HSA Core API Programmers Reference Manual

0.179

April 7, 2014

Draft

Contents

1 Introduction

1.1 Overview

Recent system-on-a-chip designs have integrated CPU, GPU, and other accelerator devices onto a single chip with a shared high-bandwidth memory system. In fact, these single-chip 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 includes requirements for virtual memory, memory coherency, architected dispatch mechanisms, and power-efficient signals. HSA refers to these accelerators as "components". The system architecture defines a consistent base for building portable applications that access the power and performance benefits of the dedicated HSA components. Many of these components, including GPUs and DSPs, are capable and flexible processors that have been extended with special hardware for accelerating parallel code. Historically these devices have been difficult to program due to a need for specialized or proprietary programming languages. HSA aims to bring the benefits of these components to mainstream programming languages using similar or identical syntax to that which is provided for accessing multi-core CPUs.

In addition to the system architecture, HSA's "Programmer's Reference Guide" defines HSAIL - a portable, low-level, compiler intermediate language designed for parallel computing.

Portable: The HSAIL language is an open-standard, supported by multiple vendors in the HSA Foundation, and is portable across vendors and product generations, so that applications which use HSAIL are guaranteed to run on future hardware that supports the HSAIL standard.

Low-level: HSAIL's representation is just above the machine instruction set. Most optimizations including register allocation are intended to be performed by the compiler that generates HSAIL. HSAIL code is translated to the host instruction set by a tool called the "finalizer". Each component will provide its own implementation of the finalizer. The finalization step is intended to be lightweight, fast, and simple. Importantly, the finalizer step does not involve a complex compiler. Applications which contain HSAIL should not see different functional or performance behavior from new finalizer versions that might be deployed in the field after the application ships.

Designed for parallel computing: HSAIL is intended to represent the parallel sections of an application. It complements but does not replace the host code âĂŞ host code still exists and is used for the

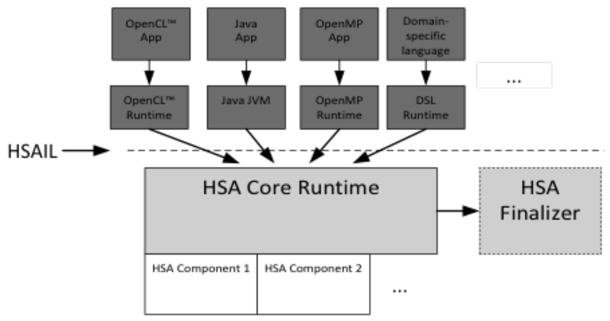

Figure 1.1: HSA Software Architecture

serial portion of the application. HSAIL represents a single "lane" of execution, and the parallelism is expressed in the grid dimensions that are specified when the HSAIL kernel is dispatched to a target component. In this way, HSAIL does not encode a specific "vector width" and can be used to represent a variety of different parallel computing devices.

For more information on HSAIL, refer to the HSA Programmer's Reference Guide.

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

- Error handling
- Runtime initialization (open/close)
- Topology Discovery
- Signals and Synchronization
- Architected Dispatch
- Memory Management

In summary, there are three specifications provided by the HSA Foundation:

HSA System Architecture Requirements: Architectural foundation for how HSA components share memory and communicate work requests.

HSA Programmer's Reference Manual: Describes HSAIL, a low-level, portable compiler IR appropriate for use as compiler intermediate language.

HSA Runtime Specification: This document. Describes the host-side API for controlling the launch of HSAIL kernels.

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

Figure 1.1 shows how the HSA Core Runtime fits into a typical software architecture stack.

At the top of the stack is a programming model such as OpenCLâĎć, 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 has the stream and lambda APIs, which provide support for both multi-core CPUs and

HSA Components. OpenMP contains OMP pragmas that mark parallel for loops and 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. HSA supports several options for when HSAIL is generated and finalized. One possibility is that the HSAIL is generated by a high-level compiler and then embedded in the application binary. In this case, the finalizer is run when the application loads and will convert the HSAIL to machine code for the target machine. Another option is that the HSAIL is finalized when the applications is built, or the machine cod The HSA Finalizer is an optional component of the HSA Core Runtime, which may 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 Core 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 work to the HSA Component. The language runtime is also responsible for initializing HSA, and may utilize other HSA core runtime features as well.

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

1.2 Execution Model

1.2.1 Architected Dispatch

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

In order for user-level applications to use the HSA system and HSA components, they need to write HSAIL programs and compile and execute these programs using user mode queues and AQL commands. The HSA Programmer's Reference Manual (PRM) defines HSAIL Virtual ISA and Programming Model, serves as a Compiler Writer's Guide, and defines Object Format (BRIG). The HSA runtime helps setup the execution via API calls and data structures to support architected features.

The HSA core runtime realizes architected dispatch. Architected dispatch is the key feature in an HSA system that enables a user-level application to directly issue commands to the HSA Component hardware. Architected dispatch differentiates it from other higher-level runtime systems and programming models: other runtime systems provide software APIs for setting arguments and launching kernels, while HSA architects these at the hardware and specification level. The critical path of the dispatch mechanism is architected at the HSA hardware level and can be done with regular memory operations and runtime provided wrapper API. Fundamentally, the user creates user mode queues and an AQL Packet in memory, and then signals the HSA component to begin executing the packet using light weight operations (which may be wrapped with API calls).

This section describes various features core runtime provides to support architected dispatch as steps that a user needs to take to utilize runtime.

1.2.2 Initial Setup

One of the first steps in the setup is that of device discovery. Device discovery is performed at the initialization of the core runtime and information is made available to the user as data structures. Section 2.3 describes these structures. The next step in the setup is creation of the component queues. Queues are an HSA architected mechanism to submit work to the HSA component HW. The interfaces for queue creation are defined in Section 2.5. Different components may provide implementation-specific code under the core API for these functions. HSA runtime also includes mechanisms to provide implementation-specific data as part of the dispatch, provided such data can be computed at compile time.

1.2.3 Compilation Flow

Once an HSAIL program is written or generated by a higher-level compilation step, it needs to be *assembled* to generate a BRIG. BRIG is the HSAIL object format and is specified in the PRM. HSA runtime defines API call to compile the BRIG and generate a code object that has sufficient information to execute the user program. The details of this compilation process and symbol resolution are discussed in Section 3.1.

1.2.4 Execution of Kernel

The Systems Architecture Requirements (SAR) document specifies the structure of the *packets* (i.e. commands) that can be placed on the HSA user mode queues for the component HW to execute them. The format of the packets is architected and they are referred to as Architected Queuing Language (AQL) packets. One of the types of AQL packets is a dispatch AQL packet. The user can now create an AQL packet and initialize it with the code object obtained from the finalization step, including the allocation of memory to hold the kernel arguments and the spill/arg/private memory. The interface for kernel arguments between the runtime and the kernel ISA (instruction set architecture) is also architected at the HSA level. This is covered in the HSAIL ABI, which specifies the in-memory layout of the kernarg segment. Users can determine the layout of the kernarg memory segment at compile time merely by examining the signature of the HSAIL function. The finalizer is required to support this ABI and thus there is no need for runtime metadata to specify the position or format of arguments. This step can be done once for each AQL packet creation.

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

HSA HW defines a packet process for processing these packets and a doorbell mechanism to inform the packet processing HW that packets have been written into the queue. The Core runtime defines a structure and update API to inform the HW that the dispatch packet has been written to the queue. Different packet formats and states of a packet are discussed in Section 2.6. Section 2.5 discusses the queue creation and various states the queue can be in, once it is created.

Once the packet is written and the HW is informed by way of the doorbell, the execution can start. The execution happens asynchronously. The user is free to write more packets for executing other kernels in the queue. This activity can overlap the actual execution of the kernel.

1.2.5 Determining Kernel Completion

HSA SAR defines signals as a mechanism for communication between different parts of a HSA system. Signals are defined as opaque objects in the HSA core runtime and APIs have been defined to send a value to the signal and wait for a value at the signal, Section 2.4 discusses signals in detail. The AQL dispatch packet has a provision for the user to pass in an opaque signal. When the HSA Component HW observes a valid signal in the AQL packet, it sends a value to this signal when execution of the kernel is complete (success or error). The user can wait on this signal to determine kernel completion. Errors and their meaning are discussed in Section 2.2.

```
int main(int argc, char **argv)
{
  unsigned int count;
  hsa_brig_direcve_offset_t code_directive_offset = atoi(argv[2]);
  hsa_control_directives_t control = atoi(argv[3]);
  uint64_t kernel_input = (uint64_t)atoi(argv[4]);
  const hsa_agent_t *component = NULL;
  static hsa_aql_dispatch_packet_t dispatch_packet;
  hsa_runtime_context_t *runtime_context;
  hsa_status_t status;
  hsa_queue_t *queue;
  hsa_brig_t *brig = (hsa_brig_t *)argv[1];
  hsa_code_object_t *code_obj;
  hsa_symbol_map_t *symbol_map;
  hsa_debug_info_t *debug_info;
  hsa_signal_handle_t signal;
  hsa_signal_value_t sigval;
  hsa_topology_header_t *header;
  hsa_platform_t *platform;

  uint64_t index;
```

```
/**** this part is the setup for a simple dispatch ****/
status = hsa_open(&runtime_context);
assert(status == HSA_STATUS_SUCCESS);

hsa_context_acquire(&runtime_context);

hsa_close(&runtime_context);

status = hsa_topology_table_create(&header);
if(status != HSA_STATUS_SUCCESS) {
  assert(HSA_STATUS_SUCCESS == hsa_close());
  exit(1);
}

platform = (hsa_platform_t *)(header->topology_table_base);

component = (hsa_agent_t *)((char *)(header->topology_table_base) +
    platform->agent_offset_list[0]);

status = hsa_queue_create(component, 1024, 0, NONE, runtime_context, &queue, NULL);
if(status != HSA_STATUS_SUCCESS) {
  assert(HSA_STATUS_SUCCESS == hsa_close());
  exit(1);
}

/**** this is the compilation part where the brig is finalized ****/
status = hsa_finalize_brig(component, brig, 1, &code_directive_offset, &control,
    NULL, &code_obj, &debug_info, &symbol_map);
if(status != HSA_STATUS_SUCCESS || symbol_map != NULL) {
  assert(HSA_STATUS_SUCCESS == hsa_queue_destroy(queue));
  assert(HSA_STATUS_SUCCESS == hsa_close());
  exit(1);
}

/**** a signal is created for completion detection ****/
sigval.value64 = 0;
status = hsa_signal_create(sigval, &signal, runtime_context);
if(status != HSA_STATUS_SUCCESS) {
  assert(HSA_STATUS_SUCCESS == hsa_queue_destroy(queue));
  assert(HSA_STATUS_SUCCESS == hsa_close());
  exit(1);
}

/**** the AQL packet is setup here for the simple kernel ****/
dispatch_packet.header.format = 2;
dispatch_packet.header.barrier = 1;
dispatch_packet.header.acquire_fence_scope = 2;
dispatch_packet.header.release_fence_scope = 2;
dispatch_packet.header.dimensions = 1;
dispatch_packet.workgroup_size_x = 256;
dispatch_packet.grid_size_x = 256;
dispatch_packet.kernel_object_address = (uint64_t)code_obj;
dispatch_packet.kernarg_address = (uint64_t)(&kernel_input);
dispatch_packet.completion_signal = signal;

memcpy(queue->base_address, (void *)&dispatch_packet, sizeof(dispatch_packet));

/**** packet processor is informed that a packet is on the queue ****/
index = hsa_queue_set_write_index(queue, 1);
```

```
  if(index != 0){
    assert(HSA_STATUS_SUCCESS == hsa_queue_destroy(queue));
    assert(HSA_STATUS_SUCCESS == hsa_close());
    exit(1);
  }
  sigval.value64 = 1;
  status = hsa_signal_send_release(queue->doorbell_signal, sigval);
  if(status != HSA_STATUS_SUCCESS) {
    assert(HSA_STATUS_SUCCESS == hsa_queue_destroy(queue));
    assert(HSA_STATUS_SUCCESS == hsa_close());
    exit(1);
  }

  /**** await completion ****/
  do {
    status = hsa_signal_wait_acquire(signal,HSA_EQUALS, sigval, &sigval);
  } while(status == HSA_STATUS_INFO_SIGNAL_TIMEOUT);
  if(status != HSA_STATUS_SUCCESS) {
    assert(HSA_STATUS_SUCCESS == hsa_queue_destroy(queue));
    assert(HSA_STATUS_SUCCESS == hsa_close());
    exit(1);
  }

  printf("\nkernel successfully executed, value %llu\n", kernel_input);

  /**** close up and destroy queue, close the runtime ****/
  status = hsa_queue_destroy(queue);
  if(status != HSA_STATUS_SUCCESS) {
    assert(HSA_STATUS_SUCCESS == hsa_close());
    exit(1);
  }
  status = hsa_close();
  assert(status == HSA_STATUS_SUCCESS);
  return 0;
}
```

2 HSA Core Programming Guide

This chapter describes HSA Core runtime API by their functional area. Note that except for any setter/getter API, the remainder of the core runtime API may be considered thread-safe. Both the signal update and the queue index update API are setter/getter API and define scope an synchronization that applies to the updates and operate on structure elements.

Several operating systems allow functions to be executed when a DLL or a shared library is loaded (e.g. 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 are outside the scope of this specification.

Similarly, any header files distributed by the HSA foundation for this spec may contain calling convention specific prefixes such as __cdecl or __stdcall. Such calling conventions are again invocation, usage and implementation specific. Hence, the calling convention specific prefix definition is outside the scope of API definition.

2.1 Initialization and Shutdown

Since the HSA core runtime is a user mode library, its state is a part of the application's process space. When the runtime is opened for the first time, a runtime instance for that application process is created. Closing a runtime destroys this instance. An application may open (or close) the HSA runtime multiple times within the same process and potentially within multiple threads – only a single instance of the runtime will exist for a given process.

The core runtime defines a runtime context that acts as a reference counting mechanism and a scheme to differentiate multiple usages of the runtime within the same application process. The runtime context is generated when the runtime is opened or when a user calls the acquire API that is defined in this Section. As an example, consider an application that is using the runtime but also uses a library, this library also creates HSA queues and submits work to them. Both the library and the application may want to register callbacks, and to capture notifications/errors of their specific usage. The runtime context helps identify the different usages (within the same process) and channel errors and notifications to appropriate callbacks. It also acts as a reference counting mechanism; while correctly acquired, the runtime context ensures that the runtime instance will not be shutdown until the context is released (this, in effect, is the reference counting part of the context).

2.1.1 Example

```
/* The runtime is opened for the first time */
status = hsa_open(&runtime_context);
assert(status == HSA_STATUS_SUCCESS);

/* now the context has been acquired, reference count associated with the
 * context incremented internally by the runtime*/
hsa_context_acquire(&runtime_context);

/* this close is just going to generate an asynchronous error and return the
 * success status */
hsa_close(&runtime_context);
assert(status == HSA_STATUS_SUCCESS);

/* this queue create will still succeed provided component is valid and
 * runtime can allocate required resources*/
status = hsa_queue_create(component, 1024, 0, NONE, runtime_context, &queue, NULL);
if(status != HSA_STATUS_SUCCESS) {
  assert(HSA_STATUS_SUCCESS == hsa_close());
  exit(1);
}

// rest of program...

/* this API will succeed, the context is now as it was after the hsa_open */
hsa_context_release(&runtime_context);
assert(status == HSA_STATUS_SUCCESS);

/* this close will attempt to release all runtime allocated releases,
 * components, topology table, queues, etc. are no longer valid */
hsa_close(&runtime_context);
assert(status == HSA_STATUS_SUCCESS);

/* this queue create will fail*/
status = hsa_queue_create(component, 1024, 0, NONE, runtime_context, &queue1, NULL);
if(status != HSA_STATUS_SUCCESS) {
  exit(1);
}
```

2.2 Errors and Notifications

Errors reported by the runtime can be synchronous or asynchronous. Synchronous errors are always reported when the call returns. They indicate if the API returned a success or an error. Asynchronous errors can occur due to various reasons: (i) Activity in packet processor, executing kernels, their actions and memory accesses. If an error is detected during execution of a kernel, the completion signal (if present) will be signaled with an error indication value. (ii) To provide *information/warning* (not as an exception in expected behavior but by definition). This information/warning may not necessarily indicate an error. For example, a timeout may be an acceptable response for a wait API but is not indicative of a failure.

2.2.1 Synchronous Errors

When a core runtime API is called by the user and does not execute successfully, the core runtime returns a status that can help determine a cause of the unsuccessful execution. Each API call discussed in this chapter defines what constitutes a successful execution. While a few error conditions can be generalized to a certain degree (e.g. failure in allocating system memory) many errors can have system/implementation specific explanations.

The HSA core runtime API defines an enumeration that captures the result of any API function that has been executed (the only exception to this behavior are setter/getter API that access core runtime structures). This enumeration is of the type hsa_status_t and enumerates *success*, *info*, and *error*. The *info* status definition is discussed in Section 2.2.2.

Success status is a single constant, HSA_STATUS_SUCCESS, with value 0. Description of every core runtime API call that returns hsa_status_t explains the expected successful behavior for that API.

Error status could be due to user input/actions that are not allowed (e.g. negative value in a size for allocation) or systemic errors (e.g. an asynchronous activity lead to a failure that cascaded into a failure in this API). The constants used for error status are restricted to the negative range of values within the hsa_status_t enumeration. Errors must always have a negative value. The name of any constant that indicates an error status is prefixed by HSA_STATUS_ERROR.

While the name of the constant in itself is informative for success, info or error status, there may be scenarios where (i) the user may request more information about the meaning of a particular status, or, (ii) the return status was implementation specific and the user needs to decode it. In the case of implementation specific status, the negative number returned for error may not correspond to a particular enumeration constant. To query additional information on synchronous errors, the core runtime provides the **hsa_status_query_description** API.

2.2.2 Asynchronous Errors and Notifications

The HSA core runtime supports user-defined callbacks to handle asynchronous errors. There are two different categories of callbacks that can be registered by the user: (i) for asynchronous information or warnings generated when the runtime is executing, or, (ii) for asynchronous errors that get generated in packet processor, or while executing a kernel . The core runtime supports a callback each for asynchronous errors and notifications. The user 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 user cannot depend on thread local storage within the callbacks implementation and may safely kill the thread that registers the callback. It is the user's responsibility to ensure that the callback function is thread-safe. The runtime does not implement any default callbacks.

The information/warning status is represented by a value greater than 0 within the hsa_status_t enumeration. The status is up to user interpretation and the runtime allows the user to register a callback to take necessary action. Consider the example where a user calls the initialize API to initialize the core runtime and the return status is HSA_STATUS_INFO_ALREADY_INITIALIZED (to indicate that the core runtime has already been initialized). This result may be interpreted differently in different usage scenarios. A callback for such notifications may be registered via **hsa_open** API discussed in Section 2.1 or via **hsa_notification_-callback_register** API.

The HSA system can have several queues in operation and several kernels executing from these queues asynchronously. When any asynchronous activity generates an error, the action that initiated the activity may have concluded. To deal with asynchronous errors, the core runtime supports asynchronous error callbacks. The asynchronous error callback may be registered by means of the **hsa_open** API discussed in Section 2.1 or via **hsa_error_callback_register** API. Details on how association of the callback can be done with asynchronous activities are discussed in Sections 2.1 and 2.5.

2.2.3 Asynchronous Notification Example

```
void error_callabck(hsa_async_error_info_t *error)
{
  printf("An error has occured at: %" PRIu64 "\n", error->timestamp);
  printf("%s \n",error->string_info);
  if(error->error_type == HSA_STATUS_ERROR_QUEUE_WRITE_INDEX_UPDATE_ERROR ||
      HSA_STATUS_QUEUE_FULL){
    printf("Queue update/full error\n");
  }
}


{
  /* The runtime is opened for the first time */
  status = hsa_open(&runtime_context);
  assert(status == HSA_STATUS_SUCCESS);

  /* error_callback is of type void error_callback(hsa_async_error_info_t
   * *info)
   */
  hsa_error_callback_register(&error_callback, NULL, &runtime_context);

  /* this queue create will succeed provided component is valid and
   * runtime can allocate required resources*/
  status = hsa_queue_create(component, 1024, 0, NONE, runtime_context, &queue, NULL);
  if(status != HSA_STATUS_SUCCESS) {
    assert(HSA_STATUS_SUCCESS == hsa_close());
    exit(1);
  }

  // rest of program
}
```

2.3 Topology Discovery

Topology discovery is provided by the runtime so users can programmatically query information about the available agents, memories, etc. This information could be utilized by the user in different ways, including decisions on where to execute a particular task. The runtime specification defines the topology table data structure and other data structures to represent topology hierarchy.

A copy of the topology information can be created using **hsa_topology_table_create** after the runtime has been initialized. The user can parse this table representing the HSA system to gather details such as the number of different agents on the system with local access to a particular set of memory resources. The topology table is designed to be allocated in a block of contiguous memory.

The table header structure includes the platform structure (hsa_platform_t). The platform information in the platform structure includes size/offset array pairs for HSA agents (hsa_agent_t), memory (hsa_memory_descriptor_t) and cache (hsa_cache_descriptor_t). The platform can have a hierarchical structure with multiple agents and physical memories. The hsa_platform_t structure also includes properties such as the clock frequency that are common across the system and also links to various elements in the topology table. Platform structure maps to the agents, cache and physical memory in the topology table for all nodes in the platform.

The user must destroy the topology table before closing the runtime. The **hsa_topology_table_destroy** API is defined by the runtime for the user to destroy the topology table. Once a table is created, some parts of it may become invalid if any HW is hot-plugged/unplugged or encounters an error. If such a change occurs, the HSA runtime generates an asynchronous error (see Section 2.2.2) with the hsa_status_t enumeration of HSA_ERROR_TOPOLOGY_CHANGE. This is an indication to the user that any current usage of topology table must be stopped and a new topology table obtained by using the **hsa_topology_table_create** API call. The runtime guarantees that any call made to **hsa_topology_table_create** API after the asynchronous error is observed will return the latest version of the topology table at the time of the API invocation. However, if the same HW was hot-swapped out and in with the same interval, or if the error encountered in a component was recovered, the topology table may look unchanged.

2.3.1 Example

TODO

2.4 Signals

In a HSA system, (coherent) global memory can serve as a means for message passing, asynchronous communication or synchronization between agents. A signal is an alternative communication mechanism, possibly more power efficient. A signal carries a value, which can be updated or conditionally waited upon via an API call or HSAIL instruction.

Signals may be utilized in many ways. For example, a running kernel, after it finishes producing a part of its computation, may set the signal in the dependency packet of another kernel dispatch so that the queue processor can resolve the dependency and launch the second kernel. Signals cannot be used for Inter-Process Communication.

Signal are represented by opaque signal handlers; signal values are represented using four or eight bytes, depending on the machine model in use. To put the signal in error state, the two most significant bits in the signal value are set and all other bits cleared. It is the users burden to check if an error has occurred by looking at the return code of the **hsa_signal_wait** invocation. Any negative value at the signal triggers the HSA_STATUS_ERROR return code from the wait API. A signal that is already in error may further be decremented to a larger negative value.

Once a signal is created for a particular context, it may be bound to other contexts. This is useful when signal is used across different components of a users application.

Sending a signal entails updating a particular value at the signal. In addition to the update of signals using Send, the API for send signal must support other atomic operations as well: *AND, OR, XOR, Exchange, Add, Subtract, Increment, Decrement, Maximum, Minimum* and *CAS*. Each operation on a signal value has the type of synchronization explicitly included in its name. For example, Send-Release is a Send on a signal value with Release synchronization. The set of (action, synchronization) signal modifiers available in the API match the corresponding HSAIL instructions [1]. For efficiency, a unique signal API has been created for each of these actions.

Acquire-Release synchronization Exchange, Maximum

Release synchronization Send, CAS, AND, OR, XOR, Add, Substract, Increment, Decrement

Relaxed synchronization Send, Exchange, AND, OR, XOR, Add, Substract, Increment, Decrement, Maximum, Minimum

Waiting on a signal returns the current value at the opaque signal object. The wait has a runtime defined timeout which indicates the maximum amount of time that an implementation can spend waiting. The signal infrastructure allows for multiple senders/waiters on a single signal.

The user may wait on a signal, with a condition specifying the terms of wait. The wait can be done either in the HSA Component via an HSAIL wait instruction or via a runtime API. Wait *reads* the value, hence Acquire and Acquire-Release synchronizations may be applied to the read. The synchronization should only assume to have been applied if the status returned by the wait API indicates a success (i.e. return value is HSA_STATUS_SUCCESS)

2.4.1 Example

TODO

2.5 Queues

HSA hardware supports kernel dispatch through user mode queues. A queue is associated with a specific component, which might have several queues attached to it. Two queue types are supported: queues which can consume any kind of AQL packets (discussed in Section 2.6), and service queues. A service queue consumes agent dispatch packets that are used to specify runtime-defined or user registered functions that will be executed on the agent (typically, the host CPU).

Agents write AQL packets to the user mode queue of a particular component. The queue memory is processed by a packet processor as though it is a ring buffer. The details on how commands can be written to the queue via AQL packets and the structure of the AQL packet are discussed in Section 2.6.

The HSA runtime allows the user to create a user mode queue by invoking **hsa_queue_create**, which is responsible for allocating memory to hold AQL packets. The pointer to the beginning of the allocated memory is stored in the *base_address* field. No memory shall be allocated by an implementation if the queue creation fails. An implementation might not initialize the hsa_queue_t structure if queue creation fails, so the user should only rely on the error code returned to determine if the queue is valid.

The hsa_queue_mailbox_t structure returned by the queue create call contains *mailbox_ptr* and *mailbox_-signal*. Their purpose is getting execution information when a **debugtrap_u32** HSAIL instruction is used in the user kernel. The user can wait on the *mailbox_signal* and process the information in the *mailbox_ptr* as discussed in Section 2.8.

This service queue is configured when a user mode queue is created. The service queue is visible to HSA agents through the queue structure *service_queue* field and is serviced by an appropriate HSA agent. The application may chose to not use a service queue, select the runtime managed service queue, or a queue managed by the application via *service_queue_type*. Address of the service queue associated with the user mode queue is returned in the queue structure. If there is no associated service queue then the NULL address will be returned. The API allows different user mode queues to have a different associated service queue. It also allows for the service queue to be user managed. The API allows the user to specify that runtime return a default shared service queue which is created when the runtime is initialized.

Internally, the queue structure contains a read index and a write index. The write index is a unique identifier for AQL packets in the queue. The read index indicates the next packet that will be consumed by the packet processor. Both indexes are not directly exposed to the user, who can only access them by using dedicated APIs. The available index functions differ on the index of interest (read or write), action to be performed (addition, compare and swap, etc.), and memory order (relaxed, release, etc.).

The read index is automatically advanced when a packet is read by the packet processor. When the agent observes that the read index matches the write index, the queue can be considered empty (it does not mean that the kernels have finished execution, just that all packets have been consumed). The write index and the read index never wrap when the write index reaches its maximum value, but an asynchronous error is generated by the packet processor and the queue is put in error state.

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

2.5.1 Queue States

There are two primary kinds of errors that impact queue processing and render a queue inactive:

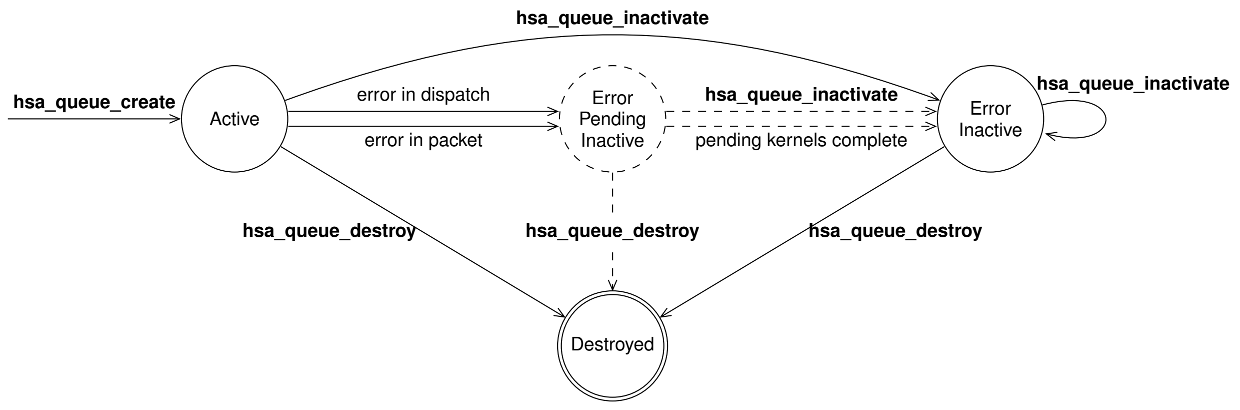

Figure 2.1: Queue state diagram.

- Errors due to packet processing, such as invalid format, field-value, invalid signal, etc.
- Errors occurring during subsequent resource/dispatch setup or system errors during dispatch.

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

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

Error pending inactive When packet processing or dispatch setup encounters one of the errors described above, the queue packet processor stops packet processing. At this point, there might be in-flight kernels and resources (such as segment allocation) that have been setup for a dispatch but have not yet been freed. So the queue is not entirely inactive, but once the asynchronous activity concludes, it will become inactive. A queue in *error pending inactive* state is not to be considered as destroyed, it still needs to be destroyed so the runtime can reclaim the memory allocated for this queue. If the user provides a callback at queue creation time, the callback is invoked after the queue is marked inactive.

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

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

The queue will report packet processing or parsing error, system error, dependency resolution error, and signaling error (signal destroyed by the time it needed to be signaled by packet processor).

The queue error reporting infrastructure supports and reports a single error per queue and attempts to inactivate the queue on the first error it encounters.

2.5.2 Example

The following sample code below shows how multiple threads can safely add packets to a queue.

```
// read the current queue write offset via intrinsic
write_index = hsa_queue_get_write_index(q);

// wait until the queue is no longer full.
while (write_index == read_index + size);

// atomically bump the WriteOffset via intrinsic
if (hsa_queue_cas_write_index(q, write_index, write_index + 1) == write_index)
{
  // calculate index
  uint32_t index = write_index & (size - 1);

  // copy over the packet, the format field is INVALID
  memcpy(q->base_address + index, pkt);

  // update format field with release semantics
  q->base_address[index].hdr.format.store(DISPATCH, std::memory_order_release);

  // ring doorbell with release semantics (could also amortize over multiple packets)
  ring_doorbell(write_index + 1);
}
```

2.6 Architected Queuing Language Support

AQL is a command interface for describing a dispatch or a dependency in a standard format for the queue packet processor.The HSA API declares structures for the different types of AQL packets described in [2]: always reserved, invalid, dispatch, agent dispatch and barrier.

2.6.1 Dispatch packet

A dispatch packet is used to submit tasks to a HSA component. It can have five different states: *on queue*, *launch*, *error*, *active* or *complete*. Figure 2.2 shows the different states of a packet and transitions leading to those states.

On queue A packet is considered to be in the on queue state once the format of the packet is changed from invalid (a value of 0) to a value of 1, 2 or 3. Any other value for format puts the packet and the queue in error state.

Launch If this dispatch packet has the barrier bit set, then the processing of this packet occurs only after all prior kernels have completed execution. Otherwise, once the packets prior to this packet are processed, the packet processor begins to process this packet and the packet enters the active state.

Error the packet processor encountered an error processing this packet. This results in a queue error (see Figure 2.1) and the packet enters the error state (the completion object is signaled with error by the packet processor). The following errors are indicated via an error signaled to the completion object: processing parsing error, dependency resolution error, system error and premature termination due to queue inactivation. When the user invokes the **hsa_queue_inactivate** API or the **hsa_queue_destroy** API while the packet is in this state, the completion object will be signaled with an error.

Active If the packet processing is successful and the kernel the packet represents is either executing or queued for execution, the packet enters the active state. From active state, either successful or failed execution both take the packet into the completed state. Alternatively, queue inactivation can also take the packet out of active state into complete state. When the user invokes the **hsa_queue_inactivate** API or the **hsa_queue_destroy** API while the packet is in this state, the completion object will be signaled with an error.

Complete A packet enters a complete state after its completion signal is signaled (either with success or error).

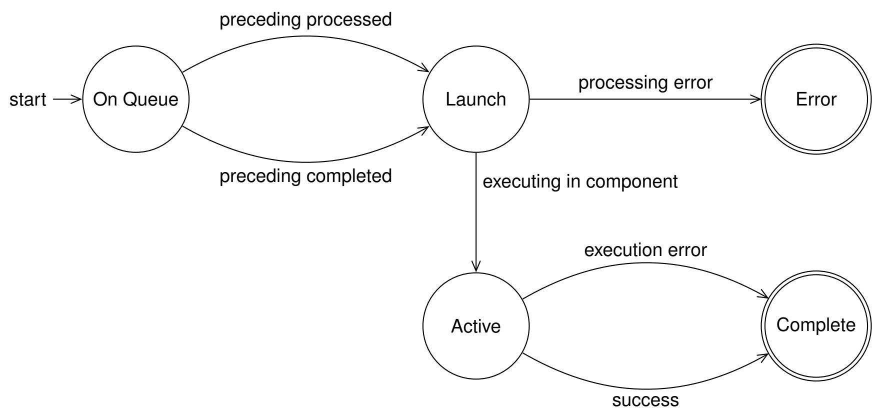

Figure 2.2: Dispatch Packet State Diagram

A dispatch packet is considered processed once the packet processor processes it and makes the queue slot occupied by this packet available. A processed dispatch packet may endure a period of time where it is awaiting its dispatch on to the HSA component. Even such packets awaiting execution are still considered as processed.

2.6.1.1 Segment sizes

If the kernel being dispatched uses private and group segments, the user is required to specify the sizes of these segments in the dispatch packet. Manually calculating this information is not feasible and requires visual inspection of the user program, which itself may have been generated by a higher-level compiler. Hence the user must rely on the finalizer to get the corresponding segment sizes. Further details about determining segment sizes are described in Section 3.1.

Of the other HSA segments, the kernarg segment is also a part of the dispatch packet, but as a pointer. This is because the kernarg segment carries the arguments required to execute the kernel being dispatched and must be setup by the user (the layout of this segment is language/finalization specific and associated with the code object generated by finalization) prior to writing the AQL packet to the queue (unlike the group and private segments, whose lifespan spans only the active state of the dispatch packet). Please refer to the HSAIL service layer for an example on how to setup the kernarg segment for the HSAIL language, which is based on the 32/64 bit modes the kernel is compiled in.

2.6.2 Agent Dispatch packet

Agent Dispatch AQL packets can be used to do dispatches on the agent queue. The HSA Queue API allows for creation of either agent queues or component queues in the core API (vendor-specific extensions may support queues that allow both agent and component dispatches, but it is not a core feature).

2.6.3 Barrier packet

The barrier packet allows the user to specify up to five dependencies as hsa_signal_t objects and requires the packet processor to resolve them before proceeding. The barrier packet is a blocking packet, in that the processing of the barrier packet completes the packet and its completion object is signaled. This is unlike a dispatch packet whose completion may occur at some future time after the packet has finished processing.

If any of the dependent signals have been signaled with a negative value, the barrier packet is complete, and will indicate failure in its completion signal. The *completion_signal* will be signaled with the error value as discussed in Section 2.4. If the queue is not already in an error state (e.g. the job generating the error was processed in a different queue) then the packet processor should consider the error code on the dependent signal to indicate an error in the queue itself and subsequently signal the *error_signal* in the queue. When all of the dependent signals have been signaled with the value 0, the *completion_signal* will be signaled with the value 0 to indicate a successful completion.

The barrier packet also has a *barrier* bit that indicates that this packet may only be processed when all previous packets have been marked as completed.

The barrier packet can be in one of the following states: *on queue*, *launch*, *completed, error* or *completed, success*. Figure 2.3 shows the state transition diagram.

On queue A packet is considered to be in the on queue state once the format of the packet is changed from invalid (a value of 0) to a value of 1 or 2 or 3. Any other value for format puts the packet and the queue in error state.

Launch If this barrier packet has the barrier bit set, then the processing of this packet occurs only after all prior dispatch packets have completed execution. Otherwise, once the packets prior to this packet are

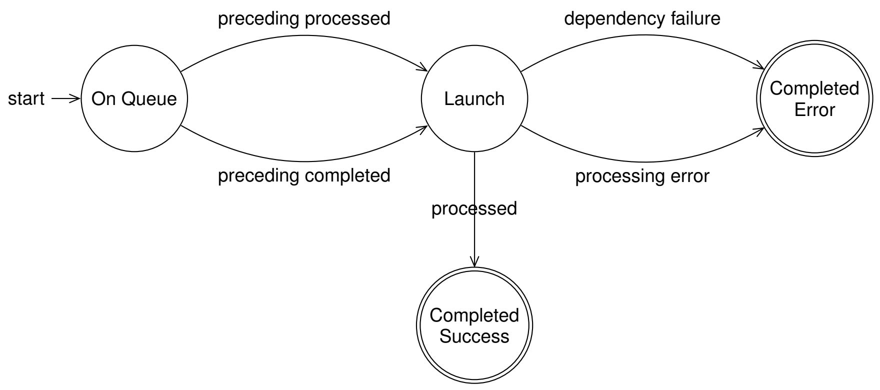

Figure 2.3: Barrier Packet State Diagram

processed, the packet processor begins to process this packet and the packet enters the processing state. From the launch state, two states are possible: completion, error or completion, success.

Completed with Error The barrier packet reaches this state from the processing state if (a) one of the dependency signals had an error, and (b) if the packet was malformed (e.g. bad signal object or invalid usage of reserved bits). A barrier packet can also reach this state when the user invokes the **hsa_queue_inactivate** API or the **hsa_queue_destroy** API while the packet is in processing state (the completion object will be appropriately signaled with an error).

Completed with Success The barrier packet had all its dependencies met, its completion object has been signaled with a value of 0.

2.6.4 Example

TODO

2.7 Memory

One of the key features of HSA is its ability to share global pointers between the host application and code executing on the component. This ability means that an application can directly pass a pointer to memory allocated on the host to a kernel function dispatched to a component without an intermediate copy, as illustrated by the example shown in Core API Documentation.

2.7.1 Registration

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

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

Register memory.

Parameters

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

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

Return Values

HSA_STATUS_SUCCESS

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

Description

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

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

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

Deregister memory.

Parameters

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

Return Values

HSA_STATUS_SUCCESS

HSA_STATUS_INFO_NOT_REGISTERED
If the pointer has not been registered before.

Description

Used for deregistering a memory region previously registered.

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

2.7.1.1 Example

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

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

is a valid program. On the other hand:

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

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

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

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

behaves identically to this program:

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

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

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

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

2.7.2 Global Memory Allocation

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

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

Allocate system memory.

Parameters

size_bytes
(in) Allocation size.

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

Return Values

HSA_STATUS_SUCCESS

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

HSA_STATUS_ERROR_INVALID_ARGUMENT
If the passed address is NULL.

Description

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

2.7.3 Kernarg Memory

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

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

Allocate kernarg memory.

Parameters

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

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

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

Return Values

HSA_STATUS_SUCCESS

HSA_STATUS_ERROR_INVALID_ARGUMENT
: If the passed address is NULL.

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

Copy between the system and kernarg segments.

Parameters

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

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

size
: (in) Number of bytes to copy.

Return Values

HSA_STATUS_SUCCESS

HSA_STATUS_ERROR_INVALID_ARGUMENT
: If the source or destination pointers are invalid.

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

Copy between the system and kernarg segments.

Parameters

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

src

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

size
(in) Number of bytes to copy.

Return Values

HSA_STATUS_SUCCESS

HSA_STATUS_ERROR_INVALID_ARGUMENT
If the source or destination pointers are invalid.

2.7.4 Component Local Memory

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

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

Allocate memory on HSA Device.

Parameters

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

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

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

Return Values

HSA_STATUS_SUCCESS

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

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

Description

Allocate global device memory associated with specified device.

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

Deallocate memory on HSA component.

Parameters

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

Return Values

HSA_STATUS_SUCCESS

Description

Deallocate component memory that was allocated with hsa_memory_allocate_component_local.

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

Copy between the system and local heaps.

Parameters

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

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

size
: (in) Number of bytes to copy.

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

Return Values

HSA_STATUS_SUCCESS

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

HSA_STATUS_ERROR_INVALID_ARGUMENT
: If any argument is invalid.

2.7.4.1 Example

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

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

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

```
 const size_t DATA_SIZE = 1024;
 void* src_ptr = malloc(DATA_SIZE);
```

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

copies 1024 bytes from system to component local memory.

The user should not register or deregister component local memory.

2.8 Execution Control

As per the systems architecture specification, the HSA system must support debugging of a HSAIL kernel. The HSA Programmers Reference Manual (PRM) describes that the debug section could hold debug data and such a section can be placed within a function. This allows the high-level compiler that generates HSAIL to embed debug specific information. This information makes its way into the debug section in the brig. This information can be used for associating a HSAIL level instruction to the higher level functionality. In addition to this, the PRM also discusses the **debugtrap_u32** that halts the current wavefront and transfers control to the agent. The single operand to **debugtrap_u32**, "src" is passed to the agent and can be used to identify the trap.

To support this infrastructure in the runtime, the Core API defines a structure that can be used to exchange information between the kernel executing on the HSA component and the agent.

The core runtime defines a structure, mailbox, whose purpose is to exchange information as a part of execution control. Mailbox is a synchronous communication mechanism between the HSA component and any agents. The HSA component indicates a **debugtrap_u32** or syscall activity by sending a signal indicating it has written to some location in the mailbox. The HSA PRM defines:

queueactivegroupcount_global_u32 *dest, address* Returns the maximum number of work-groups that can be executed in parallel for dispatches executed on the User Mode Queue with address.

activegroupid index that ranges from 0 through **queueactivegroupcount_global_u32**-1.

The mailbox is an array of structures of size **queueactivegroupcount_global_u32**. Since **activegroupid** is always unique within a queue for any concurrent execution of kernels in that queue, indexing into the mailbox by different work items happens without conflicts. When a workgroup encounters a syscall or a **debugtrap_u32**, the component indexes into its mailbox by accessing it via **activegroupid** from within the **queueptr**. Once the corresponding mailbox is accessed, pertinent information (see structure below) for each work group is populated. Subsequently the component sets the full flag, sends a signal to agent by accessing the *mailbox_signal* inside the queue structure (see Section 2.5), and waits for the full flag to be emptied. The mailbox structure is defined as follows.

enum **hsa_interrupt_condition_t**

Interrupt condition.

Values

HSA_DEBUGTRAP = 1
Caused by debugtrap_u32 instruction.

HSA_SYSCALL = 4
Caused by syscall.

HSA_OTHER_INTERRUPT = 8
Caused by other interrupt.

```
typedef struct hsa_group_execution_info_s {
    hsa_signal_handle_t full_flag;
    uint16_t workgroup_size;
    hsa_interrupt_condition_t * condition;
    uint32_t * workitem_id;
    uint32_t * compute_unit_id;
    uint64_t * aql_packet_ptr;
    uint64_t * virtual_address;
    uint64_t * current_program_counter;
    uint64_t args;
    uint64_t ** syscall_output;
} hsa_group_execution_info_t
```

Group execution information.

Data Fields

full_flag
Indicates the mailbox is full and needs to be consumed.

workgroup_size
Size of the workgroup, all pointers below are arrays of that size.

condition
What caused this execution to stop.

workitem_id
Flattend workitem IDs, array[workgroup_size].

compute_unit_id
ID of the compute unit, array[workgroup_size].

aql_packet_ptr
Pointer to the AQL packet, array[workgroup_size].

virtual_address
Any pertinent virtual address, array[workgroup_size].

current_program_counter
Current program counter, array[workgroup_size].

args
Location to where the arguments have been stored. The size and contents are written by the component and need to be decoded by the agent when reading this.

syscall_output
If the condition is syscall, location to where the outputs need to be stored. This is array[workgroup_size].

The agent waits on the signal, processes the mailbox, and clears the full flag. If this kernel had a debugtrap_-u32, a simple check for debugtrap can be written the following way:

```
assert(HSA_STATUS_SUCCESS == hsa_signal_wait(queue_ptr->mailbox_signal,
    HSA_GREATER_EQUALS, 1));
for (i = 0; i < queue->active_group_global_count; i++) {
  assert(HSA_STATUS_SUCCESS ==
      hsa_signal_query_acquire(queue->mailbox[i].full_flag,value);
  if (value.u64_value == 1){
    for (j=0; j < queue->mailbox[i].workgroup_size; j++) {
      printf("\n workitemid=%d computeunitid=%d\n",
              queue->mailbox[i].workitem_id[j],
              queue->mailbox[i].compute_unit_id[j]);
              //here we can check PC.
    }
    hsa_signal_send(queue_ptr->mailbox[i].full_flag, 0);
  }
```

```
    }
```

2.9 Agent Dispatch Support

The core runtime supports agent dispatches from an HSA component/Agent. The runtime defines a default service queue for every user mode queue created by the user. This default service queue is available to the HSAIL programs and the user applications may submit agent dispatch packets to the service queue or any user mode queue. The service queue shares the same structure as the regular HSA queue. The default service queues are monitored by the runtime.

```
hsa_status_t hsa_register_agent_dispatch_callback(
    hsa_queue_t * agent_dispatch_queue,
    void(*)(uint64_t a0, uint64_t a1, uint64_t a2, uint64_t a3, uint64_t retaddr) agent_dispatch_callback,
    hsa_runtime_context_t * context)
```

Agent dispatch runtime function registration.

Parameters

agent_dispatch_queue
Agent dispatch queue.

agent_dispatch_callback
(in) Callback that the user is registering, the callback is called with five 64 bit args as a parameter.

context
Context.

Return Values

HSA_STATUS_SUCCESS

2.10 Extensions to the Core Runtime API

When an implementor of the core runtime specification is not supporting any of the extension API, they will return HSA_STATUS_ERROR_EXTENSION_NOT_SUPPORTED as a return status for that API.

Individual vendors may define vendor extensions to HSA core runtime, or multiple vendors may collaborate to define an extension. The difference is in the naming scheme used for the symbols (defines, structures, functions, etc.) associated with the function:

- Symbols for single-vendor extensions that are defined in the global namespace must use the following naming convention:
  - *hsa_svext_<COMPANY_NAME >_*. For example, a company "ACME" defining a single-vendor extension would use the prefix *hsa_ext_acme_*. Company names must be registered with the HSA Foundation, must be unique, and may be abbreviated to improve the readability of the symbols.
- Symbols for multi-vendor extensions that are defined in the global namespace must use the following naming convention:
  - *hsa_ext_* For example, if another company embraces extension in the example above from Company "ACME", the resulting symbols would use the prefix *hsa_mvext_*.

Any constant definitions in the extension (#define/enumerations) use the same naming convention, except using all capital letters. So, using the single-vendor extension example from above, the associated defines and enumerations would have the prefix HSA_EXT_ACME_.

The symbols for all vendor extensions (both single-vendor and multi-vendor) are captured in the file **hsa/vendor_extensions.h**. This file is maintained by the HSA Foundation. This file includes the enumeration hsa_vendor_extension_t which defines a unique code for each vendor extension and multi-vendor extension. Vendors can reserve enumeration encodings through the HSA Foundation. Multi-vendor enumerations begin at the value of 1000000. For example, using the examples above, the hsa_vendor_extension_t enumeration might be:

enum **hsa_vendor_extension_t**

Vendor enumeration example.

Values

HSA_SVEXT_START = 0
: Start of the single vendor extension range.

HSA_SVEXT_ACME_FOO = 1
: Company ACME, starts with FOO symbol.

HSA_SVEXT_ACME_ANOTHER_EXT = 2
: Company ACME has another_ext symbol.

HSA_MVEXT_START = 1000000
: Multi vendor extension starts at 1000000.

HSA_MVEXT_FOO = 1000001
: Multivendor extension has a symbol foo.

HSA defines the following query function for vendor extensions:

hsa_status_t **hsa_vendor_extension_query**(
    hsa_vendor_extension_t *extension*,
    void * *extension_structure*)

Query vendor extensions.

Parameters

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

extension_structure
    (out) Extension structure.

Return Values

HSA_STATUS_SUCCESS

HSA_STATUS_ERROR_EXTENSION_UNSUPPORTED
    If the extension is not supported.

Description

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

vendor_extension.h defines a unique structure for each extension.

2.10.1 Example

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

```
// Sample hsa/vendor_extensions.h
// Company name is "ACME" and extension is "Foo"
#define HSA_EXT_ACME_MYDEFINE1 0x1000
#define HSA_EXT_ACME_MYDEFINE2 0x0100
#define HSA_EXT_ACME_MYDEFINE3 0x0010
#define HSA_EXT_ACME_MYDEFINE4 0x0001

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

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

3

HSA Extensions Programming Guide

3.1 Compilation Unit, Finalizer and ISA Linking

3.1.1 Finalization, Compilation Unit, Code, Debug and Symbol Objects

Compilation support in the HSA core runtime comprises of a finalization step. The primary functionality of this step is to translate the HSAIL to a component specific instruction set and produce a compilation unit. It resolves user defined symbols that are not already bound via callbacks provided by the user.

HSA components may support HSAIL natively or may have a native ISA that the HSAIL needs to be translated into. However, compilation is a necessary step and is required despite a HSA components' native support of HSAIL. This is because of two primary reasons:

1. The HSA kernel code objects (accessible via the hsa_compilationunit_code_t structure, discussed in Section 3.1.1.2) are an output of the finalization process and required as an input for the AQL dispatch packet; to obtain values from the finalizer for segment sizes etc. and also to act as a container for component specific execution information.

2. In order to support kernel dispatches from the HSA component, the kernel code object must reside in a memory layout specification since dispatches initiated from components are able to use memory operations to get the information necessary for a dispatch.

3.1.1.1 BRIG and .directive Section

The core runtime accepts HSAIL programs coded in the BRIG binary format, as defined in the HSA PRM [1], for its finalization process. BRIG is a binary format defined by the HSA PRM and includes 5 different sections, *.string*, *.directive*, *.code*, *.operand*, and *.debug*. More information on these sections is described in Section 19.1 of the HSA PRM. The core runtime structure that represents a BRIG is named hsa_brig_t. This structure is an in-memory representation of the BRIG. It is defined as follows:

```
typedef struct hsa_brig_s {
    uint8_t * string_section;
    uint8_t * directive_section;
    uint8_t * code_section;
    uint8_t * operand_section;
} hsa_brig_t
```

BRIG representation.

Data Fields

string_section
From PRM: string section, containing all character strings and byte data used in the compilation unit.

directive_section
The directives, which provide information for the finalizer. The directives do not generate code.

code_section
All of the executable operations. Most operations contain offsets to the .operand section.

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

Of the different sections in BRIG, the .directive section provides information to the finalizer on functions, kernels, and global declarations, etc. The symbols in the .directive section have defined placement rules (see PRM for more information). For example, immediately after a function or kernel directive, BRIG requires the directives that describe the arguments to be in a certain order. Return arguments are first, followed by input arguments, followed by the directives that apply only to the function or kernel.

The hsa_brig_t structure has the base address of the directive section. A directive for any symbol is represented using an offset into the directive section. HSA core runtime defines a type hsa_brig_directive_offset_t to represent the .directive section offset. It is typedef to an unsigned 32 bit integer and is defined by all implementations as follows:

```
typedef uint32_t hsa_brig_directive_offset_t
```

There are different types of directives specified in the PRM. Of these, the control directives are a means to allow implementations to pass information to the finalizer via HSAIL. HSA runtime defines a structure hsa_control_directives_t to represent the values of control directives both at finalization time and to record information in the kernel code object. Control directives may also be specified within the HSAIL code. When conflicting values are specified for a particular directive specified in HSAIL and at finalization time, the runtime will do one of the following: (a) perform a union (OR operation) when possible (e.g. when the value represents a bit field).

(b) when a union is not meaningful, the runtime will require that the value provided at finalization time via the hsa_control_directives_t structure match the value for this directive in the HSAIL kernel.

The hsa_control_directives_t structure is defined as follows:

```
typedef struct hsa_control_directives_s {
    hsa_control_directive_present64_t enabled_control_directives;
    hsa_exception_kind16_t enable_break_exceptions;
    hsa_exception_kind16_t enable_detect_exceptions;
    uint32_t max_dynamic_group_size;
    uint32_t max_flat_grid_size;
    uint32_t max_flat_workgroup_size;
    uint32_t requested_workgroups_per_cu;
    hsa_dim3_t required_grid_size;
    hsa_dim3_t required_workgroup_size;
    uint8_t required_dim;
    uint8_t reserved[75];
} hsa_control_directives_t
```

Control directives.

Data Fields

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

enable_break_exceptions
: If enable break exceptions is not enabled in *enabled_control_directives*, then must be 0, otherwise must be non-0 and specifies the set of HSAIL exceptions that must have the BREAK policy enabled. If the HSAIL kernel being finalized has any enablebreakexceptions control directives, then the values specified by this argument are unioned with the values in these control directives. If any of the functions the kernel calls have an enablebreakexceptions control directive, then they must be equal or a subset of, this union.

enable_detect_exceptions
: If enable detect exceptions is not enabled in *enabled_control_directives*, then must be 0, otherwise must be non-0 and specifies the set of HSAIL exceptions that must have the DETECT policy enabled. If the kernel being finalized has any enabledetectexceptions control directives, then the values specified by this argument are unioned with the values in these control directives. If any of the functions the kernel calls have an enabledetectexceptions control directive, then they must be equal or a subset of, this union.

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

max_flat_grid_size
: If this is not enabled in *enabled_control_directives* then must be 0, otherwise must be greater than 0. See HSA Programmer's Reference Manual description of maxflatgridsize control directive.

max_flat_workgroup_size
: If this is not enabled in *enabled_control_directives* then must be 0, otherwise must be greater than 0. See HSA Programmer's Reference Manual description of maxflatgridsize control directive.

requested_workgroups_per_cu

If this is not enabled in *enabled_control_directives* then must be 0 and the finalizer may generate ISA that could result in any number of work-groups executing on a single compute unit. Otherwise, the finalizer will attempt to generate ISA that will allow the specified number of work-groups to execute on a single compute unit. This is only a hint and can be ignored by the finalizer. If the kernel being finalized, or any of the functions it calls, has the same control directive, then the values must be the same or the finalization can fail. This can be used to determine the number of resources that should be allocated to a single work-group and work-item.

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

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

required_dim
: If disabled then must be 0 and the produced kernel code can be dispatched with 1, 2 or 3 dimensions. If enabled then the value is 1..3 and the code produced must only be dispatched with a dimension that matches. Other values are illegal. If the kernel being finalized, or any of the functions it calls, has a requireddimsize control directive, then the values must be the same. This can be used to optimize the code generated to compute the absolute and flat work-group and work-item id, and the dim HSAIL operations.

reserved
: Reserved. Must be 0.

Where, the hsa_control_directive_present64_t is defined as a 64bit unsigned integer.

```
typedef uint64_t hsa_control_directive_present64_t
```

The enumeration hsa_control_directive_present64_t is a bit set indicating which control directives have been specified. It is accessible via a mask, hsa_control_directives_present_mask_t, which is defined as follows:

```
enum hsa_control_directive_present_mask_t
```

Mask indicating which control directives have been specified.

Values

HSA_CONTROL_DIRECTIVE_ENABLE_BREAK_EXCEPTIONS = 0
: Mask that indicates break on exceptions is required by the user, the kernel pauses execution and the queue mailbox signal is signaled and mailbox updated.

HSA_CONTROL_DIRECTIVE_ENABLE_DETECT_EXCEPTIONS = 1
: says that exceptions are recorded.

HSA_CONTROL_DIRECTIVE_MAX_DYNAMIC_GROUP_SIZE = 2
: says that max for dynamic group size is specified.

HSA_CONTROL_DIRECTIVE_MAX_FLAT_GRID_SIZE = 4

If enabled.

HSA_CONTROL_DIRECTIVE_MAX_FLAT_WORKGROUP_SIZE = 8
If enabled.

HSA_CONTROL_DIRECTIVE_REQUESTED_WORKGROUPS_PER_CU = 16
If enabled.

HSA_CONTROL_DIRECTIVE_REQUIRED_GRID_SIZE = 32
If enabled.

HSA_CONTROL_DIRECTIVE_REQUIRED_WORKGROUP_SIZE = 64
If enabled.

HSA_CONTROL_DIRECTIVE_REQUIRED_DIM = 128
If enabled.

HSA_CONTROL_DIRECTIVE_REQUIRE_NO_PARTIAL_WORKGROUPS = 256
If enabled.

User can choose to either break on exceptions or just detect them. The PRM defines the policy to be exception-type specific, i.e. different IEEE exceptions supported by HSA (see the definition of the enumeration hsa_exception_kind_mask_t below) can be handled with different policies (BREAK vs. DETECT). The *enable_break_exceptions* field specifies the set of HSAIL exceptions that must have the BREAK policy enabled. It is possible that on some systems, enabling exceptions may result in lower code performance. If the kernel being finalized has any `enablebreakexceptions` control directives in HSAIL, then the runtime performs a union (OR operation) of values specified by this argument with the values in HSAIL control directives. If any of the functions the kernel calls have an enablebreakexceptions control directive, then they must be equal to or a subset of this union.

enum **hsa_exception_kind_mask_t**

Exception values.

Values

HSA_EXCEPTION_INVALID_OPERATION = 1
IEEE 754 INVALID operation exception.

HSA_EXCEPTION_DIVIDE_BY_ZERO = 2
An operation on finite operands gives an exact infinite result.

HSA_EXCEPTION_OVERFLOW = 4
A result is too large to be represented correctly.

HSA_EXCEPTION_UNDERFLOW = 8
A result is very small (outside the normal range) and inexact.

HSA_EXCEPTION_INEXACT = 16
Returns correctly rounded result by default.

Finally, the dimensions are defined as

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

} **hsa_dim3_t**

3D Dimensions.

Data Fields

x
: X dimension.

y
: Y dimension.

z
: Z dimension.

3.1.1.2 Code Objects

There are different code objects defined by the runtime specification in support of HSAIL: hsa_kernel_code_t, hsa_compilationunit_code_t and hsa_function_code_t. All of them are in memory and can be relocatable and/or position independent (which indicates that the object can be deep-copied to other memory locations for execution). The HSA runtime provides a query API to verify if a particular component supports position independent code objects (see Section 2.3).

The hsa_compilationunit_code_t is the header for the code object produced by the Finalizer and contains information that applies to all code entities in the compilation unit.

Since core runtime does not define a file format container, the core runtime provides API to work with HSAIL programs encoded in the BRIG binary format and supports generation of code objects that include kernels to be executed in the HSA component and binding of unresolved symbols associated with the code object generated.

Finalizer allocates a single contiguous area of memory to hold the generated code for all the code objects.

The structure of this contiguous area is as follows: Starting at offset 0, the “header” of this contiguous area is defined by the hsa_compilationunit_code_t structure. The hsa_compilationunit_code_t structure in turn contains an offset to an array of hsa_code_entry_t, one entry per code entity that the finalizer has produced code for. Each hsa_code_entry_t variable contains an offset to a hsa_*code_t object that describes that code entity (function/kernel/etc).

The kinds of code objects that can be contained in a hsa_compilationunit_code_t is defined by the following structure:

enum **hsa_code_kind_t**

TODO.

Values

HSA_CODE_NONE = 0
: Not a code object.

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

HSA_CODE_FUNCTION = 2
: HSAIL function.

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

HSA_CODE_RUNTIME_LAST = 0x7fffffff
    TODO.

HSA_CODE_VENDOR_FIRST = 0x80000000
    Vendor specific code objects.

HSA_CODE_VENDOR_LAST = 0xffffffff
    TODO.

The hsa_code_entry_t structure is defined as follows:

```
typedef struct hsa_code_entry_s {
    uint64_t code_id;
    int64_t code_byte_offset;
} hsa_code_entry_t
```

TODO.

Data Fields

code_id
    ID of the entity that generated the code. For HSAIL will be the BRIG directive offset of the kernel or function declaration. The array of hsa_code_entry_t are required to be ordered in ascending code_id to allow faster lookup.

code_byte_offset
    Byte offset from start of hsa_compilationunit_code_t to corresponding hsa_code_t. Every hsacode_t starts with a common hsa_code_t, and its code_type field indicates what specific hsa_code_t it is.

The current version number and type of the HSA code object format are defined as follows:

```
typedef uint32_t hsa_code_version32_t
```

```
enum hsa_code_version_t
```

TODO.

Values

HSA_CODE_VERSION = 0
    Code version

Every hsa_*_code_t code objects start with a common header that also contains what kind of code object it is. The common header, hsa_code_t, is defined as follows:

```
typedef struct hsa_code_s {
    hsa_code_version32_t code_version;
    uint32_t struct_byte_size;
    int64_t compilationunit_byte_offset;
    hsa_code_kind32_t code_type;
} hsa_code_t
```

TODO.

Data Fields

code_version
: The code format version. The version of this definition is specified by HSA_CODE_VERSION. Must match the value in the hsa_compilationunit_code_t that contains it.

struct_byte_size
: The byte size of the struct that contains this hsa_code_t. Must be set to sizeof(hsa_*_code_t). Used for backward compatibility.

compilationunit_byte_offset
: Offset from base of hsa_code_t to compilationunit_code_t that contains this hsa_code_t to the base of this hsa_code_t. Can be used to navigate back to the enclosing compilation unit. Since hsa_-compilationunit_code_t is always at offset 0, this value must be negative.

code_type
: Type of code object.

A bit set of flags providing information about the code in a compilation unit. Unused flags must be 0. The hsa_code_properties32_t must be used as a type for this flag. The values/mask currently supported is defined as follows:

enum **hsa_code_properties_mask_t**

TODO.

Values

HSA_CODE_PROPERTY_PIC = 1
: The code is position independent (can be executed at any address that meets the alignment requirement).

The hsa_compilationunit_code_t structure is defined as follows:

```
typedef struct hsa_compilationunit_code_s {
    hsa_code_version32_t code_version;
    uint32_t struct_byte_size;
    char component_vendor[16];
    char component_name[16];
    int64_t code_entry_byte_offset;
    uint32_t code_entry_count;
    hsa_powertwo8_t code_alignment;
    uint8_t reserved[3];
    uint64_t code_size_bytes;
    uint64_t code_base_address;
    hsa_code_properties32_t code_properties;
    uint32_t hsail_version_major;
    uint32_t hsail_version_minor;
    hsa_profile8_t hsail_profile;
    hsa_machine_model8_t hsail_machine_model;
    hsa_target_options16_t hsail_target_options;
} hsa_compilationunit_code_t
```

TODO.

Data Fields

code_version
: The code format version. The version of this definition is specified by HSA_CODE_VERSION.

struct_byte_size
: The byte size of this struct. Must be set to sizeof(hsa_compilationunit_code_t). Used for backward compatibility.

component_vendor
: The vendor of the HSA Component on which this Kernel Code object can execute. ISO/IEC 624 character encoding must be used. If the name is less than 16 characters then remaining characters must be set to 0.

component_name
: The vendor's name of the HSA Component on which this Kernel Code object can execute. ISO/IEC 646 character encoding must be used. If the name is less than 16 characters then remaining characters must be set to 0.

code_entry_byte_offset
: Byte offset from start of hsa_compilationunit_code_t to an array of code_entry_count elements of type hsa_code_entry_t. Since hsa_compilationunit_code_t is always at offset 0, this value must be positive.

code_entry_count
: Number of code entries in this compilation unit.

code_alignment
: The required alignment of this hsa_compilationunit_code_t expressed as a power of 2. The Finalizer must set this to the value required by the HSA component it will execute on and the assumptions of the machine code it contains.

reserved
: Must be 0.

code_size_bytes
: The size of the single contiguous block of memory which includes this hsa_compilationunit_code_t header and all following hsa_*_code_t and associated machine code.

code_base_address
: The base address that this hsa_compilationunit_code_t must be allocated in order to execute the code it contains. The address must be a multiple of the alignment specified by the alignment field. If the code is position independent (can be executed at any address that meets the alignment requirement), then this field must be 0.

code_properties
: A bit set of flags providing information about the code in this compilation unit. Unused flags must be 0.

hsail_version_major
: The HSAIL major version. This information is from the HSAIL version directive. If this hsa_compilationunit_code_t is not generated from an HSAIL compilation unit then must be 0.

hsail_version_minor
: The HSAIL minor version. This information is from the HSAIL version directive. If this hsa_compilationunit_code_t is not generated from an HSAIL compilation unit then must be 0.

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

hsail_machine_model

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

hsail_target_options
The HSAIL target features. There are currently no target options so this field must be 0. If target options are added they will be specified by the HSAIL version directive. If this hsa_compilationunit_code_t is not generated from an HSAIL compilation unit then must be 0.

Both the hsa_compilationunit_t and hsa_*_code_t objects can have implementation define data towards the end of the structure. All elements in the contiguous location being accessed by offsets enables such definition. This also allows the exact position of the various objects in the contiguous memory area to be implementation defined as long as memory alignment requirements are met.

Many of the hsa_*_code_t objects include a size field which is required to be set to the size of the structure. This allows forward compatibility and allows for structure definitions to change or include implementation specific information.

The hsa_kernel_code_t is required for all component dispatches and is referred to by the dispatch AQL packets placed in the HSA queue. This allows an implementation to rely on all such objects being the same size for more efficient navigation to the implementation specific data without need to first read the object's size field.

The hsa_kernel_code_t is the output of **hsa_finalize_brig** API and is defined as follows.

```
typedef struct hsa_kernel_code_s {
    hsa_code_t code;
    uint32_t workgroup_group_segment_byte_size;
    uint64_t kernarg_segment_byte_size;
    uint32_t workitem_private_segment_byte_size;
    uint32_t workgroup_fbarrier_count;
    hsa_control_directives_t control_directive;
    hsa_powertwo8_t wavefront_size;
    hsa_powertwo8_t kernarg_segment_alignment;
    hsa_powertwo8_t group_segment_alignment;
    hsa_powertwo8_t private_segment_alignment;
    uint8_t optimization_level;
    uint8_t reserved1[3];
} hsa_kernel_code_t
```

TODO.

Data Fields

code
Common header that all code objects start with. code.type must be HSA_CODE_KERNEL.

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

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

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

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

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

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

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

group_segment_alignment
: TODO.

private_segment_alignment
: TODO.

optimization_level
: The optimization level specified when the kernel was finalized.

reserved1
: Reserved. Must be 0. Component specific fields can follow this field.

3.1.1.3 Finalize BRIG API

The **hsa_finalize_brig** API accepts hsa_brig_t as an input and produces relocatable code object in which global and group symbols are bound to actual, component recognizable, addresses. A symbol in the finalization step can be a variable, a function, image or a sampler. When finalization occurs, a hsa_kernel_code_t, representing the kernel that needs to be executed on the component is generated. However, all the symbols referenced in the kernel being finalized may not be resolved with the information provided at its finalization. User is allowed to define callbacks that can resolve the symbols declared in the global segment, with the following signature:

```
typedef hsa_status_t(* hsa_map_symbol_address_t)(hsa_finalize_compilationunit_caller_t caller,
hsa_brig_directive_offset_t symbol_directive, uint64_t *address)
```

The finalization step, when successfully executed, can have two distinct outputs. The first is the compilation unit code object, hsa_compilationunit_code_t which is discussed in Section 3.1.1.2. The second output is of type hsa_compilationunit_debug_t which is only generated when the code is compiled with a debug option. This debug information is currently implementation defined.

Since the contiguous memory that *compilationunit_code* represents and the memory for *compilationunit_-debug* need to be allocated, the user is expected to provide callbacks for memory allocation.

The callback is required to have the following signature:

```
typedef void* hsa_finalize_brig_caller_t
```

Where caller is the opaque pointer passed to the **hsa_finalize_brig** that is calling back this function. *byte_-size* is the size in bytes of the memory to be allocated. *byte_alignment* is the required byte alignment of the memory allocated. Must be a power of 2. *address* is pointer to location that will be updated with address of allocated memory if successful, or NULL if not successful. *return value* is the HSA status of the allocation.

```
hsa_status_t hsa_finalize_brig(
    hsa_finalize_compilationunit_caller_t caller,
    hsa_agent_t * component,
    hsa_brig_t * brig,
    size_t code_count,
    hsa_brig_directive_offset_t * code_directive,
    hsa_control_directives_t * control_directives,
    hsa_map_symbol_address_t map_symbol_address,
    hsa_alloc_t allocate_compilationunit_code,
    hsa_alloc_t allocate_compilationunit_debug,
    uint8_t optimization_level,
    const char * options,
    hsa_compilationunit_code_t ** compilationunit_code,
    hsa_compilationunit_debug_t ** compilationunit_debug)
```

Finalize BRIG.

Parameters

caller
: Opaque pointer and will be passed to all call back functions made by this call of the finalizer.

component
: input. A valid pointer to the hsa_agent_t.

brig
: input. A pointer to the in memory BRIG structure.

code_count
: The number of kernels plus functions to produce hsa_kernel_code_t and hsa_function_code_t objects for in the generated hsa_compilationunit_code_t.

code_directive
: A pointer to an array with code_count entries of hsa_brig_directive_offset_t. Each entry is the offset in the directive section of the passed brig of a kernel or function definition. These will be the kernels and functions that will have code objects generated in the produced hsa_compilationunit_code_t.

control_directives
: The control directives that can be specified to influence how the finalizer generates code. If NULL then no control directives are used.

map_symbol_address
: Used by the finalizer to obtain the segment address for global segment symbols. The value of caller will be passed to every call.

allocate_compilationunit_code
: The callback function that the finalizer will use to allocate the contiguous block of memory that will be used for the hsa_compilationunit_code_t that is returned. It is the responsibility of the call of the finalizer to deallocate this memory, even if the finalizer does not report success.

allocate_compilationunit_debug
: The callback function that the finalizer will use to allocate the memory that will be used for the hsa_-compilationunit_debug_t that is returned. It is the responsibility of the call of the finalizer to destroy this memory, even if the finalizer does not report success.

optimization_level
: an implementation defined value that control the level of optimization performed by the finalizer.

options
: implementation defined options that can be specified to the finalizer. compilationunit_code: if the return status is success then a pointer to the generated hsa_compilationunit_code_t for the HSA component must be written.

compilationunit_code
: TODO.

compilationunit_debug
: TODO.

Return Values

HSA_STATUS_SUCCESS

HSA_STATUS_ERROR_INVALID_ARGUMENT
: If *brig* is NULL or invalid, or if *kernel_directive* is invalid.

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

HSA_STATUS_ERROR_OUT_OF_RESOURCES
: If the finalize API cannot allocate memory for compilationunit_code and compilationunitdebug or the deserialize cannot allocate memory for code_object.

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

Description

The input pointer must reference an in memory location of the BRIG. Also, the BRIG must contain at the minimum the 4 basic BRIG sections (.strings, .directives, .code, .operands). All symbols in the symbol/offset table must be resolved.

HSAIL does not define a set of options that a finalizer needs to specify. To ensure portability, **hsa_finalize_-brig** must not return an error status if a given compilation option is not recognized.

The callback for the allocation of *compilationunit_debug* is optional and is required when the user passes in a flag at finalization to generate debug information. This callback, when defined, must have the same signature as hsa_alloc_t above.

```
typedef struct hsa_compilationunit_debug_s {
    uint64_t reserved;
} hsa_compilationunit_debug_t
```

Debug unit.

Data Fields

reserved
: HSA does not define the debug format, it is device specific. It is not required to be implemented as a contiguous region of memory.

3.1.1.4 Freeing The Compilation Unit Code/Debug Object

The core runtime also provides a corresponding destruction API that destroys the compilation unit code and debug objects. This will reclaim all memory used by the hsa_compilationunit_code_t (including the ISA it contains) and associated hsa_compilationunit_debug_t. It will also unregister the ISA memory if appropriate.

hsa_status_t **hsa_compilationunit_code_destroy**(
    hsa_compilationunit_code_t * *object*)

Destroys the compilation unit code object.

Parameters

object
    (in) A pointer to the compilation unit object that needs to be destroyed.

Return Values

HSA_STATUS_SUCCESS

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

HSA_STATUS_ERROR_INVALID_ARGUMENT
    If *object* is NULL or does not point to a valid code object.

hsa_status_t **hsa_compilationunit_debug_destroy**(
    hsa_compilationunit_debug_t * *object*)

Destroys the compilation unit debug object.

Parameters

object
    (in) A pointer to the compilation unit debug object that needs to be destroyed.

Return Values

HSA_STATUS_SUCCESS

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

HSA_STATUS_ERROR_INVALID_ARGUMENT
    If *object* is NULL or does not point to a valid code object.

Note that destroying does not impact any memory segments that may have been allocated/reserved for use in a kernel from this object. It merely releases resources used to build the object.

3.1.1.5 Serializing and Deserializing a Compilation Unit

Because of the opaque nature of what the compilation unit actually represents, and in order for the users of the core runtime to build file containers, serialization and deserialization API are defined by the core runtime. The definition is as follows:

```
hsa_status_t hsa_compilationunit_serialize(
    hsa_compilationunit_code_t * code_object,
    hsa_alloc_t allocate_compilationunit_code,
    void * serialized_object)
```

Serialize a code object.

Parameters

code_object
(in) The code object to serialize.

allocate_compilationunit_code
TODO.

serialized_object
(out) Pointer to the serialized object.

Return Values

HSA_STATUS_SUCCESS

HSA_STATUS_ERROR_INVALID_ARGUMENT
If *code_object* or is NULL or does not point to a valid code object. If *serialized_object* is either null or is not valid or the size is 0.

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

```
hsa_status_t hsa_compilationunit_deserialize(
    void * serialized_object,
    hsa_compilationunit_code_t ** code_object)
```

Recreate a serialized code object.

Parameters

serialized_object
(in) Pointer to the serialized object.

code_object
(out) The code object generated as a part of serialization. Runtime allocated.

Return Values

HSA_STATUS_SUCCESS

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

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

3.1.2 Group Memory Usage

Group memory can be allocated either statically when the finalizer is called or dynamically by the user at launch time.

Static allocation is supported as follows:

- The **hsa_finalize_brig** routine writes the amount of group memory needed by the finalized ISA to the hsa_code_object_t.*workgroup_group_segment_size_byte* field. The group memory usage includes group memory which is statically allocated in the HSAIL kernel, as well as private group memory used by the finalizer. Different HSA implementations might allocate different amounts of group memory.
- The user copies the requested group segment usage to the dispatch packet field *group_segment_-size_bytes*.
- The packet processor reads the group memory usage field and reserves the required resources at dispatch time.
- Statically allocated group memory starts at a segment offset of 0.

Dynamically allocated group memory allows the user to specify the group memory size when the kernel is launched. This is useful to support dynamic group memory allocation features supported by languages such as OpenCL. Essentially, the user manually calculates the offset for each kernel argument (including the static allocation in the calculation) and passes these as arguments to the HSAIL kernel. Specifically:

- As above, the **hsa_finalize_brig** routine returns the requested static group allocation.
- HSAIL will use standard 32-bit arguments (that is, **`kernarg_u32`**) to specify group segment offsets. The user is responsible for computing the offset for each group memory argument location. The first argument must start just above the static allocation, so it always has the offset of *workgroup_group_-segment_size_byte* (see hsa_code_object_t).
- After setting the offset for each group memory argument, the user must set the AQL dispatch packet's hsa_aql_dispatch_packet_t. *group_segment_size_bytes* field to the total amount of group memory used (static and dynamic allocations).

See below for an example of setting up dynamic group memory arguments for a kernel.

```
 // ... assume setup for component, queue

 // User dynamically requests 3 group allocations of 256, 384, and 500 bytes.
 // These can be specified at launch-time.
 int size1 = 256;
 int size2 = 384;
 int size3 = 500;

hsa_aql_dispatch_packet_t aql_packet;
uint64_t* kernel_arguments =
reinterpret_cast<uint64_t *>(malloc(argument_size));

memset(kernel_arguments, 0, argument_size);

// Copy parameters into the AQL packet, computing relative offsets:
kernel_arguments[0] = kernel_object_ptr->workgroup_group_segment_size_bytes;

kernel_arguments[1] = kernel_object_ptr->workgroup_group_segment_size_bytes+size1;

kernel_arguments[2] =
    kernel_object_ptr->workgroup_group_segment_size_bytes+size1+size2;

 // Set the total group memory size:
aql_packet.group_segment_size_bytes = kernel.group_memory_usage + size1 +
size2 + size3;
```

Here is the corresponding kernel and usage model:

```
kernel &myDynamicGroupMemKernel (
                kernarg_u32 %groupOffset0 ,
                kernarg_u32 %groupOffset1 ,
                kernarg_u32 %groupOffset2 ,
                kernarg_u32 %foo )
{
        ld_kernarg_u32 $s0, [%groupOffset0 ]
        workitemid        $s1
        add               $s2, $s1, $s0
        st_group       0, [$s2]
        barrier
        ...
```

3.2 Images and Samplers

Images in HSA are accessed by an image handle hsa_image_handle_t. The image handle references the image data in memory and records information about resource layout and other properties. HSA decouples the storage of the image data and the description of how the device interprets that data. This allows the developer to control the location of the image data storage and manage memory more efficiently.

The HSA image format is specified using a format descriptor (hsa_image_format_t) that contains information about the image channel type and the channel order. The image channel type describes how the data is to be interpreted along with the bit size, and image channel order describes the number and the order. Not all image channel types and channel order combinations are valid on a HSA gent. All HSA agents have to support a required minimum set of image formats (see HSA Programmer's Reference Manual). An application can query image format capabilities using **hsa_image_get_format_capability**.

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

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

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

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

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

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

3.3 Component Initiated Dispatches

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

Some requirements to do component-initiated dispatch:

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

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

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

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

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

Appendices

A
HSA API Reference

A.1 Initialization and Shutdown

```
typedef uint64_t hsa_runtime_context_t
```

```
hsa_status_t hsa_open(
    hsa_runtime_context_t ** context)
```

Initialize the HSA runtime.

Parameters

context
(out) A type for reference counting. User allocated.

Return Values

HSA_STATUS_SUCCESS

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

HSA_STATUS_ERROR_COMPONENT_INITIALIZATION
If there is a non-specific failure in initializing one of the components.

HSA_STATUS_ERROR_CONTEXT_NULL
If the context pointer passed by the user is NULL. User is required to pass in a memory backed context pointer.

Description

Initializes the HSA runtime if it is not already initialized. It is allowed for applications to invoke hsa_open multiple times. The HSA open call returns a new context at every invocation. Reference counting is a mechanism that allows the runtime to keep a count of the number of different usages of the runtime API within the same application process. This ensures that the runtime stays active until hsa_close is called by the user when the reference count represented by that runtime context is one.

If the HSA runtime is already initialized, an asynchronous notification is generated by the runtime and HSA_-STATUS_SUCCESS is returned. If the user chooses to capture this asynchronous notification, the user should define a callback and associate it with the context returned by the hsa_open call. Each open call increments the reference count before returning success.

```
hsa_status_t hsa_close(
    hsa_runtime_context_t * context)
```

Close the HSA runtime.

Parameters

context
(in) Context to close.

Return Values

HSA_STATUS_SUCCESS

HSA_STATUS_ERROR_NOT_INITIALIZED
If invoked before the runtime was initialized, or after it has already been successfully closed.

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

Description

Decreases the context reference count for every invocation. Once the reference count is zero, it proceeds to relinquish any resources allocated for the runtime and closes the runtime instance. It is possible in a multi-threaded scenario that one thread is doing a close while the other is trying to acquire the runtime context or do an open. The core runtime specification defines that an acquire with an input context that represents a closed runtime instance will fail. However, hsa_open can be called to create a new instance of the runtime after it is closed.

An invocation when the reference count is not one it is considered successful in that HSA_STATUS_SUC-CESS is returned with status HSA_STATUS_CLOSE_CONTEXT_ACTIVE, is generated by the runtime on the context that is still active before the API returns.

```
hsa_status_t hsa_context_acquire(
    hsa_runtime_context_t * input_context)
```

Increment reference count of a context, if it is not currently zero.

Parameters

input_context
(in) The context that the user is explicitly reference counting. User allocated.

Return Values

HSA_STATUS_SUCCESS

HSA_STATUS_ERROR_NOT_INITIALIZED
If invoked before the runtime was initialized or after it has been closed.

HSA_STATUS_CONTEXT_LIMIT_REACHED
If the reference count has reached UINT64_MAX.

```
hsa_status_t hsa_context_release(
    hsa_runtime_context_t * input_context)
```

Decrement reference count of a context.

Parameters

input_context
(in) The context that the user is explicitly reference counting. User allocated.

Return Values

HSA_STATUS_SUCCESS

HSA_STATUS_ERROR_NOT_INITIALIZED
If invoked before the runtime was initialized or when the reference count is already zero.

A.2 Errors and Notifications

enum **hsa_status_t**

Values

HSA_STATUS_SUCCESS = 0
    Indicates success. The API has been successfully executed per its definition.

HSA_STATUS_FAILURE
    Indicates an error occurred, specifics were either not determinable or not encoded in the error list.

HSA_STATUS_ALREADY_INITIALIZED
    Indicates that initialization attempt failed due to prior initialization.

HSA_STATUS_INFO_SIGNAL_TIMEOUT
    Indicates that signal is timed out.

HSA_STATUS_ERROR_INVALID_ARGUMENT
    TODO.

HSA_STATUS_ERROR_OUT_OF_RESOURCES
    The runtime was not able to allocate the necessary resources. This error may also occur when the core runtime library needs to spawn threads or create internal OS-specific events.

HSA_STATUS_ERROR_INVALID_CONTEXT
    TODO.

HSA_STATUS_INFO_UNRECOGNIZED_OPTIONS
    TODO.

HSA_STATUS_ERROR_DIRECTIVE_MISMATCH
    TODO.

HSA_STATUS_ERROR_RESOURCE_FREE
    TODO.

HSA_STATUS_CLOSE_CONTEXT_ACTIVE
    TODO.

HSA_STATUS_ERROR_NOT_INITIALIZED
    TODO.

HSA_STATUS_CONTEXT_LIMIT_REACHED
    TODO.

HSA_STATUS_ERROR_COMPONENT_INITIALIZATION
    TODO.

HSA_STATUS_ERROR_CONTEXT_NULL
    TODO.

HSA_STATUS_ERROR_SIGNAL_NOT_BOUND
    TODO.

HSA_STATUS_ERROR
    TODO.

HSA_STATUS_INFO_NOT_REGISTERED
    TODO.

HSA_STATUS_ERROR_EXTENSION_UNSUPPORTED
    TODO.

HSA_STATUS_UNSUPPORTED_IMAGE_FORMAT
  TODO.

HSA_STATUS_IMAGE_SIZE_NOT_SUPPORTED
  TODO.

```
typedef struct _hsa_notification_info_s {
      hsa_status_t status;
      void * info;
      char * string_info;
      void * user_data;
} hsa_notification_info_t
```

Notification information.

Data Fields

status
  The info status enum value.

info
  A pointer to more information, this could be pointing to implementation specific details that could be useful to some tools or to binary data.

string_info
  ISO/IEC 646 character encoding must be used. A string indicating some error information. The string should be NUL terminated per ISO 646.

user_data
  A pointer to user supplied data.

```
typedef struct _hsa_async_error_info_s {
      hsa_status_t error_type;
      uint32_t queue_id;
      void * info;
      char * string_info;
      void * user_data;
      uint64_t timestamp;
      uint64_t reserved1;
      uint64_t reserved2;
      uint64_t reserved3;
} hsa_async_error_info_t
```

TODO.

Data Fields

error_type
  Indicates the type of the error, based on this, the user knows if and packet_id is available in one of the reserved words.

queue_id
  The queue that processed the entity that caused the asynchronous error.

info

A pointer to more information, this could be pointing to implementation specific details that could be useful to some tools or to binary data.

string_info
ISO/IEC 646 character encoding must be used. A string indicating some error information. The string should be NUL terminated per ISO 646.

user_data
A pointer to user supplied data

timestamp
System timestamp to indicate when the error was discovered, the implementation may chose to always return 0 and user must take it into account when utilizing the timestamp.

reserved1
Additional info to be interpreted based on *error_type*.

reserved2
Additional info to be interpreted based on *error_type*.

reserved3
Additional info to be interpreted based on *error_type*.

```
hsa_status_t hsa_status_query_description(
      hsa_status_t input_status,
      uint64_t * status_info,
      char *const * status_info_string)
```

Queries additional information on synchronous errors.

Parameters

input_status
(in) Any unsuccessful API return status that the user is seeking more information on.

status_info
(in) User allocated. In addition to the string. This value could be 0 and in itself (without *status_info_-string*) may not be independently interpreted by the user.

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

Return Values

HSA_STATUS_SUCCESS

HSA_STATUS_ERROR_INVALID_ARGUMENT
If a NULL value is passed for either of the arguments.

Description

Returns success if one or both of the *status_info* and *status_info_string* have been successfully updated with information regarding the input *input_status*.

```
hsa_status_t hsa_notification_callback_register(
      void(*)(const hsa_notification_info_t *info) callback,
      void * user_data,
    hsa_runtime_context_t * context)
```

Register a notification callback.

Parameters

callback
: (in) The callback that the user is registering, which will be invoked with info as a parameter.

user_data
: (in) The user data to call the callback with. The *user_data* field of *info* will be filled with value when the callback is called.

context
: (in) Identifies a particular runtime context that this callback is registered for. When a callback is registered for a particular context, it will only be invoked if the notification is for an action in that context.

Return Values

HSA_STATUS_SUCCESS

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

HSA_STATUS_ERROR_INVALID_ARGUMENT
: If *info* is NULL.

HSA_STATUS_ERROR_INVALID_CONTEXT
: If the context is invalid (e.g. referenced counted to 0).

```
hsa_status_t hsa_error_callback_register(
    void(*)(const hsa_async_error_info_t *info) callback,
    void * user_data,
    hsa_runtime_context_t * context)
```

Register an error callback.

Parameters

callback
: (in) The callback that the user is registering, the callback is called with info structure. User can read the structure and access its elements.

user_data
: (in) The user data to call the callback with. info.user_data will be filled with value when the callback is called.

context
: (in) The runtime context that this callback is being registered for.

Return Values

HSA_STATUS_SUCCESS

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

HSA_STATUS_ERROR_INVALID_ARGUMENT
  If *info* is NULL.

Description

When a callback is registered for a particular context, it will only be invoked if the notification is for an action in that context. For example, if a queue was created for a runtime context *c1* and a callback registered for a context *c2* but not for *c1*, any error on the queue, such as a packet processing error, will not trigger the execution of asynchronous error callback registered for context *c1*.

A.3 Topology Discovery

```
enum hsa_agent_type_t
```

Agent type.

Values

HOST = 1
: Host agent (CPU).

COMPONENT = 2
: HSA component.

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

```
typedef struct hsa_agent_s {
    uint32_t hsa_node_id;
    uint32_t agent_id;
    hsa_agent_type_t agent_type;
    char vendor[16];
    char name[16];
    uint64_t * property_table;
    uint32_t number_memory_descriptors;
    uint32_t * memory_descriptors;
    uint32_t number_cache_descriptors;
    uint32_t * cache_descriptors;
    uint32_t number_of_subagents;
    uint32_t * subagent_offset_list;
    uint32_t wavefront_size;
    uint32_t queue_size;
    uint32_t group_memory_size_bytes;
    uint32_t fbarrier_max_count;
    uint8_t is_pic_supported;
} hsa_agent_t
```

HSA agent.

Data Fields

hsa_node_id
: ID of the node this agent/component belongs to.

agent_id
: Unique identifier for an HSA agent.

agent_type
: Identifier for the type of this agent.

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

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

property_table
: Table of properties of the agent, any property that is not available has a value of 0.

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

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

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

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

number_of_subagents
: Number of subagents.

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

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

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

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

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

is_pic_supported
: Does it support position independent code?. Only applicable when the agent is a component.

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

Memory segment.

Data Fields

global
: Global segment.

privat
: Private segment.

group
: Group segment.

kernarg
: Kernarg segment.

readonly
    Readonly segment.

reserved
    Reserved.

```
typedef struct hsa_memory_descriptor_s {
    uint32_t hsa_node_id;
    uint32_t hsa_memory_id;
    hsa_segment_t supported_segment_type_mask;
    uint64_t virtual_address_base;
    uint64_t size_in_bytes;
    uint64_t peak_bandwidth_mbps;
} hsa_memory_descriptor_t
```

Memory descriptor.

Data Fields

hsa_node_id
    ID of the node this memory belongs to.

hsa_memory_id
    Unique for this memory with in the system.

supported_segment_type_mask
    Information on segments that can use this memory.

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

size_in_bytes
    Size.

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

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

Cache descriptor.

Data Fields

hsa_node_id
    ID of the node this memory belongs to.

hsa_cache_id
    Unique identified for this cache with in the system.

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

associativity
  Associativity of this cache. The array has size *levels*.

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

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

is_inclusive
  Cache inclusive with the level above?. The array has size *levels* - 1.

```
typedef struct hsa_topology_table_s {
    uint32_t header_size_bytes;
    uint32_t hsa_system_timestamp_frequency_mhz;
    uint8_t number_of_nodes;
    uint32_t * node_id;
    uint32_t number_agents;
    uint32_t * agent_offset_list_bytes;
    uint32_t number_memory_descriptors;
    uint32_t * memory_descriptor_offset_list_bytes;
    uint32_t number_cache_descriptors;
    uint32_t * cache_descriptors_offset_list_bytes;
    uint32_t table_size;
    void * topology_table_base;
} hsa_topology_table_t
```

Topology header.

Data Fields

header_size_bytes
  Size of the header.

hsa_system_timestamp_frequency_mhz
  Constant observable timestamp value increase rate is in the range 1-400MHz.

number_of_nodes
  Number of different nodes in this platform configuration.

node_id
  IDs of the nodes.

number_agents
  Number of agent offsets specified in this structure. Zero indicates that no information is available.

agent_offset_list_bytes
  Agent list, refers to the offsets in platform table.

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

memory_descriptor_offset_list_bytes
  Each element in the array carries an offset into the topology table to where memory descriptors are located. Number of elements in array equals *number_memory_descriptors*.

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

cache_descriptors_offset_list_bytes
Array of offsets (into the topology table) to cache descriptors. Number of elements in array equals number_cache_descriptors.

table_size
Size of the table.

topology_table_base
Table base address, points to the table which starts with ::hsa_platform_t structure.

hsa_status_t **hsa_topology_table_create**(
    hsa_topology_table_t ** *table*)

Retrieve topology information.

Parameters

table
(out) The topology header, this includes the base pointers to the rest of the topology table. Runtime allocated.

Return Values

HSA_STATUS_SUCCESS

HSA_STATUS_ERROR_INVALID_ARGUMENT
If *header* is NULL.

HSA_STATUS_ERROR_OUT_OF_RESOURCES
If there is a failure in allocation of an internal structure required by the core runtime or in the creation of table header or the actual table.

hsa_status_t **hsa_topology_table_destroy**(
    hsa_topology_table_t * *table*)

Release resources associated with the topology information.

Parameters

table
(in) Topology table to be destroyed.

Return Values

HSA_STATUS_SUCCESS

A.4 Signals

```
typedef uint64_t hsa_signal_handle_t
```

```
enum hsa_signal_condition_t
```

Wait condition operator.

Values

HSA_EQ
: The two operands are equal.

HSA_NE
: The two operands are not equal.

HSA_LT
: The first operand is less than the second operand.

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

```
typedef union signal_value_s {
    int32_t value32;
    int64_t value64;
} hsa_signal_value_t
```

Signal value.

Data Fields

value32
: Signal value in the small machine model.

value64
: Signal value in the large machine model.

```
hsa_status_t hsa_signal_create(
    hsa_signal_value_t initial_signal_value,
    hsa_signal_handle_t * signal_handle)
```

Create a signal.

Parameters

initial_signal_value
: (in) Initial value of the signal.

signal_handle
: (out) Signal handle.

Return Values

HSA_STATUS_SUCCESS

HSA_STATUS_ERROR_OUT_OF_RESOURCES

HSA_STATUS_ERROR_INVALID_ARGUMENT
    If *signal_handle* is NULL.

```
hsa_status_t hsa_signal_destroy(
    hsa_signal_handle_t signal_handle)
```

Destroy signal previous created by hsa_signal_create.

Parameters

signal_handle
    (in) Signal handle.

Return Values

HSA_STATUS_SUCCESS

HSA_STATUS_ERROR_INVALID_ARGUMENT
    If *signal_handle* is invalid.

```
hsa_status_t hsa_signal_bind(
    hsa_signal_handle_t signal_handle,
    hsa_runtime_context_t * context)
```

Bind a signal to a context. A signal might be bound to several contexts.

Parameters

signal_handle
    (in) Signal handle.

context
    (in) Additional context to which this signal should be bound to.

Return Values

HSA_STATUS_SUCCESS

HSA_STATUS_ERROR_INVALID_ARGUMENT
    If *signal_handle* is invalid or if *context* is NULL or invalid.

```
hsa_status_t hsa_signal_unbind(
    hsa_signal_handle_t signal_handle,
    hsa_runtime_context_t * context)
```

Unbind a signal from a context.

Parameters

signal_handle
    (in) Signal handle.

context
    (in) Unbind the signal from this context.

Return Values

HSA_STATUS_SUCCESS

HSA_STATUS_ERROR_SIGNAL_NOT_BOUND
: If the signal was not already bound to that context.

HSA_STATUS_ERROR_INVALID_ARGUMENT
: If *signal_handle* is invalid or if *context* is NULL or invalid.

Description

Signals are unbound from a particular context if the user no longer wants to receive notifications about this signal in the callback registered for that context.

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

Read the current signal value.

Parameters

signal_handle
: (in) Signal handle.

value
: (out) User-allocated pointer to where the current signal value must be read into.

Return Values

HSA_STATUS_SUCCESS

HSA_STATUS_ERROR_INVALID_ARGUMENT
: If *signal_handle* is invalid or *value* is NULL.

Description

If the signal is being updated by the component or other threads, there is no guarantee that the value returned by the query API is the value of the signal even at the instance it has been returned. This function is non-blocking and does not take any condition as an input, unlike hsa_signal_wait_acquire

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

Read the current signal value.

Parameters

signal_handle
: (in) Signal handle.

value
: (out) User-allocated pointer to where the current signal value must be read into.

Return Values

HSA_STATUS_SUCCESS

HSA_STATUS_ERROR_INVALID_ARGUMENT
: If *signal_handle* is invalid or *value* is NULL.

Description

If the signal is being updated by the component or other threads, there is no guarantee that the value returned by the query API is the value of the signal even at the instance it has been returned. This function is non-blocking and does not take any condition as an input, unlike hsa_signal_wait_acquire

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

Set the value of a signal.

Parameters

signal_handle
(in) Signal handle.

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

Return Values

HSA_STATUS_SUCCESS

HSA_STATUS_ERROR_INVALID_ARGUMENT
If *signal_handle* is invalid.

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

Set the value of a signal.

Parameters

signal_handle
(in) Signal handle.

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

Return Values

HSA_STATUS_SUCCESS

HSA_STATUS_ERROR_INVALID_ARGUMENT
If *signal_handle* is invalid.

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

Set the value of a signal and return its previous value.

Parameters

signal_handle
(in) Signal handle.

value
(inout) Value to be placed at the signal

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

Return Values

HSA_STATUS_SUCCESS

HSA_STATUS_ERROR_INVALID_ARGUMENT
  If *signal_handle* is invalid.

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

Set the value of a signal and return its previous value.

Parameters

signal_handle
  (in) Signal handle.

value
  (inout) Value to be placed at the signal

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

Return Values

HSA_STATUS_SUCCESS

HSA_STATUS_ERROR_INVALID_ARGUMENT
  If *signal_handle* is invalid.

```
hsa_status_t hsa_signal_cas_release(
    hsa_signal_handle_t signal_handle,
    hsa_signal_value_t value_compare,
    hsa_signal_value_t value_replace,
    hsa_signal_value_t * prev_value)
```

Perform a CAS on a signal.

Parameters

signal_handle
  (in) Signal handle.

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

value_replace
  (in) The new value of the signal.

prev_value
  (out) The value at the signal, prior to the atomic replace, if the comparison was successful. User allocated.

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

Increment the value of a signal by a given amount. The addition is atomic.

Parameters

signal_handle
(in) Signal handle.

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

Return Values

HSA_STATUS_SUCCESS

HSA_STATUS_ERROR_INVALID_ARGUMENT
If *signal_handle* is invalid.

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

Increment the value of a signal by a given amount. The addition is atomic.

Parameters

signal_handle
(in) Signal handle.

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

Return Values

HSA_STATUS_SUCCESS

HSA_STATUS_ERROR_INVALID_ARGUMENT
If *signal_handle* is invalid.

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

Decrement the value of a signal by a given amount.

Parameters

signal_handle
(in) Signal handle.

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

Return Values

HSA_STATUS_SUCCESS

HSA_STATUS_ERROR_INVALID_ARGUMENT
  If *signal_handle* is invalid.

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

Decrement the value of a signal by a given amount.

Parameters

signal_handle
  (in) Signal handle.

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

Return Values

HSA_STATUS_SUCCESS

HSA_STATUS_ERROR_INVALID_ARGUMENT
  If *signal_handle* is invalid.

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

Perform a logical AND of the value of a signal and a given value.

Parameters

signal_handle
  (in) Signal handle.

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

Return Values

HSA_STATUS_SUCCESS

HSA_STATUS_ERROR_INVALID_ARGUMENT
  If *signal_handle* is invalid.

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

Perform a logical AND of the value of a signal and a given value.

Parameters

signal_handle
  (in) Signal handle.

value

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

Return Values

HSA_STATUS_SUCCESS

HSA_STATUS_ERROR_INVALID_ARGUMENT
If *signal_handle* is invalid.

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

Perform a logical OR of the value of a signal and a given value.

Parameters

signal_handle
(in) Signal handle.

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

Return Values

HSA_STATUS_SUCCESS

HSA_STATUS_ERROR_INVALID_ARGUMENT
If *signal_handle* is invalid.

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

Perform a logical OR of the value of a signal and a given value.

Parameters

signal_handle
(in) Signal handle.

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

Return Values

HSA_STATUS_SUCCESS

HSA_STATUS_ERROR_INVALID_ARGUMENT
If *signal_handle* is invalid.

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

Perform a logical XOR of the value of a signal and a given value.

Parameters

signal_handle
(in) Signal handle.

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

Return Values

HSA_STATUS_SUCCESS

HSA_STATUS_ERROR_INVALID_ARGUMENT
If *signal_handle* is invalid.

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

Perform a logical XOR of the value of a signal and a given value.

Parameters

signal_handle
(in) Signal handle.

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

Return Values

HSA_STATUS_SUCCESS

HSA_STATUS_ERROR_INVALID_ARGUMENT
If *signal_handle* is invalid.

hsa_status_t **hsa_signal_max**(
hsa_signal_handle_t *signal_handle*,
hsa_signal_value_t *value*,
hsa_signal_value_t * *max_value*)

Set (increment) the signal value to a given input if it is greater than the current value.

Parameters

signal_handle
(in) Signal handle.

value
(in) User defined value.

max_value
(out) Maximum of *value* and the signal's current value.

Return Values

HSA_STATUS_SUCCESS

HSA_STATUS_ERROR_INVALID_ARGUMENT
If *signal_handle* is invalid.

```
hsa_status_t hsa_signal_min(
    hsa_signal_handle_t signal_handle,
    hsa_signal_value_t value,
    hsa_signal_value_t * min_value)
```

Set (decrement) the signal value to a given input if it is smaller than the current value.

Parameters

signal_handle
(in) Signal handle.

value
(in) User defined value.

min_value
(out) Minimum of *value* and the signal's current value.

Return Values

HSA_STATUS_SUCCESS

HSA_STATUS_ERROR_INVALID_ARGUMENT
If *signal_handle* is invalid.

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

Increment the value of a signal.

Parameters

signal_handle
(in) Signal handle.

value
(in) Value the signal is to be incremented with.

Return Values

HSA_STATUS_SUCCESS

HSA_STATUS_ERROR_INVALID_ARGUMENT
If *signal_handle* is invalid.

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

Increment the value of a signal.

Parameters

signal_handle
(in) Signal handle.

value
(in) Value the signal is to be incremented with.

Return Values

HSA_STATUS_SUCCESS

HSA_STATUS_ERROR_INVALID_ARGUMENT
: If *signal_handle* is invalid.

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

Decrement the value of a signal.

Parameters

signal_handle
: (in) Signal handle.

value
: (in) Value the signal is to be decremented with.

Return Values

HSA_STATUS_SUCCESS

HSA_STATUS_ERROR_INVALID_ARGUMENT
: If *signal_handle* is invalid.

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

Decrement the value of a signal.

Parameters

signal_handle
: (in) Signal handle.

value
: (in) Value the signal is to be decremented with.

Return Values

HSA_STATUS_SUCCESS

HSA_STATUS_ERROR_INVALID_ARGUMENT
: If *signal_handle* is invalid.

```
uint64_t hsa_signal_get_timeout()
```

Returns the maximum amount of time an implementation can spend in a wait operation on the signal.

Returns

Signal timeout. The returned value is in the units of the system-wide clock whose frequency is available in the topology table.

```
hsa_status_t hsa_signal_wait_acquire(
    hsa_signal_handle_t signal_handle,
    uint64_t timeout,
    hsa_signal_condition_t cond,
    hsa_signal_value_t compare_value,
    hsa_signal_value_t * return_value)
```

Wait until the value of a signal satisfies a given condition, or a user-provided timeout has elapsed.

Parameters

signal_handle
(in) Signal handle.

timeout
(in) Maximum wait duration. A value of zero indicates no maximum.

cond
(in) Condition used to compare the passed and signal values.

compare_value
(in) Value to compare with.

return_value
(out) Pointer to where the current value *signal_handle* must be read into. User allocated.

Return Values

HSA_STATUS_SUCCESS

HSA_STATUS_ERROR
If an error is signaled on the signal the user is waiting on. The function still returns the current value at the signal. The user may also inspect the value returned, when an error occurred.

HSA_STATUS_ERROR_INVALID_ARGUMENT
If the user is expecting an output but the pointer to the output signal value is invalid, or the passed signal is invalid.

HSA_STATUS_INFO_SIGNAL_TIMEOUT
If the signal wait has timed out.

Description

Waiting on a signal returns the current value at the signal. The wait may return before the condition is satisfied, the specified timeout has elapsed, or even before a valid value is obtained from the signal. It is the users burden to check the return status of the wait API before consuming the returned value.

```
hsa_status_t hsa_signal_wait_relaxed(
    hsa_signal_handle_t signal_handle,
    uint64_t timeout,
    hsa_signal_condition_t cond,
    hsa_signal_value_t compare_value,
    hsa_signal_value_t * return_value)
```

Wait until the value of a signal satisfies a given condition, or a user-provided timeout has elapsed.

Parameters

signal_handle
(in) Signal handle.

timeout

(in) Maximum wait duration. A value of zero indicates no maximum.

cond
: (in) Condition used to compare the passed and signal values.

compare_value
: (in) Value to compare with.

return_value
: (out) Pointer to where the current value *signal_handle* must be read into. User allocated.

Return Values

HSA_STATUS_SUCCESS

HSA_STATUS_ERROR
: If an error is signaled on the signal the user is waiting on. The function still returns the current value at the signal. The user may also inspect the value returned, when an error occurred.

HSA_STATUS_ERROR_INVALID_ARGUMENT
: If the user is expecting an output but the pointer to the output signal value is invalid, or the passed signal is invalid.

HSA_STATUS_INFO_SIGNAL_TIMEOUT
: If the signal wait has timed out.

Description

Waiting on a signal returns the current value at the signal. The wait may return before the condition is satisfied, the specified timeout has elapsed, or even before a valid value is obtained from the signal. It is the users burden to check the return status of the wait API before consuming the returned value.

A.5 Queues

enum **hsa_queue_type_t**

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

Values

HSA_QUEUE_TYPE_MULTI = 0
Multiple producers are supported.

HSA_QUEUE_TYPE_SINGLE = 1
Only a single producer is supported.

enum **hsa_queue_feature_t**

Queue features.

Values

HSA_QUEUE_FEATURE_DISPATCH = 1
Queue supports dispatch packets.

HSA_QUEUE_FEATURE_AGENT_DISPATCH = 2
Queue supports agent dispatch packets.

enum **hsa_service_queue_type_t**

Service queue type.

Values

NONE =0
Do not return a service queue, the service queue pointer will be set to NULL by the runtime.

COMMON
Use a common runtime provided service queue.

NEW
Create a new service queue.

APPLICATION_MANAGED
Create a queue that runtime will not manage, the application has to manage the queue manually.

```
typedef struct hsa_queue_s {
    hsa_queue_type_t queue_type;
    uint32_t queue_features;
    uint64_t base_address;
    hsa_signal_handle_t doorbell_signal;
    uint32_t size;
    uint32_t queue_id;
    uint64_t service_queue;
} hsa_queue_t
```

User mode queue. Queues are read-only: HSA agents can only modify the contents of the buffer pointed by *base_address*.

Data Fields

queue_type
: Queue type.

queue_features
: Queue features mask. See hsa_queue_feature_t. HSA applications should ignore any unknown set bits.

base_address
: Pointer to the base of the virtual memory which holds the AQL packets for the queue. Aligned to the AQL packet size.

doorbell_signal
: After writing a packet to the queue, the user must signal this handle with the most recent write index. Doorbell signals are allocated by the runtime during queue creation.

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

queue_id
: Queue identifier which is unique-per-process.

service_queue
: A pointer to another user mode queue that can be used by the HSAIL kernel to request system services. Provided by the application to the runtime API call when the queue is created. May be NULL, the system provided service queue or an application managed queue.

```
hsa_status_t hsa_queue_create(
    const hsa_agent_t * component,
    size_t size,
    uint32_t queue_type,
    hsa_service_queue_type_t service_queue_type,
    hsa_queue_t ** queue,
    hsa_queue_mailbox_t * mailbox)
```

Create a user mode queue.

Parameters

component
: (in) The component on which this queue is to be created.

size
: (in) Size of the queue memory in number of packets in is expected to hold. Required to be aligned with a power of two number of AQL packets.

queue_type
: (in) Type of the queue (only type 0, which is default in-order issue queue, is supported at this time).

service_queue_type
: (in) The user can choose between NONE (no service queue), COMMON (runtime provided service queue that is shared), NEW (require the runtime to create a new queue).

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

mailbox

(out) Mailbox to gather execution information to be used for debug trap. User may pass NULL here if the user doesn't want the mailbox created.

Return Values

HSA_STATUS_SUCCESS

HSA_STATUS_ERROR_INVALID_ARGUMENT
If the queue size is not a power of two, when the error message queue handle is invalid, or the component is not valid. This error code is also returned when the *queue* is NULL.

HSA_STATUS_ERROR_OUT_OF_RESOURCES
If there is a failure in allocation of an internal structure required by the core runtime library in the context of the queue creation. This error may also occur when the core runtime library needs to spawn threads or create internal OS-specific events. This error is also returned when a service queue or a user mode queue cannot be allocated.

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

Destroy a user mode queue.

Parameters

queue
(in) Queue.

Return Values

HSA_STATUS_SUCCESS

Description

After destruction it is considered undefined to access any field of the queue structure.

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

Inactivate a queue.

Parameters

queue
(in) Queue.

Return Values

HSA_STATUS_SUCCESS

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.

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

Retrieve read index of a queue.

Parameters

queue
   (in) Queue.

Returns

Read index.

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

Retrieve read index of a queue.

Parameters

queue
   (in) Queue.

Returns

Read index.

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

Retrieve write index of a queue.

Parameters

queue
   (in) Queue.

Returns

Write index.

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

Retrieve write index of a queue.

Parameters

queue
   (in) Queue.

Returns

Write index.

void **hsa_queue_store_read_index_relaxed**(
    hsa_queue_t * *queue*,
    uint64_t *val*)

Set the read index of a queue.

Parameters

queue
    (in) Queue.

val
    (in) The new value of the read index.

void **hsa_queue_store_read_index_release**(
    hsa_queue_t * *queue*,
    uint64_t *val*)

Set the read index of a queue.

Parameters

queue
    (in) Queue.

val
    (in) The new value of the read index.

void **hsa_queue_store_write_index_relaxed**(
    hsa_queue_t * *queue*,
    uint64_t *val*)

Set the write index of a queue.

Parameters

queue
    (in) Queue.

val
    (in) The new value of the write index.

void **hsa_queue_store_write_index_release**(
    hsa_queue_t * *queue*,
    uint64_t *val*)

Set the write index of a queue.

Parameters

queue
    (in) Queue.

val
    (in) The new value of the write index.

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

Atomically compare and set the write index of a queue.

Parameters

queue
(in) Queue.

expected
(in) The expected index value.

val
(in) Value to copy to the write index if *expected* matches the observed write index.

Returns

Previous value of the write index.

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

Atomically compare and set the write index of a queue.

Parameters

queue
(in) Queue.

expected
(in) The expected index value.

val
(in) Value to copy to the write index if *expected* matches the observed write index.

Returns

Previous value of the write index.

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

Atomically compare and set the write index of a queue.

Parameters

queue
(in) Queue.

expected
: (in) The expected index value.

val
: (in) Value to copy to the write index if *expected* matches the observed write index.

Returns

Previous value of the write index.

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

Atomically compare and set the write index of a queue.

Parameters

queue
: (in) Queue.

expected
: (in) The expected index value.

val
: (in) Value to copy to the write index if *expected* matches the observed write index.

Returns

Previous value of the write index.

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

Increment the write index of a queue by an offset.

Parameters

queue
: (in) Queue.

val
: (in) The value to add to the write index.

Returns

Previous value of the write index.

uint64_t **hsa_queue_add_write_index_acquire**(
    hsa_queue_t * *queue*,
    uint64_t *val*)

Increment the write index of a queue by an offset.

Parameters

queue
(in) Queue.

val
(in) The value to add to the write index.

Returns

Previous value of the write index.

uint64_t **hsa_queue_add_write_index_release**(
    hsa_queue_t * *queue*,
    uint64_t *val*)

Increment the write index of a queue by an offset.

Parameters

queue
(in) Queue.

val
(in) The value to add to the write index.

Returns

Previous value of the write index.

uint64_t **hsa_queue_add_write_index_acquire_release**(
    hsa_queue_t * *queue*,
    uint64_t *val*)

Increment the write index of a queue by an offset.

Parameters

queue
(in) Queue.

val
(in) The value to add to the write index.

Returns

Previous value of the write index.

A.6 Architected Queuing Language Support

enum **hsa_aql_packet_format_t**

Packet type.

Values

HSA_AQL_PACKET_FORMAT_ALWAYS_RESERVED = 0

HSA_AQL_PACKET_FORMAT_INVALID = 1

HSA_AQL_PACKET_FORMAT_DISPATCH = 2

HSA_AQL_PACKET_FORMAT_BARRIER = 3

HSA_AQL_PACKET_FORMAT_AGENT_DISPATCH = 4

```
typedef struct hsa_aql_packet_header_s {
    hsa_aql_packet_format_t format : 8;
    uint16_t barrier : 1;
    uint16_t acquire_fence_scope : 2;
    uint16_t release_fence_scope : 2;
    uint16_t reserved : 3;
} hsa_aql_packet_header_t
```

AQL packet header.

Data Fields

format
Packet type.

barrier
If set then processing of packets will only begin when all preceding packets are complete.

acquire_fence_scope
Determines the scope and type of the memory fence operation applied before the packet enters the active phase. The valid values are 1 (the fence is applied with component scope for the global segment) and 2 (the fence is applied across both component and sytem scope for the global segment).

release_fence_scope
Determines the scope and type of the memory fence operation applied after kernel completion but before the packet is completed. The valid values are 1 (the fence is applied with component scope for the global segment) and 2 (the fence is applied across both component and sytem scope for the global segment).

reserved
Must be zero.

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

AQL dispatch packet.

Data Fields

header
: Packet header.

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

reserved
: Reserved, must be zero.

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

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

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

reserved2
: Reserved. Must be zero.

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

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

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

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

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

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

kernarg_address
: Address of memory containing kernel arguments.

reserved3
  Reserved.

completion_signal
  HSA signaling object handle used to indicate completion of the job.

```
typedef struct hsa_aql_agent_dispatch_packet_s {
    hsa_aql_packet_header_t header;
    uint16_t type;
    uint32_t reserved2;
    uint64_t return_location;
    uint64_t arg0;
    uint64_t arg1;
    uint64_t arg2;
    uint64_t arg3;
    uint64_t reserved3;
    hsa_signal_handle_t completion_signal;
} hsa_aql_agent_dispatch_packet_t
```

Agent dispatch packet.

Data Fields

header
  Packet header.

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

reserved2
  Reserved. Must be 0.

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

arg0
  64-bit direct or indirect argument.

arg1
  64-bit direct or indirect argument.

arg2
  64-bit direct or indirect argument.

arg3
  64-bit direct or indirect argument.

reserved3
  Reserved. Must be 0.

completion_signal
  HSA signaling object handle used to indicate completion of the job.

```
typedef struct hsa_aql_barrier_packet_s {
    hsa_aql_packet_header_t header;
    uint16_t reserved2;
    uint32_t reserved3;
    hsa_signal_handle_t dep_signal0;
    hsa_signal_handle_t dep_signal1;
    hsa_signal_handle_t dep_signal2;
    hsa_signal_handle_t dep_signal3;
    hsa_signal_handle_t dep_signal4;
    uint64_t reserved4;
    hsa_signal_handle_t completion_signal;
} hsa_aql_barrier_packet_t
```

Barrier packet.

Data Fields

header
  Packet header.

reserved2
  Reserved. Must be zero.

reserved3
  Reserved. Must be zero.

dep_signal0
  Dependent signal object.

dep_signal1
  Dependent signal object.

dep_signal2
  Dependent signal object.

dep_signal3
  Dependent signal object.

dep_signal4
  Dependent signal object.

reserved4
  Reserved. Must be zero.

completion_signal
  HSA signaling object handle used to indicate completion of the job.

A.7 Images and Samplers

enum **hsa_image_format_capability_t**

Image format capability returned by hsa_image_get_format_capability.

Values

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

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

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

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

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

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

enum **hsa_agent_image_access_t**

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

Values

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

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

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

enum **hsa_image_geometry_t**

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

Values

HSA_IMAGE_GEOMETRY_1D = 0
One-dimensional image addressed by width coordinate.

HSA_IMAGE_GEOMETRY_2D = 1
Two-dimensional image addressed by width and height coordinates.

HSA_IMAGE_GEOMETRY_3D = 2
Three-dimensional image addressed by width, height, and depth coordinates.

HSA_IMAGE_GEOMETRY_1DA = 3
: Array of one-dimensional images with the same size and format. 1D arrays are addressed by index and width coordinate.

HSA_IMAGE_GEOMETRY_2DA = 4
: Array of two-dimensional images with the same size and format. 2D arrays are addressed by index and width and height coordinates.

HSA_IMAGE_GEOMETRY_1DB = 5
: One-dimensional image interpreted as a buffer with specific restrictions.

HSA_IMAGE_GEOMETRY_2DDEPTH = 6
: Two-dimensional depth image addressed by width and height coordinates.

HSA_IMAGE_GEOMETRY_2DADEPTH = 7
: Array of two-dimensional depth images with the same size and format. 2D arrays are addressed by index and width and height coordinates.

enum **hsa_image_channel_type_t**

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

Values

HSA_IMAGE_CHANNEL_TYPE_SNORM_INT8 = 0

HSA_IMAGE_CHANNEL_TYPE_SNORM_INT16 = 1

HSA_IMAGE_CHANNEL_TYPE_UNORM_INT8 = 2

HSA_IMAGE_CHANNEL_TYPE_UNORM_INT16 = 3

HSA_IMAGE_CHANNEL_TYPE_UNORM_INT24 = 4

HSA_IMAGE_CHANNEL_TYPE_UNORM_SHORT_555 = 5

HSA_IMAGE_CHANNEL_TYPE_UNORM_SHORT_565 = 6

HSA_IMAGE_CHANNEL_TYPE_UNORM_SHORT_101010 = 7

HSA_IMAGE_CHANNEL_TYPE_SIGNED_INT8 = 8

HSA_IMAGE_CHANNEL_TYPE_SIGNED_INT16 = 9

HSA_IMAGE_CHANNEL_TYPE_SIGNED_INT32 = 10

HSA_IMAGE_CHANNEL_TYPE_UNSIGNED_INT8 = 11

HSA_IMAGE_CHANNEL_TYPE_UNSIGNED_INT16 = 12

HSA_IMAGE_CHANNEL_TYPE_UNSIGNED_INT32 = 13

HSA_IMAGE_CHANNEL_TYPE_HALF_FLOAT = 14

HSA_IMAGE_CHANNEL_TYPE_FLOAT = 15

enum **hsa_image_channel_order_t**

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

Values

HSA_IMAGE_CHANNEL_ORDER_A = 0

HSA_IMAGE_CHANNEL_ORDER_R = 1

HSA_IMAGE_CHANNEL_ORDER_RX = 2

HSA_IMAGE_CHANNEL_ORDER_RG = 3

HSA_IMAGE_CHANNEL_ORDER_RGX = 4

HSA_IMAGE_CHANNEL_ORDER_RA = 5

HSA_IMAGE_CHANNEL_ORDER_RGB = 6

HSA_IMAGE_CHANNEL_ORDER_RGBX = 7

HSA_IMAGE_CHANNEL_ORDER_RGBA = 8

HSA_IMAGE_CHANNEL_ORDER_BGRA = 9

HSA_IMAGE_CHANNEL_ORDER_ARGB = 10

HSA_IMAGE_CHANNEL_ORDER_ABGR = 11

HSA_IMAGE_CHANNEL_ORDER_SRGB = 12

HSA_IMAGE_CHANNEL_ORDER_SRGBX = 13

HSA_IMAGE_CHANNEL_ORDER_SRGBA = 14

HSA_IMAGE_CHANNEL_ORDER_SBGRA = 15

HSA_IMAGE_CHANNEL_ORDER_INTENSITY = 16

HSA_IMAGE_CHANNEL_ORDER_LUMINANCE = 17

HSA_IMAGE_CHANNEL_ORDER_DEPTH = 18

HSA_IMAGE_CHANNEL_ORDER_DEPTH_STENCIL = 19

enum **hsa_sampler_addressing_mode_t**

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

Values

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

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

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

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

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

```
enum hsa_sampler_coordinate_mode_t
```

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

Values

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

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

```
enum hsa_sampler_filter_mode_t
```

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

Values

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

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

```
typedef struct hsa_image_handle_s {
    uint64_t handle;
} hsa_image_handle_t
```

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

Data Fields

handle
: HSA component specific handle to the image.

```
typedef struct hsa_image_info_s {
    size_t image_size;
    size_t image_alignment;
} hsa_image_info_t
```

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

Data Fields

image_size
: Component specific image data size in bytes.

image_alignment
: Component specific image data alignment in bytes.

```
typedef struct hsa_image_format_s {
    hsa_image_channel_type_t channel_type;
    hsa_image_channel_order_t channel_order;
} hsa_image_format_t
```

Image format descriptor (attributes of the image format).

Data Fields

channel_type
Channel type of the image.

channel_order
Channel order of the image.

```
typedef struct hsa_image_descriptor_s {
    hsa_image_geometry_t geometry;
    size_t width;
    size_t height;
    size_t depth;
    size_t array_size;
    hsa_image_format_t format;
} hsa_image_descriptor_t
```

Implementation-independent HSA Image descriptor.

Data Fields

geometry
Geometry of the image.

width
Width of the image in components.

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

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

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

format
Format of the image.

```
typedef struct hsa_image_offset_s {
    uint32_t x;
    uint32_t y;
    uint32_t z;
} hsa_image_offset_t
```

3D image coordinate offset for image manipulation.

Data Fields

x

X coordinate for the offset.

y
: Y coordinate for the offset.

z
: Z coordinate for the offset

```
typedef struct hsa_image_range_s {
    uint32_t width;
    uint32_t height;
    uint32_t depth;
} hsa_image_range_t
```

Three-dimensional image range description.

Data Fields

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

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

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

```
typedef struct hsa_image_region_s {
    hsa_image_offset_t image_offset;
    hsa_image_range_t image_range;
} hsa_image_region_t
```

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

Data Fields

image_offset
: Offset in the image (in coordinates).

image_range
: Dimensions of the image range (in coordinates).

```
typedef struct hsa_sampler_handle_s {
    uint64_t handle;
} hsa_sampler_handle_t
```

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

Data Fields

handle
: Component-specific HSA sampler.

```
typedef struct hsa_sampler_descriptor_s {
    hsa_sampler_coordinate_mode_t coordinate_mode;
    hsa_sampler_filter_mode_t filter_mode;
    hsa_sampler_addressing_mode_t address_mode;
} hsa_sampler_descriptor_t
```

Implementation-independent sampler descriptor.

Data Fields

coordinate_mode
    Sampler coordinate mode describes the normalization of image coordinates.

filter_mode
    Sampler filter type describes the type of sampling performed.

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

```
hsa_status_t hsa_image_get_format_capability(
    const hsa_agent_t * agent,
    const hsa_image_format_t * image_format,
    hsa_image_geometry_t image_geometry,
    uint32_t * capability_mask)
```

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

Parameters

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

image_format
    (in) Image format.

image_geometry
    (in) Geometry of the image.

capability_mask
    (out) Image format capability bit-mask.

Return Values

HSA_STATUS_SUCCESS

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

Description

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

```
hsa_status_t hsa_image_get_info(
    const hsa_agent_t * agent,
    const hsa_image_descriptor_t * image_descriptor,
    hsa_image_info_t * image_info)
```

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

Parameters

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

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

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

Return Values

HSA_STATUS_SUCCESS

HSA_STATUS_ERROR_INVALID_ARGUMENT
If any of the arguments is NULL.

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

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

Description

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

```
hsa_status_t hsa_image_create_handle(
    const hsa_agent_t * agent,
    const hsa_image_descriptor_t * image_descriptor,
    const void * image_data,
    hsa_agent_image_access_t agent_access,
    hsa_image_handle_t * image_handle)
```

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

Parameters

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

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

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

agent_access
(in) Access pattern of the image by the HSA agent.

image_handle
(out) Agent-specific image handle.

Return Values

HSA_STATUS_SUCCESS

HSA_STATUS_ERROR_INVALID_ARGUMENT
: If any of the arguments is NULL.

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

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

Description

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

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

agent_access defines how the HSA agent expects to use the image handle. The image format specified in *image_descriptor* must be capable by the HSA agent for the intended access pattern.

Image handles with different agent_access can be created using the same image data with exactly the same image descriptor as long as HSA_IMAGE_FORMAT_ACCESS_INVARIANT_IMAGE_DATA is reported by hsa_image_get_format_capability for the image format specified in the image descriptor. Images of non-linear SRGB channel order types can share the same image data with its equivalent linear RGB channel order types, provided the rest of image descriptor parameters are identical.

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

```
hsa_status_t hsa_image_import(
    const hsa_agent_t * agent,
    const void * src_memory,
    size_t src_row_pitch,
    size_t src_slice_pitch,
    hsa_image_handle_t dest_image_handle,
    const hsa_image_region_t * image_region,
    const hsa_signal_handle_t * completion_signal)
```

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

Parameters

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

src_memory
: (in) Source memory.

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

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

dest_image_handle

(in) Destination Image handle.

image_region
(in) Image region to be updated.

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

Return Values

HSA_STATUS_SUCCESS

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

Description

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

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

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

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

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

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

```
hsa_status_t hsa_image_export(
    const hsa_agent_t * agent,
    hsa_image_handle_t src_image_handle,
    void * dst_memory,
    size_t dst_row_pitch,
    size_t dst_slice_pitch,
    const hsa_image_region_t * image_region,
    const hsa_signal_handle_t * completion_signal)
```

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

Parameters

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

src_image_handle
(in) Source image handle.

dst_memory
(in) Destination memory.

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

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

image_region
(in) Image region to be exported.

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

Return Values

HSA_STATUS_SUCCESS

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

Description

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

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

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

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

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

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

```
hsa_status_t hsa_image_copy(
    const hsa_agent_t * agent,
    hsa_image_handle_t src_image_handle,
    hsa_image_handle_t dst_image_handle,
    const hsa_image_region_t * image_region,
    const hsa_signal_handle_t * completion_signal)
```

Copies a region from one image to another.

Parameters

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

src_image_handle
  (in) Source image handle.

dst_image_handle
  (in) Destination image handle.

image_region
  (in) Image region to be copied.

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

Return Values

HSA_STATUS_SUCCESS

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

Description

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

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

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

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

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

```
hsa_status_t hsa_image_clear(
    const hsa_agent_t * agent,
    hsa_image_handle_t image_handle,
    const float data,
    const hsa_image_region_t * image_region,
    const hsa_signal_handle_t * completion_signal)
```

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

Parameters

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

image_handle
(in) Image to be cleared.

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

image_region
(in) Image region to clear.

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

Return Values

HSA_STATUS_SUCCESS

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

Description

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

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

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

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

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

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

```
hsa_status_t hsa_image_destroy_handle(
    const hsa_agent_t * agent,
    hsa_image_handle_t * image_handle)
```

Destroys the specified image handle.

Parameters

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

image_handle
(in) Image handle.

Return Values

HSA_STATUS_SUCCESS

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

Description

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

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

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

```
hsa_status_t hsa_sampler_create_handle(
    const hsa_agent_t * agent,
    const hsa_sampler_descriptor_t * sampler_descriptor,
    hsa_sampler_handle_t * sampler_handle)
```

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

Parameters

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

sampler_descriptor
(in) Implementation-independent sampler descriptor.

sampler_handle
(out) Component-specific sampler handle.

Return Values

HSA_STATUS_SUCCESS

HSA_STATUS_ERROR_INVALID_ARGUMENT
If any of the arguments is NULL.

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

Description

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

```
hsa_status_t hsa_sampler_destroy_handle(
    const hsa_agent_t * agent,
    hsa_sampler_handle_t * sampler_handle)
```

Destroys the specified sampler handle.

Parameters

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

sampler_handle
(in) Sampler handle.

Return Values

HSA_STATUS_SUCCESS

HSA_STATUS_ERROR_INVALID_ARGUMENT
If any of the arguments is NULL.

Description

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

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

Bibliography

[1] HSA Foundation. The HSA Programmer's Reference Manual. Technical report, HSA Foundation, 2013. 14, 35

[2] HSA Foundation. The HSA Platform System Architecture Specification. Technical report, HSA Foundation, 2014. 18