa_signal_value_t hsa_signal_wait_acquire(
    hsa_signal_t signal,
    hsa_signal_condition_t condition,
    hsa_signal_value_t compare_value,
    uint64_t timeout_hint,
    hsa_wait_expectancy_t wait_expectancy_hint);

hsa_signal_value_t hsa_signal_wait_relaxed(
    hsa_signal_t signal,
    hsa_signal_condition_t condition,
    hsa_signal_value_t compare_value,
    uint64_t timeout_hint,
    hsa_wait_expectancy_t wait_expectancy_hint);
```

Wait until a signal value satisfies a specified condition, or a certain amount of time has elapsed.

Parameters

signal
: (in) Signal.

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

compare_value
: (in) Value to compare with.

timeout_hint
: (in) Maximum duration of the wait. Specified in the same unit as the system timestamp. A value of UINT64_MAX indicates no maximum.

wait_expectancy_hint
: (in) Hint indicating whether the signal value is expected to meet the given condition in a short period of time or not. The HSA runtime may use this hint to optimize the wait implementation.

Returns

Observed value of the signal, which might not satisfy the specified condition.

Description

A wait operation can spuriously resume at any time sooner than the timeout (for example, due to system or other external factors) even when the condition has not been met.

The function is guaranteed to return if the signal value satisfies the condition at some point in time during the wait, but the value returned to the application might not satisfy the condition. When the wait operation internally loads the value of the passed signal, it uses the memory order indicated in the function name.

The application might indicate a preference about the maximum wait duration. The operation might block for a shorter or longer time even if the condition is not met.

2.5 Queues

HSA hardware supports command execution through user mode queues. A user mode command queue is characterized [2] as runtime-allocated, user-level, accessible virtual memory of a certain size, containing packets (commands) defined in the Architected Queueing Language (AQL is explained in more detail in the next section). A queue is associated with a specific agent, which might have several queues attached to it. We will refer to user mode queues as just queues.

The application submits a packet to the queue of an agent by performing the following steps:

1. Create a queue on the agent, using **hsa_queue_create**. The queue should support the desired packet type. When the queue is created, the runtime allocates memory for the hsa_queue_t data structure that represents the visible part of the queue, as well as the AQL packet buffer pointed by the *base_address* field.
2. Reserve a packet ID by incrementing the write index of the queue, which is a 64-bit unsigned integer that contains the number of packets allocated so far. The runtime exposes several functions such as **hsa_queue_add_write_index_acquire** to increment the value of the write index.
3. Wait until the queue is not full before writing the packet. If the queue is full, the packet ID obtained in the previous step will be greater or equal than the sum of the current read index and the queue size. The read index of a queue is a 64-bit unsigned integer that contains 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). The application can load the read index using **hsa_queue_load_read_index_acquire** or **hsa_queue_load_read_index_relaxed**.

   If 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.
4. Populate the packet. This step does not require using any HSA API. Instead, the application directly writes the contents of the AQL packet located at *base_address* + (AQL packet size) * ((packet ID) % *size*). Note that *base_address* and *size* are fields in the queue structure, while the size of any AQL packet is 64 bytes. The different packet types are discussed in the next section.
5. Launch the packet by first setting the *type* field on its header to the corresponding value, and then storing the packet ID in *doorbell_signal* using **hsa_signal_store_release** (or any variant that uses a different memory order). The application is required to ensure that the rest of the packet is globally visible before or at the same time *type* is written.

   The doorbell signal of the queue is used 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. However, the packet might be consumed by the packet processor even before the doorbell signal has been signaled. 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, agents are required to signal the doorbell for every batch of packets they write.
6. (Optional) Wait for the packet to be complete by waiting on its completion signal, if any.
7. (Optional) Submit more packets by repeating steps 2-6
8. Destroy the queue using **hsa_queue_destroy**.

Queues are semi-opaque objects: there is a visible part, represented by the hsa_queue_t structure and the corresponding ring buffer (pointed by *base_address*), and an invisible part, which contains at least the read and write indexes. The access rules for the different queue parts are:

- The hsa_queue_t structure is read-only. If the application overwrites its contents, the behavior is undefined.
- The ring buffer can be directly accessed by the application.

- The read and write indexes of the queue can only be accessed using dedicated runtime 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 applied(relaxed, release, etc.).

2.5.1 Example - a simple dispatch

In this example, we extend the dispatch code introduced in Section 1.2 in order to illustrate how packet IDs are reserved (invocation of **hsa_queue_add_write_index_relaxed**), and how the application can wait for a packet to be complete (invocation of **hsa_signal_wait_acquire**). The application creates a signal with an initial value of 1, sets the completion signal of the Dispatch packet to be the newly created signal, and after notifying the packet processor it waits for the signal value to become zero. The decrement is performed by the packet processor, and indicates that the kernel has finished.

```
hsa_status_t get_component(hsa_agent_t agent, void* data) {
    uint32_t features = 0;
    hsa_agent_get_info(agent, HSA_AGENT_INFO_FEATURE, &features);
    if (features & HSA_AGENT_FEATURE_DISPATCH) {
        // Store component in the application-provided buffer and return
        hsa_agent_t* ret = (hsa_agent_t*) data;
        *ret = agent;
        return HSA_STATUS_INFO_BREAK;
    }
    // Keep iterating
    return HSA_STATUS_SUCCESS;
}

void packet_type_store_release(hsa_packet_header_t* header, hsa_packet_type_t type) {
    __atomic_store_n((uint8_t*) header, (uint8_t) type, __ATOMIC_RELEASE);
}

int main() {
    // Initialize the runtime
    hsa_init();

    // Retrieve the component
    hsa_agent_t component;
    hsa_iterate_agents(get_component, &component);

    // Create a queue in the component. The queue can hold 4 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);

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

    // Calculate the virtual address where to place the packet
    hsa_dispatch_packet_t* dispatch_packet = (hsa_dispatch_packet_t*) queue->base_address + packet_id;

    // Populate fields in Dispatch packet, except for the completion signal and the header type
    initialize_packet(dispatch_packet);

    // Create a signal with an initial value of one to monitor the task completion
    hsa_signal_t signal;
    hsa_signal_create(1, 0, NULL, &signal);
    dispatch_packet->completion_signal = signal;

    // Notify the queue that the packet is ready to be processed
    packet_type_store_release(&dispatch_packet->header, HSA_PACKET_TYPE_DISPATCH);
    hsa_signal_store_release(queue->doorbell_signal, packet_id);

    // Wait for the task to finish, which is the same as waiting for the value of the completion signal to be zero
    while (hsa_signal_wait_acquire(signal, HSA_EQ, 0, UINT64_MAX, HSA_WAIT_EXPECTANCY_UNKNOWN) != 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;
}
```

2.5.2 Example - error callback

The previous example can be slightly modified to illustrate the usage of queue callbacks. This time the application creates a queue passing a callback function named *callback*:

```
hsa_agent_t component;
hsa_iterate_agents(get_component, &component);

hsa_queue_t *queue;
hsa_queue_create(component, 4, HSA_QUEUE_TYPE_SINGLE, callback, NULL, &queue);
```

The callback prints the ID of the problematic queue, and the string associated with the asynchronous event:

```
void callback(hsa_status_t status, hsa_queue_t* queue) {
  const char* message;
  hsa_status_string(status, &message);
  printf("Error at queue %d: %s", queue->id, message);
}
```

Let's now assume that the application makes a mistake and submits an invalid packet to the queue. For example, the AQL packet type is set to an invalid value. When the packet processor encounters this packet, it will trigger an error that results in the runtime invoking the callback associated with the queue. The message printed to the standard output varies depending on the string returned by **hsa_status_string**. A possible output is:

```
Error at queue 0: Invalid packet format
```

2.5.3 Example - concurrent packet submissions

In previous examples the packet submission is very simple: there is a unique CPU thread submitting a single packet. In this example, we assume a more realistic scenario:

- Multiple threads concurrently submit many packets to the same queue.
- The queue might be full. Threads should avoid overwriting queue slots containing packets that have not been not processed yet.
- The number of packets submitted exceeds the size of the queue, so the submitting thread has to take wrap-around in consideration.

We start by finding a component that allows applications to create queues supporting multiple producers:

```
hsa_status_t get_component(hsa_agent_t agent, void* data) {
    uint32_t features = 0;
    hsa_agent_get_info(agent, HSA_AGENT_INFO_FEATURE, &features);
    if (features & HSA_AGENT_FEATURE_DISPATCH) {
        hsa_queue_type_t queue_type;
        hsa_agent_get_info(agent, HSA_AGENT_INFO_QUEUE_TYPE, &queue_type);
        if (queue_type == HSA_QUEUE_TYPE_MULTI) {
            hsa_agent_t* ret = (hsa_agent_t*)data;
```

```
            *ret = agent;
            return HSA_STATUS_INFO_BREAK;
        }
    }
    return HSA_STATUS_SUCCESS;
}
```

, and then creating a queue on it. The queue type is now HSA_QUEUE_TYPE_MULTI instead of HSA_-QUEUE_TYPE_SINGLE.

```
hsa_agent_t component;
hsa_iterate_agents(get_component, &component);
hsa_queue_t *queue;
hsa_queue_create(component, 4, HSA_QUEUE_TYPE_MULTI, callback, NULL, &queue);
```

Each CPU thread submits 1000 Dispatch packets by executing the function listed below. For simplicity, we omitted the code that creates the CPU threads.

```
void enqueue(hsa_queue_t* queue) {
    // Create a signal with an initial value of 1000 to monitor the overall task completion
    hsa_signal_t signal;
    hsa_signal_create(1000, 0, NULL, &signal);
    hsa_dispatch_packet_t* packets = (hsa_dispatch_packet_t*)queue->base_address;

    for (int i = 0; i < 1000; i++) {
        // Atomically request a new packet ID.
        uint64_t packet_id = hsa_queue_add_write_index_release(queue, 1);

        // Wait until the queue is not full before writing the packet
        while (packet_id - hsa_queue_load_read_index_acquire(queue) >= queue->size);

        // Compute packet offset, considering wrap-around
        hsa_dispatch_packet_t* dispatch_packet = packets + packet_id % queue->size;

        initialize_packet(dispatch_packet);
        dispatch_packet->completion_signal = signal;
        packet_type_store_release(&dispatch_packet->header, HSA_PACKET_TYPE_DISPATCH);
        hsa_signal_store_release(queue->doorbell_signal, packet_id);
    }

    // Wait until all the kernels are complete
    while (hsa_signal_wait_acquire(signal, HSA_EQ, 0, UINT64_MAX, HSA_WAIT_EXPECTANCY_UNKNOWN) != 0);
    hsa_signal_destroy(signal);
}
```

2.5.4 API

2.5.4.1 hsa_queue_type_t

```
typedef enum {
    HSA_QUEUE_TYPE_MULTI = 0,
    HSA_QUEUE_TYPE_SINGLE = 1
} hsa_queue_type_t;
```

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

Values

HSA_QUEUE_TYPE_MULTI
: Queue supports multiple producers.

HSA_QUEUE_TYPE_SINGLE
Queue only supports a single producer.

2.5.4.2 hsa_queue_feature_t

```
typedef enum {
    HSA_QUEUE_FEATURE_DISPATCH = 1,
    HSA_QUEUE_FEATURE_AGENT_DISPATCH = 2
} hsa_queue_feature_t;
```

Queue features.

Values

HSA_QUEUE_FEATURE_DISPATCH
Queue supports Dispatch packets.

HSA_QUEUE_FEATURE_AGENT_DISPATCH
Queue supports Agent Dispatch packets.

2.5.4.3 hsa_queue_t

```
typedef struct hsa_queue_s {
    hsa_queue_type_t type;
    uint32_t features;
    uint64_t base_address;
    hsa_signal_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. This is a bit-field of hsa_queue_feature_t values. Applications should ignore any unknown set bits.

base_address
Starting address of the runtime-allocated buffer used to store the AQL packets. Must be 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 manages the doorbell signal. If the application tries to replace or destroy this signal, the behavior is undefined.
If *type* is HSA_QUEUE_TYPE_SINGLE the doorbell signal value must be updated in a monotonically increasing fashion. If *type* is HSA_QUEUE_TYPE_MULTI, the doorbell signal value can be updated with any value.

size
Maximum number of packets the queue can hold. Must be a power of 2.

id
Queue identifier which is unique per process.

service_queue
A pointer to another user mode queue that can be used by an HSAIL kernel to request runtime or application-defined 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.

2.5.4.4 hsa_queue_create

```
hsa_status_t hsa_queue_create(
    hsa_agent_t agent,
    uint32_t size,
    hsa_queue_type_t type,
    void (*callback)(hsa_status_t status, hsa_queue_t *source),
    const hsa_queue_t * service_queue,
    hsa_queue_t ** queue);
```

Create a user mode queue.

Parameters

agent
(in) Agent where to create the queue.

size
(in) Number of packets the queue is expected to hold. Must be a power of 2 that does not exceed the value of HSA_AGENT_INFO_QUEUE_MAX_SIZE in *agent*.

type
(in) Type of the queue. If the value of HSA_AGENT_INFO_QUEUE_TYPE in *agent* is HSA_QUEUE_TYPE_SINGLE, then *type* must also be HSA_QUEUE_TYPE_SINGLE.

callback
(in) Callback invoked by the runtime for every asynchronous event related to the newly created queue. Might be NULL. The runtime passes two arguments to the callback: a code identifying the event that triggered the invocation, and a pointer to the queue where the event originated.

service_queue
(in) Pointer to a service queue to be associated with the newly created queue. Might be NULL. If not NULL, the queue pointed by *service_queue* must support the HSA_QUEUE_FEATURE_AGENT_DISPATCH feature.

queue
(out) Memory location where the runtime stores a pointer to the newly created 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_OUT_OF_RESOURCES
If there is failure to allocate the resources required by the implementation.

HSA_STATUS_ERROR_INVALID_AGENT

If the agent is invalid.

HSA_STATUS_ERROR_INVALID_QUEUE_CREATION
If *agent* does not support queues of the given type.

HSA_STATUS_ERROR_INVALID_ARGUMENT
If *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 creates the packet buffer, the completion signal, and the write and read indexes. The initial value of the write and read indexes is 0. The type of every packet in the buffer is initialized to HSA_PACKET_TYPE_ALWAYS_RESERVED.

The application should only rely on the error code returned to determine if the queue is valid.

2.5.4.5 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 created using **hsa_queue_create**.

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

When a queue is destroyed, the state of the AQL packets that have not been yet fully processed (their completion phase has not finished) becomes undefined. It is the responsibility of the application to ensure that all pending queue operations are finished if their results are required.

The resources allocated by the runtime during queue creation (queue structure, ring buffer, doorbell signal) are released. The queue should not be accessed after being destroyed.

2.5.4.6 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.

2.5.4.7 hsa_queue_load_read_index

```
uint64_t hsa_queue_load_read_index_acquire(
    hsa_queue_t * queue);

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*.

2.5.4.8 hsa_queue_load_write_index

```
uint64_t hsa_queue_load_write_index_acquire(
    hsa_queue_t * queue);

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*.

2.5.4.9 hsa_queue_store_write_index

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

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.

2.5.4.10 hsa_queue_cas_write_index

```
uint64_t hsa_queue_cas_write_index_acq_rel(
    hsa_queue_t * queue,
    uint64_t expected,
    uint64_t value);

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

uint64_t hsa_queue_cas_write_index_relaxed(
    hsa_queue_t * queue,
    uint64_t expected,
    uint64_t value);

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

Atomically set the write index of a queue if the observed value is equal to the expected value. The application can inspect the returned value to determine if the replacement was done.

Parameters

queue
(in) Pointer to a queue.

expected
(in) Expected value.

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

Returns

Previous value of the write index.

2.5.4.11 hsa_queue_add_write_index

```
uint64_t hsa_queue_add_write_index_acq_rel(
    hsa_queue_t * queue,
    uint64_t value);

uint64_t hsa_queue_add_write_index_acquire(
    hsa_queue_t * queue,
    uint64_t value);

uint64_t hsa_queue_add_write_index_relaxed(
    hsa_queue_t * queue,
    uint64_t value);

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.

2.5.4.12 hsa_queue_store_read_index

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

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.

Description

Modifications of the read index are not allowed and result in undefined behavior if the queue is associated with an HSA agent for which only the corresponding packet processor is permitted to update 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 kernel dispatch. An AQL packet is a user-mode buffer with a defined format encoding one command. The HSA API does not provide any functionality to create, destroy or manipulate AQL packets. Instead, the application uses regular memory operation to access the contents of packets, and user-level allocators (malloc, for example) to create a packet. Applications are not mandated to explicitly reserve any storage space for packets, since a queue already contains a command buffer where AQL packets can be written into.

The HSA API defines the format of three packet types: Dispatch, Agent Dispatch and Barrier. All formats share a common header hsa_packet_type_t that describes their type, barrier bit (force the packet processor to complete packets in order), etc.

2.6.1 Dispatch packet

An application uses a Dispatch packet (hsa_dispatch_packet_t) to submit a kernel to an HSA component. The packet contains the following bits of information:

- A pointer to the kernel ISA is stored in *kernel_object_address*.
- A pointer to the kernel arguments is stored in *kernarg_address*. The application populates this field with the address of a global memory buffer previously allocated using **hsa_memory_allocate**, which contains the dispatch parameters. Memory allocation is explained in Section 2.7, which includes an example on how to reserve space for the kernel arguments.
- Launch dimensions. The application must specify the number of dimensions of the grid (which is also the number of dimensions of the work-group), the size of each grid dimension, and the size of each work-group dimension.
- If the kernel uses group or private memory, the application must specify the storage requirements in the *group_segment_size* and *private_segment_size* fields, respectively.

The application must rely on information provided by the finalizer to retrieve the amount of kernarg, group, and private memory used by a kernel. The kernel's code descriptor (explained in Section 3.1) returned by the finalizer contains the kernarg (*kernarg_segment_byte_size*), group (*workgroup_group_segment_byte_size*) and private (*workitem_private_segment_byte_size*) static storage requirements.

2.6.1.1 Example - populating the Dispatch packet

Examples of kernel dispatches in previous sections have omitted the setup of the Dispatch packet. The code listed below shows how to configure the launch of a kernel that receives no arguments (the *kernel_object_address* field is NULL). The dispatch uses 256 work-items, all in the same work-group along the X dimension.

```
void initialize_packet(hsa_dispatch_packet_t* dispatch_packet) {
    // Contents are zeroed:
    // -Reserved fields must be 0
    // -Type is set to HSA_PACKET_TYPE_ALWAYS_RESERVED, so the packet cannot be consumed by the packet processor
    memset(dispatch_packet, 0, sizeof(hsa_dispatch_packet_t));

    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->workgroup_size_y = 1;
```

```
    dispatch_packet->workgroup_size_z = 1;
    dispatch_packet->grid_size_x = 256;
    dispatch_packet->grid_size_y = 1;
    dispatch_packet->grid_size_z = 1;

    // Indicate which ISA to run. The application is expected to have finalized a kernel (for example, using the finalization API).
    // We will assume that the kernel object location is stored in KERNEL_ADDRESS
    dispatch_packet->kernel_object_address = (uint64_t) KERNEL_ADDRESS;

    // Assume our kernel receives no arguments
    dispatch_packet->kernarg_address = 0;
}
```

2.6.2 Agent Dispatch packet

Agent Dispatch packets are used to launch built-in functions in agents. In Agent Dispatch packets there is no need to indicate the address of the function to run, the launch dimensions, or memory requirements. Instead, the application or kernel simply specifies the type of function to be performed (*type* field), the arguments (*arg*), and when applicable the memory location where to store the return value of the function (*return_address*). The HSA API defines the type hsa_agent_dispatch_packet_t to represent Agent Dispatch packets.

While the host application is allowed to submit Agent Dispatch packets to any destination agent that supports them, in a more common usage scenario the producer will be a kernel executing in a component. The following steps describe the set of actions required from the application, the kernel, and the destination agent:

1. (Application) Locate a destination agent that supports Agent Dispatch packets. The application must check whether the feature HSA_AGENT_FEATURE_AGENT_DISPATCH is present in an agent.
2. (Application) Create a so-called "service" queue on the destination agent using **hsa_queue_create**.
3. (Application) Create a queue on the component using **hsa_queue_create**, passing a pointer to the service queue obtained in the previous step. Any service request to the destination agent must use this queue. It is not possible for a work-item to request services from more than one destination agent, or submit Agent Dispatch packets to more than one service queue.
4. (Kernel) The work-item locates the service queue by accessing the *service_queue* field of the current queue. For more information on how to retrieve the address of the current pointer, please check the documentation of the **queueptr** HSAIL instruction in [1].
5. (Kernel) The work-item submits an Agent Dispatch packet to the service queue following the steps described in Section 2.5.
6. (Destination Agent) The agent parses the packet, executes the indicated service, stores the result in the memory location pointed to by *return_address*, and decrements the completion signal if present.
7. (Kernel) The work-item consumes the function's output and proceeds to the next instruction.

2.6.2.1 Example - application processes allocation service requests from component

In this example, work-items in a kernel might ask the host application to allocate memory on their behalf. This is useful because there is no HSAIL instruction to allocate virtual memory. In this scenario, the destination agent is the application running on the CPU, and the service is memory allocation.

The application starts by finding a CPU agent where it can create a queue that supports Agent Dispatch packets:

```
hsa_status_t get_agent_dispatch_agent(hsa_agent_t agent, void* data) {
    uint32_t features = 0;
    hsa_device_type_t device;
    hsa_agent_get_info(agent, HSA_AGENT_INFO_FEATURE, &features);
    if (features & HSA_AGENT_FEATURE_AGENT_DISPATCH) {
        hsa_agent_get_info(agent, HSA_AGENT_INFO_DEVICE, &device);
        if (device == HSA_DEVICE_TYPE_CPU) {
            hsa_agent_t* ret = (hsa_agent_t*) data;
            *ret = agent;
            return HSA_STATUS_INFO_BREAK;
        }
    }
    return HSA_STATUS_SUCCESS;
}
```

A service queue is created on the CPU agent:

```
hsa_agent_t agent_dispatch_agent;
hsa_iterate_agents(get_agent_dispatch_agent, &agent_dispatch_agent);
hsa_queue_t *service_queue;
hsa_queue_create(agent_dispatch_agent, 16, HSA_QUEUE_TYPE_SINGLE, NULL, NULL, &service_queue);
```

, and used in the creation of a regular queue on the component (the callback definition is omitted for simplicity):

```
hsa_agent_t component;
hsa_iterate_agents(get_component, &component);
hsa_queue_t* queue;
hsa_queue_create(component, 16, HSA_QUEUE_TYPE_MULTI, callback, service_queue, &queue);
```

The application now launches a kernel in the component queue. Every time a work-item needs more virtual memory, it will submit a Agent Dispatch packet to the service queue. The allocation size is stored in the first element of *arg*, and *return_address* contains the memory address where the application will store the starting address of the allocation. Finally, the *type* is set to an application-defined code. Let's assume that the allocation service type is 0x8000.

The following source code shows how a thread running on the host might process the Agent Dispatch packets submitted from the component. The application waits for the value of the doorbell signal in the service queue to be monotonically increased. When that happens, it simply processes the next service request by invoking malloc.

```
void process_agent_dispatch(hsa_queue_t* service_queue) {
    hsa_agent_dispatch_packet_t* packets = (hsa_agent_dispatch_packet_t*) service_queue->base_address;
    uint64_t read_index = hsa_queue_load_read_index_acquire(service_queue);
    assert(read_index == 0);
    hsa_signal_t doorbell = service_queue->doorbell_signal;

    while (true) {

        while (hsa_signal_wait_acquire(doorbell, HSA_GTE, read_index, UINT64_MAX, HSA_WAIT_EXPECTANCY_LONG) <
            (hsa_signal_value_t) read_index);

        hsa_agent_dispatch_packet_t* packet = packets + read_index % service_queue->size;
        // Agent Dispatch packet type must be an application defined function
        assert(packet->header.type == HSA_PACKET_TYPE_AGENT_DISPATCH);
        assert(packet->type >= 0x8000);

        if (packet->type == 0x8000) {
            // Component requests memory
            void** ret = (void**) packet->return_address;
            size_t size = (size_t) packet->arg[0];
            *ret = malloc(size);
```

```
        } else {
            // Process other Agent Dispatch packet types...
        }
        if (packet->completion_signal != 0) {
            hsa_signal_subtract_release(packet->completion_signal, 1);
        }
        packet->header.type = HSA_PACKET_TYPE_INVALID;
        read_index++;
        hsa_queue_store_read_index_release(service_queue, read_index);
    }
}
```

In practice, processing of Agent Dispatch packets is usually more complex because the consumer has to take into account multiple-producer scenarios.

2.6.3 Barrier packet

The Barrier packet (of type hsa_barrier_packet_t) allows an application to specify up to five signal dependencies and requires the packet processor to resolve those dependencies before proceeding. The packet processor will not launch any further packets in that queue until the Barrier packet is complete, which happens when either of these two conditions have been met:

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

2.6.3.1 Example - handling dependencies across kernels running in different components

A combination of completion signals and Barrier packets allows expressing complex dependencies between packets, queues, and agents that are automatically handled by the packet processors. For example, if kernel b executing in component B consumes the result of kernel a executing in a different component A, then b *depends* on a. In HSA, this dependency can be enforced across components by creating a signal that will be simultaneously used as I) the completion signal of a Dispatch packet $packet\_a$ corresponding to a II) the dependency signal in a Barrier packet that precedes the Dispatch packet $packet\_b$ corresponding to b. The packet processor enforces the task dependency by not launching $packet\_b$ until $packet\_a$ has completed. The following example illustrates how to programmatically express the described dependency using the HSA API.

```
int main(){
    hsa_init();

    // Find available components. Let's assume there are two, A and B
    hsa_agent_t* component = get_two_components();

    // Create queue in component A and prepare a Dispatch packet
    hsa_queue_t *queue_a;
    hsa_queue_create(component[0], 4, HSA_QUEUE_TYPE_SINGLE, NULL, NULL, &queue_a);
    uint64_t packet_id_a = hsa_queue_add_write_index_relaxed(queue_a, 1);

    hsa_dispatch_packet_t* packet_a = initialize_packet(queue_a->base_address, packet_id_a);
    // KERNEL_A is the memory location of the 1st kernel object
    packet_a->kernel_object_address = (uint64_t) KERNEL_A;

    // Create a signal with a value of 1 and attach it to the first Dispatch packet
    hsa_signal_t signal;
```

```
    hsa_signal_create(1, 0, NULL, &signal);
    packet_a->completion_signal = signal;

    // Tell packet processor of A to launch the first Dispatch packet
    packet_type_store_release(&packet_a->header, HSA_PACKET_TYPE_DISPATCH);
    hsa_signal_store_release(queue_a->doorbell_signal, packet_id_a);

    // Create queue in component B
    hsa_queue_t *queue_b;
    hsa_queue_create(component[1], 4, HSA_QUEUE_TYPE_SINGLE, NULL, NULL, &queue_b);
    uint64_t packet_id_b = hsa_queue_add_write_index_relaxed(queue_b, 2);

    // Create Barrier packet that is enqueued in a queue of B
    const size_t packet_size = sizeof(hsa_dispatch_packet_t);
    uint64_t curr_address = queue_b->base_address + packet_id_b * packet_size;
    hsa_barrier_packet_t* barrier_packet = (hsa_barrier_packet_t*)curr_address;
    memset(barrier_packet, 0, packet_size);
    barrier_packet->header.release_fence_scope = HSA_FENCE_SCOPE_COMPONENT;

    // Add dependency on the first Dispatch Packet
    barrier_packet->dep_signal[0] = signal;
    packet_type_store_release(&barrier_packet->header, HSA_PACKET_TYPE_BARRIER);

    // Create and enqueue a second Dispatch packet after the Barrier in B. The second dispatch is launched after the first
    // has completed
    hsa_dispatch_packet_t* packet_b = initialize_packet(queue_b->base_address, packet_id_b + 1);
    // KERNEL_B is the memory location of the 2nd kernel object
    packet_b->kernel_object_address = (uint64_t) KERNEL_B;
    packet_type_store_release(&packet_b->header, HSA_PACKET_TYPE_DISPATCH);

    hsa_signal_store_release(queue_b->doorbell_signal, packet_id_b + 1);

    while (hsa_signal_wait_acquire(packet_b->completion_signal, HSA_EQ, 0, UINT64_MAX,
           HSA_WAIT_EXPECTANCY_UNKNOWN) != 0);

    hsa_signal_destroy(signal);
    hsa_queue_destroy(queue_b);
    hsa_queue_destroy(queue_a);
    hsa_shut_down();
}
```

2.6.4 Packet states

After submission, a packet can be in one of the following five states: *in queue*, *launch*, *error*, *active* or *complete*. Figure 2.1 shows the state transition diagram.

In queue The packet processor has not started to parse the current packet. The transition to the launch state only occurs after all the preceding packets have completed their execution if the *barrier* bit is set in the header. If the bit is not set, the transition occurs after the preceding packets have finished their launch phase. In other words, while the packet processor is required to launch any consecutive two packets in order, it is not required to complete them in order unless the barrier bit of the second packet is set.

Launch The packet is being parsed, but it did not start executing. This phase finalizes by applying an acquire memory fence with the scope indicated by the *acquire_fence_scope* header field. Memory fences are discussed in [1], Section 6.2.6.

If an error is detected during launch, the queue transitions to the error state and the event callback associated with the queue (if present) is invoked. The runtime passes a status code to the callback that indicates the source of the problem. The following status codes might be returned:

HSA_STATUS_ERROR_INVALID_PACKET_FORMAT Malformed AQL packet. This could happen if, for example, the packet header type is invalid.

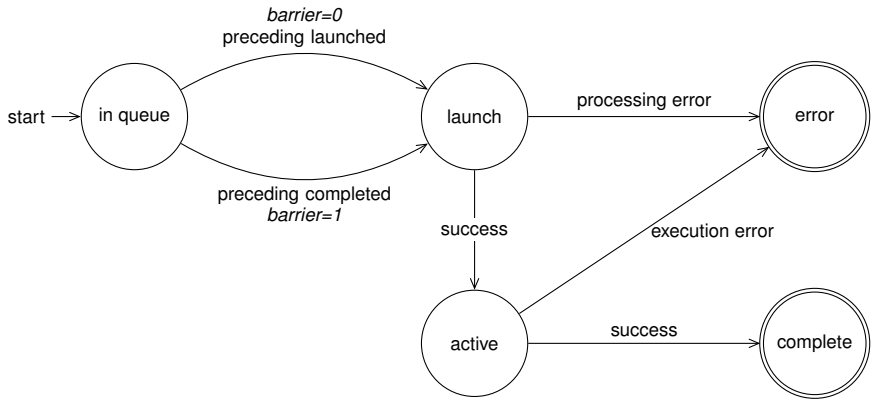

Figure 2.1: Packet State Diagram

HSA_STATUS_ERROR_OUT_OF_RESOURCES The packet processor is unable to allocate the resources required by the launch. This could happen if, for example, a Dispatch packet requests more group memory than the size of the group memory declared by the corresponding component.

Active The execution of the packet has started.

If an error is detected during this phase, the queue transitions to the error state, a release fence is applied to the packet with the scope indicated by the *release_fence_scope* header field, and the completion signal (if present) is assigned a negative value. There is no invocation of the callback associated with the queue.

If no error is detected, the transition to the complete state happens when the associated task finishes (in the case of Dispatch and Agent Dispatch packets), or when the dependencies are satisfied (in the case of a Barrier packet).

Complete A memory release fence is applied with the scope indicated by the *release_fence_scope* header field, and the completion signal (if present) decremented.

Error An error was encountered during the launch or active phases. No further packets will be launched on the queue. The queue cannot be recovered, but only inactivated or destroyed. If the application passes the queue as an argument to any HSA function other than **hsa_queue_inactivate** or **hsa_queue_destroy**, the behavior is undefined.

2.6.5 API

2.6.5.1 hsa_packet_type_t

```
typedef enum {
    HSA_PACKET_TYPE_ALWAYS_RESERVED = 0,
    HSA_PACKET_TYPE_INVALID = 1,
    HSA_PACKET_TYPE_DISPATCH = 2,
    HSA_PACKET_TYPE_BARRIER = 3,
    HSA_PACKET_TYPE_AGENT_DISPATCH = 4
} hsa_packet_type_t;
```

Packet type.

Values

HSA_PACKET_TYPE_ALWAYS_RESERVED
: Initial type of any packet when the queue is created. A packet processor must not process a packet of this type. All queues support this packet type.

HSA_PACKET_TYPE_INVALID
: The packet has been processed in the past, but has not been reassigned to the packet processor. A packet processor must not process a packet of this type. All queues support this packet type.

HSA_PACKET_TYPE_DISPATCH
: 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_PACKET_TYPE_BARRIER
: Packet used by HSA agents to delay processing of subsequent packets, and to express complex dependencies between multiple packets. All queues support this packet type.

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

2.6.5.2 hsa_fence_scope_t

```
typedef enum {
    HSA_FENCE_SCOPE_NONE = 0,
    HSA_FENCE_SCOPE_COMPONENT = 1,
    HSA_FENCE_SCOPE_SYSTEM = 2
} hsa_fence_scope_t;
```

Scope of the memory fence operation associated with a packet.

Values

HSA_FENCE_SCOPE_NONE
: No scope. Must only be used as the acquire fence scope of a Barrier packet.

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

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

2.6.5.3 hsa_packet_header_t

```
typedef struct hsa_packet_header_s {
    hsa_packet_type_t type : 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_packet_header_t
```

AQL packet header.

Data Fields

type
: 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. Must be HSA_FENCE_SCOPE_NONE for Barrier packets.

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 0.

2.6.5.4 hsa_dispatch_packet_t

```
typedef struct hsa_dispatch_packet_s {
    hsa_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;
    uint32_t group_segment_size;
    uint64_t kernel_object_address;
    uint64_t kernarg_address;
    uint64_t reserved3;
    hsa_signal_t completion_signal;
} hsa_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 0.

workgroup_size_x
X dimension of work-group, in work-items. Must be greater than 0.

workgroup_size_y
Y dimension of work-group, in work-items. Must be greater than 0. If *dimensions* is 1, the only valid value is 1.

workgroup_size_z
Z dimension of work-group, in work-items. Must be greater than 0. If *dimensions* is 1 or 2, the only valid value is 1.

reserved2
Reserved. Must be 0.

grid_size_x
X dimension of grid, in work-items. Must be greater than 0. Must not be smaller than *workgroup_size_x*.

grid_size_y
Y dimension of grid, in work-items. Must be greater than 0. If *dimensions* is 1, the only valid value is 1. Must not be smaller than *workgroup_size_y*.

grid_size_z
Z dimension of grid, in work-items. Must be greater than 0. If *dimensions* is 1 or 2, the only valid value is 1. Must not be smaller than *workgroup_size_z*.

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

group_segment_size
Size in bytes of group memory allocation request (per work-group). Must not be less than the sum of the group memory used by the kernel (and the functions it calls directly or indirectly) and the dynamically allocated group segment variables.

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

kernarg_address
Pointer to a buffer containing the kernel arguments. Might be 0.
The buffer must be allocated using **hsa_memory_allocate**, and must not be modified once the Dispatch packet is enqueued until the dispatch has completed execution.

reserved3
Reserved. Must be 0.

completion_signal
Signal used to indicate completion of the job. Might be 0 (no signal).

2.6.5.5 hsa_agent_dispatch_packet_t

```
typedef struct hsa_agent_dispatch_packet_s {
    hsa_packet_header_t header;
    uint16_t type;
    uint32_t reserved2;
    uint64_t return_address;
```

```
    uint64_t arg[4];
    uint64_t reserved3;
    hsa_signal_t completion_signal;
} hsa_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 (application defined).

reserved2
: Reserved. Must be 0.

return_address
: Address where to store the function return values, if any.

arg
: Function arguments.

reserved3
: Reserved. Must be 0.

completion_signal
: Signal used to indicate completion of the job. Might be 0 (no signal).

2.6.5.6 hsa_barrier_packet_t

```
typedef struct hsa_barrier_packet_s {
    hsa_packet_header_t header;
    uint16_t reserved2;
    uint32_t reserved3;
    hsa_signal_t dep_signal[5];
    uint64_t reserved4;
    hsa_signal_t completion_signal;
} hsa_barrier_packet_t
```

Barrier packet.

Data Fields

header
: Packet header.

reserved2
: Reserved. Must be 0.

reserved3
: Reserved. Must be 0.

dep_signal
: Array of dependent signal objects. Entries with value 0 are valid and are interpreted by the packet processor as satisfied dependencies.

reserved4

Reserved. Must be 0.

completion_signal
Signal used to indicate completion of the job. Might be 0 (no signal).

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 any component in a coherent fashion. An application can directly pass a buffer allocated on the host (for example, using malloc) to a kernel dispatched in any component. Kernels might also access memory that is not in the coherent, global segment.

The HSA runtime API provides a compact set of functions for inspecting the memory *regions* that are accessible from an agent, and (if applicable) allocating memory on those regions from the host.

A memory region represents a block of contiguous memory that is directly accessible by an agent, and exposes properties about the block of memory and how it is accessed from that particular agent. The same block of memory (virtual or physical) might be accessible from multiple agents, but the runtime creates a unique region on every agent to represent the memory block. The characteristics of a memory region include its size, addressability, access speed, and corresponding memory segment. One agent might be able to access multiple regions within the same segment.

The function **hsa_agent_iterate_regions** can be used to inspect the set of regions associated with an agent. Implementations of **hsa_agent_iterate_regions** are required to report the following regions on every agent:

- A region that starts at address 0, and is located in the global segment. This corresponds to the coherent, primary HSA memory type. All the regions with these characteristics should have identical sizes.
- A region that is located in the global segment and can be used to allocate backing storage for the kernarg segment: HSA_REGION_FLAG_KERNARG is true, and HSA_REGION_INFO_ALLOC_MAX_SIZE is not 0. All the regions with these characteristics should have identical sizes.

If the application can allocate memory in a region using **hsa_memory_allocate**, the value of the attribute HSA_REGION_INFO_ALLOC_MAX_SIZE is different from zero. The runtime allocator can only be used to allocate memory in the global segment. Memory in the private, group and kernarg segments is automatically allocated when a Dispatch packet is launched.

When the application no longer needs a buffer allocated using **hsa_memory_allocate**, it invokes **hsa_memory_free** to release the memory. The application shall not release a runtime-allocated buffer using standard libraries (for example, free). Conversely, the runtime deallocator cannot be used to release memory allocated using standard libraries (for example, malloc).

If a buffer created in the host is also accessed by a component, applications are encouraged to *register* the corresponding address range beforehand using the **hsa_memory_register** function. While kernels running on HSA components can access any regular host pointer, a registered buffer might result on improved access performance. When the application no longer needs to access a registered buffer, it should deregister that virtual address range by invoking **hsa_memory_deregister**.

2.7.1 Example - passing arguments to a kernel

In the kernel setup example listed in Section 2.6.1, the kernel receives no arguments:

```
// Indicate which ISA to run. The application is expected to have finalized a kernel (for example, using the finalization API).
// We will assume that the kernel object location is stored in KERNEL_ADDRESS
dispatch_packet->kernel_object_address = (uint64_t) KERNEL_ADDRESS;

// Assume our kernel receives no arguments.
dispatch_packet->kernarg_address = 0;
```

Let's assume now that the kernel expects a single argument, a signal handle. The application needs to populate the *kernarg_address* field of the Dispatch packet with the address of a buffer containing the signal.

The application searches for a memory region that can be used to allocate backing storage for the kernarg segment. Once found, it reserves enough space to hold the signal argument. While the actual amount of memory to be allocated is determined by the finalizer, for simplicity we will assume that it matches the natural size of the signal type, 8 bytes.

```
// Indicate which ISA to run. The application is expected to have finalized a kernel (for example, using the finalization API).
// We will assume that the kernel object location is stored in KERNEL_ADDRESS
dispatch_packet->kernel_object_address = (uint64_t) KERNEL_ADDRESS;

hsa_region_t region;
hsa_agent_iterate_regions(component, get_kernarg, &region);
hsa_memory_allocate(region, 8, (void**) &dispatch_packet->kernarg_address);
```

, where the definition of *get_kernarg* is:

```
hsa_status_t get_kernarg(hsa_region_t region, void* data) {
    hsa_region_flag_t flags;
    hsa_region_get_info(region, HSA_REGION_INFO_FLAGS, &flags);
    if (flags & HSA_REGION_FLAG_KERNARG) {
        hsa_region_t* ret = (hsa_region_t*) data;
        *ret = region;
        return HSA_STATUS_INFO_BREAK;
    }
    return HSA_STATUS_SUCCESS;
}
```

Finally, the application stores a signal handle in the buffer.

```
hsa_signal_t* buffer = (hsa_signal_t*) dispatch_packet->kernarg_address;
assert(buffer != NULL);
hsa_signal_t signal;
hsa_signal_create(1, 1, &component, &signal);
*buffer = signal;
```

The rest of the dispatch process remains the same.

2.7.2 API

2.7.2.1 hsa_region_t

```
typedef uint64_t hsa_region_t;
```

A memory region represents a block of contiguous memory that is directly accessible by an agent, and exposes properties about the block of memory and how it is accessed from that particular agent.

2.7.2.2 hsa_segment_t

```
typedef enum {
    HSA_SEGMENT_GLOBAL,
    HSA_SEGMENT_PRIVATE,
    HSA_SEGMENT_GROUP,
    HSA_SEGMENT_KERNARG,
    HSA_SEGMENT_READONLY,
    HSA_SEGMENT_SPILL,
    HSA_SEGMENT_ARG
```

} **hsa_segment_t**;

Types of memory segments.

Values

HSA_SEGMENT_GLOBAL
: Global segment. Used to hold data that is shared by all agents.

HSA_SEGMENT_PRIVATE
: Private segment. Used to hold data that is local to a single work-item.

HSA_SEGMENT_GROUP
: Group segment. Used to hold data that is shared by the work-items of a work-group.

HSA_SEGMENT_KERNARG
: Kernarg segment. Used to pass arguments into a kernel. Memory in this segment is visible to all work-items of the kernel dispatch with which it is associated.

HSA_SEGMENT_READONLY
: Read-only segment. Used to hold data that remains constant during the execution of a kernel dispatch.

HSA_SEGMENT_SPILL
: Spill segment. Used to load or store register spills.

HSA_SEGMENT_ARG
: Arg segment. Used to pass arguments into and out of functions.

2.7.2.3 hsa_region_flag_t

```
typedef enum {
    HSA_REGION_FLAG_KERNARG = 1,
    HSA_REGION_FLAG_CACHED_L1 = 2,
    HSA_REGION_FLAG_CACHED_L2 = 4,
    HSA_REGION_FLAG_CACHED_L3 = 8,
    HSA_REGION_FLAG_CACHED_L4 = 16
} hsa_region_flag_t;
```

Region flags.

Values

HSA_REGION_FLAG_KERNARG
: The application can use memory in the region to store kernel arguments, and provide the values for the kernarg segment of a kernel dispatch. If the region is not in the global segment, this flag must not be set.

HSA_REGION_FLAG_CACHED_L1
: Accesses to data in the region are cached in the L1 data cache of the region's agent. If the agent does not have an L1 cache (as reported by the attribute HSA_AGENT_INFO_CACHE_SIZE), this flag must not be set.

HSA_REGION_FLAG_CACHED_L2
: Accesses to data in the region are cached in the L2 data cache of the region's agent. If the agent does not have an L2 cache (as reported by the attribute HSA_AGENT_INFO_CACHE_SIZE), this flag must not be set.

HSA_REGION_FLAG_CACHED_L3

Accesses to data in the region are cached in the L3 data cache of the region's agent. If the agent does not have an L3 cache (as reported by the attribute HSA_AGENT_INFO_CACHE_SIZE), this flag must not be set.

HSA_REGION_FLAG_CACHED_L4
Accesses to data in the region are cached in the L4 data cache of the region's agent. If the agent does not have an L4 cache (as reported by the attribute HSA_AGENT_INFO_CACHE_SIZE), this flag must not be set.

2.7.2.4 hsa_region_info_t

```
typedef enum {
    HSA_REGION_INFO_BASE,
    HSA_REGION_INFO_SIZE,
    HSA_REGION_INFO_AGENT,
    HSA_REGION_INFO_FLAGS,
    HSA_REGION_INFO_SEGMENT,
    HSA_REGION_INFO_ALLOC_MAX_SIZE,
    HSA_REGION_INFO_ALLOC_GRANULE,
    HSA_REGION_INFO_ALLOC_ALIGNMENT,
    HSA_REGION_INFO_BANDWIDTH,
    HSA_REGION_INFO_NODE
} hsa_region_info_t;
```

Attributes of a memory region.

Values

HSA_REGION_INFO_BASE
Base (starting) address. The type of this attribute is void*.

HSA_REGION_INFO_SIZE
Size, in bytes. The type of this attribute is size_t.

HSA_REGION_INFO_AGENT
Agent associated with this region. The type of this attribute is hsa_agent_t.

HSA_REGION_INFO_FLAGS
Flag mask. The type of this attribute is uint32_t, a bit-field of hsa_region_flag_t values.

HSA_REGION_INFO_SEGMENT
Segment where memory in the region can be used. The type of this attribute is hsa_segment_t.

HSA_REGION_INFO_ALLOC_MAX_SIZE
Maximum allocation size in this region, in bytes. A value of 0 indicates that the host cannot allocate memory in the region using **hsa_memory_allocate**. If the value of HSA_REGION_INFO_SEGMENT is other than HSA_SEGMENT_GLOBAL, the maximum allocation size must be 0. The type of this attribute is size_t.

HSA_REGION_INFO_ALLOC_GRANULE
Allocation granularity of buffers allocated by **hsa_memory_allocate** in this region. The size of a buffer allocated in this region is a multiple of the value of this attribute. If HSA_REGION_INFO_ALLOC_-MAX_SIZE is 0, the allocation granularity must be 0. The type of this attribute is size_t.

HSA_REGION_INFO_ALLOC_ALIGNMENT
Alignment of buffers allocated by **hsa_memory_allocate** in this region. If HSA_REGION_INFO_-ALLOC_MAX_SIZE is 0, the alignment must be 0. Otherwise, it must be a power of 2. The type of this attribute is size_t.

HSA_REGION_INFO_BANDWIDTH
Peak bandwidth, in MB/s. The type of this attribute is uint32_t.

HSA_REGION_INFO_NODE
NUMA node associated with this region. The type of this attribute is uint32_t.

2.7.2.5 hsa_region_get_info

```
hsa_status_t hsa_region_get_info(
    hsa_region_t region,
    hsa_region_info_t attribute,
    void * value);
```

Get the current value of an attribute of a region.

Parameters

region
(in) A valid region.

attribute
(in) Attribute to query.

value
(out) Pointer to a application-allocated buffer where to store the value of the attribute. If the buffer passed by the application is not large enough to hold the value of *attribute*, the behavior is undefined.

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_REGION
If the region is invalid.

HSA_STATUS_ERROR_INVALID_ARGUMENT
If *attribute* is not a valid region attribute, or *value* is NULL.

2.7.2.6 hsa_agent_iterate_regions

```
hsa_status_t hsa_agent_iterate_regions(
    hsa_agent_t agent,
    hsa_status_t (*callback)(hsa_region_t region, void *data),
    void * data);
```

Iterate over the memory regions associated with a given agent, and invoke an application-defined callback on every iteration.

Parameters

agent
(in) A valid agent.

callback

(in) Callback to be invoked once per region that is directly accessible from the agent. The runtime passes two arguments to the callback, the region and the application data. If *callback* returns a status other than HSA_STATUS_SUCCESS for a particular iteration, the traversal stops and **hsa_agent_iterate_regions** returns that status value.

data
(in) Application data that is passed to *callback* on every iteration. Might be NULL.

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_AGENT
If the agent is invalid.

HSA_STATUS_ERROR_INVALID_ARGUMENT
If *callback* is NULL.

2.7.2.7 hsa_memory_allocate

```
hsa_status_t hsa_memory_allocate(
    hsa_region_t region,
    size_t size,
    void ** ptr);
```

Allocate a block of memory in a given region.

Parameters

region
(in) Region where to allocate memory from.

size
(in) Allocation size, in bytes. This value is rounded up to the nearest multiple of HSA_REGION_INFO_ALLOC_GRANULE in *region*. Allocations of size 0 are allowed and return a NULL pointer.

ptr
(out) Pointer to the location where to store the base address of the allocated block. The returned base address is aligned to the value of HSA_REGION_INFO_ALLOC_ALIGNMENT in *region*.

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 no memory is available.

HSA_STATUS_ERROR_INVALID_REGION
If the region is invalid.

HSA_STATUS_ERROR_INVALID_ALLOCATION
If the host is not allowed to allocate memory in *region*, or *size* is greater than the value of HSA_REGION_INFO_ALLOC_MAX_SIZE in *region*.

HSA_STATUS_ERROR_INVALID_ARGUMENT

If *ptr* is NULL.

2.7.2.8 hsa_memory_free

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

Deallocate a block of memory previously allocated using **hsa_memory_allocate**.

Parameters

ptr
(in) Pointer to a memory block. If *ptr* is 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.

2.7.2.9 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. A size of zero is only allowed if *address* is NULL.

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.

HSA_STATUS_ERROR_INVALID_ARGUMENT
If *size* is 0 but *address* is not NULL.

Description

Registering a buffer serves as an indication to the runtime that the passed buffer might be accessed from a component other than the host. Registration is a performance hint that allows the runtime implementation to know which buffers will be accessed by some of the components ahead of time.

Registrations should not overlap.

2.7.2.10 hsa_memory_deregister

```
hsa_status_t hsa_memory_deregister(
    void * address,
    size_t size);
```

Deregister memory previously registered using **hsa_memory_register**.

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.

size
: (in) Size of the region to be deregistered.

Return Values

HSA_STATUS_SUCCESS
: The function has been executed successfully.

HSA_STATUS_ERROR_NOT_INITIALIZED
: The runtime has not been initialized.

Description

If the memory interval being deregistered does not match a previous registration (start and end addresses), the behavior is undefined.

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

```
typedef enum {
    HSA_EXT_START = 0,
    HSA_EXT_FINALIZER = HSA_EXT_START,
    HSA_EXT_LINKER = 1,
    HSA_EXT_IMAGES = 2,
    HSA_SVEXT_START = 10000
} hsa_extension_t;
```

HSA extensions.

Values

HSA_EXT_START
: Start of the multi vendor extension range.

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

HSA_EXT_LINKER
: Linker extension.

HSA_EXT_IMAGES
: Images extension.

HSA_SVEXT_START
: 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 an 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.

```
// The structure which defines the version, functions, and data for the extension:
typedef struct hsa_ext_acme_foo_s {
    // Major version number of the extension.
    int major_version;
    // Minor version number of the extension.
    int minor_version;
    // 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 2.

2.9.1.2 hsa_powertwo_t

```
typedef enum {
    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
} hsa_powertwo_t;
```

Power of two between 1 and 256.

2.9.1.3 hsa_dim3_t

```
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

```
typedef enum {
    HSA_DIM_X = 0,
    HSA_DIM_Y = 1,
```

```
    HSA_DIM_Z = 2
} hsa_dim_t;
```

Dimensions in a 3D space.

Values

HSA_DIM_X
: X dimension.

HSA_DIM_Y
: Y dimension.

HSA_DIM_Z
: Z dimension.

2.9.1.5 hsa_runtime_caller_t

```
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

The finalization API is used to finalize given kernels and/or indirect functions. The finalization API is a vendor-specific low-level API. It can be used to generate code for kernels and indirect functions from a specific program for a specific HSA component using **hsa_ext_finalize**. A kernel can only be finalized once per program per agent. An indirect function can only be finalized once per program per agent per call convention. Only code for the HSA components specified when the program was created can be requested.

The program must contain a definition for the requested kernels and indirect functions among the modules that have been added to the program. The modules of the program must collectively define all variables, fbarriers, kernels and functions referenced by operations in the code block of:

- The kernel and indirect functions being finalized.
- The transitive closure of all functions specified by **call** or **scall** operations starting with the kernel and indirect functions being finalized. Refer to the HSA Programmer's Reference Manual [1], Chapter 10 for Function Operations.

When invoking the finalizer, one or more kernels and indirect functions can be requested. Requested kernels and indirect functions are represented as an array of hsa_ext_finalization_request_t. On some implementations, specifying multiple kernels and indirect functions can produce code with better performance than finalizing the kernels and indirect function individually. For example, it can allow the finalizer to generate common code for shared functions which can reduce code footprint and improve instruction cache performance.

The finalization API allocates a kernel descriptor for every kernel definition in a program, and an indirect function descriptor for every indirect function definition in the program. Both kinds of descriptors are represented as an array of hsa_ext_code_descriptor_t in global segment memory. Each code descriptor provides the information about a finalization of the kernel or indirect function for a specific HSA component, and for indirect functions, a specific call convention of that HSA component. An HSA runtime uses an array of finalization handles hsa_ext_finalization_handle_t to access code descriptors.

The kernel descriptor array is indexed by the agent ID hsa_ext_program_agent_id_t The kernel descriptor and agent ID are available by the **ldk** and **agentid** operations respectively, or by the HSA runtime queries **hsa_ext_query_kernel_descriptor_address** and **hsa_ext_query_program_agent_id**. Refer to the HSA Programmer's Reference Manual [1], Chapter 4.2.1 for Agent ID and 11.3 for User Mode Queue Operations.

The indirect function descriptor is indexed by the call convention ID hsa_ext_program_call_convention_-id32_t which is performed implicitly by the **icall** operation. The indirect function descriptor address is

available by the **ldi** operation or the HSA runtime query **hsa_ext_query_indirect_function_descriptor_address**. Refer to the HSA Programmer's Reference Manual [1], Chapter 4.2.2 for Call Convention ID and 10.8 for Indirect Call (**icall**, **ldi**) Operations.

The finalizer updates the code descriptor corresponding to the agent or call convention for each kernel or indirect function that it finalizes.

The layout of a hsa_ext_code_descriptor_t is defined by the HSA runtime. It includes a kind field hsa_ext_code_kind32_t that indicates whether the code descriptor contains finalized code. If it has been finalized, then for kernels the information needed to create a User Mode Queue kernel dispatch packet is available, including:

- The byte size of the group segment for a single work-group:

  Includes module scope and function scope group segment variables used by the kernel or any functions it calls directly or indirectly.

  Includes any finalizer allocated temporary space. For example, in the implementation of exception operations or fbarriers.

  Does not include any dynamically allocated group segment space. Refer to the HSA Programmer's Reference Manual [1], Chapter 4.20 Dynamic Group Memory Allocation.

- The byte size of the private segment for a single work-item. This includes:

  Module scope and function scope private segment variables.

  Space for function scope spill segment variables allocated in memory.

  Space for argument scope arg segment variables allocated in memory.

  Any space needed for saved HSAIL or ISA registers due to calls.

  Any other finalizer introduced temporaries including spilled ISA registers and space for function call stack.

- The 64-bit opaque code handle to the finalized code that includes the executable ISA for the HSA component. It can be used for the kernel dispatch packet kernel object address field.

The code descriptor also includes other information that may be useful to a high-level language runtime to invoke and manage the kernel's execution. For example, the size and alignment of the kernarg segment and the call convention used by the code of the kernel.

For an indirect function, a code descriptor includes:

- The 64-bit opaque code handle to the finalized code that includes the executable ISA for a single call convention of the HSA component.

Once code has been generated for a kernel for a specific HSA component, it can be executed by adding a kernel dispatch packet to a User Mode Queue associated with the HSA component. The information required to create the kernel dispatch packet is available in the code descriptor addressed by using the agent ID to index the kernel descriptor.

For an indirect function, the code is only made available to kernel dispatches launched after the indirect function has been finalized. Therefore, prior to executing a kernel, all indirect functions that it will call must have been finalized for the HSA component with the call convention used by the kernel code. The **icall** operation, used to call indirect functions, implicitly access the code descriptor addressed by using the call convention ID of the executing kernel to index the indirect function descriptor. Refer to the HSA Programmer's Reference Manual [1], Chapter 4.2.2 for Call Convention ID and 10.8 for Indirect Call (**icall**, **ldi**) Operations.

Appropriate memory synchronization is needed to access the code descriptor since it is updated concurrently by the HSA runtime. Memory synchronization may include using an acquire fence at system scope on

the kernel dispatch packet if the code descriptor may have changed since the HSA component executed the last packet acquire fence at system scope, or using a load acquire at system scope on the code descriptor kind field if the code descriptor may change during the execution of the kernel dispatch. This applies both to accesses performed explicitly to create kernel dispatch packets, and implicitly by the **icall** operation.

The code will remain available to execute until the HSA runtime is used to destroy the HSAIL program with which the code is associated using **hsa_ext_program_destroy**. All HSAIL programs created by the application are implicitly destroyed when the application terminates.

3.1.1 API

3.1.1.1 hsa_ext_brig_profile8_t

```
typedef uint8_t hsa_ext_brig_profile8_t;
```

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

3.1.1.2 hsa_ext_brig_profile_t

```
typedef enum {
    HSA_EXT_BRIG_PROFILE_BASE = 0,
    HSA_EXT_BRIG_PROFILE_FULL = 1
} hsa_ext_brig_profile_t;
```

Profile kinds. For more information see the HSA Programmer's Reference Manual.

Values

HSA_EXT_BRIG_PROFILE_BASE
: Base profile.

HSA_EXT_BRIG_PROFILE_FULL
: 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 the HSA Programmer's Reference Manual.

3.1.1.4 hsa_ext_brig_machine_model_t

```
typedef enum {
    HSA_EXT_BRIG_MACHINE_SMALL = 0,
    HSA_EXT_BRIG_MACHINE_LARGE = 1
} hsa_ext_brig_machine_model_t;
```

Machine model kinds. For more information see the HSA Programmer's Reference Manual.

Values

HSA_EXT_BRIG_MACHINE_SMALL
: Use 32 bit addresses for global segment and flat addresses.

HSA_EXT_BRIG_MACHINE_LARGE
: 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

```
typedef enum {
    HSA_EXT_BRIG_SECTION_DATA = 0,
    HSA_EXT_BRIG_SECTION_CODE = 1,
    HSA_EXT_BRIG_SECTION_OPERAND = 2
} hsa_ext_brig_section_id_t;
```

Predefined BRIG section kinds.

Values

HSA_EXT_BRIG_SECTION_DATA
: Textual character strings and byte data used in the module. Also contains variable length arrays of offsets into other sections that are used by entries in the hsa_code and hsa_operand sections. For more information see the HSA Programmer's Reference Manual.

HSA_EXT_BRIG_SECTION_CODE
: All of the directives and instructions of the module. Most entries contain offsets to the hsa_operand or hsa_data sections. Directives provide information to the finalizer, and instructions correspond to HSAIL operations which the finalizer uses to generate executable ISA code. For more information see the HSA Programmer's Reference Manual.

HSA_EXT_BRIG_SECTION_OPERAND
: The operands of directives and instructions in the code section. For example, immediate constants, registers and address expressions. For more information see the HSA Programmer's Reference Manual.

3.1.1.7 hsa_ext_brig_section_header_t

```
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. Every section starts with a hsa_ext_brig_section_header_t which contains the section size, name and offset to the first entry. For more information see the HSA Programmer's Reference Manual.

Data Fields

byte_count
Size in bytes of the section, including the size of the hsa_ext_brig_section_header_t. Must be a multiple of 4.

header_byte_count
Size of the header in bytes, which is also equal to the offset of the first entry in the section. Must be a multiple of 4.

name_length
Length of the section name in bytes.

name
Section name, *name_length* bytes long.

3.1.1.8 hsa_ext_brig_module_t

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

A module is the basic building block for HSAIL programs. When HSAIL is generated it is represented as a module.

Data Fields

section_count
Number of sections in the module. Must be at least 3.

section
A variable-sized array containing pointers to the BRIG sections. Must have *section_count* elements. Indexed by hsa_ext_brig_section_id32_t. The first three elements must be for the following predefined sections in the following order: HSA_EXT_BRIG_SECTION_DATA, HSA_EXT_BRIG_SECTION_CODE, HSA_EXT_BRIG_SECTION_OPERAND.

3.1.1.9 hsa_ext_brig_module_handle_t

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

An opaque handle to the hsa_ext_brig_module_t.

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;
```

An entry offset into the code section of the BRIG module. The value is the byte offset relative to the start of the section to the beginning of the referenced entry. The value 0 is reserved to indicate that the offset does

not reference any entry.

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

```
typedef enum {
    HSA_EXT_EXCEPTION_INVALID_OPERATION = 1,
    HSA_EXT_EXCEPTION_DIVIDE_BY_ZERO = 2,
    HSA_EXT_EXCEPTION_OVERFLOW = 4,
    HSA_EXT_EXCEPTION_UNDERFLOW = 8,
    HSA_EXT_EXCEPTION_INEXACT = 16
} hsa_ext_exception_kind_t;
```

HSAIL exception kinds. For more information see the HSA Programmer's Reference Manual.

Values

HSA_EXT_EXCEPTION_INVALID_OPERATION
: 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, ∞) or mul(∞, 0.0).
  - Fused multiply add: fma(0.0, ∞, c) or fma(∞, 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($+\infty$, $-\infty$), sub($+\infty$, $+\infty$).
  - Division: div(0.0, 0.0) or div(∞, ∞).
  - 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, ∞, 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
: 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
: The floating-point exponent of a value is too large to be represented.

HSA_EXT_EXCEPTION_UNDERFLOW
: 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
: 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 the 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 or 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

```
typedef enum {
    HSA_EXT_CONTROL_DIRECTIVE_ENABLE_BREAK_EXCEPTIONS = 0,
    HSA_EXT_CONTROL_DIRECTIVE_ENABLE_DETECT_EXCEPTIONS = 1,
    HSA_EXT_CONTROL_DIRECTIVE_MAX_DYNAMIC_GROUP_SIZE = 2,
    HSA_EXT_CONTROL_DIRECTIVE_MAX_FLAT_GRID_SIZE = 4,
    HSA_EXT_CONTROL_DIRECTIVE_MAX_FLAT_WORKGROUP_SIZE = 8,
    HSA_EXT_CONTROL_DIRECTIVE_REQUESTED_WORKGROUPS_PER_CU = 16,
    HSA_EXT_CONTROL_DIRECTIVE_REQUIRED_GRID_SIZE = 32,
    HSA_EXT_CONTROL_DIRECTIVE_REQUIRED_WORKGROUP_SIZE = 64,
    HSA_EXT_CONTROL_DIRECTIVE_REQUIRED_DIM = 128,
    HSA_EXT_CONTROL_DIRECTIVE_REQUIRE_NO_PARTIAL_WORKGROUPS = 256
} hsa_ext_control_directive_present_t;
```

HSAIL control directive kinds. For more information see the HSA Programmer's Reference Manual.

Values

HSA_EXT_CONTROL_DIRECTIVE_ENABLE_BREAK_EXCEPTIONS
: 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
: 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 endeavor 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

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
: 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 the HSA Programmer's Reference Manual.

HSA_EXT_CONTROL_DIRECTIVE_MAX_FLAT_WORKGROUP_SIZE
: 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 the HSA Programmer's Reference Manual.

HSA_EXT_CONTROL_DIRECTIVE_REQUESTED_WORKGROUPS_PER_CU
: 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
: 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 the HSA Programmer's Reference Manual.

HSA_EXT_CONTROL_DIRECTIVE_REQUIRED_WORKGROUP_SIZE
: 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 than the WAVESIZE then barrier operations can be optimized to just a memory fence.

HSA_EXT_CONTROL_DIRECTIVE_REQUIRED_DIM
: 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

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 control directive is present.

3.1.1.15 hsa_ext_control_directives_t

```
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 **hsa_ext_finalize** to specify values for the control directives, and is used in hsa_ext_code_descriptor_t to record the values of the control directives that the finalizer 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 the 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 hsa_ext_control_directive_present_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 endeavor 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 the 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 the 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 the 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

```
typedef enum {
    HSA_EXT_CODE_NONE = 0,
    HSA_EXT_CODE_KERNEL = 1,
    HSA_EXT_CODE_INDIRECT_FUNCTION = 2,
    HSA_EXT_CODE_RUNTIME_FIRST = 0x40000000,
    HSA_EXT_CODE_RUNTIME_LAST = 0x7fffffff,
    HSA_EXT_CODE_VENDOR_FIRST = 0x80000000,
    HSA_EXT_CODE_VENDOR_LAST = 0xffffffff
} hsa_ext_code_kind_t;
```

Kinds of code object. For more information see the HSA Programmer's Reference Manual.

Values

HSA_EXT_CODE_NONE
Not a code object.

HSA_EXT_CODE_KERNEL
HSAIL kernel that can be used with an AQL Dispatch packet.

HSA_EXT_CODE_INDIRECT_FUNCTION
HSAIL indirect function.

HSA_EXT_CODE_RUNTIME_FIRST
HSA runtime code objects. For example, partially linked code objects.

HSA_EXT_CODE_RUNTIME_LAST

HSA_EXT_CODE_VENDOR_FIRST
: Vendor specific code objects.

HSA_EXT_CODE_VENDOR_LAST

3.1.1.18 hsa_ext_program_call_convention_id32_t

```
typedef uint32_t hsa_ext_program_call_convention_id32_t;
```

Each HSA component can support one or more call conventions. For example, an HSA component may have different call conventions that each use a different number of ISA registers to allow different numbers of wavefronts to execute on a compute unit.

3.1.1.19 hsa_ext_program_call_convention_id_t

```
typedef enum {
    HSA_EXT_PROGRAM_CALL_CONVENTION_FINALIZER_DETERMINED = -1
} hsa_ext_program_call_convention_id_t;
```

Kinds of program call convention IDs.

Values

HSA_EXT_PROGRAM_CALL_CONVENTION_FINALIZER_DETERMINED
: Finalizer determined call convention ID.

3.1.1.20 hsa_ext_code_handle_t

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

The 64-bit opaque code handle to the finalized code that includes the executable ISA for the HSA component. It can be used for the kernel Dispatch packet kernel object address field.

Data Fields

handle
: HSA component specific handle to the code.

3.1.1.21 hsa_ext_debug_information_handle_t

```
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_t

```
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
```

Provides the information about a finalization of the kernel or indirect function for a specific HSA component, and for indirect functions, a specific call convention of that HSA component.

Data Fields

code_type
: Type of code object this code descriptor associated with.

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
: The 64-bit opaque code handle to the finalized code that includes the executable ISA for the HSA component. It can be used for the kernel Dispatch packet kernel object address field.

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 allocate 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 it 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 it 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 it 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 it must be 0.

reserved2
Reserved. Must be 0.

control_directive

The values should be the actual 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_t

```
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. Holds information about the module needed to be finalized. If the module contains an indirect function, also holds information about call convention ID.

Data Fields

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

symbol
Entry offset into the code section.

program_call_convention
If this finalization request is for indirect function, desired program call convention.

3.1.1.24 hsa_ext_finalization_handle_t

```
typedef struct hsa_ext_finalization_handle_s {
    uint64_t handle;
} hsa_ext_finalization_handle_t
```

Finalization handle is the handle to the object produced by the finalizer that contains the ISA code and related information needed to execute that code for a specific agent for the set of kernels/indirect functions specified in the finalization request.

Data Fields

handle
HSA component specific handle to the finalization information.

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);
```

Callback 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);
```

Callback 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);
```

Callback 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,
    hsa_ext_program_agent_id_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_handle_t * finalization);
```

Finalizes provided list of kernel and/or indirect functions.

Parameters

caller

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

agent

(in) HSA agents 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/or indirect needed to be finalized.

finalization_request_list
(in) List of kernels and/or indirect functions needed to be finalized.

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) Callback function to get the definition of a module scope variable/fbarrier or kernel/function.

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

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

optimization_level
(in) An implementation defined value that controls 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*. 0 - exclude debug information, 1 - include debug information.

finalization
(out) Handle to the object produced that contains the ISA code and related information needed to execute that code for the specific *agent* for the set of kernels/indirect functions specified in the *finalization_request_list*.

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*.

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. A kernel can only be finalized once per program per agent. An indirect function can only be finalized once per program per agent per call

convention. Only code for the HSA components specified when the program was created can be requested. The program must contain a definition for the requested kernels and indirect functions among the modules that have been added to the program.

3.1.1.29 hsa_ext_query_finalization_code_descriptor_count

```
hsa_status_t hsa_ext_query_finalization_code_descriptor_count(
    hsa_agent_t agent,
    hsa_ext_finalization_handle_t finalization,
    uint32_t * code_descriptor_count);
```

Queries the total number of kernel and indirect functions that have been finalized as part of the finalization object.

Parameters

agent
(in) Agent for which the finalization object contains code.

finalization
(in) Finalization handle that references the finalization object for *agent*.

code_descriptor_count
(out) Number of kernel and indirect functions that have been finalized as part of the finalization object.

Return Values

HSA_STATUS_SUCCESS
The function has been executed successfully, and number of code descriptors is queried.

HSA_STATUS_ERROR_INVALID_ARGUMENT
If provided *agent* is NULL or not valid. If *finalization* points to invalid finalization. If *code_descriptor_count* is NULL.

3.1.1.30 hsa_ext_query_finalization_code_descriptor

```
hsa_status_t hsa_ext_query_finalization_code_descriptor(
    hsa_agent_t agent,
    hsa_ext_finalization_handle_t finalization,
    uint32_t index,
    hsa_ext_code_descriptor_t * code_descriptor);
```

Queries information about one of the kernel or indirect functions that have been finalized as part of a finalization object.

Parameters

agent
(in) Agent for which the finalization object contains code.

finalization
(in) Finalization handle that references the finalization object for *agent*.

index
(in) Specifies which kernel or indirect function information is being requested. Must be in the range 0 to **hsa_ext_query_finalization_code_descriptor_count** - 1.

code_descriptor
(out) The information about the requested kernel or indirect function.

Return Values

HSA_STATUS_SUCCESS
: The function has been executed successfully, and code descriptor is queried.

HSA_STATUS_ERROR_INVALID_ARGUMENT
: If provided *agent* is NULL or not valid. If *finalization* points to invalid finalization. If *code_descriptor* is NULL.

3.1.1.31 hsa_ext_destroy_finalization

```
hsa_status_t hsa_ext_destroy_finalization(
    hsa_agent_t agent,
    hsa_ext_finalization_handle_t finalization);
```

Destroys a finalization. This may reclaim the memory occupied by the finalization object, and remove corresponding ISA code from the associated agent. Once destroyed, all code that is part of the finalization object is invalidated. It is undefined if any dispatch is executing, or will subsequently be executed, when the finalization containing its code is destroyed.

Parameters

agent
: (in) Agent for which the finalization object contains code.

finalization
: (in) Handle to the finalization to be destroyed.

Return Values

HSA_STATUS_SUCCESS
: The function has been executed successfully.

HSA_STATUS_ERROR_INVALID_ARGUMENT
: If *finalization* is NULL or does not point to a valid finalization structure.

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

3.1.1.32 hsa_ext_serialize_finalization

```
hsa_status_t hsa_ext_serialize_finalization(
    hsa_runtime_caller_t caller,
    hsa_agent_t agent,
    hsa_ext_finalization_handle_t finalization,
    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.

Parameters

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

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

finalization
(in) Handle to the finalization to be serialized.

alloc_serialize_data_callback
(in) Callback function for allocation.

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

debug_information
(in) The flag for including/excluding the debug information for *finalization*. 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* 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.

3.1.1.33 hsa_ext_deserialize_finalization

```
hsa_status_t hsa_ext_deserialize_finalization(
    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_handle_t * finalization);
```

Deserializes the finalization.

Parameters

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

serialized_object
(in) Serialized object to be deserialized.

agent
(in) The HSA agent for which *finalization* must be deserialized.

program_agent_id
(in) ID of the agent to deserialize the finalization for. Used to implement agentid_u32 operation.

program_agent_count

(in) Number of agents in the program. Used to implement agentcount_u32 operation.

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

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

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

finalization
(out) Handle to the deserialized finalization.

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*.

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

The HSAIL linking API is a vendor-neutral high level API that provides semantics for HSAIL inter-module linking. Linking between modules is done within the context of the HSAIL program.

An application can use the HSA runtime to create zero or more HSAIL programs using **hsa_ext_program_create**. Created HSAIL programs can be accessed through hsa_ext_program_handle_t, which is returned from the program creation call. When the program is created, one or more HSA components hsa_agent_t that are part of HSA platform must be specified, together with the machine model hsa_ext_brig_machine_model8_t and profile hsa_ext_brig_profile8_t.

The set of agents associated with a program cannot be changed after it has been created. Within a program, each of these HSA component members has a unique identifier of type hsa_ext_program_agent_id_t in the range 0 to the number of agent members minus one. The same HSA component may have a different agent identifier in different HSAIL programs that it is a member. In addition, each HSA component can support one or more call conventions hsa_ext_program_call_convention_id32_t. For example, an HSA component may have different call conventions that each use a different number of ISA registers to allow different numbers of wavefronts to execute on a compute unit. When the HSA runtime is used to create an HSAIL program, it determines a dense call convention ID ordering for the program. The first agent is assigned call convention IDs 0 to the number of call conventions it supports minus one. The next agent is assigned the next range of call conventions according to the number it supports, and so on. Note that the same agent may have a different range of call convention IDs in different programs of which it is a member. An HSA runtime query **hsa_ext_query_call_convention** is available to determine the range of call convention IDs used for a particular agent of a particular program.

The machine model address size for the global segment must match the size used by the application.

An application can add zero or more HSAIL modules hsa_ext_brig_module_t to the HSAIL program using **hsa_ext_add_module**. An HSAIL module is the unit of HSAIL generation, and can contain multiple symbol declarations and definitions. Distinct instances of the symbols it defines are created within each program, and symbol declarations are only linked to the definitions provided by other modules in the same program. The same HSAIL module can be added to multiple HSAIL programs, which allows multiple instances of the same kernel and indirect functions that reference distinct allocations of global segment variables. The same HSAIL module cannot be added to the same HSAIL program more than once. The machine model and profile of the HSAIL module that is being added has to match the machine model and profile of the HSAIL program it is added to. HSAIL modules and their handles can be queried from the program using several query operations, for example **hsa_ext_query_program_modules** queries the specified number of module handles, **hsa_ext_query_program_brig_module** queries the module with the specified module handle. HSAIL modules contained in the program can be accessed through the module handle hsa_ext_brig_module_handle_t.

The linking API manages linking of symbol declarations to symbol definitions between modules, therefore the low-level APIs can request certain definitions and addresses using hsa_ext_symbol_definition_callback_t and hsa_ext_symbol_address_callback_t respectively. In addition, the application can provide symbol definitions to an HSAIL program using **hsa_ext_define_agent_allocation_global_variable_address** and **hsa_ext_define_readonly_variable_address**, and request the address of symbols defined by the HSAIL program using **hsa_ext_query_symbol_definition**, **hsa_ext_query_program_allocation_global_variable_address**, **hsa_ext_query_agent_global_variable_address**, **hsa_ext_query_readonly_variable_address**.

Once the program is linked and finalized, it is the application's responsibility to destroy the program. Destroying the program will deallocate all code objects, so the code will become unavailable.

3.2.1 API

3.2.1.1 hsa_ext_program_handle_t

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

An opaque handle to the HSAIL program.

Data Fields

handle
HSA component specific handle to the program.

Description

An application can use the HSA runtime to create zero or more HSAIL programs, to which it can add zero or more HSAIL modules. HSAIL program manages linking of symbol declaration to symbol definitions between modules. In addition, the application can provide symbol definitions to an HSAIL program, and can obtain the address of symbols defined by the HSAIL program using the HSA runtime. An HSAIL program can be created with **hsa_ext_program_create**, which returns a handle to the created program hsa_ext_program_handle_t. A program handle has to be further used to add particular modules to the program using **hsa_ext_add_module**, perform various define, query and validation operations, and finalize the program using **hsa_ext_finalize_program**. A program has to be destroyed once not needed any more using **hsa_ext_program_destroy**.

3.2.1.2 hsa_ext_program_agent_id_t

```
typedef uint32_t hsa_ext_program_agent_id_t;
```

3.2.1.3 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 components that are part of the HSA platform to create a program for.

agent_count
(in) Number of HSA components to create an HSAIL program for.

machine_model
(in) The kind of machine model this HSAIL program is created for. The machine model address size for the global segment must match the size used by the applications. All module added to the program must have the same machine model.

profile
(in) The kind of profile this HSAIL program is created for. All modules added to the program must have the same profile as the program.

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, and an HSAIL program is created.

HSA_STATUS_ERROR_INVALID_ARGUMENT
If *agents* is NULL, or not valid. If *agent_count* is 0. If *machine_model* is not valid. If *profile* is not valid. In this case *program* will be NULL.

HSA_STATUS_ERROR_OUT_OF_RESOURCES
If there is a failure to allocate resources required for program creation.

HSA_EXT_STATUS_INFO_ALREADY_INITIALIZED
If *program* is already a valid program. No error is returned, just an info status is used to indicate invalid options.

Description

Creates an HSAIL program. When an HSAIL program is created, one or more HSA components that are part of the HSA platform must be specified (hsa_agent_t), together with the machine model (hsa_ext_brig_machine_model8_t) and profile (hsa_ext_brig_profile8_t). The set of agents associated with the HSAIL being created cannot be changed after the program is created. The machine model address size for the global segment must match the size used by the application. All modules added to the program must have the same machine model and profile as the program. Once the program is created, the program handle (hsa_ext_program_handle_t) is returned. See hsa_ext_program_handle_t for more details.

3.2.1.4 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, and an HSAIL program is destroyed.

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*. When the program is destroyed, member code objects are destroyed as well.

3.2.1.5 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 HSAIL module to an existing HSAIL program.

Parameters

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

brig_module
(in) HSAIL 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, and the module is added successfully.

HSA_STATUS_ERROR_INVALID_ARGUMENT
If the *program* is not a valid HSAIL program. If the *brig_module* is not a valid HSAIL module. If the machine model and/or profile of the *brig_module* do not match the machine model and/or profile *program*.

HSA_STATUS_ERROR_OUT_OF_RESOURCES
If there is a failure to allocate resources required for module addition.

HSA_EXT_STATUS_INFO_ALREADY_INITIALIZED
If the *brig_module* is already added to the HSAIL program. No error is returned, just an info status is used to indicate invalid options.

Description

Adds an existing HSAIL module to an existing HSAIL program. An HSAIL module is the unit of HSAIL generation, and can contain multiple symbol declarations and definitions. An HSAIL module can be added to zero or more HSAIL programs. Distinct instances of the symbols it defines are created within each program, and symbol declarations are only linked to the definitions provided by other modules in the same program. The same HSAIL module can be added to multiple HSAIL programs, which allows multiple instances of the same kernel and indirect functions that reference distinct allocations of global segment variables. The same HSAIL module cannot be added to the same HSAIL program more than once. The machine model and profile of the HSAIL module that is being added has to match the machine model and profile of the HSAIL program it is added to. HSAIL modules and their handles can be queried from the program using several query operations.

3.2.1.6 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 HSAIL program.

Parameters

program
(in) HSAIL program to be finalized.

agent
(in) Specific HSA component(s) to finalize program for.

finalization_request_count
(in) The number of kernels and indirect functions that are requested to be finalized.

finalization_request_list
(in) List of kernels and indirect functions that are requested to be finalized.

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) Callback function to get the string representation of the error message.

optimization_level
(in) An implementation defined value that controls the level of optimization performed by the finalizer. For more information see the HSA Programmer's Reference Manual.

options
(in) Implementation defined options that can be passed to the finalizer. For more information see the HSA Programmer's Reference Manual.

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

Return Values

HSA_STATUS_SUCCESS
The function has been executed successfully, and the requested list of kernels/functions is finalized.

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. The *error_message_callback* can be used to get the string representation of the error.

HSA_STATUS_ERROR_INVALID_ARGUMENT
If *agent* is NULL or invalid (if one of the specified HSA components is not the part of the HSAIL program). If the *program* is not a valid HSAIL program. If *finalization_request_list* is NULL or invalid. If *finalization_request_count* is 0. The *error_message_callback* can be used to get the string representation of the error.

HSA_STATUS_ERROR_OUT_OF_RESOURCES
If there is a failure to allocate resources required for finalization. The *error_message_callback* can be used to get the string representation of the error.

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

Description

Finalizes provided HSAIL program. The HSA runtime finalizer can be used to generate code for kernels and indirect functions from a specific program for a specific HSA component. A kernel can only be finalized once per program per agent. An indirect function can only finalized once per program per agent per call convention. Only code for HSA components specified when the program was created can be requested. The program must contain a definition for the requested kernels and indirect functions among the modules that have been added to the program. The modules of the program must collectively define all variables, fbarriers, kernels and functions referenced by operations in the code block. In addition, the caller of this

function can specify control directives as an input argument, which will be passed to the finalizer. These control directives can be used for low-level performance tuning, for more information on control directives see the HSA Programmer's Reference Manual.

3.2.1.7 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,
    hsa_ext_program_agent_id_t * program_agent_id);
```

Queries HSA component's ID for specified HSA component contained in specified HSAIL program.

Parameters

program
(in) HSAIL program to query HSA component's ID from.

agent
(in) HSA component for which the ID is queried.

program_agent_id
(out) HSA component's ID contained in specified HSAIL program.

Return Values

HSA_STATUS_SUCCESS
The function has been executed successfully, and HSA component's ID is queried.

HSA_STATUS_ERROR_INVALID_ARGUMENT
If provided *program* or *agent* is not valid. If *program_agent_id* is NULL.

3.2.1.8 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 the number of HSA components contained in specified HSAIL program.

Parameters

program
(in) HSAIL program to query number of HSA components from.

program_agent_count
(out) Number of HSA components contained in specified HSAIL program.

Return Values

HSA_STATUS_SUCCESS
The function has been executed successfully, and number of HSA components in the HSAIL program is queried.

HSA_STATUS_ERROR_INVALID_ARGUMENT
If provided *program* is not valid. If *program_agent_count* is NULL.

3.2.1.9 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 specified number of HSA components contained in specified HSAIL program.

Parameters

program
(in) HSAIL program to query HSA agents from.

program_agent_count
(in) Number of HSA agents to query.

agents
(out) HSA agents contained in specified HSAIL program.

Return Values

HSA_STATUS_SUCCESS
The function has been executed successfully, and HSA agents contained in the HSAIL program are queried.

HSA_STATUS_ERROR_INVALID_ARGUMENT
If provided *program* is invalid.

3.2.1.10 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 the number of HSAIL modules contained in the HSAIL program.

Parameters

program
(in) HSAIL program to query number of HSAIL modules from.

program_module_count
(out) Number of HSAIL modules in specified HSAIL program.

Return Values

HSA_STATUS_SUCCESS
The function has been executed successfully, and the number of HSAIL modules contained in the HSAIL program is queried.

HSA_STATUS_ERROR_INVALID_ARGUMENT
If provided *program* is invalid. If *program_module_count* is NULL.

3.2.1.11 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 specified number of HSAIL module handles contained in specified HSAIL program.

Parameters

program
(in) HSAIL program to query HSAIL module handles from.

program_module_count
(in) Number of HSAIL module handles to query.

modules
(out) HSAIL module handles in specified HSAIL program.

Return Values

HSA_STATUS_SUCCESS
The function has been executed successfully, and HSAIL module handles contained in the HSAIL program are queried.

HSA_STATUS_ERROR_INVALID_ARGUMENT
If provided *program* is invalid. If *modules* is NULL.

3.2.1.12 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 HSAIL module with specified handle that is contained in the specified HSAIL program.

Parameters

program
(in) HSAIL program to query HSAIL modules from.

module
(in) HSAIL module handle for which to query the HSAIL module.

brig_module
(out) HSAIL module contained in specified HSAIL program.

Return Values

HSA_STATUS_SUCCESS
The function has been executed successfully, and HSAIL module contained in the HSAIL program is queried.

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

3.2.1.13 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 IDs used for a specified HSA agent of a specified HSAIL program.

Parameters

program

(in) HSAIL program to query call convention IDs from.

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

first_call_convention_id
(out) Set of call convention IDs for specified HSA agent of a specified HSAIL program.

call_convention_count
(out) Number of call convention IDs available for specified HSA agent of a specified program.

Return Values

HSA_STATUS_SUCCESS
The function has been executed successfully, and call convention IDs contained in the HSAIL program are queried.

HSA_STATUS_ERROR_INVALID_ARGUMENT
If provided *program* is not valid. If *agent* is not valid or NULL. If *first_call_convention_id* is NULL. If *call_convention_count* is NULL.

3.2.1.14 hsa_ext_query_symbol_definition

```
hsa_status_t hsa_ext_query_symbol_definition(
    hsa_ext_program_handle_t program,
    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);
```

Queries the definition of a module scope variable/fbarrier or kernel/function for a specified HSAIL program.

Parameters

program
(in) HSAIL program to query symbol definition from.

module
(in) HSAIL module to query symbol definition from.

symbol
(in) Offset to query symbol definition from.

definition_module
(out) Queried HSAIL module handle.

definition_module_brig
(out) Queried HSAIL module.

definition_symbol
(out) Queried symbol.

Return Values

HSA_STATUS_SUCCESS
The function has been executed successfully, and symbol definition contained in the HSAIL program is queried.

HSA_STATUS_ERROR_INVALID_ARGUMENT
If provided *program* is not valid.

3.2.1.15 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 global variable address in specified HSAIL program. Allows direct access to host variables from HSAIL.

Parameters

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

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

symbol
(in) Offset in the HSAIL module to put the address on.

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

address
(in) Address to define in HSAIL program.

Return Values

HSA_STATUS_SUCCESS
The function has been executed successfully, and specified global variable address is defined in specified HSAIL program.

HSA_STATUS_ERROR_INVALID_ARGUMENT
If provided *program* is not valid. If *module* is not valid. If *address* is NULL.

3.2.1.16 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 global variable address from specified HSAIL program. Allows host program to directly access variables.

Parameters

program
(in) HSAIL program to query global variable address from.

module
(in) HSAIL module to query global variable address from.

symbol
(in) Offset in the HSAIL module to get the address from.

address
(out) Queried address.

Return Values

HSA_STATUS_SUCCESS
: The function has been executed successfully, and the global variable address is queried from specified HSAIL program.

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

3.2.1.17 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 global variable address for specified HSA agent in specified HSAIL program. Allows direct access to host variables from HSAIL.

Parameters

program
: (in) HSAIL program to define global variable address for.

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

module
: (in) HSAIL module to define global variable address for.

symbol
: (in) Offset in the HSAIL module to put the address on.

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

address
: (in) Address to define for HSA agent in HSAIL program.

Return Values

HSA_STATUS_SUCCESS
: The function has been executed successfully, and specified global variable address is defined for specified HSA agent in specified HSAIL program.

HSA_STATUS_ERROR_INVALID_ARGUMENT
: If provided *program* is not valid. If *agent* is NULL or not valid. If *module* is not valid. If *address* is NULL.

3.2.1.18 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 global variable address for specified HSA agent from specified HSAIL program. Allows host program to directly access variables.

Parameters

program
(in) HSAIL program to query global variable address from.

agent
(in) HSA agent to query global variable address from.

module
(in) HSAIL module to query global variable address from.

symbol
(in) Offset in the HSAIL module to get the address from.

address
(out) Queried address.

Return Values

HSA_STATUS_SUCCESS
The function has been executed successfully, and the global variable address is queried from specified HSAIL program.

HSA_STATUS_ERROR_INVALID_ARGUMENT
If provided *program* is not valid. If *agent* is NULL or not valid. If *module* is not valid. If *address* is NULL.

3.2.1.19 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 readonly variable address for specified HSA agent in specified HSAIL program. Allows direct access to host variables from HSAIL.

Parameters

program
(in) HSAIL program to define readonly variable address for.

agent
(in) HSA agent to define readonly variable address for.

module
(in) HSAIL module to define readonly variable address for.

symbol
(in) Offset in the HSAIL module to put the address on.

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

address
(in) Address to define for HSA agent in HSAIL program.

Return Values

HSA_STATUS_SUCCESS
: The function has been executed successfully, and specified readonly variable address is defined for specified HSA agent in specified HSAIL program.

HSA_STATUS_ERROR_INVALID_ARGUMENT
: If provided *program* is not valid. If *agent* is NULL or not valid. If *module* is not valid. If *address* is NULL.

3.2.1.20 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 readonly variable address for specified HSA agent from specified HSAIL program. Allows host program to directly access variables.

Parameters

program
: (in) HSAIL program to query readonly variable address from.

agent
: (in) HSA agent to query readonly variable address from.

module
: (in) HSAIL module to query readonly variable address from.

symbol
: (in) Offset in the HSAIL module to get the address from.

address
: (out) Queried address.

Return Values

HSA_STATUS_SUCCESS
: The function has been executed successfully, and the readonly variable address is queried from specified HSAIL program.

HSA_STATUS_ERROR_INVALID_ARGUMENT
: If provided *program* is not valid. If *agent* is NULL or not valid. If *module* is not valid. If *address* is NULL.

3.2.1.21 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,
    hsa_ext_code_descriptor_t ** kernel_descriptor);
```

Queries kernel descriptor address from specified HSAIL program. Needed to create a Dispatch packet.

Parameters

program
: (in) HSAIL program to query kernel descriptor address from.

module
(in) HSAIL module to query kernel descriptor address from.

symbol
(in) Offset in the HSAIL module to get the address from.

kernel_descriptor
(out) The address of the kernel descriptor for the requested kernel, which is an array of hsa_ext_code_descriptor_t indexed by hsa_ext_program_agent_id_t.

Return Values

HSA_STATUS_SUCCESS
The function has been executed successfully, and the kernel descriptor address is queried from specified HSAIL program.

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

3.2.1.22 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,
    hsa_ext_code_descriptor_t ** indirect_function_descriptor);
```

Queries indirect function descriptor address from specified HSAIL program. Allows host program to perform indirect function table variable initialization.

Parameters

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

module
(in) HSAIL module to query indirect function descriptor address from.

symbol
(in) Offset in the HSAIL module to get the address from.

indirect_function_descriptor
(out) The address of the indirect function descriptor for the requested indirect function, which is an array of hsa_ext_code_descriptor_t indexed by hsa_ext_program_call_convention_id32_t.

Return Values

HSA_STATUS_SUCCESS
The function has been executed successfully, and the indirect function descriptor address is queried from specified HSAIL program.

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

3.2.1.23 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 with specified HSAIL program handle. Checks if all declarations and definitions match, if there is at most one definition. The caller decides when to call validation routines. For example, it can be done in the debug mode.

Parameters

program
(in) HSAIL program to validate.

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

Return Values

HSA_STATUS_SUCCESS
The function has been executed successfully, and specified HSAIL program is a valid program.

HSA_STATUS_ERROR
If specified HSAIL program is not valid, refer to the error callback function to get string representation of the failure.

3.2.1.24 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 specified HSAIL module with specified HSAIL module handle for specified HSAIL program. Checks if BRIG for specified module is legal: operation operand type rules, etc. For more information about BRIG see the HSA Programmer's Reference Manual. The caller decides when to call validation routines. For example, it can be done in the debug mode.

Parameters

program
(in) HSAIL program to validate HSAIL module in.

module
(in) HSAIL module handle to validate.

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

Return Values

HSA_STATUS_SUCCESS
The function has been executed successfully, and specified HSAIL module is a valid module..

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

3.2.1.25 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 specified HSAIL program. Used for offline compilation.

Parameters

caller
(in) Opaque pointer to the caller of this function.

program
(in) HSAIL program to be serialized.

alloc_serialize_data_callback
(in) Callback function for memory allocation.

error_message_callback
(in) Callback function to get the string representation of the error message (if any).

debug_information
(in) The flag for including/excluding the debug information. 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, and specified program is serialized.

HSA_STATUS_ERROR_INVALID_ARGUMENT
If *program* is not a valid program.

HSA_STATUS_ERROR_OUT_OF_RESOURCES
If there is a failure to allocate resources required for serialization. The *error_message_callback* can be used to get the string representation of the error.

3.2.1.26 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);
```

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

3.2.1.27 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);
```

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

3.2.1.28 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 from a given serialized object. Used for offline compilation. Includes callback functions, where callback functions take symbolic name. This allows symbols defined by application to be relocated.

Parameters

caller
(in) Opaque pointer to the caller of this function.

serialized_object
(in) Serialized HSAIL program.

program_allocation_symbol_address
(in) Callback 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) Callback 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) Callback function to get the string representation of the error message.

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

program
(out) Deserialized HSAIL program.

Return Values

HSA_STATUS_SUCCESS
The function has been executed successfully, and HSAIL program is deserialized.

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 there is a failure to allocate resources required for deserialization. The *error_message_callback* can be used to get the string representation of the error.

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 an 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 **rdimage**, **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 is 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_t

```
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

```
typedef enum {
    HSA_EXT_IMAGE_FORMAT_NOT_SUPPORTED = 0x0,
    HSA_EXT_IMAGE_FORMAT_READ_ONLY = 0x1,
    HSA_EXT_IMAGE_FORMAT_WRITE_ONLY = 0x2,
    HSA_EXT_IMAGE_FORMAT_READ_WRITE = 0x4,
    HSA_EXT_IMAGE_FORMAT_READ_MODIFY_WRITE = 0x8,
    HSA_EXT_IMAGE_FORMAT_ACCESS_INVARIANT_IMAGE_DATA = 0x10
} hsa_ext_image_format_capability_t;
```

Image format capability returned by **hsa_ext_image_get_format_capability**.

Values

HSA_EXT_IMAGE_FORMAT_NOT_SUPPORTED
Images of this format are not supported.

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

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

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

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

HSA_EXT_IMAGE_FORMAT_ACCESS_INVARIANT_IMAGE_DATA
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_t

```
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

```
typedef enum {
    HSA_EXT_IMAGE_ACCESS_PERMISSION_READ_ONLY,
    HSA_EXT_IMAGE_ACCESS_PERMISSION_WRITE_ONLY,
    HSA_EXT_IMAGE_ACCESS_PERMISSION_READ_WRITE
} 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

```
typedef enum {
    HSA_EXT_IMAGE_GEOMETRY_1D = 0,
    HSA_EXT_IMAGE_GEOMETRY_2D = 1,
    HSA_EXT_IMAGE_GEOMETRY_3D = 2,
    HSA_EXT_IMAGE_GEOMETRY_1DA = 3,
    HSA_EXT_IMAGE_GEOMETRY_2DA = 4,
    HSA_EXT_IMAGE_GEOMETRY_1DB = 5,
    HSA_EXT_IMAGE_GEOMETRY_2DDEPTH = 6,
    HSA_EXT_IMAGE_GEOMETRY_2DADEPTH = 7
} 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
One-dimensional image addressed by width coordinate.

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

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

HSA_EXT_IMAGE_GEOMETRY_1DA
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

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

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

HSA_EXT_IMAGE_GEOMETRY_2DADEPTH
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

```
typedef enum {
    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
} 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.

3.3.1.7 hsa_ext_image_channel_order_t

```
typedef enum {
    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
} 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.

3.3.1.8 hsa_ext_image_format_t

```
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_t

```
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_t

```
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_t

```
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_t

```
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

```
typedef enum {
    HSA_EXT_SAMPLER_ADDRESSING_UNDEFINED = 0,
    HSA_EXT_SAMPLER_ADDRESSING_CLAMP_TO_EDGE = 1,
    HSA_EXT_SAMPLER_ADDRESSING_CLAMP_TO_BORDER = 2,
    HSA_EXT_SAMPLER_ADDRESSING_REPEAT = 3,
    HSA_EXT_SAMPLER_ADDRESSING_MIRRORED_REPEAT = 4
} 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
: Out-of-range coordinates are not handled.

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

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

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

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

3.3.1.14 hsa_ext_sampler_coordinate_mode_t

```
typedef enum {
    HSA_EXT_SAMPLER_COORD_NORMALIZED = 0,
    HSA_EXT_SAMPLER_COORD_UNNORMALIZED = 1
} 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
: Coordinates are all in the range of 0.0 to 1.0.

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

3.3.1.15 hsa_ext_sampler_filter_mode_t

```
typedef enum {
    HSA_EXT_SAMPLER_FILTER_NEAREST = 0,
    HSA_EXT_SAMPLER_FILTER_LINEAR = 1
} hsa_ext_sampler_filter_mode_t;
```

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

Values

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

HSA_EXT_SAMPLER_FILTER_LINEAR
: 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_t

```
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(
    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(
    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 are 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(
    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 *access_permission*.

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 supported 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(
    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_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(
    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_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(
    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_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(
    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_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(
    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(
    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(
    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.

Appendix A

Glossary

Architected Queuing Language (AQL) A command interface for the dispatch of HSA Agent commands.

Arg segment A memory segment used to pass arguments into and out of functions.

BRIG The HSAIL binary format.

Compute unit A piece of virtual hardware capable of executing the HSAIL instruction set. The work-items of a work-group are executed on the same compute unit. An HSA component is composed of one or more compute units.

Finalizer A back-end compiler that translates HSAIL code into native ISA for a compute unit.

Global segment A memory segment in which memory is visible to all work-items in all HSA components and to all host CPUs.

Grid A multidimensional, rectangular structure containing work-groups. A Grid is formed when a program launches a kernel.

Group segment A memory segment in which memory is visible to a single work-group.

Host CPU An HSA Agent that also supports the native CPU instruction set and runs the host operating system and the HSA runtime. As an HSA Agent, the host CPU can dispatch commands to an HSA Component using memory operations to construct and enqueue AQL packets. In some systems, a host CPU can also act as an HSA Component (with appropriate HSAIL finalizer and AQL mechanisms).

HSA Agent A hardware or software component that participates in the HSA memory model. An HSA Agent can submit AQL packets for execution. An HSA Agent may also but is not required to be an HSA Component. It is possible for a system to include HSA Agents that are neither HSA Components nor host CPUs.

HSA application A program written in the host CPU instruction set architecture (ISA). In addition to the host CPU code, it may include zero or more HSAIL programs.

HSA Component An HSA Agent that supports the HSAIL instruction set and the AQL packet format. As an HSA Agent, an HSA Component can dispatch commands to any HSA Component (including itself) using memory operations to construct and enqueue AQL packets. An HSA Component is composed of one or more compute units.

HSA implementation A combination of (1) hardware components that execute one or more machine instruction set architectures (ISAs), (2) a compiler, linker, and loader, (3) a finalizer that translates HSAIL code into the appropriate native ISA if the hardware components cannot support HSAIL natively, and (4) a runtime system.

HSA Packet Processor HSA Packet Processors are tightly bound to one or more HSA Agents, and provide the user mode queue functionality for the HSA Agents. HSA Packet Processors participate in the HSA memory model and are HSA Agents.

HSA runtime A library of services that can be executed by the application on a host CPU that supports the execution of HSAIL programs. This includes a finalizer that translates HSAIL code into the appropriate native ISA for each HSA component that is part of the HSA system. In addition, it supports a runtime queue that can be used by any agent, including HSA components, to submit agent dispatch packets to perform runtime functions.

HSAIL Heterogeneous System Architecture Intermediate Language. A virtual machine and a language. The instruction set of the HSA virtual machine that preserves virtual machine abstractions and allows for inexpensive translation to machine code.

Image handle An opaque handle to an image that includes information about the properties of the image and access to the image data.

Kernarg segment A memory segment used to pass arguments into a kernel.

Kernel A section of code executed in a data-parallel way by an HSA Component. Kernels are written in HSAIL and then separately translated by a finalizer to the target instruction set.

Packet ID Each AQL packet has a 64-bit identifier unique to the queue scope. The identifier is assigned as the sequential number of the packet slot allocated in the queue.

Private segment A memory segment in which memory is visible only to a single work-item. Used for read-write memory.

Readonly segment A memory segment for read-only memory.

Sampler handle An opaque handle to a sampler which specifies how coordinates are processed when an image is read within a kernel.

Segment A contiguous addressable block of memory. Segments have size, addressability, access speed, access rights, and level of sharing between work-items. Also called memory segment.

Signal (handle) An opaque handle to a signal which can be used for notification between threads and work-items belonging to a single process potentially executing on different agents in the HSA system.

Spill segment A memory segment used to load or store register spills.

Wavefront A group of work-items that share a single program counter.

Work-group A collection of work-items.

Work-item The simplest element of work.

Index - Core APIs

Index - Extension APIs

Bibliography

[1] HSA Programmer's Reference Manual. Provisional 1.0 - Ratified, HSA Foundation, 2014/06/05.

[2] HSA Platform System Architecture Specification. Provisional 1.0 - Ratified, HSA Foundation, 2014/04/18.