HSA Runtime Programmer's Reference Manual

Revision: Version 1.0 • Issue Date: 6 Feb 2015

Acknowledgments

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

AMD

- Apte, Prasad
- Blinzer, Paul
- Cao, Fan
- Cornwall, Jay
- Ding, Wei
- Edwards, Adrian
- Errabolu, Ramesh
- Keely, Sean
- Méndez-Lojo, Mario (spec editor)
- Ramalingam, Shreyas
- Sander, Ben
- Thangirala, Hari
- Tipparaju, Vinod
- Tye, Tony
- Wicaksono, Besar
- Xiao, Shucai
- Yao, Ming
- 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

- Bellows, Greg
- Bin, Lihan
- Bourd, Alex
- Gaster, Ben
- Howes, Lee
- Rychlik, Bob
- Simpson, Robert J.

Samsung Electronics

- Kohli, Soma
- Llamas, Ignacio
- Ryu, Soojung
- Shebanow, Michael

Sandia National Laboratories

- Hammond, Simon D.
- Stark, Dylan

Via Alliance Technologies

- Hong, Mike

Contents

Figures

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 kernel agents.

The HSA system architecture defines a consistent base for building portable applications that access the power and performance benefits of the dedicated kernel agents. Many of these kernel agents, 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 kernel agents 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*.

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 kernel agent provides its own implementation of the finalizer. For more information on HSAIL, refer to the *HSA Programmer's Reference Manual Version 1.0*.

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 kernel agents. 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

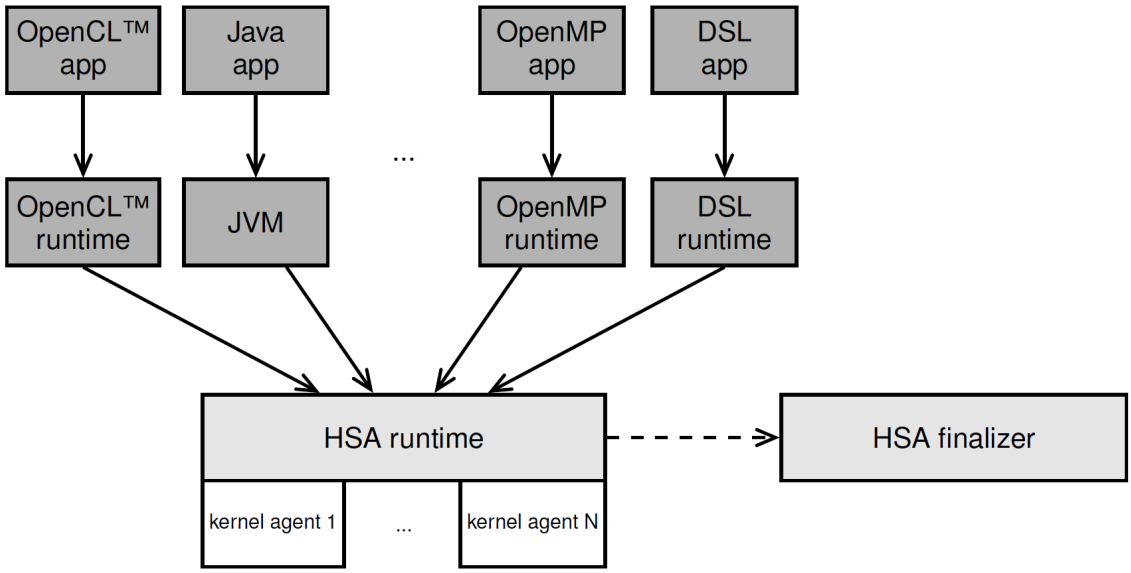


Figure 1–1 (above) shows how the HSA runtime fits into a typical software architecture stack. At the top of the stack is a programming model such as OpenCL™, Java, OpenMP, or a domain-specific language (DSL). The programming model must include some way to indicate a parallel region that can be accelerated. 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 kernel agents. 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 precompiled 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 element 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 kernel agent. The language runtime is also responsible for initializing the HSA runtime, selecting target devices, creating execution queues, managing memory. The language runtime 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. This means that languages that use the HSA runtime can execute on different vendors' platforms that support the API. Each vendor is responsible for supplying their own HSA runtime implementation that supports all of the kernel agents 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 only include user-space components (for example, simulators or CPU implementations).

Figure 1–1 (previous page) shows the "AQL" (Architected Queuing Language) path that application runtimes use to send commands directly to kernel agents. For more information on AQL, see 2.6. Architected Queuing Language Packets (page 53).

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 Agent Discovery

The first step any HSA application must perform is to initialize the runtime before invoking any other calls to the API:

```
hsa_init();
```

The next step the application performs is to find a device where it can launch the kernel. In HSA parlance, a regular device is called an *agent*, and if the agent can run kernels then it is also an *kernel agent*. The glossary at the end of this document contains more precise definitions of these terms. The HSA API uses opaque handles of type hsa_agent_t to represent agents and kernel agents.

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 kernel agent. If this is the case, the kernel agent is stored in the buffer and the iteration ends:

```
hsa_agent_t kernel_agent;
hsa_iterate_agents(get_kernel_agent, &kernel_agent);
```

where the application-provided callback *get_kernel_agent* is:

```
hsa_status_t get_kernel_agent(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_KERNEL_DISPATCH) {
        // Store kernel agent 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. System and Agent Information (page 18) 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 needs to launch a kernel in a kernel agent, it does so by placing an *AQL packet* in a *queue* owned by the kernel agent. 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 *kernel dispatch packet*.

The binary structure of the different packet types is defined in the *HSA Platform System Architecture Specification Version 1.0*. For example, all the packets types occupy 64 bytes of storage and share a common header, and the kernel dispatch packets should specify a handle to the executable code at offset 32. The packet structure is known to the application (kernel dispatch packets correspond to the hsa_kernel_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 is associated with 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 kernel agent 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(kernel_agent, 256, HSA_QUEUE_TYPE_SINGLE, NULL, NULL, UINT32_MAX, UINT32_MAX, &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 kernel dispatch packet format (type hsa_kernel_dispatch_packet_t The location of the packet buffer is available in the *base_address* field of any queue:

```
hsa_kernel_dispatch_packet_t* packet = (hsa_kernel_dispatch_packet_t*) queue->base_address;

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

// Configuration of the rest of the kernel 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 which buffer index to use to write a packet, and when to write it. For more information on queues, see 2.5. Queues (page 40). For more information on AQL packets, see 2.6. Architected Queuing Language Packets (page 53).

1.2.3 Signals and Packet launch

The kernel 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 contents of the first 32 bits of the packet (which include the *header* and the *setup* fields) must be atomically set using a release memory ordering. This ensures that previous modifications to the rest of the packet are globally visible by the time the first 32 bits of the packet are also visible. The most relevant information passed in the header is the packet's type (in this case, HSA_PACKET_TYPE_KERNEL_DISPATCH). For simplicity we omit the details on how to setup the header and setup fields (see hsa_kernel_dispatch_packet_t for the source code of the helper functions used in the snippet). One possible implementation of the atomic update in GCC is:

```
uint16_t hdr = header(HSA_PACKET_TYPE_KERNEL_DISPATCH);
uint16_t setup = kernel_dispatch_setup();
__atomic_store_n(packet, hdr | (setup << 16),  ATOMIC_RELEASE);
```

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, see 2.4. Signals (page 30).

When the runtime creates a queue, it also automatically creates a "doorbell" signal that must be used by the application to inform the packet processor 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 show how to compile a kernel, indicate which executable code to run in the kernel dispatch packet, nor 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, see 2.9. Common Definitions (page 107). The API follows the requirements listed in the *HSA Programmer's Reference Manual Version 1.0* and the *HSA Platform System Architecture Specification Version 1.0*.

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 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 Initialization and Shut Down API

2.1.1.1 hsa_init

Initialize the HSA runtime.

Signature

```
hsa_status_t hsa_init();
```

Return Values

HSA_STATUS_SUCCESS
: The function has been executed successfully.

HSA_STATUS_ERROR_OUT_OF_RESOURCES
: There is failure to allocate the resources required by the implementation.

HSA_STATUS_ERROR_REFCOUNT_OVERFLOW
: The HSA 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

Shut down the HSA runtime.

Signature

```
hsa_status_t hsa_shut_down();
```

Return Values

HSA_STATUS_SUCCESS
: The function has been executed successfully.

HSA_STATUS_ERROR_NOT_INITIALIZED
: The HSA runtime has not been initialized.

Description

Decreases the reference count of the HSA runtime instance. When the reference count reaches 0, the HSA 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 HSA runtime reaches 0, 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 0, 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 (see 2.4. Signals (page 30)) can 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 may 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 (see 2.5. Queues (page 40)) 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. An application can associate a callback with a queue at creation time.

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 Runtime Notifications API

2.2.1.1 hsa_status_t

Status codes.

Signature

```
typedef enum {
    HSA_STATUS_SUCCESS = 0x0,
    HSA_STATUS_INFO_BREAK = 0x1,
    HSA_STATUS_ERROR = 0x1000,
    HSA_STATUS_ERROR_INVALID_ARGUMENT = 0x1001,
    HSA_STATUS_ERROR_INVALID_QUEUE_CREATION = 0x1002,
    HSA_STATUS_ERROR_INVALID_ALLOCATION = 0x1003,
    HSA_STATUS_ERROR_INVALID_AGENT = 0x1004,
    HSA_STATUS_ERROR_INVALID_REGION = 0x1005,
    HSA_STATUS_ERROR_INVALID_SIGNAL = 0x1006,
    HSA_STATUS_ERROR_INVALID_QUEUE = 0x1007,
    HSA_STATUS_ERROR_OUT_OF_RESOURCES = 0x1008,
    HSA_STATUS_ERROR_INVALID_PACKET_FORMAT = 0x1009,
    HSA_STATUS_ERROR_RESOURCE_FREE = 0x100A,
    HSA_STATUS_ERROR_NOT_INITIALIZED = 0x100B,
    HSA_STATUS_ERROR_REFCOUNT_OVERFLOW = 0x100C,
    HSA_STATUS_ERROR_INCOMPATIBLE_ARGUMENTS = 0x100D,
    HSA_STATUS_ERROR_INVALID_INDEX = 0x100E,
    HSA_STATUS_ERROR_INVALID_ISA = 0x100F,
    HSA_STATUS_ERROR_INVALID_ISA_NAME = 0x1017,
    HSA_STATUS_ERROR_INVALID_CODE_OBJECT = 0x1010,
    HSA_STATUS_ERROR_INVALID_EXECUTABLE = 0x1011,
    HSA_STATUS_ERROR_FROZEN_EXECUTABLE = 0x1012,
    HSA_STATUS_ERROR_INVALID_SYMBOL_NAME = 0x1013,
    HSA_STATUS_ERROR_VARIABLE_ALREADY_DEFINED = 0x1014,
    HSA_STATUS_ERROR_VARIABLE_UNDEFINED = 0x1015,
    HSA_STATUS_ERROR_EXCEPTION = 0x1016
} 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_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 HSA runtime failed to allocate the necessary resources. This error may also occur when the HSA runtime 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 0.

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

HSA_STATUS_ERROR_INCOMPATIBLE_ARGUMENTS
The arguments passed to a functions are not compatible.

HSA_STATUS_ERROR_INVALID_INDEX
The index is invalid.

HSA_STATUS_ERROR_INVALID_ISA
The instruction set architecture is invalid.

HSA_STATUS_ERROR_INVALID_ISA_NAME
The instruction set architecture name is invalid.

HSA_STATUS_ERROR_INVALID_CODE_OBJECT
The code object is invalid.

HSA_STATUS_ERROR_INVALID_EXECUTABLE
The executable is invalid.

HSA_STATUS_ERROR_FROZEN_EXECUTABLE
The executable is frozen.

HSA_STATUS_ERROR_INVALID_SYMBOL_NAME
There is no symbol with the given name.

HSA_STATUS_ERROR_VARIABLE_ALREADY_DEFINED
The variable is already defined.

HSA_STATUS_ERROR_VARIABLE_UNDEFINED
The variable is undefined.

HSA_STATUS_ERROR_EXCEPTION
An HSAIL operation resulted on a hardware exception.

2.2.1.2 hsa_status_string

Query additional information about a status code.

Signature

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

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 HSA runtime has not been initialized.

HSA_STATUS_ERROR_INVALID_ARGUMENT
status is an invalid status code, or *status_string* is NULL.

2.3 System and Agent Information

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

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

The application can use **hsa_system_get_info** to query system-wide attributes. Note that the value of some attributes is not constant. For example, the current timestamp HSA_SYSTEM_INFO_TIMESTAMP value returned by the runtime can increase as time progresses. For more information on timestamps, see the *HSA Platform System Architecture Specification Version 1.0*.

2.3.1 System and Agent Information API

2.3.1.1 hsa_endianness_t

Endianness. A convention used to interpret the bytes making up a data word.

Signature

```
typedef enum {
    HSA_ENDIANNESS_LITTLE = 0,
    HSA_ENDIANNESS_BIG = 1
} hsa_endianness_t;
```

Values

HSA_ENDIANNESS_LITTLE
: The least significant byte is stored in the smallest address.

HSA_ENDIANNESS_BIG
: The most significant byte is stored in the smallest address.

2.3.1.2 hsa_machine_model_t

Machine model. A machine model determines the size of certain data types in HSA runtime and an agent.

Signature

```
typedef enum {
    HSA_MACHINE_MODEL_SMALL = 0,
    HSA_MACHINE_MODEL_LARGE = 1
} hsa_machine_model_t;
```

Values

HSA_MACHINE_MODEL_SMALL
: Small machine model. Addresses use 32 bits.

HSA_MACHINE_MODEL_LARGE
: Large machine model. Addresses use 64 bits.

2.3.1.3 hsa_profile_t

Profile. A profile indicates a particular level of feature support. For example, in the base profile the application must use the HSA runtime allocator to reserve Shared Virtual Memory, while in the full profile any host pointer can be shared across all the agents.

Signature

```
typedef enum {
    HSA_PROFILE_BASE = 0,
    HSA_PROFILE_FULL = 1
} hsa_profile_t;
```

Values

HSA_PROFILE_BASE
: Base profile.

HSA_PROFILE_FULL
: Full profile.

2.3.1.4 hsa_system_info_t

System attributes.

Signature

```
typedef enum {
    HSA_SYSTEM_INFO_VERSION_MAJOR = 0,
    HSA_SYSTEM_INFO_VERSION_MINOR = 1,
    HSA_SYSTEM_INFO_TIMESTAMP = 2,
    HSA_SYSTEM_INFO_TIMESTAMP_FREQUENCY = 3,
    HSA_SYSTEM_INFO_SIGNAL_MAX_WAIT = 4,
    HSA_SYSTEM_INFO_ENDIANNESS = 5,
    HSA_SYSTEM_INFO_MACHINE_MODEL = 6,
    HSA_SYSTEM_INFO_EXTENSIONS = 7
} hsa_system_info_t;
```

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 Hz. The timestamp (clock) frequency is in the range 1-400MHz. The type of this attribute is uint64_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.

HSA_SYSTEM_INFO_ENDIANNESS
Endianness of the system. The type of this attribute is hsa_endianness_t

HSA_SYSTEM_INFO_MACHINE_MODEL
Machine model supported by the HSA runtime. The type of this attribute is hsa_machine_model_t

HSA_SYSTEM_INFO_EXTENSIONS
Bit-mask indicating which extensions are supported by the implementation. An extension with an ID of *i* is supported if the bit at position *i* is set. The type of this attribute is uint8_t[128].

2.3.1.5 hsa_system_get_info

Get the current value of a system attribute.

Signature

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

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 HSA runtime has not been initialized.

HSA_STATUS_ERROR_INVALID_ARGUMENT
attribute is an invalid system attribute, or *value* is NULL.

2.3.1.6 hsa_extension_t

HSA extensions.

Signature

```
typedef enum {
    HSA_EXTENSION_FINALIZER = 0,
    HSA_EXTENSION_IMAGES = 1
} hsa_extension_t;
```

Values

HSA_EXTENSION_FINALIZER
Finalizer extension.

HSA_EXTENSION_IMAGES
Images extension.

2.3.1.7 hsa_system_extension_supported

Query if a given version of an extension is supported by the HSA implementation.

Signature

```
hsa_status_t hsa_system_extension_supported(
    uint16_t extension,
    uint16_t version_major,
    uint16_t version_minor,
    bool *result );
```

Parameters

extension
(in) Extension identifier.

version_major
(in) Major version number.

version_minor
(in) Minor version number.

result
(out) Pointer to a memory location where the HSA runtime stores the result of the check. The result is true if the specified version of the extension is supported, and false otherwise.

Return Values

HSA_STATUS_SUCCESS
The function has been executed successfully.

HSA_STATUS_ERROR_NOT_INITIALIZED
The HSA runtime has not been initialized.

HSA_STATUS_ERROR_INVALID_ARGUMENT
extension is not a valid extension, or *result* is NULL.

2.3.1.8 hsa_system_get_extension_table

Retrieve the function pointers corresponding to a given version of an extension. Portable applications are expected to invoke the extension API using the returned function pointers.

Signature

```
hsa_status_t hsa_system_get_extension_table(
    uint16_t extension,
    uint16_t version_major,
    uint16_t version_minor,
```

```
    void *table);
```

Parameters

extension
(in) Extension identifier.

version_major
(in) Major version number for which to retrieve the function pointer table.

version_minor
(in) Minor version number for which to retrieve the function pointer table.

table
(out) Pointer to an application-allocated function pointer table that is populated by the HSA runtime. Must not be NULL. The memory associated with table can be reused or freed after the function returns.

Return Values

HSA_STATUS_SUCCESS
The function has been executed successfully.

HSA_STATUS_ERROR_NOT_INITIALIZED
The HSA runtime has not been initialized.

HSA_STATUS_ERROR_INVALID_ARGUMENT
extension is not a valid extension, or *table* is NULL.

Description

The application is responsible for verifying that the given version of the extension is supported by the HSA implementation (see **hsa_system_extension_supported**). If the given combination of extension, major version, and minor version is not supported by the implementation, the behavior is undefined.

2.3.1.9 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 kernel dispatch packets launching HSAIL-derived binaries).

Signature

```
typedef struct hsa_agent_s {
    uint64_t handle;
} hsa_agent_t
```

Data Fields

handle
Opaque handle.

2.3.1.10 hsa_agent_feature_t

Agent features.

Signature

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

Values

HSA_AGENT_FEATURE_KERNEL_DISPATCH
The agent supports AQL packets of kernel dispatch type. If this feature is enabled, the agent is also a kernel agent.

HSA_AGENT_FEATURE_AGENT_DISPATCH
The agent supports AQL packets of agent dispatch type.

2.3.1.11 hsa_device_type_t

Hardware device type.

Signature

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

Values

HSA_DEVICE_TYPE_CPU
CPU device.

HSA_DEVICE_TYPE_GPU
GPU device.

HSA_DEVICE_TYPE_DSP
DSP device.

2.3.1.12 hsa_default_float_rounding_mode_t

Default floating-point rounding mode.

Signature

```
typedef enum {
    HSA_DEFAULT_FLOAT_ROUNDING_MODE_DEFAULT = 0,
    HSA_DEFAULT_FLOAT_ROUNDING_MODE_ZERO = 1,
    HSA_DEFAULT_FLOAT_ROUNDING_MODE_NEAR = 2
} hsa_default_float_rounding_mode_t;
```

Values

HSA_DEFAULT_FLOAT_ROUNDING_MODE_DEFAULT
Use a default floating-point rounding mode specified elsewhere.

HSA_DEFAULT_FLOAT_ROUNDING_MODE_ZERO
: Operations that specify the default floating-point mode are rounded to zero by default.

HSA_DEFAULT_FLOAT_ROUNDING_MODE_NEAR
: Operations that specify the default floating-point mode are rounded to the nearest representable number and that ties should be broken by selecting the value with an even least significant bit.

2.3.1.13 hsa_agent_info_t

Agent attributes.

Signature

```
typedef enum {
    HSA_AGENT_INFO_NAME = 0,
    HSA_AGENT_INFO_VENDOR_NAME = 1,
    HSA_AGENT_INFO_FEATURE = 2,
    HSA_AGENT_INFO_MACHINE_MODEL = 3,
    HSA_AGENT_INFO_PROFILE = 4,
    HSA_AGENT_INFO_DEFAULT_FLOAT_ROUNDING_MODE = 5,
    HSA_AGENT_INFO_BASE_PROFILE_DEFAULT_FLOAT_ROUNDING_MODES = 23,
    HSA_AGENT_INFO_FAST_F16_OPERATION = 24,
    HSA_AGENT_INFO_WAVEFRONT_SIZE = 6,
    HSA_AGENT_INFO_WORKGROUP_MAX_DIM = 7,
    HSA_AGENT_INFO_WORKGROUP_MAX_SIZE = 8,
    HSA_AGENT_INFO_GRID_MAX_DIM = 9,
    HSA_AGENT_INFO_GRID_MAX_SIZE = 10,
    HSA_AGENT_INFO_FBARRIER_MAX_SIZE = 11,
    HSA_AGENT_INFO_QUEUES_MAX = 12,
    HSA_AGENT_INFO_QUEUE_MIN_SIZE = 13,
    HSA_AGENT_INFO_QUEUE_MAX_SIZE = 14,
    HSA_AGENT_INFO_QUEUE_TYPE = 15,
    HSA_AGENT_INFO_NODE = 16,
    HSA_AGENT_INFO_DEVICE = 17,
    HSA_AGENT_INFO_CACHE_SIZE = 18,
    HSA_AGENT_INFO_ISA = 19,
    HSA_AGENT_INFO_EXTENSIONS = 20,
    HSA_AGENT_INFO_VERSION_MAJOR = 21,
    HSA_AGENT_INFO_VERSION_MINOR = 22
} hsa_agent_info_t;
```

Values

HSA_AGENT_INFO_NAME
: Agent name. The type of this attribute is a NUL-terminated char[64]. If the name of the agent uses less than 63 characters, the rest of the array must be filled with NULs.

HSA_AGENT_INFO_VENDOR_NAME
: Name of vendor. The type of this attribute is a NUL-terminated char[64]. If the name of the vendor uses less than 63 characters, the rest of the array must be filled with NULs.

HSA_AGENT_INFO_FEATURE
: Agent capability. The type of this attribute is hsa_agent_feature_t.

HSA_AGENT_INFO_MACHINE_MODEL
: Machine model supported by the agent. The type of this attribute is hsa_machine_model_t.

HSA_AGENT_INFO_PROFILE
Profile supported by the agent. The type of this attribute is hsa_profile_t.

HSA_AGENT_INFO_DEFAULT_FLOAT_ROUNDING_MODE
Default floating-point rounding mode. The type of this attribute is hsa_default_float_rounding_mode_t but the value HSA_DEFAULT_FLOAT_ROUNDING_MODE_DEFAULT is not allowed.

HSA_AGENT_INFO_BASE_PROFILE_DEFAULT_FLOAT_ROUNDING_MODES
Default floating-point rounding modes supported by the agent in the Base profile. The type of this attribute is a mask of hsa_default_float_rounding_mode_t. The default floating-point rounding mode (HSA_AGENT_INFO_DEFAULT_FLOAT_ROUNDING_MODE) bit must not be set.

HSA_AGENT_INFO_FAST_F16_OPERATION
Flag indicating that the f16 HSAIL operation is at least as fast as the f32 operation in the current agent. The value of this attribute is undefined if the agent is not a kernel agent. The type of this attribute is bool.

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 kernel agent. 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 kernel agent. 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 kernel agent. 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_DIM The value of this attribute is undefined if the agent is not a kernel agent. 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 kernel agent. 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 kernel agent. 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_MIN_SIZE
Minimum number of packets that a queue created in the agent can hold. Must be a power of 2 greater than 0. Must not exceed the value of HSA_AGENT_INFO_QUEUE_MAX_SIZE 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 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_AGENT_INFO_ISA
Instruction set architecture of the agent. The type of this attribute is hsa_isa_t.

HSA_AGENT_INFO_EXTENSIONS
Bit-mask indicating which extensions are supported by the agent. An extension with an ID of *i* is supported if the bit at position *i* is set. The type of this attribute is uint8_t[128].

HSA_AGENT_INFO_VERSION_MAJOR
Major version of the HSA runtime specification supported by the agent. The type of this attribute is uint16_t.

HSA_AGENT_INFO_VERSION_MINOR
Minor version of the HSA runtime specification supported by the agent. The type of this attribute is uint16_t.

2.3.1.14 hsa_agent_get_info

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

Signature

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

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 HSA runtime has not been initialized.

HSA_STATUS_ERROR_INVALID_AGENT
The agent is invalid.

HSA_STATUS_ERROR_INVALID_ARGUMENT
attribute is an invalid agent attribute, or *value* is NULL.

2.3.1.15 hsa_iterate_agents

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

Signature

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

Parameters

callback
(in) Callback to be invoked once per agent. The HSA 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. May be NULL.

Return Values

HSA_STATUS_SUCCESS
The function has been executed successfully.

HSA_STATUS_ERROR_NOT_INITIALIZED
The HSA runtime has not been initialized.

HSA_STATUS_ERROR_INVALID_ARGUMENT
callback is NULL.

2.3.1.16 hsa_exception_policy_t

Exception policies applied in the presence of hardware exceptions.

Signature

```
typedef enum {
    HSA_EXCEPTION_POLICY_BREAK = 1,
    HSA_EXCEPTION_POLICY_DETECT = 2
} hsa_exception_policy_t;
```

Values

HSA_EXCEPTION_POLICY_BREAK
If a hardware exception is detected, a work-item signals an exception.

HSA_EXCEPTION_POLICY_DETECT
If a hardware exception is detected, a hardware status bit is set.

2.3.1.17 hsa_agent_get_exception_policies

Retrieve the exception policy support for a given combination of agent and profile.

Signature

```
hsa_status_t hsa_agent_get_exception_policies(
    hsa_agent_t agent,
    hsa_profile_t profile,
    uint16_t *mask);
```

Parameters

agent
(in) Agent.

profile
(in) Profile.

mask
(out) Pointer to a memory location where the HSA runtime stores a mask of hsa_exception_policy_t values. Must not be NULL.

Return Values

HSA_STATUS_SUCCESS
The function has been executed successfully.

HSA_STATUS_ERROR_NOT_INITIALIZED
The HSA runtime has not been initialized.

HSA_STATUS_ERROR_INVALID_AGENT
The agent is invalid.

HSA_STATUS_ERROR_INVALID_ARGUMENT
profile is not a valid profile, or *mask* is NULL.

2.3.1.18 hsa_agent_extension_supported

Query if a given version of an extension is supported by an agent.

Signature

```
hsa_status_t hsa_agent_extension_supported(
    uint16_t extension,
    hsa_agent_t agent,
    uint16_t version_major,
    uint16_t version_minor,
    bool *result);
```

Parameters

extension
(in) Extension identifier.

agent
(in) Agent.

version_major
(in) Major version number.

version_minor
(in) Minor version number.

result
(out) Pointer to a memory location where the HSA runtime stores the result of the check. The result is true if the specified version of the extension is supported, and false otherwise.

Return Values

HSA_STATUS_SUCCESS
The function has been executed successfully.

HSA_STATUS_ERROR_NOT_INITIALIZED
The HSA runtime has not been initialized.

HSA_STATUS_ERROR_INVALID_AGENT
The agent is invalid.

HSA_STATUS_ERROR_INVALID_ARGUMENT
extension is not a valid extension, or *result* is NULL.

2.4 Signals

Agents can communicate with each other by using coherent shared (global) memory or by using signals. Agents can perform operations on signals similar to operations performed on shared memory locations. For example, an agent can 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 a signed 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 the function **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 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 (*HSA Platform System Architecture Specification Version 1.0*; *HSA Programmer's Reference Manual Version 1.0*).

The application may wait on a signal, with a condition specifying the terms of the wait. The wait can be done either in the kernel agent 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 to read it.

2.4.1 Signals API

2.4.1.1 hsa_signal_value_t

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

Signature

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

2.4.1.2 hsa_signal_t

Signal handle.

Signature

```
typedef struct hsa_signal_s {
    uint64_t handle;
} hsa_signal_t
```

Data Fields

handle
Opaque handle. The value 0 is reserved.

2.4.1.3 hsa_signal_create

Create a signal.

Signature

```
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);
```

Parameters

initial_value
(in) Initial value of the signal.

num_consumers
(in) Size of consumers. A value of 0 indicates that any 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 HSA 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 HSA 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 HSA runtime has not been initialized.

HSA_STATUS_ERROR_OUT_OF_RESOURCES
There is failure to allocate the resources required by the implementation.

HSA_STATUS_ERROR_INVALID_ARGUMENT
signal is NULL, *num_consumers* is greater than 0 but *consumers* is NULL, or *consumers* contains duplicates.

2.4.1.4 hsa_signal_destroy

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

Signature

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

Parameters

signal
(in) Signal.

Return Values

HSA_STATUS_SUCCESS
The function has been executed successfully.

HSA_STATUS_ERROR_NOT_INITIALIZED
The HSA runtime has not been initialized.

HSA_STATUS_ERROR_INVALID_SIGNAL
signal is invalid.

HSA_STATUS_ERROR_INVALID_ARGUMENT
The handle in *signal* is 0.

2.4.1.5 hsa_signal_load

Atomically read the current value of a signal.

Signature

```
hsa_signal_value_t  hsa_signal_load_acquire(
    hsa_signal_t signal);

hsa_signal_value_t  hsa_signal_load_relaxed(
    hsa_signal_t signal);
```

Parameters

signal
(in) Signal.

Returns

Value of the signal.

2.4.1.6 hsa_signal_store

Atomically set the value of a signal.

Signature

```
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);
```

Parameters

signal
(in) Signal.

value
(in) New signal value.

Description

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

2.4.1.7 hsa_signal_exchange

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

Signature

```
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);
```

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 changed, all the agents waiting on *signal* for which *value* satisfies their wait condition are awakened.

2.4.1.8 hsa_signal_cas

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.

Signature

```
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);
```

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 changed, all the agents waiting on *signal* for which *value* satisfies their wait condition are awakened.

2.4.1.9 hsa_signal_add

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

Signature

```
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);
```

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 changed, all the agents waiting on *signal* for which *value* satisfies their wait condition are awakened.

2.4.1.10 hsa_signal_subtract

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

Signature

```
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);
```

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 changed, all the agents waiting on *signal* for which *value* satisfies their wait condition are awakened.

2.4.1.11 hsa_signal_and

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

Signature

```
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);
```

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 changed, all the agents waiting on *signal* for which *value* satisfies their wait condition are awakened.

2.4.1.12 hsa_signal_or

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

Signature

```
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);
```

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 changed, all the agents waiting on *signal* for which *value* satisfies their wait condition are awakened.

2.4.1.13 hsa_signal_xor

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

Signature

```
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);
```

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 changed, all the agents waiting on *signal* for which *value* satisfies their wait condition are awakened.

2.4.1.14 hsa_signal_condition_t

Wait condition operator.

Signature

```
typedef enum {
    HSA_SIGNAL_CONDITION_EQ = 0,
    HSA_SIGNAL_CONDITION_NE = 1,
    HSA_SIGNAL_CONDITION_LT = 2,
    HSA_SIGNAL_CONDITION_GTE = 3
    } hsa_signal_condition_t;
```

Values

HSA_SIGNAL_CONDITION_EQ
The two operands are equal.

HSA_SIGNAL_CONDITION_NE
The two operands are not equal.

HSA_SIGNAL_CONDITION_LT
The first operand is less than the second operand.

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

2.4.1.15 hsa_wait_state_t

State of the application thread during a signal wait.

Signature

```
typedef enum {
    HSA_WAIT_STATE_BLOCKED = 0,
    HSA_WAIT_STATE_ACTIVE = 1
    } hsa_wait_state_t;
```

Values

HSA_WAIT_STATE_BLOCKED
The application thread may be rescheduled while waiting on the signal.

HSA_WAIT_STATE_ACTIVE
The application thread stays active while waiting on a signal.

2.4.1.16 hsa_signal_wait

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

Signature

```
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_state_t wait_state_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_state_t wait_state_hint );
```

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. The operation might block for a shorter or longer time even if the condition is not met. A value of UINT64_MAX indicates no maximum.

wait_state_hint
(in) Hint used by the application to indicate the preferred waiting state. The actual waiting state is ultimately decided by HSA runtime and may not match the provided hint. A value of HSA_WAIT_STATE_ACTIVE may improve the latency of response to a signal update by avoiding rescheduling overhead.

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. The application must ensure that signals are used in such way that wait wakeup conditions are not invalidated before dependent threads have woken up.

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

2.5 Queues

HSA hardware supports command execution through user mode queues. A user mode command queue is characterized (see the *HSA Platform System Architecture Specification Version 1.0*) 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. An agent may 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 (has space for the packet) 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 plus 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 of the packet 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 the 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 to 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 1.2. Programming Model (page 11) 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 kernel 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.

```
void simple_dispatch() {
    // Initialize the runtime
    hsa_init();

    // Retrieve the kernel agent
    hsa_agent_t kernel_agent;
    hsa_iterate_agents(get_kernel_agent,  &kernel_agent);

    // Create a queue in the kernel agent. The queue can hold 4 packets, and has no callback or service queue associated with it
    hsa_queue_t *queue;
    hsa_queue_create(kernel_agent, 4, HSA_QUEUE_TYPE_SINGLE, NULL, NULL, 0, 0, &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_kernel_dispatch_packet_t* packet = (hsa_kernel_dispatch_packet_t*) queue->base_address + packet_id;

    // Populate fields in kernel dispatch packet, except for the header, the setup, and
    // the completion signal fields
    initialize_packet(packet);

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

    // Notify the queue that the packet is ready to be processed
    packet_store_release((uint32_t*) packet, header(HSA_PACKET_TYPE_KERNEL_DISPATCH), kernel_dispatch_setup());
    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(packet->completion_signal, HSA_SIGNAL_CONDITION_EQ, 0, UINT64_MAX, HSA_WAIT_
        STATE_ACTIVE) != 0);

    // Done! The kernel has completed. Time to cleanup resources and leave
    hsa_signal_destroy(packet->completion_signal);
    hsa_queue_destroy(queue);
    hsa_shut_down();
}
```

The definitions of the helper functions (such as initialize_packet) are listed in 2.6.1. Kernel dispatch packet (page 53).

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 kernel_agent;
hsa_iterate_agents(get_kernel_agent,  &kernel_agent);
hsa_queue_t *queue;
hsa_queue_create(kernel_agent, 4, HSA_QUEUE_TYPE_SINGLE, callback, NULL, UINT32_MAX, UINT32_MAX, &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, void* data) {
    const char* message;
    hsa_status_string(status, &message);
    printf("Error at queue %" PRIu64 ": %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 kernel agent that allows applications to create queues supporting multiple producers:

```
hsa_status_t get_multi_kernel_agent(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_KERNEL_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 kernel_agent;
hsa_iterate_agents(get_kernel_agent,  &kernel_agent);
hsa_queue_t *queue;
hsa_queue_create(kernel_agent, 4, HSA_QUEUE_TYPE_MULTI, callback, NULL, UINT32_MAX, UINT32_MAX, &queue);
```

Each CPU thread submits 1,000 kernel 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_kernel_dispatch_packet_t* packets = (hsa_kernel_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_kernel_dispatch_packet_t* packet = packets + packet_id % queue->size;

        initialize_packet(packet);
        packet->kernarg_address = counter;
        packet->completion_signal = signal;
        packet_store_release((uint32_t*) packet, header(HSA_PACKET_TYPE_KERNEL_DISPATCH), kernel_dispatch_
            setup());
        hsa_signal_store_release(queue->doorbell_signal,   packet_id);
        }

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

2.5.4 Queues API

2.5.4.1 hsa_queue_type_t

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

Signature

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

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

Queue features.

Signature

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

Values

HSA_QUEUE_FEATURE_KERNEL_DISPATCH
Queue supports kernel dispatch packets.

HSA_QUEUE_FEATURE_AGENT_DISPATCH
Queue supports agent dispatch packets.

2.5.4.3 hsa_queue_t

User mode queue.

Signature

```
typedef struct hsa_queue_s {
    hsa_queue_type_t type;
    uint32_t features;

#ifdef HSA_LARGE_MODEL
    void * base_address;
#elif defined HSA_LITTLE_ENDIAN
    void * base_address;
    uint32_t reserved0;
#else
    uint32_t reserved0;
    void * base_address;
#endif

    hsa_signal_t doorbell_signal;
    uint32_t size;
    uint32_t reserved1;
    uint64_t id ;
} hsa_queue_t
```

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 HSA runtime-allocated buffer used to store the AQL packets. Must be aligned to the size of an AQL packet.

reserved0
Reserved. Must be 0.

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.

reserved1
Reserved. Must be 0.

id
Queue identifier, which is unique over the lifetime of the application.

Description

The queue structure is read-only and allocated by the HSA runtime, but agents can directly modify the contents of the buffer pointed by *base_address*, or use HSA runtime APIs to access the doorbell signal.

2.5.4.4 hsa_queue_create

Create a user mode queue.

Signature

```
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, void *data),
    void *data,
    uint32_t private_segment_size,
    uint32_t group_segment_size,
    hsa_queue_t **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 between 1 and the value of HSA_AGENT_INFO_QUEUE_MAX_SIZE in *agent*. The size of the newly created queue is the maximum of *size* and the value of HSA_AGENT_INFO_QUEUE_MIN_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 HSA runtime for every asynchronous event related to the newly created queue. May be NULL. The HSA runtime passes three arguments to the callback: a code identifying the event that triggered the invocation, a pointer to the queue where the event originated, and the application data.

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

private_segment_size
(in) Hint indicating the maximum expected private segment usage per work-item, in bytes. There may be performance degradation if the application places a kernel dispatch packet in the queue and the corresponding private segment usage exceeds *private_segment_size*. If the application does not want to specify any particular value for this argument, *private_segment_size* must be UINT32_MAX. If the queue does not support kernel dispatch packets, this argument is ignored.

group_segment_size
(in) Hint indicating the maximum expected group segment usage per work-group, in bytes. There may be performance degradation if the application places a kernel dispatch packet in the queue and the corresponding group segment usage exceeds *group_segment_size*. If the application does not want to specify any particular value for this argument, *group_segment_size* must be UINT32_MAX. If the queue does not support kernel dispatch packets, this argument is ignored.

queue
(out) Memory location where the HSA 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 HSA runtime has not been initialized.

HSA_STATUS_ERROR_OUT_OF_RESOURCES
There is failure to allocate the resources required by the implementation.

HSA_STATUS_ERROR_INVALID_AGENT
The agent is invalid.

HSA_STATUS_ERROR_INVALID_QUEUE_CREATION
agent does not support queues of the given type.

HSA_STATUS_ERROR_INVALID_ARGUMENT
size is not a power of two, *size* is 0, *type* is an invalid queue type, or *queue* is NULL.

Description

The HSA runtime creates the queue structure, the underlying 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_INVALID.

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

2.5.4.5 hsa_soft_queue_create

Create a queue for which the application or a kernel is responsible for processing the AQL packets.

Signature

```
hsa_status_t hsa_soft_queue_create(
```

```
    hsa_region_t region,
    uint32_t size,
    hsa_queue_type_t type,
    uint32_t features,
    hsa_signal_t doorbell_signal,
    hsa_queue_t **queue);
```

Parameters

region
(in) Memory region that the HSA runtime should use to allocate the AQL packet buffer and any other queue metadata.

size
(in) Number of packets the queue is expected to hold. Must be a power of 2 greater than 0.

type
(in) Queue type.

features
(in) Supported queue features. This is a bit-field of hsa_queue_feature_t values.

doorbell_signal
(in) Doorbell signal that the HSA runtime must associate with the returned queue. The signal handle must not be 0.

queue
(out) Memory location where the HSA runtime stores a pointer to the newly created queue. The application should not rely on the value returned for this argument but only in the status code to determine if the queue is valid. Must not be NULL.

Return Values

HSA_STATUS_SUCCESS
The function has been executed successfully.

HSA_STATUS_ERROR_NOT_INITIALIZED
The HSA runtime has not been initialized.

HSA_STATUS_ERROR_OUT_OF_RESOURCES
There is failure to allocate the resources required by the implementation.

HSA_STATUS_ERROR_INVALID_ARGUMENT
size is not a power of two, *size* is 0, *type* is an invalid queue type, the doorbell signal handle is 0, or *queue* is NULL.

Description

The application can use this function to create queues where AQL packets are not parsed by the packet processor associated with an agent, but rather by a unit of execution running on that agent (for example, a thread in the host application).

The application is responsible for ensuring that all the producers and consumers of the resulting queue can access the provided doorbell signal and memory region. The application is also responsible for ensuring that the unit of execution processing the queue packets supports the indicated features (AQL packet types).

When the queue is created, the HSA runtime allocates the packet buffer using *region*, and the write and read indices. The initial value of the write and read indices is 0, and the type of every packet in the buffer is initialized to HSA_PACKET_TYPE_INVALID. The value of the *size*, *type*, *features*, and *doorbell_signal* fields in the returned queue match the values passed by the application.

2.5.4.6 hsa_queue_destroy

Destroy a user mode queue.

Signature

```
hsa_status_t hsa_queue_destroy(
    hsa_queue_t *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 HSA runtime has not been initialized.

HSA_STATUS_ERROR_INVALID_QUEUE
The queue is invalid.

HSA_STATUS_ERROR_INVALID_ARGUMENT
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 HSA runtime during queue creation (queue structure, ring buffer, doorbell signal) are released. The queue should not be accessed after being destroyed.

2.5.4.7 hsa_queue_inactivate

Inactivate a queue.

Signature

```
hsa_status_t hsa_queue_inactivate(
    hsa_queue_t *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 HSA runtime has not been initialized.

HSA_STATUS_ERROR_INVALID_QUEUE
The queue is invalid.

HSA_STATUS_ERROR_INVALID_ARGUMENT
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.

2.5.4.8 hsa_queue_load_read_index

Atomically load the read index of a queue.

Signature

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

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

Parameters

queue
(in) Pointer to a queue.

Returns

Read index of the queue pointed by *queue*.

2.5.4.9 hsa_queue_load_write_index

Atomically load the write index of a queue.

Signature

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

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

Parameters

queue
(in) Pointer to a queue.

Returns

Write index of the queue pointed by *queue*.

2.5.4.10 hsa_queue_store_write_index

Atomically set the write index of a queue.

Signature

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

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

Parameters

queue
(in) Pointer to a queue.

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

2.5.4.11 hsa_queue_cas_write_index

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.

Signature

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

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

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

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

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.

2.5.4.12 hsa_queue_add_write_index

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

Signature

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

uint64_t  hsa_queue_add_write_index_acquire(
      const hsa_queue_t *queue,
      uint64_t value);

uint64_t  hsa_queue_add_write_index_relaxed(
      const hsa_queue_t *queue,
      uint64_t value);

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

Parameters

queue
(in) Pointer to a queue.

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

Returns

Previous value of the write index.

2.5.4.13 hsa_queue_store_read_index

Atomically set the read index of a queue.

Signature

```
void  hsa_queue_store_read_index_relaxed(
      const hsa_queue_t *queue,
```

```
    uint64_t value);

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

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 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 specific format that encodes 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 required to explicitly reserve storage space for packets because a queue already contains a command buffer where AQL packets can be written.

The HSA API defines the format of the different packet types: kernel dispatch, agent dispatch, barrier-AND and barrier-OR. 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), and other properties.

2.6.1 Kernel dispatch packet

An application uses a kernel dispatch packet hsa_kernel_dispatch_packet_t to submit a kernel to a kernel agent. The packet contains the following bits of information:

- A pointer to the kernel executable code is stored in *kernel_object*.
- 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 2.7. Memory (page 66), 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. Each executable symbol (hsa_executable_symbol_t) associated with a kernel exposes the kernarg (HSA_EXECUTABLE_SYMBOL_INFO_KERNEL_KERNARG_SEGMENT_SIZE), group (HSA_EXECUTABLE_SYMBOL_INFO_KERNEL_GROUP_SEGMENT_SIZE), and private (HSA_EXECUTABLE_SYMBOL_INFO_KERNEL_PRIVATE_SEGMENT_SIZE) static storage requirements.

2.6.1.1 Example: populating the kernel dispatch packet

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

```
void initialize_packet(hsa_kernel_dispatch_packet_t* packet) {
    // Reserved fields, private and group memory, and completion signal are all set to 0.
    memset(((uint8_t*) packet) + 4, 0, sizeof(hsa_kernel_dispatch_packet_t) - 4);

    packet->workgroup_size_x = 256;
    packet->workgroup_size_y = 1;
    packet->workgroup_size_z = 1;
    packet->grid_size_x = 256;
    packet->grid_size_y = 1;
    packet->grid_size_z = 1;

    // Indicate which executable code to run.
    // The application is expected to have finalized a kernel (for example, using the finalization API).
    // We will assume that the kernel object containing the executable code is stored in KERNEL_OBJECT
    packet->kernel_object = KERNEL_OBJECT;

    // Assume our kernel receives no arguments
    packet->kernarg_address = NULL;
}
```

The definition of the function that atomically sets the first 32 bits of an AQL packet (the header and setup fields, in the case of a kernel dispatch packet) depends on the library used by the application to perform atomic memory updates. In GCC, a possible definition would be:

```
void packet_store_release(uint32_t* packet, uint16_t header, uint16_t rest) {
    __atomic_store_n(packet, header | (rest << 16),  __ATOMIC_RELEASE);
}
```

The header is built using the following function:

```
uint16_t header(hsa_packet_type_t type) {
    uint16_t header = type << HSA_PACKET_HEADER_TYPE;
    header |= HSA_FENCE_SCOPE_SYSTEM << HSA_PACKET_HEADER_ACQUIRE_FENCE_SCOPE;
    header |= HSA_FENCE_SCOPE_SYSTEM << HSA_PACKET_HEADER_RELEASE_FENCE_SCOPE;
    return header;
}
```

The *setup* contents indicate that the dispatch uses one dimension:

```
uint16_t kernel_dispatch_setup() {
    return 1 << HSA_KERNEL_DISPATCH_PACKET_SETUP_DIMENSIONS;
}
```

2.6.2 Agent dispatch packet

Applications use agent dispatch packets 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.

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

1. (Application) Locate the agent associated with the host (CPU) by calling **hsa_iterate_agents**.
2. (Application) Locate a memory *region* that is accessible to the host and the kernel agent – for example, a global fine-grained region. The function **hsa_agent_iterate_regions** lists the memory regions associated with a given agent. Memory regions are explained in 2.7. Memory (page 66).
3. (Application) Create a signal by calling **hsa_signal_create**.
4. (Application) Create a *soft* queue (using the signal and region handles retrieved before) that supports agent dispatch packets using **hsa_soft_queue_create**. The application is responsible for processing the AQL packets enqueued in the soft queue.
5. (Application) Launch a kernel in a kernel agent. The work-items executing the kernel have access to the soft queue – for example, it has been passed as an argument in a kernel dispatch packet.
6. (Kernel) When a work-item needs to execute a given built-in (service), it submits an agent dispatch packet to the soft queue following the steps described in 2.5. Queues (page 40).
7. (Application) The application 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.
8. (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 kernel agent

In this example, work-items in a kernel can 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 the CPU agent:

```
hsa_status_t get_cpu_agent(hsa_agent_t agent, void* data) {
    hsa_device_type_t device;
    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;
}
```

, and the associated fine-grained memory region:

```
hsa_status_t get_fine_grained_region(hsa_region_t region, void* data) {
```

```
    hsa_region_segment_t segment;
    hsa_region_get_info(region, HSA_REGION_INFO_SEGMENT, &segment);
    if (segment != HSA_REGION_SEGMENT_GLOBAL) {
        return HSA_STATUS_SUCCESS;
    }
    hsa_region_global_flag_t flags;
    hsa_region_get_info(region, HSA_REGION_INFO_GLOBAL_FLAGS, &flags);
    if (flags & HSA_REGION_GLOBAL_FLAG_FINE_GRAINED) {
        hsa_region_t* ret = (hsa_region_t*) data;
        *ret = region;
        return HSA_STATUS_INFO_BREAK;
    }
    return HSA_STATUS_SUCCESS;
}
```

Afterwards, the application creates the soft queue where the allocation requests will be enqueued:

```
hsa_agent_t cpu_agent;
hsa_iterate_agents(get_cpu_agent,  &cpu_agent);

hsa_region_t region;
hsa_agent_iterate_regions(cpu_agent, get_fine_grained_region, &region);

hsa_signal_t completion_signal;
hsa_signal_create(1, 0, NULL, &completion_signal);

hsa_queue_t *soft_queue;
hsa_soft_queue_create(region, 16, HSA_QUEUE_TYPE_MULTI, HSA_QUEUE_FEATURE_AGENT_DISPATCH,
    completion_signal, &soft_queue);
```

Note how the application (and not the HSA runtime) decides which completion signal and memory must be used when creating the queue. This is a major difference with respect to "regular" queues created using **hsa_queue_create**.

The application now creates a regular queue on a kernel agent and launches a kernel in that queue. A pointer to the soft queue is passed as an argument to the kernel dispatch packet by placing it into the buffer stored in the *kernarg_address* field. 2.7.1. Global memory (page 67) describes the HSA runtime elements needed for allocating memory that can be used to pass kernel arguments.

Every time a work-item needs more virtual memory, it will submit an agent dispatch packet to the soft 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 example code shows how a thread running on the host might process the agent dispatch packets submitted from the kernel agent. The application waits for the value of the doorbell signal in the soft queue to be monotonically increased. When that happens, the thread processes the allocation request by invoking malloc.

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

    while (read_index < 100) {

        while (hsa_signal_wait_acquire(doorbell, HSA_SIGNAL_CONDITION_GTE, read_index, UINT64_MAX,
            HSA_WAIT_STATE_BLOCKED) <
            (hsa_signal_value_t) read_index);
```

```
            hsa_agent_dispatch_packet_t* packet = packets + read_index % queue->size;

            if (packet->type == 0x8000) {
                // kernel agent 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.handle != 0) {
                hsa_signal_subtract_release(packet->completion_signal, 1);
            }
            packet_store_release((uint32_t*) packet, header(HSA_PACKET_TYPE_INVALID), packet->type);
            read_index++;
            hsa_queue_store_read_index_release(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-AND and barrier-OR packets

The barrier-AND packet (of type hsa_barrier_and_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-AND packet is complete. A barrier-AND packet is complete when all of the dependent signals have been observed with the value 0 after the barrier-AND packet launched. It is not required that all dependent signals are observed to be 0 at the same time.

The barrier-OR packet (of type hsa_barrier_or_packet_t) is very similar to the barrier-AND packet, but it becomes complete when the packet processor observes that any of the dependent signals have a value of 0.

2.6.3.1 Example: handling dependencies across kernels running in different kernel agents

A combination of completion signals and barrier-AND 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 kernel agent *B* consumes the result of kernel *a* executing in a different kernel agent *A*, then *b* depends on *a*. In HSA, this dependency can be enforced across kernel agents by creating a signal that will be simultaneously used as 1) the completion signal of a kernel dispatch packet *packet_a* corresponding to *a* 2) the dependency signal in a barrier-AND packet that precedes the kernel 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.

```
void barrier(){
    hsa_init();

    // Find available kernel agents. Let's assume there are two, A and B
    hsa_agent_t* kernel_agent = get_kernel_agents();

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

```
    hsa_kernel_dispatch_packet_t* packet_a = (hsa_kernel_dispatch_packet_t*) queue_a->base_address + packet_id_a;
    initialize_packet(packet_a);
    // KERNEL_OBJECT_A is the 1st kernel object
    packet_a->kernel_object = (uint64_t) KERNEL_OBJECT_A;

    // Create a signal with a value of 1 and attach it to the first kernel dispatch packet
    hsa_signal_create(1, 0, NULL, &packet_a->completion_signal);

    // Tell packet processor of A to launch the first kernel dispatch packet
    packet_store_release((uint32_t*) packet_a, header(HSA_PACKET_TYPE_KERNEL_DISPATCH), kernel_dispatch_setup());
    hsa_signal_store_release(queue_a->doorbell_signal, packet_id_a);

    // Create queue in kernel agent B
    hsa_queue_t *queue_b;
    hsa_queue_create(kernel_agent[1], 4, HSA_QUEUE_TYPE_SINGLE, NULL, NULL, 0, 0, &queue_b);
    uint64_t packet_id_b = hsa_queue_add_write_index_relaxed(queue_b, 2);

    // Create barrier-AND packet that is enqueued in a queue of B
    hsa_barrier_and_packet_t* barrier_and_packet = (hsa_barrier_and_packet_t*)queue_b->base_address + packet_id_b;
    memset(((uint8_t*) barrier_and_packet) + 4, 0, sizeof(*barrier_and_packet) - 4);

    // Add dependency on the first kernel dispatch packet
    barrier_and_packet->dep_signal[0] = packet_a->completion_signal;
    packet_store_release((uint32_t*) barrier_and_packet, header(HSA_PACKET_TYPE_BARRIER_AND), 0);

    // Create and enqueue a second kernel dispatch packet after the barrier-AND in B. The second dispatch is launched after
    // the first has completed
    hsa_kernel_dispatch_packet_t* packet_b = (hsa_kernel_dispatch_packet_t*) queue_b->base_address + packet_id_b;
    initialize_packet(packet_b);
    // KERNEL_OBJECT_B is the 2nd kernel object
    packet_b->kernel_object = (uint64_t) KERNEL_OBJECT_B;
    hsa_signal_create(1, 0, NULL, &packet_b->completion_signal);

    packet_store_release((uint32_t*) packet_b, header(HSA_PACKET_TYPE_KERNEL_DISPATCH), kernel_dispatch_setup());
    hsa_signal_store_release(queue_b->doorbell_signal, packet_id_b + 1);

    while (hsa_signal_wait_acquire(packet_b->completion_signal, HSA_SIGNAL_CONDITION_EQ, 0, UINT64_MAX,
        HSA_WAIT_STATE_ACTIVE) != 0);

    hsa_signal_destroy(packet_b->completion_signal);
    hsa_queue_destroy(queue_b);
    hsa_signal_destroy(packet_a->completion_signal);
    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 (facing page) shows the state transition diagram.

In queue The packet processor has not started to parse the current packet. If the barrier bit is set in the header, the transition to the launch state occurs only after all the preceding packets have completed their execution. If the barrier 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 has not started execution. This phase finalizes by applying an acquire memory fence with the scope indicated by the acquire fence scope field in the header. Memory fences are explained in the *HSA Programmer's Reference Manual Version 1.0*.

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 can be returned:

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

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

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 field in the header, and the HSA runtime invokes the application callback associated with the queue. The following status codes can be returned:

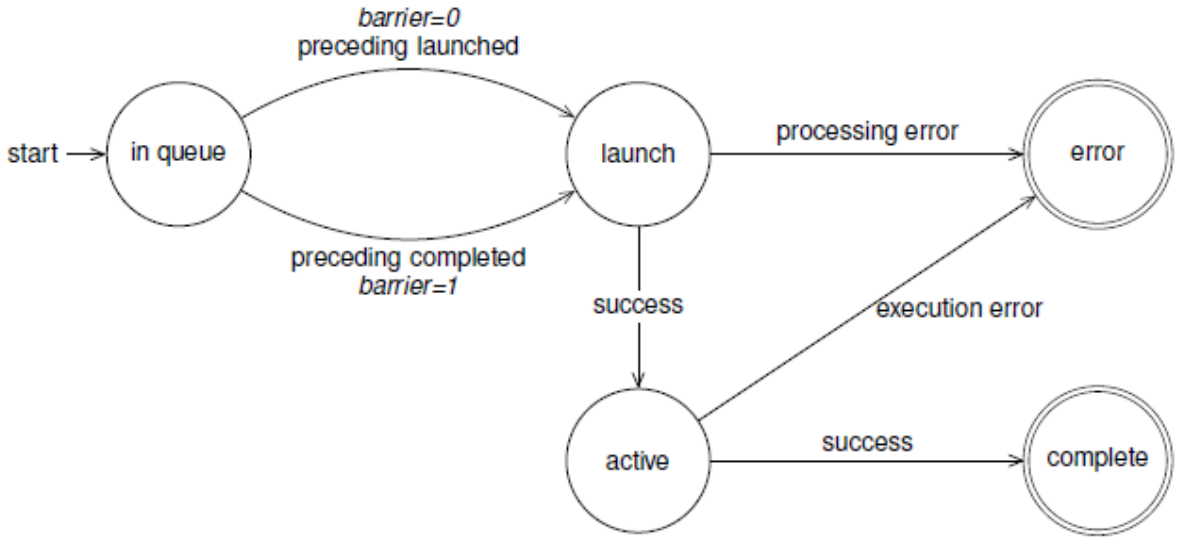
HSA_STATUS_ERROR_EXCEPTION An HSAIL exception has been triggered during the execution of a kernel dispatch packet. For example, a floating point operation has resulted in an overflow.

If no error is detected, the transition to the complete state happens when the associated task finishes (in the case of kernel dispatch and agent dispatch packets), or when the dependencies are satisfied (in the case of a barrier-AND and barrier-OR packets).

Complete A memory release fence is applied with the scope indicated by the release fence scope field in the header, 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.

Figure 2–1 Packet State Diagram



2.6.5 Architected Queuing Language Packets API

2.6.5.1 hsa_packet_type_t

Packet type.

Signature

```
typedef enum {
    HSA_PACKET_TYPE_VENDOR_SPECIFIC = 0,
    HSA_PACKET_TYPE_INVALID = 1,
    HSA_PACKET_TYPE_KERNEL_DISPATCH = 2,
    HSA_PACKET_TYPE_BARRIER_AND = 3,
    HSA_PACKET_TYPE_AGENT_DISPATCH = 4,
    HSA_PACKET_TYPE_BARRIER_OR = 5
} hsa_packet_type_t;
```

Values

HSA_PACKET_TYPE_VENDOR_SPECIFIC
: Vendor-specific packet.

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

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

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

2.6.5.2 hsa_fence_scope_t

Scope of the memory fence operation associated with a packet.

Signature

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

Values

HSA_FENCE_SCOPE_NONE
No scope (no fence is applied). The packet relies on external fences to ensure visibility of memory updates.

HSA_FENCE_SCOPE_AGENT
The fence is applied with agent scope for the global segment.

HSA_FENCE_SCOPE_SYSTEM
The fence is applied across both agent and system scope for the global segment.

2.6.5.3 hsa_packet_header_t

Sub-fields of the header field that is present in any AQL packet. The offset (with respect to the address of *header*) of a sub-field is identical to its enumeration constant. The width of each sub-field is determined by the corresponding value in hsa_packet_header_width_t.The offset and the width are expressed in bits.

Signature

```
typedef enum {
    HSA_PACKET_HEADER_TYPE = 0,
    HSA_PACKET_HEADER_BARRIER = 8,
    HSA_PACKET_HEADER_ACQUIRE_FENCE_SCOPE = 9,
    HSA_PACKET_HEADER_RELEASE_FENCE_SCOPE = 11
} hsa_packet_header_t;
```

Values

HSA_PACKET_HEADER_TYPE
Packet type. The value of this sub-field must be one of hsa_packet_type_t. If the type is HSA_PACKET_TYPE_VENDOR_SPECIFIC, the packet layout is vendor-specific.

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

HSA_PACKET_HEADER_ACQUIRE_FENCE_SCOPE
Acquire fence scope. The value of this sub-field determines the scope and type of the memory fence operation applied before the packet enters the active phase. An acquire fence ensures that any subsequent global segment or image loads by any unit of execution that belongs to a dispatch that has not yet entered the active phase on any queue of the same kernel agent, sees any data previously released at the scopes specified by the acquire fence. The value of this sub-field must be one of hsa_fence_scope_t.

HSA_PACKET_HEADER_RELEASE_FENCE_SCOPE
Release fence scope, The value of this sub-field determines the scope and type of the memory fence operation applied after kernel completion but before the packet is completed. A release fence makes any global segment or image data that was stored by any unit of execution that belonged to a dispatch that has completed the active phase on any queue of the same kernel agent visible in all the scopes specified by the release fence. The value of this sub-field must be one of hsa_fence_scope_t.

2.6.5.4 hsa_packet_header_width_t

Width (in bits) of the sub-fields in hsa_packet_header_t.

Signature

```
typedef enum {
    HSA_PACKET_HEADER_WIDTH_TYPE = 8,
    HSA_PACKET_HEADER_WIDTH_BARRIER = 1,
    HSA_PACKET_HEADER_WIDTH_ACQUIRE_FENCE_SCOPE = 2,
    HSA_PACKET_HEADER_WIDTH_RELEASE_FENCE_SCOPE = 2
} hsa_packet_header_width_t;
```

2.6.5.5 hsa_kernel_dispatch_packet_setup_t

Sub-fields of the kernel dispatch packet *setup* field. The offset (with respect to the address of *setup*) of a sub-field is identical to its enumeration constant. The width of each sub-field is determined by the corresponding value in hsa_kernel_dispatch_packet_setup_width_t. The offset and the width are expressed in bits.

Signature

```
typedef enum {
    HSA_KERNEL_DISPATCH_PACKET_SETUP_DIMENSIONS = 0
} hsa_kernel_dispatch_packet_setup_t;
```

Values

HSA_KERNEL_DISPATCH_PACKET_SETUP_DIMENSIONS
: Number of dimensions of the grid. Valid values are 1, 2, or 3.

2.6.5.6 hsa_kernel_dispatch_packet_setup_width_t

Width (in bits) of the sub-fields in hsa_kernel_dispatch_packet_setup_t.

Signature

```
typedef enum {
    HSA_KERNEL_DISPATCH_PACKET_SETUP_WIDTH_DIMENSIONS = 2
} hsa_kernel_dispatch_packet_setup_width_t;
```

2.6.5.7 hsa_kernel_dispatch_packet_t

AQL kernel dispatch packet.

Signature

```
typedef struct hsa_kernel_dispatch_packet_s {
    uint16_t header;
    uint16_t setup;
    uint16_t workgroup_size_x;
    uint16_t workgroup_size_y;
    uint16_t workgroup_size_z;
    uint16_t reserved0;
    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;

#ifdef HSA_LARGE_MODEL
    void * kernarg_address;
#elif defined HSA_LITTLE_ENDIAN
    void * kernarg_address;
    uint32_t reserved1;
#else
    uint32_t reserved1;
    void * kernarg_address;
#endif
    uint64_t reserved2;
    hsa_signal_t completion_signal;
} hsa_kernel_dispatch_packet_t
```

Data Fields

header
Packet header. Used to configure multiple packet parameters such as the packet type. The parameters are described by hsa_packet_type_t.

setup
Dispatch setup parameters. Used to configure kernel dispatch parameters such as the number of dimensions in the grid. The parameters are described by hsa_kernel_dispatch_packet_setup_t.

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 the grid has 1 dimension, the only valid value is 1.

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

reserved0
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 the grid has 1 dimension, 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 the grid has 1 or 2 dimensions, 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
Opaque handle to a code object that includes an implementation-defined executable code for the kernel.

kernarg_address
Pointer to a buffer containing the kernel arguments. May be NULL.

The buffer must be allocated using **hsa_memory_allocate**, and must not be modified once the kernel dispatch packet is enqueued until the dispatch has completed execution.

reserved1
Reserved. Must be 0.

reserved2
Reserved. Must be 0.

completion_signal
Signal used to indicate completion of the job. The application can use the special signal handle 0 to indicate that no signal is used.

2.6.5.8 hsa_agent_dispatch_packet_t

Agent dispatch packet.

Signature

```
typedef struct hsa_agent_dispatch_packet_s {
    uint16_t header;
    uint16_t type;
    uint32_t reserved0;

#ifdef HSA_LARGE_MODEL
    void * return_address;
#elif defined HSA_LITTLE_ENDIAN
    void * return_address;
    uint32_t reserved1;
#else
    uint32_t reserved1;
    void * return_address;
#endif
    uint64_t arg[4];
    uint64_t reserved2;
    hsa_signal_t completion_signal;
} hsa_agent_dispatch_packet_t
```

Data Fields

header
Packet header. Used to configure multiple packet parameters such as the packet type. The parameters are described by hsa_packet_header_t.

type
Application-defined function to be performed by the destination agent.

reserved0
Reserved. Must be 0.

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

reserved1
Reserved. Must be 0.

arg
Function arguments.

reserved2
Reserved. Must be 0.

completion_signal
Signal used to indicate completion of the job. The application can use the special signal handle 0 to indicate that no signal is used.

2.6.5.9 hsa_barrier_and_packet_t

Barrier-AND packet.

Signature

```
typedef struct hsa_barrier_and_packet_s {
    uint16_t header;
    uint16_t reserved0;
    uint32_t reserved1;
    hsa_signal_t dep_signal[5];
    uint64_t reserved2;
    hsa_signal_t completion_signal;
} hsa_barrier_and_packet_t
```

Data Fields

header
Packet header. Used to configure multiple packet parameters such as the packet type. The parameters are described by hsa_packet_header_t.

reserved0
Reserved. Must be 0.

reserved1
Reserved. Must be 0.

dep_signal
Array of dependent signal objects. Signals with a handle value of 0 are allowed and are interpreted by the packet processor as satisfied dependencies.

reserved2
Reserved. Must be 0.

completion_signal
Signal used to indicate completion of the job. The application can use the special signal handle 0 to indicate that no signal is used.

2.6.5.10 hsa_barrier_or_packet_t

Barrier-OR packet.

Signature

```
typedef struct hsa_barrier_or_packet_s {
    uint16_t header;
    uint16_t reserved0;
    uint32_t reserved1;
    hsa_signal_t dep_signal[5];
    uint64_t reserved2;
    hsa_signal_t completion_signal;
} hsa_barrier_or_packet_t
```

Data Fields

header
Packet header. Used to configure multiple packet parameters such as the packet type. The parameters are described by hsa_packet_header_t.

reserved0
Reserved. Must be 0.

reserved1
Reserved. Must be 0.

dep_signal
Array of dependent signal objects. Signals with a handle value of 0 are allowed and are interpreted by the packet processor as dependencies not satisfied.

reserved2
Reserved. Must be 0.

completion_signal
Signal used to indicate completion of the job. The application can use the special signal handle 0 to indicate that no signal is used.

2.7 Memory

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.

A memory region represents a block of virtual memory with certain characteristics that is accessible by one or more agents. The region object hsa_region_t exposes properties about the block of memory such as the associated memory segment, size, and in some cases allocation characteristics.

The function **hsa_agent_iterate_regions** can be used to inspect the set of regions associated with an agent. If the application can allocate memory in a region using the function **hsa_memory_allocate**, the flag HSA_REGION_INFO_RUNTIME_ALLOC_ALLOWED is set for that region. The HSA runtime allocator can only be used to allocate memory in the global and readonly segments. Memory in the private, group and kernarg segments is automatically allocated when a kernel dispatch packet is launched.

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

2.7.1 Global memory

Regions associated with the global segment are divided into two broad categories: fine-grained and coarse-grained. The main difference between these memory types is that fine-grained memory is directly accessible to all the agents in the system at the same time (under the terms of the HSA memory model), while coarse-grained memory may be accessible to multiple agents, but never at the same time: the application is responsible for explicitly assigning ownership of a buffer to a specific agent. In addition to this, the application can only use memory allocated from a fine-grained region in order to pass arguments to a kernel, but not all fine-grained regions can be used for this purpose.

Implementations of the HSA runtime are required to report at least the following fine-grained regions on every HSA system:

- A fine-grained region that is located in the global segment and corresponds to the coherent, primary HSA memory type (see the *HSA Platform System Architecture Specification Version 1.0*). The value of the attribute HSA_REGION_INFO_SEGMENT in this region is HSA_REGION_SEGMENT_GLOBAL and the HSA_REGION_GLOBAL_FLAG_FINE_GRAINED flag must be set.
- If the HSA system exposes at least one kernel agent, a fine-grained region that is located in the global segment and can be used to allocate backing storage for the kernarg segment: HSA_REGION_GLOBAL_FLAG_KERNARG is true, and HSA_REGION_INFO_RUNTIME_ALLOC_ALLOWED is true.

Memory allocated outside of the HSA API (for example, using malloc) is considered fine-grained only for those agents in the system that support the Full profile, but cannot be used to pass arguments to a kernel. In agents that only support the Base profile, fine-grained semantics are constrained to buffers allocated using **hsa_memory_allocate**.

If a buffer allocated outside of the HSA API is accessed by a kernel agent that supports the Full profile, the application is encouraged to register the corresponding address range beforehand using the **hsa_memory_register** function. While kernels running on kernel agents with Full profile support can access any regular host pointer, a registered buffer can 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**.

Coarse-grained regions are visible to one or more agents. The application can determine that a region supports coarse-grained semantics because the value of the attribute HSA_REGION_INFO_SEGMENT is HSA_REGION_SEGMENT_GLOBAL and the HSA_REGION_GLOBAL_FLAG_COARSE_GRAINED flag is set. If the same region handle is accessible to several agents, the application can explicitly transfer the ownership of buffers allocated in that region to any of those agents, but only one owner is allowed at a time. The HSA runtime exposes the function **hsa_memory_assign_agent** to assign ownership of a buffer to an agent. It is important to note that:

- The ownership change affects a buffer within a region, and not the entire region. Different buffers within the same coarse-grained region can have different owners.
- If the new owner cannot access the region associated with the buffer, the behavior is undefined.
- Ownership change is a no-op for fine-grained buffers.

When a coarse-grained region is visible to a unique agent (i.e., the region is only reported by **hsa_agent_iterate_regions** for that agent), the application can only assign ownership of memory within the region to that same agent. This particular case of coarse-grained memory is also known as agent allocation (see the *HSA Programmer's Reference Manual Version 1.0*). An application can still access the contents of an agent allocation buffer by invoking the synchronous copy function (**hsa_memory_copy**).

2.7.1.1 Example: passing arguments to a kernel

In the kernel setup example listed in 2.6.1. Kernel dispatch packet (page 53), the kernel receives no arguments:

```
packet->kernarg_address = NULL;
```

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 kernel 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 size of a signal handle.

```
hsa_region_t region;
hsa_agent_iterate_regions(kernel_agent, get_kernarg, &region);

// Allocate a buffer where to place the kernel arguments.
hsa_memory_allocate(region, sizeof(hsa_signal_t), (void**) &packet->kernarg_address);

// Place the signal the argument buffer
hsa_signal_t* buffer = (hsa_signal_t*) packet->kernarg_address;
assert(buffer != NULL);
hsa_signal_t signal;
hsa_signal_create(128, 1, &kernel_agent, &signal);
*buffer = signal;
```

The definition of *get_kernarg* is:

```
hsa_status_t get_kernarg(hsa_region_t region, void* data) {
    hsa_region_segment_t segment;
    hsa_region_get_info(region, HSA_REGION_INFO_SEGMENT, &segment);
    if (segment != HSA_REGION_SEGMENT_GLOBAL) {
        return HSA_STATUS_SUCCESS;
    }
    hsa_region_global_flag_t flags;
    hsa_region_get_info(region, HSA_REGION_INFO_GLOBAL_FLAGS, &flags);
    if (flags & HSA_REGION_GLOBAL_FLAG_KERNARG) {
        hsa_region_t* ret = (hsa_region_t*) data;
        *ret = region;
        return HSA_STATUS_INFO_BREAK;
    }
    return HSA_STATUS_SUCCESS;
}
```

The rest of the dispatch process remains the same.

2.7.2 Readonly memory

The application can allocate memory in a readonly region in order to store information that remains constant during the execution of a kernel. Kernel agents are only permitted to perform read operations on the addresses of variables that reside in readonly memory. The contents of a readonly buffer can be initialized or changed from one kernel dispatch execution to another by the application using the copy function (**hsa_memory_copy**).

Each kernel agent exposes one or more readonly regions, which are private to that kernel agent. Passing a readonly buffer associated with one agent in a kernel dispatch packet that is executed to a different agent results in undefined behavior.

Accesses to readonly buffers might perform better than accesses to global buffers on some HSA implementations. All readonly memory is persistent across the lifetime of an application.

2.7.3 Group and Private memory

Memory in the group segment is used to store information that is shared by all the work-items in a work-group. Group memory is visible to the work-items of a single work-group of a kernel dispatch. An address of a variable in group memory can be read and written by any work-item in the work-group with which it is associated, but not by work-items in other work-groups or by other agents. Group memory is persistent across the execution of the work-items in the work-group of the kernel dispatch with which it is associated, and it is uninitialized when the work-group starts execution.

Memory in the private segment is used to store information local to a single work-item. Private memory is visible only to a single work-item of a kernel dispatch. An address of a variable in private memory can be read and written only by the work-item with which it is associated, but not by any other work-items or other agents. Private memory is persistent across the execution of the work-item with which it is associated, and it is uninitialized when the work-item starts.

Memory in the group and private segments is represented in the HSA runtime API in a similar fashion to memory in the global and readonly segments: using regions. Each kernel agent exposes a group and a private regions. However, the application is not allowed to explicitly allocate memory in these regions using **hsa_memory_allocate**, nor it can copy any contents into them using **hsa_memory_copy**. On the other hand, the application must specify the amount of group and private memory that needs to be allocated for a particular execution of a kernel, by populating the *group_segment_size* and *private_segment_size* fields of the kernel dispatch packet.

The actual allocation of group and private memory happens automatically, before a kernel starts execution. The application must ensure that the request amount of group memory per work-group does not exceed the maximum allocation size declared by the kernel agent where the kernel dispatch packet is enqueued, which is the value of the HSA_REGION_INFO_ALLOC_MAX_SIZE attribute in the group region associated with that kernel agent. Similarly, the private memory usage of a kernel dispatch packet must not exceed the value of HSA_REGION_INFO_ALLOC_MAX_SIZE for the corresponding private region.

2.7.4 Memory API

2.7.4.1 hsa_region_t

A memory region represents a block of virtual memory with certain properties. For example, the HSA runtime represents fine-grained memory in the global segment using a region. A region might be associated with more than one agent.

Signature

```
typedef struct hsa_region_s {
    uint64_t handle;
} hsa_region_t
```

Data Fields

handle
Opaque handle.

2.7.4.2 hsa_region_segment_t

Memory segments associated with a region.

Signature

```
typedef enum {
    HSA_REGION_SEGMENT_GLOBAL = 0,
    HSA_REGION_SEGMENT_READONLY = 1,
    HSA_REGION_SEGMENT_PRIVATE = 2,
    HSA_REGION_SEGMENT_GROUP = 3
} hsa_region_segment_t;
```

Values

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

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

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

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

2.7.4.3 hsa_region_global_flag_t

Global region flags.

Signature

```
typedef enum {
    HSA_REGION_GLOBAL_FLAG_KERNARG = 1,
    HSA_REGION_GLOBAL_FLAG_FINE_GRAINED = 2,
    HSA_REGION_GLOBAL_FLAG_COARSE_GRAINED = 4
} hsa_region_global_flag_t;
```

Values

HSA_REGION_GLOBAL_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 this flag is set, then HSA_REGION_GLOBAL_FLAG_FINE_GRAINED must be set.

HSA_REGION_GLOBAL_FLAG_FINE_GRAINED
: Updates to memory in this region are immediately visible to all the agents under the terms of the HSA memory model. If this flag is set, then HSA_REGION_GLOBAL_FLAG_COARSE_GRAINED must not be set.

HSA_REGION_GLOBAL_FLAG_COARSE_GRAINED
: Updates to memory in this region can be performed by a single agent at a time. If a different agent in the system is allowed to access the region, the application must explicitly invoke **hsa_memory_assign_agent** in order to transfer ownership to that agent for a particular buffer.

2.7.4.4 hsa_region_info_t

Attributes of a memory region.

Signature

```
typedef enum {
    HSA_REGION_INFO_SEGMENT = 0,
    HSA_REGION_INFO_GLOBAL_FLAGS = 1,
    HSA_REGION_INFO_SIZE = 2,
    HSA_REGION_INFO_ALLOC_MAX_SIZE = 4,
    HSA_REGION_INFO_RUNTIME_ALLOC_ALLOWED = 5,
    HSA_REGION_INFO_RUNTIME_ALLOC_GRANULE = 6,
    HSA_REGION_INFO_RUNTIME_ALLOC_ALIGNMENT = 7
} hsa_region_info_t;
```

Values

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

HSA_REGION_INFO_GLOBAL_FLAGS
: Flag mask. The value of this attribute is undefined if the value of HSA_REGION_INFO_SEGMENT is not HSA_REGION_SEGMENT_GLOBAL. The type of this attribute is uint32_t, a bit-field of hsa_region_global_flag_t values.

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

HSA_REGION_INFO_ALLOC_MAX_SIZE
: Maximum allocation size in this region, in bytes. Must not exceed the value of HSA_REGION_INFO_SIZE. The type of this attribute is size_t.

  If the region is in the global or readonly segments, this is the maximum size that the application can pass to **hsa_memory_allocate**. If the region is in the group segment, this is the maximum size (per work-group) that can be requested for a given kernel dispatch. If the region is in the private segment, this is the maximum size (per work-item) that can be request for a specific kernel dispatch.

HSA_REGION_INFO_RUNTIME_ALLOC_ALLOWED
: Indicates whether memory in this region can be allocated using **hsa_memory_allocate**. The type of this attribute is bool.

  The value of this flag is always false for regions in the group and private segments.

HSA_REGION_INFO_RUNTIME_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. The value of this attribute is only defined if HSA_REGION_INFO_RUNTIME_ALLOC_ALLOWED is true for this region. The type of this attribute is size_t.

HSA_REGION_INFO_RUNTIME_ALLOC_ALIGNMENT
Alignment of buffers allocated by **hsa_memory_allocate** in this region. The value of this attribute is only defined if HSA_REGION_INFO_RUNTIME_ALLOC_ALLOWED is true for this region, and must be a power of 2. The type of this attribute is size_t.

2.7.4.5 hsa_region_get_info

Get the current value of an attribute of a region.

Signature

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

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 HSA runtime has not been initialized.

HSA_STATUS_ERROR_INVALID_REGION
The region is invalid.

HSA_STATUS_ERROR_INVALID_ARGUMENT
attribute is an invalid region attribute, or *value* is NULL.

2.7.4.6 hsa_agent_iterate_regions

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

Signature

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

Parameters

agent
(in) A valid agent.

callback
(in) Callback to be invoked once per region that is accessible from the agent. The HSA 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. May be NULL.

Return Values

HSA_STATUS_SUCCESS
The function has been executed successfully.

HSA_STATUS_ERROR_NOT_INITIALIZED
The HSA runtime has not been initialized.

HSA_STATUS_ERROR_INVALID_AGENT
The agent is invalid.

HSA_STATUS_ERROR_INVALID_ARGUMENT
callback is NULL.

2.7.4.7 hsa_memory_allocate

Allocate a block of memory in a given region.

Signature

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

Parameters

region
(in) Region where to allocate memory from. The region must have the HSA_REGION_INFO_RUNTIME_ALLOC_ALLOWED flag set.

size
(in) Allocation size, in bytes. Must not be zero. This value is rounded up to the nearest multiple of HSA_REGION_INFO_RUNTIME_ALLOC_GRANULE in *region*.

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_RUNTIME_ALLOC_ALIGNMENT in region. If the allocation fails, the returned value is undefined.

Return Values

HSA_STATUS_SUCCESS
The function has been executed successfully.

HSA_STATUS_ERROR_NOT_INITIALIZED
The HSA runtime has not been initialized.

HSA_STATUS_ERROR_OUT_OF_RESOURCES
No memory is available.

HSA_STATUS_ERROR_INVALID_REGION
The region is invalid.

HSA_STATUS_ERROR_INVALID_ALLOCATION
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
ptr is NULL, or *size* is 0.

2.7.4.8 hsa_memory_free

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

Signature

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

Parameters

ptr
(in) Pointer to a memory block. If *ptr* does not match a value previously returned by **hsa_memory_allocate**, the behavior is undefined.

Return Values

HSA_STATUS_SUCCESS
The function has been executed successfully.

HSA_STATUS_ERROR_NOT_INITIALIZED
The HSA runtime has not been initialized.

2.7.4.9 hsa_memory_copy

Copy a block of memory.

Signature

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

Parameters

dst
(out) Buffer where the content is to be copied.

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

size
(in) Number of bytes to copy. If *size* is 0, no copy is performed and the function returns success. Copying a number of bytes larger than the size of the buffers pointed by *dst* or *src* results in undefined behavior.

Return Values

HSA_STATUS_SUCCESS
The function has been executed successfully.

HSA_STATUS_ERROR_NOT_INITIALIZED
The HSA runtime has not been initialized.

HSA_STATUS_ERROR_INVALID_ARGUMENT
The source or destination pointers is NULL.

2.7.4.10 hsa_memory_assign_agent

Change the ownership of a global, coarse-grained buffer.

Signature

```
hsa_status_t hsa_memory_assign_agent(
    void *ptr,
    hsa_agent_t agent,
    hsa_access_permission_t access);
```

Parameters

ptr
(in) Base address of a global buffer. The pointer should match an address previously returned by **hsa_memory_allocate**. The size of the buffer affected by the ownership change is identical to the size of that previous allocation. If *ptr* points to a fine-grained global buffer, no operation is performed and the function returns success. If *ptr* does not point to global memory, the behavior is undefined.

agent
(in) Agent that becomes the owner of the buffer. The application is responsible for ensuring that *agent* has access to the region that contains the buffer. It is allowed to change ownership to an agent that is already the owner of the buffer, with the same or different access permissions.

access
(in) Access permissions requested for the new owner.

Return Values

HSA_STATUS_SUCCESS
The function has been executed successfully.

HSA_STATUS_ERROR_NOT_INITIALIZED
The HSA runtime has not been initialized.

HSA_STATUS_ERROR_INVALID_AGENT
The agent is invalid.

HSA_STATUS_ERROR_OUT_OF_RESOURCES
The HSA runtime is unable to acquire the resources required by the operation.

HSA_STATUS_ERROR_INVALID_ARGUMENT

ptr is NULL, or *access* is not a valid access value.

Description

The contents of a coarse-grained buffer are visible to an agent only after ownership has been explicitely transferred to that agent. Once the operation completes, the previous owner cannot longer access the data in the buffer.

An implementation of the HSA runtime is allowed, but not required, to change the physical location of the buffer when ownership is transferred to a different agent. In general the application must not assume this behavior. The virtual location (address) of the passed buffer is never modified.

2.7.4.11 hsa_memory_register

Register a global, fine-grained buffer.

Signature

```
hsa_status_t hsa_memory_register(
    void *ptr,
    size_t size);
```

Parameters

ptr
(in) A buffer in global memory. If a NULL pointer is passed, no operation is performed.

size
(in) Requested registration size in bytes. A size of 0 is only allowed if *ptr* is NULL.

Return Values

HSA_STATUS_SUCCESS
The function has been executed successfully.

HSA_STATUS_ERROR_NOT_INITIALIZED
The HSA runtime has not been initialized.

HSA_STATUS_ERROR_OUT_OF_RESOURCES
There is a failure in allocating the necessary resources.

HSA_STATUS_ERROR_INVALID_ARGUMENT
size is 0 but *ptr* is not NULL.

Description

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

Registration is only recommended for buffers in the global segment that have not been allocated using the HSA allocator (**hsa_memory_allocate**), but an OS allocator instead.

Registrations should not overlap.

2.7.4.12 hsa_memory_deregister

Deregister memory previously registered using **hsa_memory_register**.

Signature

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

Parameters

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

size
(in) Size of the buffer to be deregistered.

Return Values

HSA_STATUS_SUCCESS
The function has been executed successfully.

HSA_STATUS_ERROR_NOT_INITIALIZED
The HSA 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 Code Objects and Executables

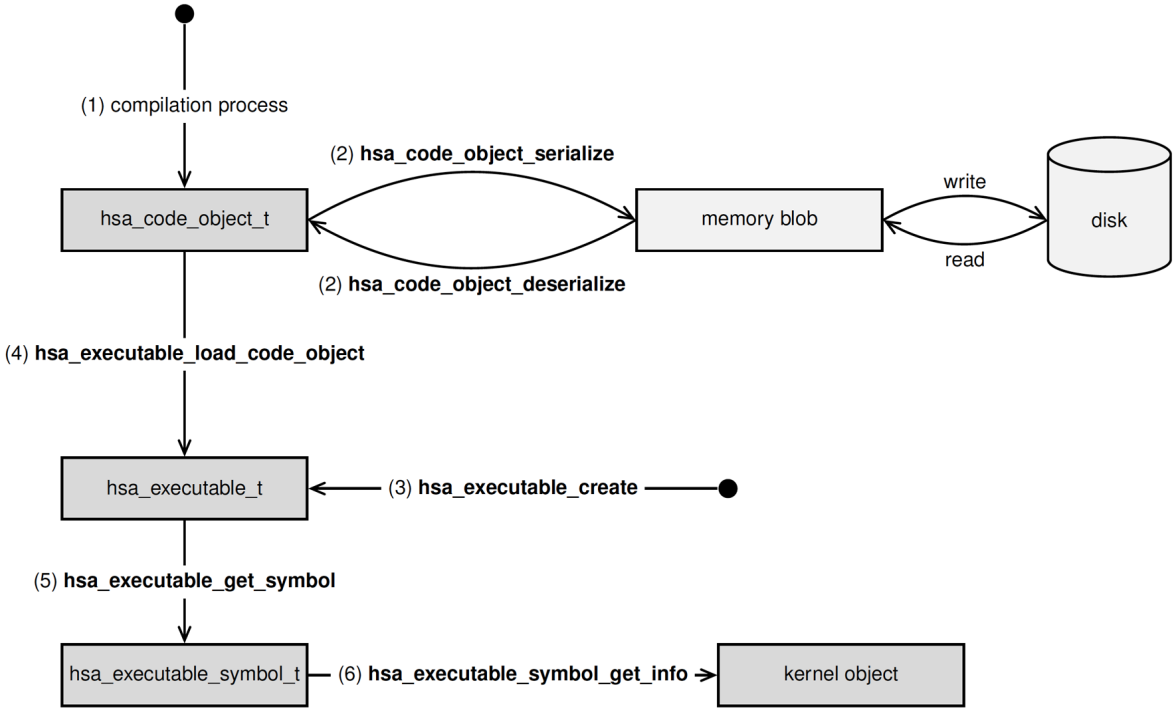
When the application populates a kernel dispatch packet, it must specify a kernel object, a handle to the machine code to be executed. Kernel object handles are created by first compiling the kernel source down to an HSA runtime, ISA-specific representation called *code object*, and then loading the code object into another HSA runtime object, the *executable*. The application can performs queries on the executable object using the HSA runtime API in order to obtain the kernel object.

The following algorithm is one example of the stages that could be passed through to generate a kernel object and assumes use of the HSAIL finalization extension described in 3.2. HSAIL Finalization (page 110). Other methods for producing a code object are valid but are not directly described in this document.

1. The application is in possession of one or more HSAIL modules that are the result of compiling a high-level language such as OpenMP or OpenCL. One of the HSAIL modules contains the kernel to be executed. Manipulation of BRIG is out of the scope of the HSA runtime API, so the application must resort to external libraries in order to locate a kernel in an HSAIL module. Using the HSA runtime extension described in 3.2. HSAIL Finalization (page 110), the application finalizes the HSAIL program, which results on a code descriptor handle of type hsa_code_object_t.
2. Code object handles are ISA-specific representations that contain the code for a set of kernels and indirect functions. The application can obtain a code object not only by finalizing an HSAIL program, but also by compiling a vendor-specific intermediate representation. Code objects can be serialized (**hsa_code_object_serialize**) and deserialized (**hsa_code_object_deserialize**), which enables the application to perform offline compilation. Each code object is associated with multiple symbols (hsa_code_symbol_t) which are the representation of a variable, kernel, or indirect function in the original source program. The application cannot use the kernel symbols contained in a code object to populate the *kernel_object* field in the kernel dispatch packet. Instead, the application must use an executable kernel symbol, which is part of an executable.
3. The application creates an executable handle using **hsa_executable_create**. The HSA runtime uses the executable handle hsa_executable_t to load a set of code object handles, possibly associated with different instruction set architectures.
4. The code object is added (and loaded) to the executable by invoking **hsa_executable_load_code_object**.
5. The application queries the executable in order to retrieve the executable symbol corresponding to the given kernel and agent. This can be achieved by passing the kernel's name, the kernel's module name, and the agent to **hsa_executable_get_symbol**.
6. Executable symbols that represent kernels expose the attribute HSA_EXECUTABLE_SYMBOL_INFO_KERNEL_OBJECT, which is a handle to the machine code that is ultimately used to launch a kernel. The application uses the getter function **hsa_executable_symbol_get_info** to retrieve the value of executable symbol attributes.
7. The application uses the kernel object value to populate the *kernel_object* field of a kernel dispatch packet. The application may use some other information in the symbol (for example, the kernel's memory usage) to fill other fields of the kernel dispatch packet.

This basic workflow is represented in Figure 2–2 (facing page). The functions write and read represent operating system calls to write and read a number of bytes to a file.

Figure 2–2 Retrieving a kernel object handle from a code object



2.8.1 Code Objects and Executables API

2.8.1.1 hsa_symbol_kind_t

Symbol type.

Signature

```
typedef enum {
    HSA_SYMBOL_KIND_VARIABLE = 0,
    HSA_SYMBOL_KIND_KERNEL = 1,
    HSA_SYMBOL_KIND_INDIRECT_FUNCTION = 2
} hsa_symbol_kind_t;
```

Values

HSA_SYMBOL_KIND_VARIABLE
Variable.

HSA_SYMBOL_KIND_KERNEL
Kernel.

HSA_SYMBOL_KIND_INDIRECT_FUNCTION
Indirect function.

2.8.1.2 hsa_variable_allocation_t

Allocation type of a variable.

Signature

```
typedef enum {
    HSA_VARIABLE_ALLOCATION_AGENT = 0,
    HSA_VARIABLE_ALLOCATION_PROGRAM = 1
} hsa_variable_allocation_t;
```

Values

HSA_VARIABLE_ALLOCATION_AGENT
Agent allocation.

HSA_VARIABLE_ALLOCATION_PROGRAM
Program allocation.

2.8.1.3 hsa_symbol_kind_linkage_t

Linkage type of a symbol.

Signature

```
typedef enum {
    HSA_SYMBOL_KIND_LINKAGE_MODULE = 0,
    HSA_SYMBOL_KIND_LINKAGE_PROGRAM = 1,
} hsa_symbol_kind_linkage_t;
```

Values

HSA_SYMBOL_KIND_LINKAGE_MODULE
Module linkage.

HSA_SYMBOL_KIND_LINKAGE_PROGRAM
Program linkage.

2.8.1.4 hsa_variable_segment_t

Memory segment associated with a variable.

Signature

```
typedef enum {
    HSA_VARIABLE_SEGMENT_GLOBAL = 0,
    HSA_VARIABLE_SEGMENT_READONLY = 1
} hsa_variable_segment_t;
```

Values

HSA_VARIABLE_SEGMENT_GLOBAL
Global memory segment.

HSA_VARIABLE_SEGMENT_READONLY
Readonly memory segment.

2.8.1.5 hsa_isa_t

Instruction set architecture.

Signature

```
typedef struct hsa_isa_s {
    uint64_t handle;
} hsa_isa_t
```

Data Fields

handle
Opaque handle.

2.8.1.6 hsa_isa_from_name

Retrieve a reference to an ISA handle out of a symbolic name.

Signature

```
hsa_status_t hsa_isa_from_name(
    const char *name,
    hsa_isa_t *isa);
```

Parameters

name
(in) Vendor-specific name associated with a particular instruction set architecture. Must be a NUL-terminated string.

isa
(out) Memory location where the HSA runtime stores the ISA handle corresponding to the given name. Must not be NULL.

Return Values

HSA_STATUS_SUCCESS
The function has been executed successfully.

HSA_STATUS_ERROR_INVALID_ARGUMENT
name is NULL, or *isa* is NULL.

HSA_STATUS_ERROR_INVALID_ISA_NAME
The given name does not correspond to any instruction set architecture.

2.8.1.7 hsa_isa_info_t

Instruction set architecture attributes.

Signature

```
typedef enum {
    HSA_ISA_INFO_NAME_LENGTH = 0,
    HSA_ISA_INFO_NAME = 1,
    HSA_ISA_INFO_CALL_CONVENTION_COUNT = 2,
    HSA_ISA_INFO_CALL_CONVENTION_INFO_WAVEFRONT_SIZE = 3,
    HSA_ISA_INFO_CALL_CONVENTION_INFO_WAVEFRONTS_PER_COMPUTE_UNIT = 4
} hsa_isa_info_t;
```

Values

HSA_ISA_INFO_NAME_LENGTH
: The length of the ISA name. The type of this attribute is uint32_t.

HSA_ISA_INFO_NAME
: Human-readable description. The type of this attribute is character array with the length equal to the value of HSA_ISA_INFO_NAME_LENGTH attribute.

HSA_ISA_INFO_CALL_CONVENTION_COUNT
: Number of call conventions supported by the instruction set architecture. The type of this attribute is uint32_t.

HSA_ISA_INFO_CALL_CONVENTION_INFO_WAVEFRONT_SIZE
: Number of work-items in a wavefront for a given call convention. Must be a power of 2 in the range [1,256]. The type of this attribute is uint32_t.

HSA_ISA_INFO_CALL_CONVENTION_INFO_WAVEFRONTS_PER_COMPUTE_UNIT
: Number of wavefronts per compute unit for a given call convention. In practice, other factors (for example, the amount of group memory used by a work-group) may further limit the number of wavefronts per compute unit. The type of this attribute is uint32_t.

2.8.1.8 hsa_isa_get_info

Get the current value of an attribute for a given instruction set architecture (ISA).

Signature

```
hsa_status_t hsa_isa_get_info(
    hsa_isa_t isa,
    hsa_isa_info_t attribute,
    uint32_t index,
    void *value);
```

Parameters

isa
: (in) A valid instruction set architecture.

attribute
: (in) Attribute to query.

index
: (in) Call convention index. Used only for call convention attributes, otherwise ignored. Must have a value between 0 (inclusive) and the value of the attribute HSA_ISA_INFO_CALL_CONVENTION_COUNT (not inclusive) in *isa*.

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 HSA runtime has not been initialized.

HSA_STATUS_ERROR_INVALID_ISA
The instruction set architecture is invalid.

HSA_STATUS_ERROR_INVALID_INDEX
index out of range.

HSA_STATUS_ERROR_INVALID_ARGUMENT
attribute is an invalid instruction set architecture attribute, or *value* is NULL.

2.8.1.9 hsa_isa_compatible

Check if the instruction set architecture of a code object can be executed on an agent associated with another architecture.

Signature

```
hsa_status_t hsa_isa_compatible(
    hsa_isa_t code_object_isa,
    hsa_isa_t agent_isa,
    bool *result );
```

Parameters

code_object_isa
(in) Instruction set architecture associated with a code object.

agent_isa
(in) Instruction set architecture associated with an agent.

result
(out) Pointer to a memory location where the HSA runtime stores the result of the check. If the two architectures are compatible, the result is true; if they are incompatible, the result is false.

Return Values

HSA_STATUS_SUCCESS
The function has been executed successfully.

HSA_STATUS_ERROR_INVALID_ISA
*code_object*_isa or *agent_isa* is invalid.

HSA_STATUS_ERROR_INVALID_ARGUMENT
result is NULL.

2.8.1.10 hsa_code_object_t

An opaque handle to a code object, which contains executable code for finalized kernels and indirect functions together with information about the global/readonly segment variables they reference.

Signature

```
typedef struct hsa_code_object_s {
    uint64_t handle;
} hsa_code_object_t
```

Data Fields

handle
Opaque handle.

2.8.1.11 hsa_callback_data_t

Opaque handle to application data that is passed to the serialization and deserialization functions.

Signature

```
typedef struct hsa_callback_data_s {
    uint64_t handle;
} hsa_callback_data_t
```

Data Fields

handle
Opaque handle.

2.8.1.12 hsa_code_object_serialize

Serialize a code object. Can be used for offline finalization, install-time finalization, disk code caching, etc.

Signature

```
hsa_status_t hsa_code_object_serialize(
    hsa_code_object_t code_object,
    hsa_status_t (*alloc_callback)(size_t size, hsa_callback_data_t data, void **address),
    hsa_callback_data_t callback_data,
    const char *options,
    void **serialized_code_object,
    size_t *serialized_code_object_size);
```

Parameters

code_object
(in) Code object.

alloc_callback
(in) Callback function for memory allocation. Must not be NULL. The HSA runtime passes three arguments to the callback: the allocation size, the application data, and a pointer to a memory location where the application stores the allocation result. The HSA runtime invokes *alloc_callback* once to allocate a buffer that contains the serialized version of *code_object*. If the callback returns a status code other than HSA_STATUS_SUCCESS this function returns the same code.

callback_data
(in) Application data that is passed to *alloc_callback*. May be NULL.

options
(in) Vendor-specific options. May be NULL.

serialized_code_object
(out) Memory location where the HSA runtime stores a pointer to the serialized code object. Must not be NULL.

serialized_code_object_size
(out) Memory location where the HSA runtime stores the size (in bytes) of *serialized_code_object*. The returned value matches the allocation size passed by the HSA runtime to *alloc_callback*. Must not be NULL.

Return Values

HSA_STATUS_SUCCESS
The function has been executed successfully.

HSA_STATUS_ERROR_NOT_INITIALIZED
The HSA runtime has not been initialized.

HSA_STATUS_ERROR_OUT_OF_RESOURCES
There is a failure to allocate resources required for the operation.

HSA_STATUS_ERROR_INVALID_CODE_OBJECT
code_object is invalid.

HSA_STATUS_ERROR_INVALID_ARGUMENT
alloc_callback, *serialized_code_object*, or *serialized_code_object_size* is NULL.

2.8.1.13 hsa_code_object_deserialize

Deserialize a code object.

Signature

```
hsa_status_t hsa_code_object_deserialize(
    void *serialized_code_object,
    size_t serialized_code_object_size,
    const char *options,
    hsa_code_object_t *code_object );
```

Parameters

serialized_code_object
(in) A serialized code object. Must not be NULL.

serialized_code_object_size
(in) The size (in bytes) of *serialized_code_object*. Must not be 0.

options
(in) Vendor-specific options. May be NULL.

code_object
(out) Memory location where the HSA runtime stores the deserialized code object.

Return Values

HSA_STATUS_SUCCESS
The function has been executed successfully.

HSA_STATUS_ERROR_NOT_INITIALIZED
The HSA runtime has not been initialized.

HSA_STATUS_ERROR_OUT_OF_RESOURCES
There is a failure to allocate resources required for the operation.

HSA_STATUS_ERROR_INVALID_ARGUMENT
serialized_code_object, or *code_object* is NULL, or *serialized_code_object_size* is 0.

2.8.1.14 hsa_code_object_destroy

Destroy a code object.

Signature

```
hsa_status_t hsa_code_object_destroy(
    hsa_code_object_t code_object );
```

Parameters

code_object
(in) Code object. The handle becomes invalid after it has been destroyed.

Return Values

HSA_STATUS_SUCCESS
The function has been executed successfully.

HSA_STATUS_ERROR_NOT_INITIALIZED
The HSA runtime has not been initialized.

HSA_STATUS_ERROR_INVALID_CODE_OBJECT
code_object is invalid.

Description

The lifetime of a code object must exceed that of any executable where it has been loaded. If an executable that loaded *code_object* has not been destroyed, the behavior is undefined.

2.8.1.15 hsa_code_object_type_t

Code object type.

Signature

```
typedef enum {
    HSA_CODE_OBJECT_TYPE_PROGRAM = 0
} hsa_code_object_type_t;
```

Values

HSA_CODE_OBJECT_TYPE_PROGRAM
: Produces code object that contains ISA for all kernels and indirect functions in HSA source.

2.8.1.16 hsa_code_object_info_t

Code object attributes.

Signature

```
typedef enum {
    HSA_CODE_OBJECT_INFO_VERSION = 0,
    HSA_CODE_OBJECT_INFO_TYPE = 1,
    HSA_CODE_OBJECT_INFO_ISA = 2,
    HSA_CODE_OBJECT_INFO_MACHINE_MODEL = 3,
    HSA_CODE_OBJECT_INFO_PROFILE = 4,
    HSA_CODE_OBJECT_INFO_DEFAULT_FLOAT_ROUNDING_MODE = 5
} hsa_code_object_info_t;
```

Values

HSA_CODE_OBJECT_INFO_VERSION
: The version of the code object. The type of this attribute is a NUL-terminated char[64]. If the version of the code object uses fewer than 63 characters, the rest of the array must be filled with NULs.

HSA_CODE_OBJECT_INFO_TYPE
: Type of code object. The type of this attribute is hsa_code_object_type_t.

HSA_CODE_OBJECT_INFO_ISA
: Instruction set architecture this code object is produced for. The type of this attribute is hsa_isa_t.

HSA_CODE_OBJECT_INFO_MACHINE_MODEL
: Machine model this code object is produced for. The type of this attribute is hsa_machine_model_t.

HSA_CODE_OBJECT_INFO_PROFILE
: Profile this code object is produced for. The type of this attribute is hsa_profile_t.

HSA_CODE_OBJECT_INFO_DEFAULT_FLOAT_ROUNDING_MODE
: Default floating-point rounding mode used when the code object is produced. The type of this attribute is hsa_default_float_rounding_mode_t.

2.8.1.17 hsa_code_object_get_info

Get the current value of an attribute for a given code object.

Signature

```
hsa_status_t hsa_code_object_get_info(
    hsa_code_object_t code_object,
    hsa_code_object_info_t attribute,
    void *value);
```

Parameters

code_object
(in) Code object.

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 HSA runtime has not been initialized.

HSA_STATUS_ERROR_INVALID_ARGUMENT
attribute is an invalid code object attribute, or *value* is NULL.

HSA_STATUS_ERROR_INVALID_CODE_OBJECT
code_object is invalid.

2.8.1.18 hsa_code_symbol_t

Code object symbol.

Signature

```
typedef struct hsa_code_symbol_s {
    uint64_t handle;
} hsa_code_symbol_t
```

Data Fields

handle
Opaque handle.

2.8.1.19 hsa_code_object_get_symbol

Get the symbol handle within a code object for a given a symbol name.

Signature

```
hsa_status_t hsa_code_object_get_symbol(
    hsa_code_object_t code_object,
    const char *symbol_name,
    hsa_code_symbol_t *symbol);
```

Parameters

code_object
(in) Code object.

symbol_name
(in) Symbol name.

symbol
(out) Memory location where the HSA runtime stores the symbol handle.

Return Values

HSA_STATUS_SUCCESS
The function has been executed successfully.

HSA_STATUS_ERROR_NOT_INITIALIZED
The HSA runtime has not been initialized.

HSA_STATUS_ERROR_INVALID_CODE_OBJECT
code_object is invalid.

HSA_STATUS_ERROR_INVALID_SYMBOL_NAME
There is no symbol with a name that matches *symbol_name*.

HSA_STATUS_ERROR_INVALID_ARGUMENT
symbol_name is NULL, or *symbol* is NULL.

2.8.1.20 hsa_code_symbol_info_t

Code object symbol attributes.

Signature

```
typedef enum {
    HSA_CODE_SYMBOL_INFO_TYPE = 0,
    HSA_CODE_SYMBOL_INFO_NAME_LENGTH = 1,
    HSA_CODE_SYMBOL_INFO_NAME = 2,
    HSA_CODE_SYMBOL_INFO_MODULE_NAME_LENGTH = 3,
    HSA_CODE_SYMBOL_INFO_MODULE_NAME = 4,
    HSA_CODE_SYMBOL_INFO_LINKAGE = 5,
    HSA_CODE_SYMBOL_INFO_IS_DEFINITION = 17,
    HSA_CODE_SYMBOL_INFO_VARIABLE_ALLOCATION = 6,
    HSA_CODE_SYMBOL_INFO_VARIABLE_SEGMENT = 7,
    HSA_CODE_SYMBOL_INFO_VARIABLE_ALIGNMENT = 8,
    HSA_CODE_SYMBOL_INFO_VARIABLE_SIZE = 9,
    HSA_CODE_SYMBOL_INFO_VARIABLE_IS_CONST = 10,
    HSA_CODE_SYMBOL_INFO_KERNEL_KERNARG_SEGMENT_SIZE = 11,
    HSA_CODE_SYMBOL_INFO_KERNEL_KERNARG_SEGMENT_ALIGNMENT = 12,
    HSA_CODE_SYMBOL_INFO_KERNEL_GROUP_SEGMENT_SIZE = 13,
    HSA_CODE_SYMBOL_INFO_KERNEL_PRIVATE_SEGMENT_SIZE = 14,
    HSA_CODE_SYMBOL_INFO_KERNEL_DYNAMIC_CALLSTACK = 15,
    HSA_CODE_SYMBOL_INFO_INDIRECT_FUNCTION_CALL_CONVENTION = 16
} hsa_code_symbol_info_t;
```

Values

HSA_CODE_SYMBOL_INFO_TYPE
The type of the symbol. The type of this attribute is hsa_symbol_kind_t.

HSA_CODE_SYMBOL_INFO_NAME_LENGTH
The length of the symbol name. The type of this attribute is uint32_t.

HSA_CODE_SYMBOL_INFO_NAME
The name of the symbol. The type of this attribute is character array with the length equal to the value of HSA_CODE_SYMBOL_INFO_NAME_LENGTH attribute.

HSA_CODE_SYMBOL_INFO_MODULE_NAME_LENGTH
The length of the module name to which this symbol belongs if this symbol has module linkage, otherwise 0 is returned. The type of this attribute is uint32_t.

HSA_CODE_SYMBOL_INFO_MODULE_NAME
The module name to which this symbol belongs if this symbol has module linkage, otherwise empty string is returned. The type of this attribute is character array with the length equal to the value of HSA_CODE_SYMBOL_INFO_MODULE_NAME_LENGTH attribute.

HSA_CODE_SYMBOL_INFO_LINKAGE
The linkage kind of the symbol. The type of this attribute is hsa_symbol_kind_linkage_t.

HSA_CODE_SYMBOL_INFO_IS_DEFINITION
Indicates whether the symbol corresponds to a definition. The type of this attribute is bool.

HSA_CODE_SYMBOL_INFO_VARIABLE_ALLOCATION
The allocation kind of the variable. The value of this attribute is undefined if the symbol is not a variable. The type of this attribute is hsa_variable_allocation_t.

HSA_CODE_SYMBOL_INFO_VARIABLE_SEGMENT
The segment kind of the variable. The value of this attribute is undefined if the symbol is not a variable. The type of this attribute is hsa_variable_segment_t.

HSA_CODE_SYMBOL_INFO_VARIABLE_ALIGNMENT
Alignment of the variable. The value of this attribute is undefined if the symbol is not a variable. The type of this attribute is uint32_t.

HSA_CODE_SYMBOL_INFO_VARIABLE_SIZE
Size of the variable. The value of this attribute is undefined if the symbol is not a variable. The type of this attribute is uint32_t.

A size of 0 is returned if the variable is an external variable and has an unknown dimension.

HSA_CODE_SYMBOL_INFO_VARIABLE_IS_CONST
Indicates whether the variable is constant. The value of this attribute is undefined if the symbol is not a variable. The type of this attribute is bool.

HSA_CODE_SYMBOL_INFO_KERNEL_KERNARG_SEGMENT_SIZE
Size of kernarg segment memory that is required to hold the values of the kernel arguments, in bytes. The value of this attribute is undefined if the symbol is not a kernel. The type of this attribute is uint32_t.

HSA_CODE_SYMBOL_INFO_KERNEL_KERNARG_SEGMENT_ALIGNMENT
: Alignment (in bytes) of the buffer used to pass arguments to the kernel, which is the maximum of 16 and the maximum alignment of any of the kernel arguments. The value of this attribute is undefined if the symbol is not a kernel. The type of this attribute is uint32_t.

HSA_CODE_SYMBOL_INFO_KERNEL_GROUP_SEGMENT_SIZE
: Size of static group segment memory required by the kernel (per work-group), in bytes. The value of this attribute is undefined if the symbol is not a kernel. The type of this attribute is uint32_t.

  The reported amount does not include any dynamically allocated group segment memory that may be requested by the application when a kernel is dispatched.

HSA_CODE_SYMBOL_INFO_KERNEL_PRIVATE_SEGMENT_SIZE
: Size of static private, spill, and arg segment memory required by this kernel (per work-item), in bytes. The value of this attribute is undefined if the symbol is not a kernel. The type of this attribute is uint32_t.

  If the value of HSA_CODE_SYMBOL_INFO_KERNEL_DYNAMIC_CALLSTACK is true, the kernel may use more private memory than the reported value, and the application must add the dynamic call stack usage to *private_segment_size* when populating a kernel dispatch packet.

HSA_CODE_SYMBOL_INFO_KERNEL_DYNAMIC_CALLSTACK
: Dynamic callstack flag. The value of this attribute is undefined if the symbol is not a kernel. The type of this attribute is bool.

  If this flag is set (the value is true), the kernel uses a dynamically sized call stack. This can happen if recursive calls, calls to indirect functions, or the HSAIL alloca instruction are present in the kernel.

HSA_CODE_SYMBOL_INFO_INDIRECT_FUNCTION_CALL_CONVENTION
: Call convention of the indirect function. The value of this attribute is undefined if the symbol is not an indirect function. The type of this attribute is uint32_t.

2.8.1.21 hsa_code_symbol_get_info

Get the current value of an attribute for a given code symbol.

Signature

```
hsa_status_t hsa_code_symbol_get_info(
    hsa_code_symbol_t code_symbol,
    hsa_code_symbol_info_t attribute,
    void *value);
```

Parameters

code_symbol
: (in) Code symbol.

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 HSA runtime has not been initialized.

HSA_STATUS_ERROR_INVALID_ARGUMENT
attribute is an invalid code symbol attribute, or *value* is NULL.

2.8.1.22 hsa_code_object_iterate_symbols

Iterate over the symbols in a code object, and invoke an application-defined callback on every iteration.

Signature

```
hsa_status_t hsa_code_object_iterate_symbols(
    hsa_code_object_t code_object,
    hsa_status_t (*callback)(hsa_code_object_t code_object, hsa_code_symbol_t symbol, void *data),
    void *data);
```

Parameters

code_object
(in) Code object.

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

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

Return Values

HSA_STATUS_SUCCESS
The function has been executed successfully.

HSA_STATUS_ERROR_NOT_INITIALIZED
The HSA runtime has not been initialized.

HSA_STATUS_ERROR_INVALID_CODE_OBJECT
code_object is invalid.

HSA_STATUS_ERROR_INVALID_ARGUMENT
callback is NULL.

2.8.1.23 hsa_executable_t

An opaque handle to an executable, which contains ISA for finalized kernels and indirect functions together with the allocated global/readonly segment variables they reference.

Signature

```
typedef struct hsa_executable_s {
    uint64_t handle;
} hsa_executable_t
```

Data Fields

handle
Opaque handle.

2.8.1.24 hsa_executable_state_t

Executable state.

Signature

```
typedef enum {
    HSA_EXECUTABLE_STATE_UNFROZEN = 0,
    HSA_EXECUTABLE_STATE_FROZEN = 1
} hsa_executable_state_t;
```

Values

HSA_EXECUTABLE_STATE_UNFROZEN
Executable state, which allows the user to load code objects and define external variables. Variable addresses, kernel code handles, and indirect function code handles are not available in query operations until the executable is frozen (zero always returned).

HSA_EXECUTABLE_STATE_FROZEN
Executable state, which allows the user to query variable addresses, kernel code handles, and indirect function code handles using query operation. Loading new code objects, as well as defining external variables is not allowed in this state.

2.8.1.25 hsa_executable_create

Create an empty executable.

Signature

```
hsa_status_t hsa_executable_create(
    hsa_profile_t profile,
    hsa_executable_state_t executable_state,
    const char *options,
    hsa_executable_t *executable);
```

Parameters

profile
(in) Profile used in the executable.

executable_state
(in) Executable state. If the state is HSA_EXECUTABLE_STATE_FROZEN, the resulting executable is useless because no code objects can be loaded, and no variables can be defined.

options
(in) Vendor-specific options. May be NULL.

executable
(out) Memory location where the HSA runtime stores newly created executable handle.

Return Values

HSA_STATUS_SUCCESS
The function has been executed successfully.

HSA_STATUS_ERROR_NOT_INITIALIZED
The HSA runtime has not been initialized.

HSA_STATUS_ERROR_OUT_OF_RESOURCES
There is a failure to allocate resources required for the operation.

HSA_STATUS_ERROR_INVALID_ARGUMENT
profile is invalid, or *executable* is NULL.

2.8.1.26 hsa_executable_destroy

Destroy an executable.

Signature

```
hsa_status_t hsa_executable_destroy(
    hsa_executable_t executable);
```

Parameters

executable
(in) Executable.

Return Values

HSA_STATUS_SUCCESS
The function has been executed successfully.

HSA_STATUS_ERROR_NOT_INITIALIZED
The HSA runtime has not been initialized.

HSA_STATUS_ERROR_INVALID_EXECUTABLE
The executable is invalid.

Description

Executable handle becomes invalid after the executable has been destroyed. Code object handles that were loaded into this executable are still valid after the executable has been destroyed, and can be used as intended. Resources allocated outside and associated with this executable (such as external global/readonly variables) can be released after the executable has been destroyed.

Executable should not be destroyed while kernels are in flight.

2.8.1.27 hsa_executable_load_code_object

Load code object into the executable.

Signature

```
hsa_status_t hsa_executable_load_code_object(
    hsa_executable_t executable,
    hsa_agent_t agent,
    hsa_code_object_t code_object,
    const char *options);
```

Parameters

executable
(in) Executable.

agent
(in) Agent to load code object for. The agent must support the default floating-point rounding mode used by *code_object*.

code_object
(in) Code object to load. The lifetime of the code object must exceed that of the executable. If *code_object* is destroyed before *executable*, the behavior is undefined.

options
(in) Vendor-specific options. May be NULL.

Return Values

HSA_STATUS_SUCCESS
The function has been executed successfully.

HSA_STATUS_ERROR_NOT_INITIALIZED
The HSA runtime has not been initialized.

HSA_STATUS_ERROR_OUT_OF_RESOURCES
There is a failure to allocate resources required for the operation.

HSA_STATUS_ERROR_INVALID_EXECUTABLE
The executable is invalid.

HSA_STATUS_ERROR_INVALID_AGENT
The agent is invalid.

HSA_STATUS_ERROR_INVALID_CODE_OBJECT
code_object is invalid.

HSA_STATUS_ERROR_INCOMPATIBLE_ARGUMENTS
agent is not compatible with *code_object* (for example, *agent* does not support the default floating-point rounding mode specified by *code_object*), or *code_object* is not compatible with *executable* (for example, *code_object* and *executable* have different machine models or profiles).

HSA_STATUS_ERROR_FROZEN_EXECUTABLE
executable is frozen.

Description

Every global/readonly variable that is external must be defined using define set of operations before loading code objects. Internal global/readonly variable is allocated once the code object, that is being loaded, references this variable and this variable is not allocated.

Any module linkage declaration must have been defined either by a define variable or by loading a code object that has a symbol with module linkage definition.

2.8.1.28 hsa_executable_freeze

Freeze the executable.

Signature

```
hsa_status_t hsa_executable_freeze(
    hsa_extension_t executable,
    const char *options);
```

Parameters

executable
(in) Executable.

options
(in) Vendor-specific options. May be NULL.

Return Values

HSA_STATUS_SUCCESS
The function has been executed successfully.

HSA_STATUS_ERROR_NOT_INITIALIZED
The HSA runtime has not been initialized.

HSA_STATUS_ERROR_INVALID_EXECUTABLE
The executable is invalid.

HSA_STATUS_ERROR_VARIABLE_UNDEFINED
One or more variable is undefined in the executable.

HSA_STATUS_ERROR_FROZEN_EXECUTABLE
executable is already frozen.

Description

No modifications to executable can be made after freezing: no code objects can be loaded to the executable, no external variables can be defined. Freezing the executable does not prevent querying executable's attributes.

2.8.1.29 hsa_executable_info_t

Executable attributes.

Signature

```
typedef enum {
    HSA_EXECUTABLE_INFO_PROFILE = 1,
    HSA_EXECUTABLE_INFO_STATE = 2
} hsa_executable_info_t;
```

Values

HSA_EXECUTABLE_INFO_PROFILE
Profile this executable is created for. The type of this attribute is hsa_profile_t.

HSA_EXECUTABLE_INFO_STATE
Executable state. The type of this attribute is hsa_executable_state_t.

2.8.1.30 hsa_executable_get_info

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

Signature

```
hsa_status_t hsa_executable_get_info(
    hsa_executable_t executable,
    hsa_executable_info_t attribute,
    void *value);
```

Parameters

executable
(in) Executable.

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 HSA runtime has not been initialized.

HSA_STATUS_ERROR_INVALID_EXECUTABLE
The executable is invalid.

HSA_STATUS_ERROR_INVALID_ARGUMENT
attribute is an invalid executable attribute, or *value* is NULL.

2.8.1.31 hsa_executable_global_variable_define

Define an external global variable with program allocation.

Signature

```
hsa_status_t hsa_executable_global_variable_define(
    hsa_executable_t executable,
    const char *variable_name,
    void *address);
```

Parameters

executable
(in) Executable.

variable_name
(in) Name of the variable.

address
(in) Address where the variable is defined. The buffer pointed by *address* is owned by the application, and cannot be deallocated before *executable* is destroyed.

Return Values

HSA_STATUS_SUCCESS
The function has been executed successfully.

HSA_STATUS_ERROR_NOT_INITIALIZED
The HSA runtime has not been initialized.

HSA_STATUS_ERROR_OUT_OF_RESOURCES
There is a failure to allocate resources required for the operation.

HSA_STATUS_ERROR_INVALID_ARGUMENT
variable_name is NULL.

HSA_STATUS_ERROR_INVALID_EXECUTABLE
The executable is invalid.

HSA_STATUS_ERROR_VARIABLE_ALREADY_DEFINED
The variable is already defined.

HSA_STATUS_ERROR_INVALID_SYMBOL_NAME
There is no variable with the *variable_name*.

HSA_STATUS_ERROR_FROZEN_EXECUTABLE
executable is frozen.

Description

This function allows the application to provide the definition of a variable in the global segment memory with program allocation. The variable must be defined before loading a code object into an executable. In addition, code objects loaded must not define the variable.

2.8.1.32 hsa_executable_agent_global_variable_define

Define an external global variable with agent allocation.

Signature

```
hsa_status_t hsa_executable_agent_global_variable_define(
```

```
    hsa_executable_t executable,
    hsa_agent_t agent,
    const char *variable_name,
    void *address);
```

Parameters

executable
(in) Executable.

agent
(in) Agent for which the variable is being defined.

variable_name
(in) Name of the variable.

address
(in) Address where the variable is defined. The buffer pointed by address is owned by the application, and cannot be deallocated before *executable* is destroyed.

Return Values

HSA_STATUS_SUCCESS
The function has been executed successfully.

HSA_STATUS_ERROR_NOT_INITIALIZED
The HSA runtime has not been initialized.

HSA_STATUS_ERROR_OUT_OF_RESOURCES
There is a failure to allocate resources required for the operation.

HSA_STATUS_ERROR_INVALID_ARGUMENT
variable_name is NULL.

HSA_STATUS_ERROR_INVALID_EXECUTABLE
The executable is invalid.

HSA_STATUS_ERROR_INVALID_AGENT
agent is invalid.

HSA_STATUS_ERROR_VARIABLE_ALREADY_DEFINED
The variable is already defined.

HSA_STATUS_ERROR_INVALID_SYMBOL_NAME
There is no variable with the *variable_name*.

HSA_STATUS_ERROR_FROZEN_EXECUTABLE
executable is frozen.

Description

This function allows the application to provide the definition of a variable in the global segment memory with agent allocation. The variable must be defined before loading a code object into an executable. In addition, code objects loaded must not define the variable.

2.8.1.33 hsa_executable_readonly_variable_define

Define an external readonly variable.

Signature

```
hsa_status_thsa_executable_readonly_variable_define(
    hsa_executable_t executable,
    hsa_agent_t agent,
    const char *variable_name,
    void *address);
```

Parameters

executable
(in) Executable.

agent
(in) Agent for which the variable is being defined.

variable_name
(in) Name of the variable.

address
(in) Address where the variable is defined. The buffer pointed by address is owned by the application, and cannot be deallocated before *executable* is destroyed.

Return Values

HSA_STATUS_SUCCESS
The function has been executed successfully.

HSA_STATUS_ERROR_NOT_INITIALIZED
The HSA runtime has not been initialized.

HSA_STATUS_ERROR_OUT_OF_RESOURCES
There is a failure to allocate resources required for the operation.

HSA_STATUS_ERROR_INVALID_ARGUMENT
variable_name is NULL.

HSA_STATUS_ERROR_INVALID_EXECUTABLE
executable is invalid.

HSA_STATUS_ERROR_INVALID_AGENT
agent is invalid.

HSA_STATUS_ERROR_VARIABLE_ALREADY_DEFINED
The variable is already defined.

HSA_STATUS_ERROR_INVALID_SYMBOL_NAME
There is no variable with the *variable_name*.

HSA_STATUS_ERROR_FROZEN_EXECUTABLE
executable is frozen.

Description

This function allows the application to provide the definition of a variable in the readonly segment memory. The variable must be defined before loading a code object into an executable. In addition, code objects loaded must not define the variable.

2.8.1.34 hsa_executable_validate

Validate executable. Checks that all code objects have matching machine model, profile, and default floating-point rounding mode. Checks that all declarations have definitions. Checks declaration-definition compatibility (see *HSA Programmer's Reference Manual Version 1.0* for compatibility rules).

Signature

```
hsa_status_t hsa_executable_validate(
    hsa_executable_t executable,
    uint32_t *result );
```

Parameters

executable
(in) Executable.

result
(out) Memory location where the HSA runtime stores the validation result. If the executable is valid, the result is 0.

Return Values

HSA_STATUS_SUCCESS
The function has been executed successfully.

HSA_STATUS_ERROR_NOT_INITIALIZED
The HSA runtime has not been initialized.

HSA_STATUS_ERROR_INVALID_EXECUTABLE
executable is invalid.

HSA_STATUS_ERROR_INVALID_ARGUMENT
result is NULL.

2.8.1.35 hsa_executable_symbol_t

Executable symbol.

Signature

```
typedef struct hsa_executable_symbol_s {
    uint64_t handle;
} hsa_executable_symbol_t
```

Data Fields

handle
Opaque handle.

2.8.1.36 hsa_executable_get_symbol

Get the symbol handle for a given a symbol name.

Signature

```
hsa_status_t hsa_executable_get_symbol(
    hsa_executable_t executable,
    const char *module_name,
    const char *symbol_name,
    hsa_agent_t agent,
    int32_t call_convention,
    hsa_executable_symbol_t *symbol);
```

Parameters

executable
(in) Executable.

module_name
(in) Module name. Must be NULL if the symbol has program linkage.

symbol_name
(in) Symbol name.

agent
(in) Agent associated with the symbol. If the symbol is independent of any agent (for example, a variable with program allocation), this argument is ignored.

call_convention
(in) Call convention associated with the symbol. If the symbol does not correspond to an indirect function, this argument is ignored.

symbol
(out) Memory location where the HSA runtime stores the symbol handle.

Return Values

HSA_STATUS_SUCCESS
The function has been executed successfully.

HSA_STATUS_ERROR_NOT_INITIALIZED
The HSA runtime has not been initialized.

HSA_STATUS_ERROR_INVALID_EXECUTABLE
The executable is invalid.

HSA_STATUS_ERROR_INVALID_SYMBOL_NAME
There is no symbol with a name that matches *symbol_name*.

HSA_STATUS_ERROR_INVALID_ARGUMENT
symbol_name is NULL, or *symbol* is NULL.

2.8.1.37 hsa_executable_symbol_info_t

Executable symbol attributes.

Signature

```
typedef enum {
    HSA_EXECUTABLE_SYMBOL_INFO_TYPE = 0,
    HSA_EXECUTABLE_SYMBOL_INFO_NAME_LENGTH = 1,
    HSA_EXECUTABLE_SYMBOL_INFO_NAME = 2,
    HSA_EXECUTABLE_SYMBOL_INFO_MODULE_NAME_LENGTH = 3,
    HSA_EXECUTABLE_SYMBOL_INFO_MODULE_NAME = 4,
    HSA_EXECUTABLE_SYMBOL_INFO_AGENT = 20,
    HSA_EXECUTABLE_SYMBOL_INFO_VARIABLE_ADDRESS = 21,
    HSA_EXECUTABLE_SYMBOL_INFO_LINKAGE = 5,
    HSA_EXECUTABLE_SYMBOL_INFO_IS_DEFINITION = 17,
    HSA_EXECUTABLE_SYMBOL_INFO_VARIABLE_ALLOCATION = 6,
    HSA_EXECUTABLE_SYMBOL_INFO_VARIABLE_SEGMENT = 7,
    HSA_EXECUTABLE_SYMBOL_INFO_VARIABLE_ALIGNMENT = 8,
    HSA_EXECUTABLE_SYMBOL_INFO_VARIABLE_SIZE = 9,
    HSA_EXECUTABLE_SYMBOL_INFO_VARIABLE_IS_CONST = 10,
    HSA_EXECUTABLE_SYMBOL_INFO_KERNEL_OBJECT = 22,
    HSA_EXECUTABLE_SYMBOL_INFO_KERNEL_KERNARG_SEGMENT_SIZE = 11,
    HSA_EXECUTABLE_SYMBOL_INFO_KERNEL_KERNARG_SEGMENT_ALIGNMENT = 12,
    HSA_EXECUTABLE_SYMBOL_INFO_KERNEL_GROUP_SEGMENT_SIZE = 13,
    HSA_EXECUTABLE_SYMBOL_INFO_KERNEL_PRIVATE_SEGMENT_SIZE = 14,
    HSA_EXECUTABLE_SYMBOL_INFO_KERNEL_DYNAMIC_CALLSTACK = 15,
    HSA_EXECUTABLE_SYMBOL_INFO_INDIRECT_FUNCTION_OBJECT = 23,
    HSA_EXECUTABLE_SYMBOL_INFO_INDIRECT_FUNCTION_CALL_CONVENTION = 16
} hsa_executable_symbol_info_t;
```

Values

HSA_EXECUTABLE_SYMBOL_INFO_TYPE
The kind of the symbol. The type of this attribute is hsa_symbol_kind_t.

HSA_EXECUTABLE_SYMBOL_INFO_NAME_LENGTH
The length of the symbol name. The type of this attribute is uint32_t.

HSA_EXECUTABLE_SYMBOL_INFO_NAME
The name of the symbol. The type of this attribute is character array with the length equal to the value of HSA_EXECUTABLE_SYMBOL_INFO_NAME_LENGTH attribute.

HSA_EXECUTABLE_SYMBOL_INFO_MODULE_NAME_LENGTH
The length of the module name to which this symbol belongs if this symbol has module linkage, otherwise 0 is returned. The type of this attribute is uint32_t.

HSA_EXECUTABLE_SYMBOL_INFO_MODULE_NAME
The module name to which this symbol belongs if this symbol has module linkage, otherwise empty string is returned. The type of this attribute is character array with the length equal to the value of HSA_EXECUTABLE_SYMBOL_INFO_MODULE_NAME_LENGTH attribute.

HSA_EXECUTABLE_SYMBOL_INFO_AGENT
: Agent associated with this symbol. If the symbol is a variable, the value of this attribute is only defined if HSA_EXECUTABLE_SYMBOL_INFO_VARIABLE_ALLOCATION is HSA_VARIABLE_ALLOCATION_AGENT. The type of this attribute is hsa_agent_t.

HSA_EXECUTABLE_SYMBOL_INFO_VARIABLE_ADDRESS
: The address of the variable. The value of this attribute is undefined if the symbol is not a variable. The type of this attribute is uint64_t.

  If executable's state is HSA_EXECUTABLE_STATE_UNFROZEN, then 0 is returned.

HSA_EXECUTABLE_SYMBOL_INFO_LINKAGE
: The linkage kind of the symbol. The type of this attribute is hsa_symbol_kind_linkage_t.

HSA_EXECUTABLE_SYMBOL_INFO_IS_DEFINITION
: Indicates whether the symbol corresponds to a definition. The type of this attribute is bool.

HSA_EXECUTABLE_SYMBOL_INFO_VARIABLE_ALLOCATION
: The allocation kind of the variable. The value of this attribute is undefined if the symbol is not a variable. The type of this attribute is hsa_variable_allocation_t.

HSA_EXECUTABLE_SYMBOL_INFO_VARIABLE_SEGMENT
: The segment kind of the variable. The value of this attribute is undefined if the symbol is not a variable. The type of this attribute is hsa_variable_segment_t.

HSA_EXECUTABLE_SYMBOL_INFO_VARIABLE_ALIGNMENT
: Alignment of the variable. The value of this attribute is undefined if the symbol is not a variable. The type of this attribute is uint32_t.

HSA_EXECUTABLE_SYMBOL_INFO_VARIABLE_SIZE
: Size of the variable. The value of this attribute is undefined if the symbol is not a variable. The type of this attribute is uint32_t.

  A value of 0 is returned if the variable is an external variable and has an unknown dimension.

HSA_EXECUTABLE_SYMBOL_INFO_VARIABLE_IS_CONST
: Indicates whether the variable is constant. The value of this attribute is undefined if the symbol is not a variable. The type of this attribute is bool.

HSA_EXECUTABLE_SYMBOL_INFO_KERNEL_OBJECT
: Kernel object handle, used in the kernel dispatch packet. The value of this attribute is undefined if the symbol is not a kernel. The type of this attribute is uint64_t.

  If the state of the executable is HSA_EXECUTABLE_STATE_UNFROZEN, then 0 is returned.

HSA_EXECUTABLE_SYMBOL_INFO_KERNEL_KERNARG_SEGMENT_SIZE
: Size of kernarg segment memory that is required to hold the values of the kernel arguments, in bytes. The value of this attribute is undefined if the symbol is not a kernel. The type of this attribute is uint32_t.

HSA_EXECUTABLE_SYMBOL_INFO_KERNEL_KERNARG_SEGMENT_ALIGNMENT
: Alignment (in bytes) of the buffer used to pass arguments to the kernel, which is the maximum of 16 and the maximum alignment of any of the kernel arguments. The value of this attribute is undefined if the symbol is not a kernel. The type of this attribute is uint32_t.

HSA_EXECUTABLE_SYMBOL_INFO_KERNEL_GROUP_SEGMENT_SIZE
Size of static group segment memory required by the kernel (per work-group), in bytes. The value of this attribute is undefined if the symbol is not a kernel. The type of this attribute is uint32_t.

The reported amount does not include any dynamically allocated group segment memory that may be requested by the application when a kernel is dispatched.

HSA_EXECUTABLE_SYMBOL_INFO_KERNEL_PRIVATE_SEGMENT_SIZE
Size of static private, spill, and arg segment memory required by this kernel (per work-item), in bytes. The value of this attribute is undefined if the symbol is not a kernel. The type of this attribute is uint32_t.

If the value of HSA_EXECUTABLE_SYMBOL_INFO_KERNEL_DYNAMIC_CALLSTACK is true, the kernel may use more private memory than the reported value, and the application must add the dynamic call stack usage to *private_segment_size* when populating a kernel dispatch packet.

HSA_EXECUTABLE_SYMBOL_INFO_KERNEL_DYNAMIC_CALLSTACK
Dynamic callstack flag. The value of this attribute is undefined if the symbol is not a kernel. The type of this attribute is bool.

If this flag is set (the value is true), the kernel uses a dynamically sized call stack. This can happen if recursive calls, calls to indirect functions, or the HSAIL alloca instruction are present in the kernel.

HSA_EXECUTABLE_SYMBOL_INFO_INDIRECT_FUNCTION_OBJECT
Indirect function object handle. The value of this attribute is undefined if the symbol is not an indirect function, or the associated agent does not support the Full Profile. The type of this attribute depends on the machine model: if machine model is small, then the type is uint32_t, if machine model is large, then the type is uint64_t.

If the state of the executable is HSA_EXECUTABLE_STATE_UNFROZEN, then 0 is returned.

HSA_EXECUTABLE_SYMBOL_INFO_INDIRECT_FUNCTION_CALL_CONVENTION
Call convention of the indirect function. The value of this attribute is undefined if the symbol is not an indirect function, or the associated agent does not support the Full Profile. The type of this attribute is uint32_t.

2.8.1.38 hsa_executable_symbol_get_info

Get the current value of an attribute for a given executable symbol.

Signature

```
hsa_status_t hsa_executable_symbol_get_info(
    hsa_executable_symbol_t executable_symbol,
    hsa_executable_symbol_info_t attribute,
    void *value);
```

2.8.1.38.1 Parameters

executable_symbol
(in) Executable symbol.

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 HSA runtime has not been initialized.

HSA_STATUS_ERROR_INVALID_ARGUMENT
attribute is an invalid executable symbol attribute, or *value* is NULL.

2.8.1.39 hsa_executable_iterate_symbols

Iterate over the symbols in a executable, and invoke an application-defined callback on every iteration.

Signature

```
hsa_status_t hsa_executable_iterate_symbols(
    hsa_executable_t executable,
    hsa_status_t (*callback)(hsa_executable_t executable, hsa_executable_symbol_t symbol, void *data),
    void *data);
```

Parameters

executable
(in) Executable.

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

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

Return Values

HSA_STATUS_SUCCESS
The function has been executed successfully.

HSA_STATUS_ERROR_NOT_INITIALIZED
The HSA runtime has not been initialized.

HSA_STATUS_ERROR_INVALID_EXECUTABLE
The executable is invalid.

HSA_STATUS_ERROR_INVALID_ARGUMENT
callback is NULL.

2.9 Common Definitions

2.9.1 Common Definitions API

2.9.1.1 hsa_dim3_t

Three-dimensional coordinate.

Signature

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

Data Fields

x
X dimension.

y
Y dimension.

z
Z dimension.

2.9.1.2 hsa_access_permission_t

Access permissions.

Signature

```
typedef enum {
    HSA_ACCESS_PERMISSION_RO = 1,
    HSA_ACCESS_PERMISSION_WO = 2,
    HSA_ACCESS_PERMISSION_RW = 3
} hsa_access_permission_t;
```

Values

HSA_ACCESS_PERMISSION_RO
Read-only access.

HSA_ACCESS_PERMISSION_WO
Write-only access.

HSA_ACCESS_PERMISSION_RW
Read and write access.

CHAPTER 3.
HSA Extensions Programming Guide

3.1 Extensions in HSA

Extensions to the HSA core runtime API can be HSA-approved or vendor-specific. HSA-approved extensions are not required to be supported by a conforming HSA implementation, but are expected to be widely available; they define functionality that is likely to move into the core API in a future version of the HSA specification. Currently, there are two HSA-approved extensions: Finalization (see 3.2. HSAIL Finalization (page 110)) and Images (see 3.3. Images and Samplers (page 119)).

Extensions approved by the HSA Foundation can be promoted to the core API in later versions. When this occurs, the extension specification is added to the core specification. Functions, types, and enumeration constants that are part of a promoted extension will have the extension prefix removed. HSA implementations of such later revisions must also declare support for the original extensions and expose the original versions of functions, types, and enumeration constants as a transition aid.

3.1.1 Extension requirements

Each extension must be assigned a name of the form:

- *hsa_ext_* (HSA-approved extension)
- *hsa_<vendor(s) name>_<label>* (other)

The label is one or more words separated by underscores, providing a short name for the extension. A label must be short, meaningful, and should not collide with the vendor name, the *hsa* token, or the *ext* token. Vendor names must be registered with the HSA Foundation, must be unique, and may be abbreviated to improve the readability of the symbols. An extension name must not contain upper case characters. For instance, if vendor *HAL* wants to create an extension related to label *foo*, the resulting extension name is *hsa_hal_foo*.

All the functions and types declared in the extension must be prefixed by the extension name, and follow HSA naming conventions. For example, a vendor-specific extension *hsa_hal_foo* could declare the following identifiers:

```
hsa_status_t hsa_hal_foo_do_something();

typedef enum {
    HSA_HAL_FOO_CATEGORY_VALUE = 1,
} hsa_hal_foo_category_t;
```

An extension can add new enumeration constants to an existing core enumeration. For example, an extension may add agent attributes to hsa_agent_info_t. In order to avoid enumeration value collisions in core enumerations, the enumeration constants used by an extension must be assigned by the HSA Foundation.

Every extension must create a preprocessor definition that matches the extension name. The value associated with the identifier encodes the version number. For example, the *hsa_hal_foo* extension (version 1.1) would include the following preprocessing directive in the header:

```
#define hsa_hal_foo 001001
```

If the extension API exposes any functions, the extension interface must declare a function table (structure) in which each field is a pointer to a function exported by the extension. The function pointer table must have as many entries as functions are exported by the extension API. For example, the header associated with the extension *hsa_hal_foo* would contain the following declaration:

```
typedef struct hsa_hal_foo_pfn_s {
    hsa_status_t  (*hsa_hal_foo_pfn_do_something)();
} hsa_hal_foo_pfn_t;
```

The HSA Foundation assigns a unique integer ID in the [0, 0x400) interval to each extension. The identifier remains the same throughout all the versions of the same extension. In the HSA runtime API, the application uses the identifier to refer to a specific extension. Identifiers are listed in the hsa_extension_t enumeration. For example, the extension *hsa_hal_foo* would add the enumeration constant HSA_EXTENSION_HAL_FOO associated with a unique constant expression (the identifier).

3.1.2 Extension support: HSA runtime and agents

The HSA runtime indicates which extensions it supports in the HSA_SYSTEM_INFO_EXTENSIONS bit-mask attribute. If bit i is set in the bit-mask, then the extension with an ID of i is supported by the implementation. Because the bit-mask does not expose any information about which revision of the extension is supported, the application must query the function **hsa_system_extension_supported** when needed.

A portable application must use the function pointers exported by an extension to invoke its API. The application can retrieve a copy of the extension function pointer table by calling **hsa_system_get_extension_table**. Some HSA implementations may choose to also export its functions statically from the object libraries implementing those functions. However, portable applications cannot rely on this behavior. In the following code snippet, the application invokes an extension function once the HSA runtime has populated the function pointer table corresponding to the version 1.0 of the *hsa_hal_foo* extension.

```
bool system_support, agent_support;
hsa_system_extension_supported(HSA_EXTENSION_HAL_FOO, 1, 0, &system_support);
hsa_agent_extension_supported(HSA_EXTENSION_HAL_FOO, agent, 1, 0, &agent_support);
if (system_support && agent_support) {
    hsa_hal_foo_pfn_t pfns;
    hsa_system_get_extension_table(HSA_EXTENSION_HAL_FOO, 1, 0, &pfns);
    pfns.hsa_hal_foo_pfn_do_something();
}
```

An agent indicates which extensions it supports in the HSA_AGENT_INFO_EXTENSIONS bit-mask attribute. The agent may support an extension even if the implementation does not support it (see HSA_SYSTEM_INFO_EXTENSIONS and vice versa. The application can query if a version of the extension is supported by an agent using **hsa_agent_extension_supported**.

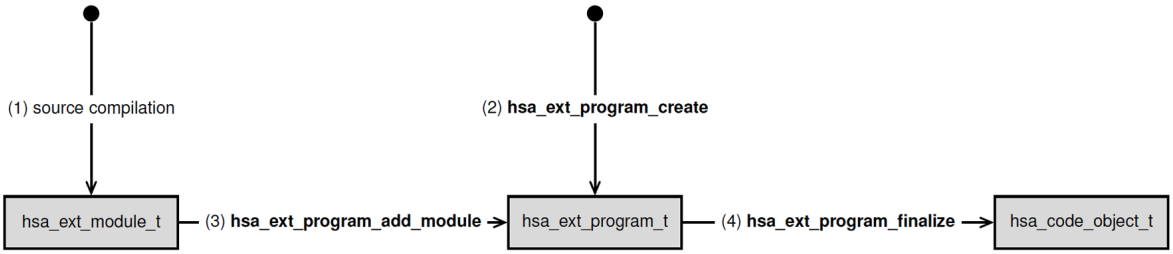
3.2 HSAIL Finalization

The finalization API allows the application to compile a set of HSAIL modules in binary format (BRIG), and retrieve the corresponding code object (see 2.8. Code Objects and Executables (page 78)). In the most basic usage scenario, the application wishes to obtain the code object for a given kernel of interest (a kernel that is to be executed). Using the HSA runtime data structures and functions, the basic workflow is:

1. The application is in possession of one or more HSAIL modules that are the result of compiling a high-level language such as OpenMP or OpenCL. One of the HSAIL modules contains the kernel of interest. This step is performed outside of the HSA runtime API.
2. The application creates an HSAIL program by invoking **hsa_ext_program_create**.
3. The HSAIL module containing the kernel is added to the HSAIL program by using **hsa_ext_program_add_module**.
4. The application finalizes the HSAIL program (**hsa_ext_program_finalize**), which creates a code object handle.
5. The code object handle can be serialized to disk or further processed in order to launch a kernel.

This basic workflow is represented in Figure 3–1 (below).

Figure 3–1 From source to code object



3.2.1 HSAIL Finalization API

3.2.1.1 hsa_status_t finalizer constants

Enumeration constants added to hsa_status_t by this extension.

Signature

```
enum {
    HSA_EXT_STATUS_ERROR_INVALID_PROGRAM = 0x2000,
    HSA_EXT_STATUS_ERROR_INVALID_MODULE = 0x2001,
    HSA_EXT_STATUS_ERROR_INCOMPATIBLE_MODULE = 0x2002,
    HSA_EXT_STATUS_ERROR_MODULE_ALREADY_INCLUDED = 0x2003,
    HSA_EXT_STATUS_ERROR_SYMBOL_MISMATCH = 0x2004,
    HSA_EXT_STATUS_ERROR_FINALIZATION_FAILED = 0x2005,
    HSA_EXT_STATUS_ERROR_DIRECTIVE_MISMATCH = 0x2006
};
```

Values

HSA_EXT_STATUS_ERROR_INVALID_PROGRAM
The HSAIL program is invalid.

HSA_EXT_STATUS_ERROR_INVALID_MODULE
The HSAIL module is invalid.

HSA_EXT_STATUS_ERROR_INCOMPATIBLE_MODULE
Machine model or profile of the HSAIL module do not match the machine model or profile of the HSAIL program.

HSA_EXT_STATUS_ERROR_MODULE_ALREADY_INCLUDED
The HSAIL module is already a part of the HSAIL program.

HSA_EXT_STATUS_ERROR_SYMBOL_MISMATCH
Compatibility mismatch between symbol declaration and symbol definition.

HSA_EXT_STATUS_ERROR_FINALIZATION_FAILED
The finalization encountered an error while finalizing a kernel or indirect function.

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

3.2.1.2 hsa_ext_module_t

HSAIL (BRIG) module. For a definition of the BrigModule_t type, see the *HSA Programmer's Reference Manual Version 1.0*.

Signature

```
typedef BrigModule_t hsa_ext_module_t;
```

3.2.1.3 hsa_ext_program_t

An opaque handle to a HSAIL program, which groups a set of HSAIL modules that collectively define functions and variables used by kernels and indirect functions.

Signature

```
typedef struct hsa_ext_program_s {
    uint64_t handle;
} hsa_ext_program_t
```

Data Fields

handle
Opaque handle.

3.2.1.4 hsa_ext_program_create

Create an empty HSAIL program.

Signature

```
hsa_status_t hsa_ext_program_create(
    hsa_machine_model_t machine_model,
    hsa_profile_t profile,
    hsa_default_float_rounding_mode_t default_float_rounding_mode, const char *options,
    hsa_ext_program_t *program);
```

Parameters

machine_model
(in) Machine model used in the HSAIL program.

profile
(in) Profile used in the HSAIL program.

default_float_rounding_mode
(in) Default floating-point rounding mode used in the HSAIL program.

options
(in) Vendor-specific options. May be NULL.

program
(out) Memory location where the HSA runtime stores the newly created HSAIL program handle.

Return Values

HSA_STATUS_SUCCESS
The function has been executed successfully.

HSA_STATUS_ERROR_NOT_INITIALIZED
The HSA runtime has not been initialized.

HSA_STATUS_ERROR_OUT_OF_RESOURCES
There is a failure to allocate resources required for the operation.

HSA_STATUS_ERROR_INVALID_ARGUMENT
machine_model is invalid, *profile* is invalid, *default_float_rounding_mode* is invalid, or *program* is NULL.

3.2.1.5 hsa_ext_program_destroy

Destroy a HSAIL program.

Signature

```
hsa_status_t hsa_ext_program_destroy(
    hsa_ext_program_t program);
```

Parameters

program
(in) HSAIL program.

Return Values

HSA_STATUS_SUCCESS
The function has been executed successfully.

HSA_STATUS_ERROR_NOT_INITIALIZED
The HSA runtime has not been initialized.

HSA_EXT_STATUS_ERROR_INVALID_PROGRAM
The HSAIL program is invalid.

Description

The HSAIL program handle becomes invalid after it has been destroyed. Code object handles produced by **hsa_ext_program_finalize** are still valid after the HSAIL program has been destroyed, and can be used as intended. Resources allocated outside and associated with the HSAIL program (such as HSAIL modules that are added to the HSAIL program) can be released after the finalization program has been destroyed.

3.2.1.6 hsa_ext_program_add_module

Add a HSAIL module to an existing HSAIL program.

Signature

```
hsa_status_t hsa_ext_program_add_module(
    hsa_ext_program_t program,
    hsa_ext_module_t module);
```

Parameters

program
(in) HSAIL program.

module
(in) HSAIL module. The application can add the same HSAIL module to *program* at most once. The HSAIL module must specify the same machine model and profile as *program*. If the default floating-point rounding mode of *module* is not default, then it should match that of *program*.

Return Values

HSA_STATUS_SUCCESS
The function has been executed successfully.

HSA_STATUS_ERROR_NOT_INITIALIZED
The HSA runtime has not been initialized.

HSA_STATUS_ERROR_OUT_OF_RESOURCES
There is a failure to allocate resources required for the operation.

HSA_EXT_STATUS_ERROR_INVALID_PROGRAM
The HSAIL program is invalid.

HSA_EXT_STATUS_ERROR_INVALID_MODULE
The HSAIL module is invalid.

HSA_EXT_STATUS_ERROR_INCOMPATIBLE_MODULE
The machine model of *module* does not match machine model of *program*, or the profile of *module* does not match profile of *program*.

HSA_EXT_STATUS_ERROR_MODULE_ALREADY_INCLUDED
The HSAIL module is already a part of the HSAIL program.

HSA_EXT_STATUS_ERROR_SYMBOL_MISMATCH
Symbol declaration and symbol definition compatibility mismatch. See the symbol compatibility rules in the *HSA Programmer's Reference Manual Version 1.0*.

Description

The HSA runtime does not perform a deep copy of the HSAIL module upon addition. Instead, it stores a pointer to the HSAIL module. The ownership of the HSAIL module belongs to the application, which must ensure that *module* is not released before destroying the HSAIL program.

The HSAIL module is successfully added to the HSAIL program if *module* is valid, if all the declarations and definitions for the same symbol are compatible, and if *module* specify machine model and profile that matches the HSAIL program.

3.2.1.7 hsa_ext_program_iterate_modules

Iterate over the HSAIL modules in a program, and invoke an application-defined callback on every iteration.

Signature

```
hsa_status_t hsa_ext_program_iterate_modules(
    hsa_ext_program_t program,
    hsa_status_t (*callback)(hsa_ext_program_t program, hsa_ext_module_t module, void *data),
    void *data);
```

Parameters

program
(in) HSAIL program.

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

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

Return Values

HSA_STATUS_SUCCESS
The function has been executed successfully.

HSA_STATUS_ERROR_NOT_INITIALIZED
The HSA runtime has not been initialized.

HSA_EXT_STATUS_ERROR_INVALID_PROGRAM
The HSAIL program is invalid.

HSA_STATUS_ERROR_INVALID_ARGUMENT
callback is NULL.

3.2.1.8 hsa_ext_program_info_t

HSAIL program attributes.

Signature

```
typedef enum {
    HSA_EXT_PROGRAM_INFO_MACHINE_MODEL = 0,
    HSA_EXT_PROGRAM_INFO_PROFILE = 1,
    HSA_EXT_PROGRAM_INFO_DEFAULT_FLOAT_ROUNDING_MODE = 2
} hsa_ext_program_info_t;
```

Values

HSA_EXT_PROGRAM_INFO_MACHINE_MODEL
: Machine model specified when the HSAIL program was created. The type of this attribute is hsa_machine_model_t.

HSA_EXT_PROGRAM_INFO_PROFILE
: Profile specified when the HSAIL program was created. The type of this attribute is hsa_profile_t.

HSA_EXT_PROGRAM_INFO_DEFAULT_FLOAT_ROUNDING_MODE
: Default floating-point rounding mode specified when the HSAIL program was created. The type of this attribute is hsa_default_float_rounding_mode_t.

3.2.1.9 hsa_ext_program_get_info

Get the current value of an attribute for a given HSAIL program.

Signature

```
hsa_status_t hsa_ext_program_get_info(
    hsa_ext_program_t program,
    hsa_ext_program_info_t attribute,
    void *value);
```

Parameters

program
: (in) HSAIL program.

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 HSA runtime has not been initialized.

HSA_EXT_STATUS_ERROR_INVALID_PROGRAM
The HSAIL program is invalid.

HSA_STATUS_ERROR_INVALID_ARGUMENT
attribute is an invalid HSAIL program attribute, or *value* is NULL.

3.2.1.10 hsa_ext_finalizer_call_convention_t

Finalizer-determined call convention.

Signature

```
typedef enum {
    HSA_EXT_FINALIZER_CALL_CONVENTION_AUTO = -1
} hsa_ext_finalizer_call_convention_t;
```

Values

HSA_EXT_FINALIZER_CALL_CONVENTION_AUTO
Finalizer-determined call convention.

3.2.1.11 hsa_ext_control_directives_t

Control directives specify low-level information about the finalization process.

Signature

```
typedef struct hsa_ext_control_directives_s {
    uint64_t control_directives_mask;
    uint16_t break_exceptions_mask;
    uint16_t detect_exceptions_mask;
    uint32_t max_dynamic_group_size;
    uint64_t max_flat_grid_size;
    uint32_t max_flat_workgroup_size;
    uint32_t reserved1;
    uint64_t required_grid_size[3];
    hsa_dim3_t required_workgroup_size;
    uint8_t required_dim;
    uint8_t reserved2[75];
} hsa_ext_control_directives_t
```

Data Fields

control_directives_mask
Bitset indicating which control directives are enabled. The bit assigned to a control directive is determined by the corresponding value in BrigControlDirective.

If a control directive is disabled, its corresponding field value (if any) must be 0. Control directives that are only present or absent (such as partial workgroups) have no corresponding field as the presence of the bit in this mask is sufficient.

break_exceptions_mask
Bitset of HSAIL exceptions that must have the BREAK policy enabled. The bit assigned to an HSAIL exception is determined by the corresponding value in BrigExceptionsMask. If the kernel contains a enablebreakexceptions control directive, the finalizer uses the union of the two masks.

detect_exceptions_mask

Bitset of HSAIL exceptions that must have the DETECT policy enabled. The bit assigned to an HSAIL exception is determined by the corresponding value in BrigExceptionsMask. If the kernel contains a enabledetectexceptions control directive, the finalizer uses the union of the two masks.

max_dynamic_group_size

Maximum size (in bytes) of dynamic group memory that will be allocated by the application for any dispatch of the kernel. If the kernel contains a maxdynamicsize control directive, the two values should match.

max_flat_grid_size

Maximum number of grid work-items that will be used by the application to launch the kernel. If the kernel contains a maxflatgridsize control directive, the value of *max_flat_grid_size* must not be greater than the value of the directive, and takes precedence.

The value specified for maximum absolute grid size must be greater than or equal to the product of the values specified by *required_grid_size*.

If the bit at position BRIG_CONTROL_MAXFLATGRIDSIZE is set in *control_directives_mask*, this field must be greater than 0.

max_flat_workgroup_size

Maximum number of work-group work-items that will be used by the application to launch the kernel. If the kernel contains a maxflatworkgroupsize control directive, the value of *max_flat_workgroup_size* must not be greater than the value of the directive, and takes precedence.

The value specified for maximum absolute grid size must be greater than or equal to the product of the values specified by *required_workgroup_size*.

If the bit at position BRIG_CONTROL_MAXFLATWORKGROUPSIZE is set in *control_directives_mask*, this field must be greater than 0.

reserved1

Reserved. Must be 0.

required_grid_size

Grid size that will be used by the application in any dispatch of the kernel. If the kernel contains a requiredgridsize control directive, the dimensions should match.

The specified grid size must be consistent with *required_workgroup_size* and *required_dim*. Also, the product of the three dimensions must not exceed *max_flat_grid_size*. Note that the listed invariants must hold only if all the corresponding control directives are enabled.

If the bit at position BRIG_CONTROL_REQUIREDGRIDSIZE is set in *control_directives_mask*, the three dimension values must be greater than 0.

required_workgroup_size

Work-group size that will be used by the application in any dispatch of the kernel. If the kernel contains a requiredworkgroupsize control directive, the dimensions should match.

The specified work-group size must be consistent with *required_grid_size* and *required_dim*. Also, the product of the three dimensions must not exceed *max_flat_workgroup_size*. Note that the listed invariants must hold only if all the corresponding control directives are enabled.

If the bit at position BRIG_CONTROL_REQUIREDWORKGROUPSIZE is set in *control_directives_mask*, the three dimension values must be greater than 0.

required_dim
Number of dimensions that will be used by the application to launch the kernel. If the kernel contains a requireddim control directive, the two values should match.

The specified dimensions must be consistent with *required_grid_size* and *required_workgroup_size*. This invariant must hold only if all the corresponding control directives are enabled.

If the bit at position BRIG_CONTROL_REQUIREDDIM is set in *control_directives_mask*, this field must be 1, 2, or 3.

reserved2
Reserved. Must be 0.

3.2.1.12 hsa_ext_program_finalize

Finalize an HSAIL program for a given instruction set architecture.

Signature

```
hsa_status_t hsa_ext_program_finalize(
    hsa_ext_program_t program,
    hsa_isa_t isa,
    int32_t call_convention,
    hsa_ext_control_directives_t  control_directives,
    const char *options,
    hsa_code_object_type_t  code_object_type,
    hsa_code_object_t *code_object );
```

Parameters

program
(in) HSAIL program.

isa
(in) Instruction set architecture to finalize for.

call_convention
(in) A call convention used in a finalization. Must have a value between HSA_EXT_FINALIZER_CALL_CONVENTION_AUTO (inclusive) and the value of the attribute HSA_ISA_INFO_CALL_CONVENTION_COUNT in *isa* (not inclusive).

control_directives
(in) Low-level control directives that influence the finalization process.

options
(in) Vendor-specific options. May be NULL.

code_object_type
(in) Type of code object to produce.

code_object
(out) Code object generated by the Finalizer, which contains the machine code for the kernels and indirect functions in the HSAIL program. The code object is independent of the HSAIL module that was used to generate it.

Return Values

HSA_STATUS_SUCCESS
The function has been executed successfully.

HSA_STATUS_ERROR_NOT_INITIALIZED
The HSA runtime has not been initialized.

HSA_STATUS_ERROR_OUT_OF_RESOURCES
There is a failure to allocate resources required for the operation.

HSA_EXT_STATUS_ERROR_INVALID_PROGRAM
The HSAIL program is invalid.

HSA_STATUS_ERROR_INVALID_ISA
isa is invalid.

HSA_EXT_STATUS_ERROR_DIRECTIVE_MISMATCH
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_EXT_STATUS_ERROR_FINALIZATION_FAILED
The Finalizer encountered an error while compiling a kernel or an indirect function.

Description

Finalize all of the kernels and indirect functions that belong to the same HSAIL program for a specific instruction set architecture (ISA). The transitive closure of all functions specified by call or scall must be defined. Kernels and indirect functions that are being finalized must be defined. Kernels and indirect functions that are referenced in kernels and indirect functions being finalized may or may not be defined, but must be declared. All the global/readonly segment variables that are referenced in kernels and indirect functions being finalized may or may not be defined, but must be declared.

3.3 Images and Samplers

The HSA runtime uses an opaque image handle (hsa_ext_image_t) to represent images. The image handle references the image data in memory and stores information about resource layout and other properties. HSA decouples the storage of the image data and the description of how the agent interprets that data. This allows the application to control the location of the image data storage and manage memory more efficiently.

An image *format* is specified using a channel type and a channel order. The channel type describes how the data is to be interpreted along with the bit size, and the channel order describes the number and the order of memory components. Not all image channel types and channel order combinations are valid on an agent, but an agent must support a minimum set of image formats. For more information, see the *HSA Programmer's Reference Manual Version 1.0*. An application can use **hsa_ext_image_get_capability** to obtain the image format capabilities for a given combination of agent, geometry, and image format.

An implementation-independent image format descriptor (hsa_ext_image_descriptor_t) is composed of a geometry along with the image format. The image descriptor is used to inquire the runtime for the agent-specific image data size and alignment details by calling **hsa_ext_image_data_get_info** for the purpose of determining the implementation's storage requirements. The memory requirements (hsa_ext_image_data_info_t) include the size of the memory needed as well as any alignment constraints. An application can either allocate new memory for storing the image data, or use an existing buffer. Before the image data is used, an agent-specific image handle must be created using it and if necessary, cleared and prepared according to the intended use.

The function **hsa_ext_image_create** creates an agent-specific image from an image format descriptor, an application-allocated buffer that conforms to the requirements provided by **hsa_ext_image_data_get_info**, and access permissions. The returned handle can used by the HSAIL operations **rdimage**, **ldimage**, and **stimage**.

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 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. The *HSA Platform System Architecture Specification Version 1.0* details the HSA memory model for images.

An agent-specific sampler handle (hsa_ext_sampler_t) is used by the HSAIL language to describe how images are processed by the **rdimage** HSAIL operation. The function **hsa_ext_sampler_create** creates a sampler handle from an agent-independent sampler descriptor (hsa_ext_sampler_descriptor_t).

3.3.1 Images and Samplers API

3.3.1.1 hsa_ext_image_t

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

Signature

```
typedef struct hsa_ext_image_s {
    uint64_t handle;
} hsa_ext_image_t
```

Data Fields

handle
Opaque handle.

3.3.1.2 hsa_ext_image_geometry_t

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

Signature

```
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;
```

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.

3.3.1.3 hsa_ext_image_channel_type_t

Channel type associated with the elements of an image. For definitions on each component type, see the *HSA Programmer's Reference Manual Version 1.0*. The enumeration values match the BRIG type BrigImageChannelType.

Signature

```
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;
```

3.3.1.4 hsa_ext_image_channel_order_t

Channel order associated with the elements of an image. For definitions on each component order, see the *HSA Programmer's Reference Manual Version 1.0*. The enumeration values match the BRIG type BrigImageChannelOrder.

Signature

```
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;
```

3.3.1.5 hsa_ext_image_format_t

Image format.

Signature

```
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
```

Data Fields

channel_type
Channel type.

channel_order
Channel order.

3.3.1.6 hsa_ext_image_descriptor_t

Implementation-independent image descriptor.

Signature

```
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
```

Data Fields

geometry
Image geometry.

width
Width of the image, in components.

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

depth
Depth of the image, in components. Only defined if *geometry* is HSA_EXT_IMAGE_GEOMETRY_3D. A depth of 0 is same as a depth of 1.

array_size
Number of images in the image array. Only defined if *geometry* is HSA_EXT_IMAGE_GEOMETRY_1DA, HSA_EXT_IMAGE_GEOMETRY_2DA, or HSA_EXT_IMAGE_GEOMETRY_2DADEPTH.

format
Image format.

3.3.1.7 hsa_ext_image_capability_t

Image capability.

Signature

```
typedef enum {
    HSA_EXT_IMAGE_CAPABILITY_NOT_SUPPORTED = 0x0,
    HSA_EXT_IMAGE_CAPABILITY_READ_ONLY = 0x1,
    HSA_EXT_IMAGE_CAPABILITY_WRITE_ONLY = 0x2,
    HSA_EXT_IMAGE_CAPABILITY_READ_WRITE = 0x4,
    HSA_EXT_IMAGE_CAPABILITY_READ_MODIFY_WRITE = 0x8,
    HSA_EXT_IMAGE_CAPABILITY_ACCESS_INVARIANT_DATA_LAYOUT = 0x10
```

```
} hsa_ext_image_capability_t;
```

Values

HSA_EXT_IMAGE_CAPABILITY_NOT_SUPPORTED
Images of this geometry and format are not supported in the agent.

HSA_EXT_IMAGE_CAPABILITY_READ_ONLY
Read-only images of this geometry and format are supported by the agent.

HSA_EXT_IMAGE_CAPABILITY_WRITE_ONLY
Write-only images of this geometry and format are supported by the agent.

HSA_EXT_IMAGE_CAPABILITY_READ_WRITE
Read-write images of this geometry and format are supported by the agent.

HSA_EXT_IMAGE_CAPABILITY_READ_MODIFY_WRITE
Images of this geometry and format can be accessed from read-modify-write operations in the agent.

HSA_EXT_IMAGE_CAPABILITY_ACCESS_INVARIANT_DATA_LAYOUT
Images of this geometry and format are guaranteed to have a consistent data layout regardless of how they are accessed by the associated agent.

3.3.1.8 hsa_ext_image_get_capability

Retrieve the supported image capabilities for a given combination of agent, image format, and geometry.

Signature

```
hsa_status_t hsa_ext_image_get_capability(
    hsa_agent_t agent,
    hsa_ext_image_geometry_t geometry,
    const hsa_ext_image_format_t *image_format,
    uint32_t *capability_mask );
```

Parameters

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

geometry
(in) Geometry.

image_format
(in) Pointer to an image format. Must not be NULL.

capability_mask
(out) Pointer to a memory location where the HSA runtime stores a bit-mask of supported image capability (hsa_ext_image_capability_t) values. Must not be NULL.

Return Values

HSA_STATUS_SUCCESS
The function has been executed successfully.

HSA_STATUS_ERROR_NOT_INITIALIZED
The HSA runtime has not been initialized.

HSA_STATUS_ERROR_INVALID_AGENT
The agent is invalid.

HSA_STATUS_ERROR_INVALID_ARGUMENT
geometry is not a valid image geometry value, *image_format* is NULL, or *capability_mask* is NULL.

3.3.1.9 hsa_ext_image_data_info_t

Agent-specific image size and alignment requirements, populated by **hsa_ext_image_data_get_info**.

Signature

```
typedef struct hsa_ext_image_data_info_s {
    size_t size;
    size_t alignment;
} hsa_ext_image_data_info_t
```

Data Fields

size
Image data size, in bytes.

alignment
Image data alignment, in bytes.

3.3.1.10 hsa_ext_image_data_get_info

Retrieve the image data requirements for a given combination of image descriptor, access permission, and agent.

Signature

```
hsa_status_t hsa_ext_image_data_get_info(
    hsa_agent_t agent,
    const hsa_ext_image_descriptor_t *image_descriptor,
    hsa_access_permission_t  access_permission,
    hsa_ext_image_data_info_t  *image_data_info);
```

Parameters

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

image_descriptor
(in) Pointer to an image descriptor. Must not be NULL.

access_permission
(in) Image access mode for *agent*.

image_data_info
(out) Memory location where the runtime stores the size and alignment requirements. Must not be NULL.

Return Values

HSA_STATUS_SUCCESS
The function has been executed successfully.

HSA_STATUS_ERROR_NOT_INITIALIZED
The HSA runtime has not been initialized.

HSA_STATUS_ERROR_INVALID_AGENT
The agent is invalid.

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

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

HSA_STATUS_ERROR_INVALID_ARGUMENT
image_descriptor is NULL, *access_permission* is not a valid access permission value, or *image_data_info* is NULL.

3.3.1.10.1 Description

The optimal image data size and alignment requirements may vary depending on the image attributes specified in *image_descriptor*. Also, different implementation of the HSA runtime may return different requirements for the same input values.

The implementation must return the same image data requirements for different access permissions with exactly the same image descriptor as long as **hsa_ext_image_get_capability** reports HSA_EXT_IMAGE_CAPABILITY_ACCESS_INVARIANT_DATA_LAYOUT for the geometry and image format contained in the image descriptor.

3.3.1.11 hsa_ext_image_create

Creates a agent-defined image handle from an implementation-independent image descriptor and a agent-specific image data.

Signature

```
hsa_status_t hsa_ext_image_create(
    hsa_agent_t agent,
    const hsa_ext_image_descriptor_t *image_descriptor,
    const void *image_data,
    hsa_access_permission_t  access_permission,
    hsa_ext_image_t *image);
```

Parameters

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

image_descriptor
(in) Pointer to an image descriptor. Must not be NULL.

image_data
(in) Image data buffer that must have been allocated according to the size and alignment requirements dictated by **hsa_ext_image_data_get_info**. Must not be NULL.

access_permission
(in) Access permission of the image by the agent. The access permission defines how the agent expects to use the image and must match the corresponding HSAIL image handle type. The agent must support the image format specified in *image_descriptor* for the given permission.

image
(out) Pointer to a memory location where the HSA runtime stores the newly created image handle. Must not be NULL.

Return Values

HSA_STATUS_SUCCESS
The function has been executed successfully.

HSA_STATUS_ERROR_NOT_INITIALIZED
The HSA runtime has not been initialized.

HSA_STATUS_ERROR_INVALID_AGENT
The agent is invalid.

HSA_EXT_STATUS_ERROR_IMAGE_FORMAT_UNSUPPORTED
The agent does not have the capability to support the image format contained in the image descriptor using the specified access permission.

HSA_STATUS_ERROR_OUT_OF_RESOURCES
The HSA runtime cannot create the image because it is out of resources (for example, the agent does not support the creation of more image handles with the given access permission).

HSA_STATUS_ERROR_INVALID_ARGUMENT
image_descriptor is NULL, *image_data* is NULL, *access_permission* is not a valid access permission value, or *image* is NULL.

Description

Image created with different access permissions but the same image descriptor can share the same image data if HSA_EXT_IMAGE_CAPABILITY_ACCESS_INVARIANT_DATA_LAYOUT is reported by **hsa_ext_image_get_capability** for the image format specified in the image descriptor. Images with a s-form channel order can share the same image data with other images that have the corresponding non-s-form channel order, provided the rest of their image descriptors 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.

3.3.1.12 hsa_ext_image_destroy

Destroy an image previously created using **hsa_ext_image_create**.

Signature

```
hsa_status_t hsa_ext_image_destroy(
    hsa_agent_t agent,
```

```
    hsa_ext_image_t image);
```

Parameters

agent
(in) Agent associated with the image.

image
(in) Image.

Return Values

HSA_STATUS_SUCCESS
The function has been executed successfully.

HSA_STATUS_ERROR_NOT_INITIALIZED
The HSA runtime has not been initialized.

HSA_STATUS_ERROR_INVALID_AGENT
The agent is invalid.

Description

Destroying the image handle does not free the associated image data or modify its contents. The application should not destroy an image while there are references to it queued for execution or currently being used in a kernel.

3.3.1.13 hsa_ext_image_copy

Copies a portion of one image (the source) to another image (the destination).

Signature

```
hsa_status_t hsa_ext_image_copy(
    hsa_agent_t agent,
    hsa_ext_image_t src_image,
    const hsa_dim3_t *src_offset,
    hsa_ext_image_t dst_image,
    const hsa_dim3_t *dst_offset,
    const hsa_dim3_t *range);
```

Parameters

agent
(in) Agent associated with both images.

src_image
(in) Source image. The agent associated with the source image must be identical to that of the destination image.

src_offset
(in) Pointer to the offset within the source image where to copy the data from. Must not be NULL.

dst_image
(in) Destination image.

dst_offset
(in) Pointer to the offset within the destination image where to copy the data. Must not be NULL.

range
(in) Dimensions of the image portion to be copied. The HSA runtime computes the size of the image data to be copied using this argument. Must not be NULL.

Return Values

HSA_STATUS_SUCCESS
The function has been executed successfully.

HSA_STATUS_ERROR_NOT_INITIALIZED
The HSA runtime has not been initialized.

HSA_STATUS_ERROR_INVALID_AGENT
The agent is invalid.

HSA_STATUS_ERROR_INVALID_ARGUMENT
src_offset is NULL, *dst_offset* is NULL, or *range* is NULL.

Description

The source and destination image formats should match, except if the channel type of one of the images is the standard form of the channel type of the other image. For example, it is allowed to copy a source image with a channel type of HSA_EXT_IMAGE_CHANNEL_ORDER_SRGB to a destination image with a channel type of HSA_EXT_IMAGE_CHANNEL_ORDER_RGB.

The source and destination images do not have to be of the same geometry and appropriate scaling is performed by the HSA runtime. It is possible to copy subregions between any combinations of source and destination types, provided that the dimensions of the subregions are the same. For example, it is allowed to copy a rectangular region from a 2D image to a slice of a 3D image.

If the source and destination image data overlap, or the combination of offset and range references an out-out-bounds element in any of the images, the behavior is undefined.

3.3.1.14 hsa_ext_image_region_t

Image region.

Signature

```
typedef struct hsa_ext_image_region_s {
    hsa_dim3_t offset;
    hsa_dim3_t range;
} hsa_ext_image_region_t
```

Data Fields

offset
Offset within an image (in coordinates).

range
Dimensions of the image range (in coordinates). The x, y, and z dimensions correspond to width, height, and depth respectively.

3.3.1.15 hsa_ext_image_import

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

Signature

```
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_t dst_image,
    const hsa_ext_image_region_t *image_region);
```

Parameters

agent
(in) Agent associated with the image.

src_memory
(in) Source memory. Must not be NULL.

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
(in) Destination image.

image_region
(in) Pointer to the image region to be updated. Must not be NULL.

Return Values

HSA_STATUS_SUCCESS
The function has been executed successfully.

HSA_STATUS_ERROR_NOT_INITIALIZED
The HSA runtime has not been initialized.

HSA_STATUS_ERROR_INVALID_AGENT
The agent is invalid.

HSA_STATUS_ERROR_INVALID_ARGUMENT
src_memory is NULL, or *image_region* is 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 *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** image handle can overlap. Overlapping of any of the source and destination memory within the import operation produces undefined results.

3.3.1.16 hsa_ext_image_export

Export the image data to linearly organized memory.

Signature

```
hsa_status_t hsa_ext_image_export(
    hsa_agent_t agent,
    hsa_ext_image_t src_image,
    void *dst_memory,
    size_t dst_row_pitch,
    size_t dst_slice_pitch,
    const hsa_ext_image_region_t *image_region);
```

Parameters

agent
(in) Agent associated with the image.

src_image
(in) Source image.

dst_memory
(in) Destination memory. Must not be NULL.

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) Pointer to the image region to be exported. Must not be NULL.

Return Values

HSA_STATUS_SUCCESS
The function has been executed successfully.

HSA_STATUS_ERROR_NOT_INITIALIZED
The HSA runtime has not been initialized.

HSA_STATUS_ERROR_INVALID_AGENT
The agent is invalid.

HSA_STATUS_ERROR_INVALID_ARGUMENT
dst_memory is NULL, or *image_region* is NULL.

Description

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

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** image handle can overlap. Overlapping of any of the source and destination memory within the export operation produces undefined results.

3.3.1.17 hsa_ext_image_clear

Clear an image to the specified value.

Signature

```
hsa_status_thsa_ext_image_clear(
    hsa_agent_t agent,
    hsa_ext_image_t image,
    const void *data,
    const hsa_ext_image_region_t *image_region);
```

Parameters

agent
(in) Agent associated with the image.

image
(in) Image to be cleared.

data
(in) Clear value array. Specifying a clear value outside of the range that can be represented by an image format results in undefined behavior. Must not be NULL.

image_region
(in) Pointer to the image region to clear. Must not be NULL. If the region references an out-out-bounds element, the behavior is undefined.

Return Values

HSA_STATUS_SUCCESS
The function has been executed successfully.

HSA_STATUS_ERROR_NOT_INITIALIZED
The HSA runtime has not been initialized.

HSA_STATUS_ERROR_INVALID_AGENT
The agent is invalid.

HSA_STATUS_ERROR_INVALID_ARGUMENT
data is NULL, or *image_region* is NULL.

Description

Clearing an image does not perform any format conversion and the provided clear data is directly stored regardless of the image format. The lowest bits of the data (number of bits depending on the image component type) stored in the cleared image are based on the image component order.

The number of elements in *data* should match the number of access components for the channel order of *image*, as determined by the *HSA Programmer's Reference Manual Version 1.0*. A single element is required for HSA_EXT_IMAGE_CHANNEL_ORDER_DEPTH and HSA_EXT_IMAGE_CHANNEL_ORDER_DEPTH_STENCIL, while any other channel order requires 4 elements.

Each element in *data* is a 32-bit value. The type of each element should match the access type associated with the channel type of *image*, as determined by the *HSA Programmer's Reference Manual Version 1.0*:

- HSA_EXT_IMAGE_CHANNEL_TYPE_SIGNED_INT8, HSA_EXT_IMAGE_CHANNEL_TYPE_SIGNED_INT16, and HSA_EXT_IMAGE_CHANNEL_TYPE_SIGNED_INT32 map to int32_t.
- HSA_EXT_IMAGE_CHANNEL_TYPE_UNSIGNED_INT8, HSA_EXT_IMAGE_CHANNEL_TYPE_UNSIGNED_INT16, and HSA_EXT_IMAGE_CHANNEL_TYPE_UNSIGNED_INT32 map to uint32_t.
- Any other channel type maps to a 32-bit float.

3.3.1.18 hsa_ext_sampler_t

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

Signature

```
typedef struct hsa_ext_sampler_s {
    uint64_t handle;
} hsa_ext_sampler_t
```

Data Fields

handle
Opaque handle.

3.3.1.19 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 BRIG type BrigSamplerAddressing.

Signature

```
typedef enum {
    HSA_EXT_SAMPLER_ADDRESSING_MODE_UNDEFINED = 0,
    HSA_EXT_SAMPLER_ADDRESSING_MODE_CLAMP_TO_EDGE = 1,
    HSA_EXT_SAMPLER_ADDRESSING_MODE_CLAMP_TO_BORDER = 2,
    HSA_EXT_SAMPLER_ADDRESSING_MODE_REPEAT = 3,
    HSA_EXT_SAMPLER_ADDRESSING_MODE_MIRRORED_REPEAT = 4
} hsa_ext_sampler_addressing_mode_t;
```

Values

HSA_EXT_SAMPLER_ADDRESSING_MODE_UNDEFINED
Out-of-range coordinates are not handled.

HSA_EXT_SAMPLER_ADDRESSING_MODE_CLAMP_TO_EDGE
Clamp out-of-range coordinates to the image edge.

HSA_EXT_SAMPLER_ADDRESSING_MODE_CLAMP_TO_BORDER
Clamp out-of-range coordinates to the image border.

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

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

3.3.1.20 hsa_ext_sampler_coordinate_mode_t

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

Signature

```
typedef enum {
    HSA_EXT_SAMPLER_COORDINATE_MODE_UNNORMALIZED = 0,
    HSA_EXT_SAMPLER_COORDINATE_MODE_NORMALIZED = 1
} hsa_ext_sampler_coordinate_mode_t;
```

Values

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

HSA_EXT_SAMPLER_COORDINATE_MODE_NORMALIZED
Coordinates are all in the range of 0.0 to 1.0.

3.3.1.21 hsa_ext_sampler_filter_mode_t

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

Signature

```
typedef enum {
    HSA_EXT_SAMPLER_FILTER_MODE_NEAREST = 0,
    HSA_EXT_SAMPLER_FILTER_MODE_LINEAR = 1
} hsa_ext_sampler_filter_mode_t;
```

Values

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

HSA_EXT_SAMPLER_FILTER_MODE_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.

3.3.1.22 hsa_ext_sampler_descriptor_t

Implementation-independent sampler descriptor.

Signature

```
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
```

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.

3.3.1.23 hsa_ext_sampler_create

Create a kernel agent defined sampler handle for a given combination of a (agent-independent) sampler descriptor and agent.

Signature

```
hsa_status_t hsa_ext_sampler_create(
    hsa_agent_t agent,
    const hsa_ext_sampler_descriptor_t *sampler_descriptor,
    hsa_ext_sampler_t *sampler );
```

Parameters

agent
(in) Agent to be associated with the sampler.

sampler_descriptor
(in) Pointer to a sampler descriptor. Must not be NULL.

sampler
(out) Memory location where the HSA runtime stores the newly created sampler handle. Must not be NULL.

Return Values

HSA_STATUS_SUCCESS
The function has been executed successfully.

HSA_STATUS_ERROR_NOT_INITIALIZED
The HSA runtime has not been initialized.

HSA_STATUS_ERROR_INVALID_AGENT
The agent is invalid.

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

HSA_STATUS_ERROR_INVALID_ARGUMENT
sampler_descriptor is NULL, or *sampler* is NULL.

3.3.1.24 hsa_ext_sampler_destroy

Destroy a sampler previously created using **hsa_ext_sampler_create**.

Signature

```
hsa_status_t hsa_ext_sampler_destroy(
    hsa_agent_t agent,
    hsa_ext_sampler_t sampler );
```

Parameters

agent
(in) Agent associated with the sampler.

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

Return Values

HSA_STATUS_SUCCESS
The function has been executed successfully.

HSA_STATUS_ERROR_NOT_INITIALIZED
The HSA runtime has not been initialized.

HSA_STATUS_ERROR_INVALID_AGENT
The agent is invalid.

3.3.1.25 hsa_status_t images constants

Enumeration constants added to hsa_status_t by this extension.

Signature

```
enum {
    HSA_EXT_STATUS_ERROR_IMAGE_FORMAT_UNSUPPORTED = 0x3000,
    HSA_EXT_STATUS_ERROR_IMAGE_SIZE_UNSUPPORTED = 0x3001
};
```

Values

HSA_EXT_STATUS_ERROR_IMAGE_FORMAT_UNSUPPORTED
Image format is not supported.

HSA_EXT_STATUS_ERROR_IMAGE_SIZE_UNSUPPORTED
Image size is not supported.

3.3.1.26 hsa_agent_info_t images constants

Enumeration constants added to hsa_agent_info_t by this extension. The value of any of these attributes is undefined if the agent is not a kernel agent, or the implementation does not support images.

Signature

```
enum {
    HSA_EXT_AGENT_INFO_IMAGE_1D_MAX_ELEMENTS = 0x3000,
    HSA_EXT_AGENT_INFO_IMAGE_1DA_MAX_ELEMENTS = 0x3001,
    HSA_EXT_AGENT_INFO_IMAGE_1DB_MAX_ELEMENTS = 0x3002,
    HSA_EXT_AGENT_INFO_IMAGE_2D_MAX_ELEMENTS = 0x3003,
    HSA_EXT_AGENT_INFO_IMAGE_2DA_MAX_ELEMENTS = 0x3004,
    HSA_EXT_AGENT_INFO_IMAGE_2DDEPTH_MAX_ELEMENTS = 0x3005,
    HSA_EXT_AGENT_INFO_IMAGE_2DADEPTH_MAX_ELEMENTS = 0x3006,
    HSA_EXT_AGENT_INFO_IMAGE_3D_MAX_ELEMENTS = 0x3007,
    HSA_EXT_AGENT_INFO_IMAGE_ARRAY_MAX_LAYERS = 0x3008,
    HSA_EXT_AGENT_INFO_MAX_IMAGE_RD_HANDLES = 0x3009,
    HSA_EXT_AGENT_INFO_MAX_IMAGE_RORW_HANDLES = 0x300A,
    HSA_EXT_AGENT_INFO_MAX_SAMPLER_HANDLERS = 0x300B
};
```

Values

HSA_EXT_AGENT_INFO_IMAGE_1D_MAX_ELEMENTS
Maximum number of elements in 1D images. Must be at most 16384. The type of this attribute is uint32_t.

HSA_EXT_AGENT_INFO_IMAGE_1DA_MAX_ELEMENTS
Maximum number of elements in 1DA images. Must be at most 16384. The type of this attribute is uint32_t.

HSA_EXT_AGENT_INFO_IMAGE_1DB_MAX_ELEMENTS
Maximum number of elements in 1DB images. Must be at most 65536. The type of this attribute is uint32_t.

HSA_EXT_AGENT_INFO_IMAGE_2D_MAX_ELEMENTS
: Maximum dimensions (width, height) of 2D images, in image elements. The X and Y maximums must be at most 16384. The type of this attribute is uint32_t[2].

HSA_EXT_AGENT_INFO_IMAGE_2DA_MAX_ELEMENTS
: Maximum dimensions (width, height) of 2DA images, in image elements. The X and Y maximums must be at most 16384. The type of this attribute is uint32_t[2].

HSA_EXT_AGENT_INFO_IMAGE_2DDEPTH_MAX_ELEMENTS
: Maximum dimensions (width, height) of 2DDEPTH images, in image elements. The X and Y maximums must be at most 16384. The type of this attribute is uint32_t[2].

HSA_EXT_AGENT_INFO_IMAGE_2DADEPTH_MAX_ELEMENTS
: Maximum dimensions (width, height) of 2DADEPTH images, in image elements. The X and Y maximums must be at most 16384. The type of this attribute is uint32_t[2].

HSA_EXT_AGENT_INFO_IMAGE_3D_MAX_ELEMENTS
: Maximum dimensions (width, height, depth) of 3D images, in image elements. The maximum along any dimension cannot exceed 2048. The type of this attribute is uint32_t[3].

HSA_EXT_AGENT_INFO_IMAGE_ARRAY_MAX_LAYERS
: Maximum number of image layers in a image array. Must not exceed 2048. The type of this attribute is uint32_t.

HSA_EXT_AGENT_INFO_MAX_IMAGE_RD_HANDLES
: Maximum number of read-only image handles that can be created at any one time. Must be at least 128. The type of this attribute is uint32_t.

HSA_EXT_AGENT_INFO_MAX_IMAGE_RORW_HANDLES
: Maximum number of write-only and read-write image handles (combined) that can be created at any one time. Must be at least 64. The type of this attribute is uint32_t.

HSA_EXT_AGENT_INFO_MAX_SAMPLER_HANDLERS
: Maximum number of sampler handlers that can be created at any one time. Must be at least 16. The type of this attribute is uint32_t.

APPENDIX A. Glossary

agent

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

Architected Queuing Language (AQL)

An AQL packet is an HSA-standard packet format. AQL kernel dispatch packets are used to dispatch kernels on the kernel agent and specify the launch dimensions, instruction code, kernel arguments, completion detection, and more. Other AQL packets control aspects of a kernel agent such as when to execute AQL packets and making the results of memory instructions visible. AQL packets are queued on User Mode Queues. See *HSA Platform System Architecture Specification Version 1.0* .

AQL packet

User-mode buffer with a specific format (determined by the Architected Queuing Language) that encodes one command.

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. A kernel agent is composed of one or more compute units.

finalizer

A finalizer is part of the HSA runtime and translates HSAIL code in the form of BRIG into an HSA code object that contains the appropriate native machine code for a kernel agent that is part of an HSA system. When an application uses the HSA runtime it can optionally include the finalizer.

global segment

A memory segment in which memory is visible to all work-items in all kernel agents 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 agent that also supports the native CPU instruction set and runs the host operating system and the HSA runtime. As an agent, the host CPU can dispatch commands to a kernel agent using memory instructions to construct and enqueue AQL packets. In some systems, a host CPU can also act as a kernel agent (with appropriate HSAIL finalizer and AQL mechanisms).

HSA application

A program written in the host CPU instruction set. In addition to the host CPU code, it may include zero or more HSAIL programs.

HSA implementation

A combination of one or more host CPU agents able to execute the HSA runtime, one or more kernel agents able to execute HSAIL programs, and zero or more other agents that participate in the HSA memory model.

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: support for User Mode Queues, signals and memory management; optional support for images and samplers; a finalizer; and a loader. See the *HSA Runtime Programmer's Reference Manual*.

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 a compute unit. Kernels are written in HSAIL and then separately translated by a finalizer to the target instruction set.

kernel agent

An agent that supports the HSAIL instruction set and supports execution of AQL kernel dispatch packets. As an agent, a kernel agent can dispatch commands to any kernel agent (including itself) using memory instructions to construct and enqueue AQL packets. A kernel agent is composed of one or more compute units.

packet ID

Each AQL packet has a 64-bit packet ID unique to the User Mode Queue on which it is enqueued. The packet ID is assigned as a monotonically increasing sequential number of the logical packet slot allocated in the User Mode queue. The combination of the packet ID and the queue ID is unique for a process.

packet processor

Packet processors are tightly bound to one or more agents, and provide the functionality to process AQL packets enqueued on User Mode Queues of those agents. The packet processor function may be performed by the same or by a different agent to the one with which the User Mode Queue is associated that will execute the kernel dispatch packet or agent dispatch packet function.

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 by an `rdimage` image instruction.

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 executing on a single instruction pointer.

work-group

A collection of work-items from the same kernel dispatch.

work-item

The simplest element of work. Another name for a unit of execution in a kernel dispatch.

Index

F

G

H