HSA Core API Programmers Reference Manual

[c1] 0.170

10th November 2013

Draft

[c1] *AMD*: ~~0.169~~

Contents

1 Introduction

1.1 Overview

The runtime system contains the user mode management software required to execute a compiled HSA program. It ties in the information required in the compilation unit to execution on queues, it abstracts HSA Component functionality, and also exposes HSA features to developers of applications, libraries and programming tools.

The HSA core runtime API aims to be a thin layer that abstracts a common set of HSA features and allows for composition and support for different higher-level functionality that various programming models and languages can in turn be built on top of. The core base runtime aims to be a portable target to higher-level services and subsystems and programming model/language runtime systems.

This document specifies and describes the HSA core base API.

1.1.1 Goals

The HSA systems architecture requirements working group has specified a set of requirements that form the minimal feature-set of a HSA system. These features must be abstracted via API to enable users to program and utilize an HSA system. The goal of this document is to describe the API that abstracts these features and makes them available to a HSA user. The goal of this document is to specify necessary and sufficient API to support and enable utilization of following features that have been defined as requirements in the systems architecture requirements specification:

- Shared Virtual Memory

- Cache Coherency Domains
- HSA Platform Topology Discovery
- Memory-Based Signaling and Synchronization
- User Mode Queuing
- Preemptive HSA Component Context Switching
- Architected Queuing Language (AQL)
- HSA Component IEEE754-2008 Floating Point Exception Reporting
- HSA Component Hardware Debug Infrastructure
- Efficient Syscall Infrastructure

The user mode queue and AQL feature definitions in the HSA Systems Architecture Requirements (SAR) Specification enable architected support for performing a user mode dispatch in an HSA component – creating an AQL packet in a user mode queue and signaling the HW that such a packet is written is sufficient to initiate execution on an HSA component. The goal of this specification is to ensure that API definition doesn't preclude architected dispatch.

1.1.2 Approach

This document defines and groups API by functional area and describes how the features described in HSA Systems Architecture Requirement Specification document are abstracted.

The primary memory type, as defined in the systems architecture requirements document, requires no specific allocation or support from the runtime API. Hence no specific API are required to enable users to utilize Shared Virtual Memory or Cache Coherence Domains – these are realized in the HW implementation of an HSA system. The other features listed in 1.1.1 each form their on functional area in the HSA core base API specification.

1.2 Infrastructure and Execution Flow

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 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 Programmers Reference Manual (PRM) [**?**] defines HSAIL Virtual ISA and Programming Model, serves as a Compiler Writers 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.

1.2.1 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 the information 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.

1.2.2 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 can has sufficient information to execute the user program. The details of this compilation process and symbol resolution are discussed in Section 2.8.

1.2.3 Execution of Kernel

The Systems Architecture Requirements (SAR) document specifies the structure of the *packets* (i.e. commands) that can 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 (this is

discussed in Section 2.7), 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. Core runtime defines a structure and update API to inform the HW that 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 via. 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.

1.2.4 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 API 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 the execution of the kernel complete (success or error). The user can wait on this signal to determine kernel completion. Different kinds of errors and their meaning are discussed in Section 2.1.

```
int main(int argc, char **argv)
{
unsigned int count;
uint32_t kernel_directive = atoi(argv[2]);
uint64_t kernel_input = (uint64_t)atoi(argv[3]);
const hsa_agent_t *component_list = NULL;
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;
uint64_t index;

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

  status = hsa_component_get_list(&count, &component_list);
  if(count <= 0 || status != HSA_STATUS_SUCCESS) {
    assert(HSA_STATUS_SUCCESS == hsa_close());
    exit(1);
  }

  component = &component_list[0];

  status = hsa_queue_create(component, 1024, 0, runtime_context,
     &queue);
  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, kernel_directive,
     NULL, 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 API Specification and Description

2.1 Core Runtime API Synchronous and Asynchronous Errors and Asynchronous Notification

Errors handling in the core runtime can broadly be classified into two categories: synchronous error handling and asynchronous error~~handling.~~ /notification handling.

Synchronous errors are always reported when the ~~API returns, in some cases that warrant an application abort, even before the API returns and control is returned to the application.~~ call returns. They indicate if the API returned a success or an error.

Asynchronous errors can occur ~~because HSA system because~~ due to various reasons: (i) the HSA system supports asynchronous execution and ~~they~~ errors can be triggered by activity in packet processor, executing kernels, their actions and memory accesses. ~~Asynchronous errors can also occur~~ (ii) when an error is detected during the execution of a kernel, the completion signal (if present) will be signaled with an error indication value. (iii) for providing *information/warning* (not as an exception in expected behavior but by definition) and 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.

2.1.1 Synchronous Errors

When a core runtime API is called and does not execute successfully (each API call discussed in this chapter defines what constitutes a successful execution), the core runtime returns a status that can potentially help ~~user determine the~~ the user determine a cause of the such unsuccessful 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 explanation. ~~In~~

addition to being a mechanism for reporting successful or erroneous execution, the status returned by the core runtime API calls will also be used for providing (not as an exception in expected behavior but by definition) and this information 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.

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 four categories of the results: *success* , *info* , *error* and . . The *info* status definition is discussed in Section 2.1.2.

Success status is a single value, `HSA_STATUS_SUCCESS`. Description of every core runtime API call that returns `hsa_status_t` explains what a successful behavior is.

status requires users interpretation. Consider the example where a user calls the initialize API to initialize the core runtime and the return status is(to indicate that the core runtime has already been initialized). This result may be interpreted differently in different usage scenarios. The constants used for info status are restricted to a particular range of values with in the enumeration. They start at a value greater than and end at a value less than . Either of the or constants will not deliberately be returned as a status for any API by the core runtime. Name of any constant that indicates info status is prefixed by the expected successful behavior for that API is. The value of `HSA_STATUS_SUCCESS` is always 0.

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 leads lead to a failure that cascaded into a failure in this API). The constants used for error status are restricted to a particular the negative range of values with in the `hsa_status_t` enumeration. They start at a value greater than and end at a value less than . Either of the or constants will not deliberately be returned as a status for any API by the core runtime. Errors must always have a negative value. Name of any constant that indicates error status is prefixed by `HSA_STATUS_ERROR`. Errors could potentially be implementation.

, or status can be either info or error but is implementationdefined. The implementation specific status starts at a constant value and ends at – both these constants will not deliberately be returned as a status for any API by the core runtime.

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 this statusa 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 number returned will need to be decoded by the usernegative number returned for error may not correspond to a particular enumeration constant. To support thisquery of additional information on synchronous errors, the core runtime defines the following API:

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

input_status is input argument. Any unsuccessful API return status that the user is seeking more information on.

status_info user allocated, output. In addition to the string. This value could be 0 and in itself (without status_info_string) may not be independently interpretable by the user.

status_info_string output from the API, a ISO/IEC 646 encoded english language string that potentially describes the error status. The string terminates in a ISO 646 defined NUL char.

This API returns `HSA_STATUS_SUCCESS` if one or both of the *status_info* and *status_info_string* have been successfully updated with information regarding the input *input_status*. Otherwise it returns one of the following errors:

- `HSA_STATUS_NONE` when no additional information is available regarding the status user requested.
- `HSA_STATUS_ERROR_INVALID_ARGUMENT` if a NULL value is passed for either of the arguments

2.1.2 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.

Asynchronous Notification of Information or Warning

The information/warning status is represented by by a value greater than 0 within the `hsa_status_t` enumeration. The status is up to users 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 2.2 or via **`hsa_notification_callback_register`** API, which is defined as follows:

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

notification_callback input. the callback that the user is registering, the callback is called with info as a parameter. User can read the structure and access its elements.

user_data input. the user data to call the callback with. info → user_data will be filled with value when the callback is called.

context the runtime context that this callback is being registered for.

The *context* parameter is used to identify 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. Section 2.2 discusses the context in detail. The **hsa_notification_callback_register** API can return one of the following errors:

- ▷ 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.

One of the arguments of the notification callback is a structure that contains notification information. The structure is defined as follows:

```
typedef struct hsa_notification_info_s{
    hsa_status_t status;
    void         *ptr_info;
    char         *string_info;
    void         *user_data;
} hsa_notification_info_t;
```

status the info status enum value

ptr_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

Asynchronous Notification of Errors

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. In order to deal with asynchronous errors, the core runtime ~~defines error message queues. Error message queues are message queues of fixed size, created by the user or the runtime in order to communicate asynchronous error information. Each element that can be in the error message queue represents an error. It has information on the type of error, identification information on which packet on which queue caused it and a platform clock value (see Section 2.3) indicating when the error was placed on the queue. The structure that represents an individual error in the error message queue~~ supports asynchronous error callbacks. The asynchronous error callback may be registered via **hsa_open** API discussed in Section 2.2 or via **hsa_error_callback_register** API, which is defined as follows:

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

error_callback input. 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 input. the user data to call the callback with. info-¿user_data will be filled with value when the callback is called.

context the runtime context that this callback is being registered for.

~~The user can create and destroy a message queue using the following API:~~

~~Size of the queue represents the number of asynchronous messages this queue can handle. When the error message queue is full, new asynchronous messages will be ignored until user has the opportunity to consume the old ones. Note that the~~ Details on how association of the callback could be done with asynchronous activities are discussed in Sections 2.2 and 2.5. The *message*~~*context* is a pointer to – this is an opaque handle to the message queue and is an input to all message queue related operations.~~

~~This API returns if an error message queue of a requested size has been successfully created . At the time of creation, the error message queuehas no association to any particular kind of asynchronous activities. Details on how association of this queue to asynchronous activities are discussed in Sections 2.2 and 2.5.~~ parameter is used to identify 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. 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*. This API can return one of the following errors:

- ▷ 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 message queue creation~~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 ~~*size*~~*info* is ~~zero or if messageis~~ NULL.

~~The API to wait on a message queue for error~~ One of the arguments of the notification callback is a structure that contains notification information. The structure is defined as follows:

~~Note that that the errors~~is user allocated, and an output to the API. ~~The wait API is a blocking API. When errors occur, the first error in the message queue is copied into the error~~before the status ~~is returned.~~

~~Error message queues can work across different components.~~

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

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.

ptr_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 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 intepreted based on the error_type

reserved2 additional info to be intepreted based on the error_type

reserved3 additional info to be intepreted based on the error_type

2.2 Open and close API

Since 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 ~~initialize and finish~~ open (or close) the HSA runtime multiple times with-in the same process ~~context~~ and potentially with in multiple threads – only a single instance of the runtime, per-process, will exist. ~~Initialization of the HSA core runtime is required for an application to use the runtime API calls~~

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 for the first time 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 also uses a library that creates queues and submits work to them. Both the library and the application want to register callbacks, but to capture notifications/errors of their specific usage. Context helps identify the different users (with in 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* . ~~The application is also expected to finish the runtime once it no longer needs to use the runtime API. Calling any runtime API function without initializing itresults in an undefined behavior~~

This section defines four new API, **hsa_open** to open the runtime instance, **hsa_close** to close it, **hsa_context_acquire** to create a new context (and increments the reference count), and, **hsa_context_release** to release the acquired context.

Invocation of **hsa_open** initializes the HSA runtime if it is already not initialized. It is allowed for applications to ~~initialize core runtime~~ invoke **hsa_open** multiple times and do ~~a matching set of finishes~~multiple **hsa_close** API calls. Any subsequent initialization of the HSA runtime library, while it is still initialized, is allowed. The runtime implementation ~~relies on internal reference counting~~ exposes a reference counting mechanism to the user via the runtime context. Reference counting is a mechanism that allows the runtime to keep ~~an internal~~ a count of the number of ~~calls to~~ different libraries or threads using the runtime API. This ensures that the runtime stays active until ~~the last call.~~

~~An error message queue is an output of the API. This error message queue serves as the error message queue for the runtime (this is discussed again in Section 2.5). All invocations of the return the same default error message queue handle, i.e.there is only one default error message queue per instance runtime. This API does return a failure when no HSA components were discovered~~ a **hsa_close** is called by the user when the reference count represented by the runtime ~~.~~ context is 1.

The definition of the **hsa_open** API is as follows:

```
hsa_status_t hsa_open(void ( *notification_callback)(const
    hsa_notification_info_t *info),
    void *notification_user_data,
    void ( *error_callback)(const hsa_async_error_info_t *info),
    void *async_error_user_data, hsa_runtime_context_t **context);
```

notification_callback input. the callback that the user is registering, the callback is called with info structure. User can read the structure and access its elements.

notification_user_data input. the user data to call the callback with. info-¿user_data will be filled with value when the callback is called.

error_callback input. the callback that the user is registering, the callback is called with info structure. User can read the structure and access its elements.

error_user_data input. the user data to call the callback with. info-¿user_data will be filled with value when the callback is called.

context output, user allocated. A type for reference counting.

The ~~initialize~~ open API returns `HSA_STATUS_SUCCESS` if the initialization was successful. Otherwise it returns one of the following errors:

- ~~when the core runtime library has already been initialized.~~

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

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

- ~~Any implementation specific error has a error value >(see 2.1 for details)~~`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.

If the HSA runtime is already initialized, an asynchronous notification is generated by the runtime and `HSA_STATUS_SUCCESS` is returned. It is the users burden to define a callback that would potentially invoke the *acquire* API on reference counting as multiple **`hsa_open`** calls do not automatically increment the reference count.

The runtime defines **`hsa_close`** as the corresponding API call to finalize the use of the runtime API. This API does not take in any input. This API ~~merely~~ updates the reference count ~~until~~ once upon its first invocation. Once the reference count ~~indicates that number of initializations has been matched with the number of finishes. Once this match is determined, the runtime proceeds to freeing resources allocated during initialization~~is 0, 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 ~~finish~~ close while the other is trying to ~~initialize~~acquire the runtime context. The core runtime implementation handles such scenarios by defining that an acquire on context that represents a closed runtime instance will fail. The API for **`hsa_close`** is defined as follows:

```
hsa_status_t hsa_close();
```

The ~~finish~~ close API returns HSA_STATUS_SUCCESS if the ~~finish~~ close was successful. Otherwise, it returns one of the following errors:

- ~~if the number of times initialize has been called is more than the number of time finish has been called.~~

  HSA_STATUS_ERROR_NOT_INITIALIZED if the ~~finish~~ close was called (a) either before the runtime was every initialized, or (b) after it has already been ~~finished~~successfully closed.

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

Once runtime is opened and a context obtained, user can control its reference counting and generate new contexts. For example, if an **hsa_open** call resulted in an asynchronous notification that a prior open successfully initialized the runtime, and the user callback implementation catches this notification, the user's callback can explicitly request an acquire on the context.

The HSA core runtime API for an acquire on a context, **hsa_context_acquire**, is defined as follows:

```
hsa_status_t hsa_context_acquire(hsa_runtime_context_t *input_context,
    hsa_runtime_context_t **output_context);
```

input_context input, user allocated. the context that the user is expilicitly reference counting, increment reference count if not 0

output_context output, user allocated. the implementation may chose to return a different context on an acquire.

The open API returns HSA_STATUS_SUCCESS if the acquire was successful and if *output_context* holds the new context generated. Otherwise it returns one of the following errors:

- ~~Any implementation specific error has a error value >(see 2.1 for details) .~~
  HSA_STATUS_ERROR_NOT_INITIALIZED if the **hsa_acquire_context** was called (a) either before the runtime was every initialized, or (b) after it has already been closed.

The corresponding release API, **hsa_context_release** is defined as follows:

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

context input, user allocated. the context that the user is expilicitly reference counting, decrement reference count if not 1

The **hsa_context_release** API returns HSA_STATUS_SUCCESS if the release was successful. Otherwise it returns one of the following errors:

- HSA_STATUS_ERROR_NOT_INITIALIZED if the **hsa_release_context** was called (a) either before the runtime was after reference count has already reached a value of 0.

2.3 HSA Topology and Component

The HSA platform topology information is provided by the runtime via. data structures so user can gather details about how a HSA system/platform decided to expose its architectural details such as components, memory, caches and connectivity (platform topology requirement is described in the SAR document. The SAR document also depicts possible complex topologies and potential data-structure associations). This information could be utilized by the user in different ways including decisions on where to execute a particular user task. Core runtime specification defines the topology table data structure and other data structures to represent topology hierarchy. As a part of the HSA core base API functionalityAfter the core runtime is initialized with **hsa_open**, the user may query information about the create a local copy of the topology information using the API **hsa_topology_table_create**. The user can parse this table representing the HSA system to gather details such as the number of host compute units and different HSA Components on the system with local access to a particular set of memory resourcesattached to the node's memory controller and appropriate HSA-compliant access attributes. This information could be utilized by the user in different ways including decisions on where to execute a particular user task. .

Platform topology is represented as a table in the HSA runtime implementation. See SAR section on **Requirement: HSA Platform Topology Discovery** for description on the potential structureof this table . There are five elements to the topology information : agents , caches, physical memory, and when applicable, Translation Lookaside Buffer (TLB)and IThe **hsa_topology_table_create** API is defined as follows:

```
hsa_status_t hsa_topology_table_create(hsa_topology_header_t **header);
```

header output, runtime allocated. The topology header, this includes the base pointers to the rest of the topology table.

The API returns HSA_STATUS_SUCCESS if the table has been successfully created and returned via *header*. Otherwise, it returns one of the following errors:

- 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 create of table header or the actual table.

The table structure is shown in Figure 2.1. The first entity in the table is a table header. This is the output of the **hsa_topology_table_create** API. The table header is defined by the following structure:

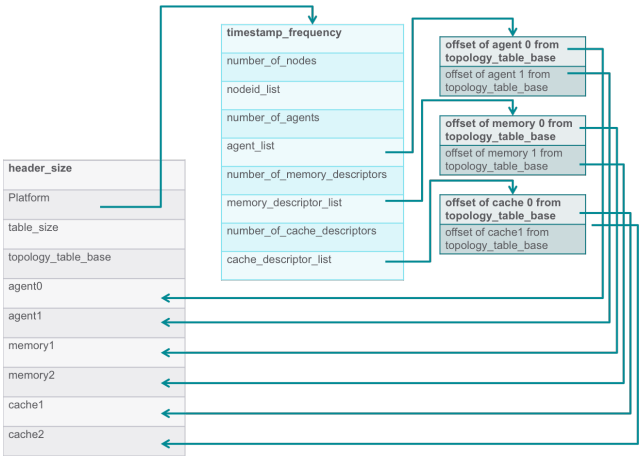

Figure 2.1: Structure of the Topology Table

```
typedef struct hsa_topology_header_s{
    uint32_t        header_size;
    hsa_platform_t  platform;
    uint32_t        table_size;
    void            *topology_table_base;
} hsa_topology_header_t;
```

header_size size of the header

platform the hierarchical platform structure that abstracts the table

table_size size of the table

topology_table_base table base address

The table header structure includes the platform structure (`hsa_platform_t`). The platform information in the platform structure includes size/~~O interconnects~~offset-array pairs for HSA agents (`hsa_agent_t`), memory(`hsa_memory_descriptor_t` and cache (`hsa_cache_descriptor_t`). HSA platform can have a ~~complex~~ hierarchical structure with multiple ~~nodes and each node with multiple components~~components/agents and physical memories. ~~For its user, the core runtime encapsulates the topology information. The topology table is internal to runtime. The runtime defines a structure,~~ The `hsa_platform_t` ~~that represents~~ structure also includes properties such as the clock frequency that are common across the platform and also links to various elements in the topology table (see Figure ~~??~~ ~~). It~~ 2.1).

The platform structure is defined as follows:

```
typedef struct hsa_platform_s{
    uint32_t                hsa_system_timestamp_frequency_mhz;
```

```
    uint8_t                 number_of_nodes;
    uint32_t                *node_id;
    uint32_t                number_of_agents;
    uint32_t                *agent_offset_list;
    uint32_t                number_memory_descriptors;
    uint32_t                **memory_descriptor_offset_list;
    uint32_t                number_cache_descriptors;
    uint32_t                *cache_descriptors;
} hsa_platform_t;
```

hsa_system_timestamp_frequency_mhz 1-400MHz.

number_of_nodes number of different nodes in this platform configuration.

node_id ids of the nodes.

number_of_agents number of agents.

agent_offset_list agent list, refers to the offsets in platform table.

number_memory_descriptors number of the different types of memories available to this agent.

memory_descriptor_offset_list 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

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

~~maptoagentMapping of Agent and Platform Structures to Topology Table~~

When no information is available about a particular element, the corresponding *number_<element>s* field is set to zero by the runtime in the platform structure. Platform structure maps to the agents, cache and physical memory, etc. in the topology table for all the nodes in the platform.

The core runtime defines the following structure to represent cache:

```
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;
    bool     *is_inclusive;
} hsa_cache_descriptor_t;
```

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 mult-level cache)

associativity associativity of this cache, array with number of entries = number of levels

cache_size size at each level, this array is of size = levels

cache_line_size cache line size at each level, this array is of size = levels

is_inclusive is the cache inclusive with the level above? The size of this array is level-1

The structure holds associativity, cache size, cache line size for all levels of cache and the inclusivity property for all but the last level. Each cache in the HSA system has a unique cache ID identifying it.

The ~~structure representing TLB is very similar to the one for cache, with associativity, levels and inclusivity. It is defined as follows:~~

~~The~~ memory descriptor structure represents a physical memory block or region and includes elements to provide provide bandwidth, interleave characteristics and latency for accessing memory. Implementations may choose not to provide memory bandwidth or latency information. The memory descriptor structure is defined as follows:

```
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       physical_address_base;
    uint64_t       virtual_address_base;
    uint64_t       size_in_bytes;
    uint64_t       peak_bandwidth_mbps;
} hsa_memory_descriptor_t;
```

hsa_node_id id of the node this memory belongs to.

hsa_memory_id unique identified for this memory with in the system.

supported_segment_type_mask information on segments that can use this memory.

physical_address_base base of the physical address for 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

The structure:

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

global:1 if bit is set, the element/mask represents global segment

private:1 if bit is set, the element/mask represents private segment

group:1 if bit is set, the element/mask represents group segment

kernarg:1 if bit is set, the element/mask represents kernarg segment

readonly:1 if bit is set, the element/mask represents readonly segment

spill:1 if bit is set, the element/mask represents spill segment

arg:1 if bit is set, the element/mask represents arg segment

reserved:1 reserved

can represent any combination of of the 7 HSA segments, a single bit for each segment.

The HSA Agent data structure represents an HSA component when the ~~*is*~~*agent_type* field in the agent structure is set to a ~~1.~~ 1 (i.e. bit 0 is set to 1). The structure contains elements that describe its properties. Each component has access to coherent global memory (the HSA global segment, and as per the requirement defined in SAR, has access to other segments as well). The *agent_type* is utilized as a bit-field. Setting bit 2 indicates that the agent is a host, bit 3 indicates that agent can participate in agent dispatches. All the three bits or a combination of them can be set by the HSA runtime.

The structure of the HSA agent/component is defined as follows:

```
typedef struct hsa_agent_s{
    bool                    is_pic_supported;
    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;
} hsa_agent_t;
```

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

hsa_node_id id of the node this agent/component belongs to.

agent_id Unique identifier for an HSA agent.

agent_type an identifier for the type of this agent.

vendor[16] 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[16] 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.

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

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.

With in the agent, the agent type is an enumeration that is defined as follows:

```
typedef enum hsa_agent_type_t{
    HOST=1,
    COMPONENT=2,
    AGENT_DISPATCH=4
}hsa_agent_type_t;
```

HOST=1 indcates that the agent represents the host.

COMPONENT=2 indicates that agent represents an HSA component.

AGENT_DISPATCH=4 indicates that the agent is capable of agent dispatches, and can serve as a target for them.

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 2.1.2) 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 not look different from the users perception.

2.4 Memory based Signals and Synchronization in HSA

In a HSA system, memory is coherent and can serve as a means for message passing, asynchronous communication or synchronization between various elements. A signal is an

alternative, possibly more power-efficient, communication mechanism between two entities in a HSA system. A signal carries a value, which can be updated or conditionally waited upon via an API call or an HSAIL instruction [**?**]. A signal structure is opaque ~~so implementations~~ and is always typedef'ed to `uint64_t`. Implementations can use the most power-efficient send-propagation and wait techniques available to them on the HSA system.

The HSA SAR Specification [**?**] identifies HSA Agent as a participant in a HSA memory based signaling and synchronization. This feature, as stated in the HSA SAR Specification, requires a runtime API for allocation of signals that may be used for synchronization and states that the signal is opaque and may contain implementation specific information.

An API, **`hsa_signal_create`**, to support creation of signals, is defined as follows:

```
hsa_status_t hsa_signal_create(hsa_signal_value_t intial_signal_value,
                               hsa_signal_handle_t *signal_handle,
                               hsa_runtime_context_t *context);
```

initial_signal_value input. Initial value at the signal, the signal is initialized with this value.

signal_handle User allocated, output. The handle of the signal that this API creates. This handle is opaque.

context input. The context in which this signal is being created. Any errors/notifications will be reported via callbacks registered in the same context.

The signal create API returns `HSA_STATUS_SUCCESS` if the signal object has been successfully created. Otherwise, it returns one of the following:

- `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 message queue creation. 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 ~~*signal*~~*signal_handle* is NULL or an invalid pointer of it an invalid/NULL context is passed in as an argument.

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. An API to bind the signal to a particular runtime context is defined as follows:

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

signal_handle input. Opaque handle that was obtained from **`hsa_signal_create`** API.

context input. Additional context to which this signal should be bound to

This API returns HSA_STATUS_SUCCESS if the bind was successful. Otherwise, it returns one of the following errors:

- HSA_STATUS_ERROR_INVALID_ARGUMENT is *signal_handle* is NULL or invalid or if the *context* is NULL or invalid.

The corresponding signal destruction API is defined as follows:

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

signal_handle input. Opaque handle that was obtained from **hsa_signal_create** API.

The signal destroy API returns HSA_STATUS_SUCCESS if the signal object has been successfully destroyed. Otherwise, it returns one of the following:

- HSA_STATUS_ERROR_INVALID_ARGUMENT if *signal_handle* is invalid.

~~There are three participants in signal-based communication: the Kernel executing in the HSA component, the HSA agent, the HSA packet processor~~A signal can also be unbound from a particular context if the user no longer wants to receive notifications about this signal in the callback registered for that context. The API to unbind is defined as follows:

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

signal_handle input. Opaque handle that was obtained from **hsa_signal_create** API.

context input. Unbind the signal from this context.

The API returns HSA_STATUS_SUCCESS if the signal is successfully unbound from the context. Otherwise, it can return one of the following errors:

- HSA_STATUS_ERROR_SIGNAL_NOT_BOUND if the signal was not already bound to that context.
- HSA_STATUS_ERROR_INVALID_ARGUMENT is *signal_handle* is NULL or invalid or if the *context* is NULL or invalid.

As per the HSA SAR specification the signals may only be created and operated on by either instructions in HSAIL or the HSA runtime API. Sending a signal entails updating a particular value at the signal. 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 for a particular value before returning.

The API to query the timeout is defined as:

```
uint64_t hsa_signal_get_timeout();
```

This getter API does not return a status. This API returns the timeout, which indicates the maximum amount of time an implementation can spend in a wait operation on the signal. The return value is in the ~~unit of clock ticks, or, as SAR describes, timestamp~~units of the system-wide clock who's frequency is available via the `hsa_platform_t` structure (see Section 2.3). As per SAR, the HSA system has a system-wide timestamp that operates at a fixed frequency. ~~Both the timestamp and the~~ The frequency can be queried via the `hsa_platform_t` structure defined in Section 2.3. The timeout is incremented at the same frequency. The user can use this information to translate the timeout to a different frequency domain.

~~When multiple threads are attempting to signal without the use of atomics, no ordering guarantee is given by the HSA system.~~

The send signal API sets the signal handle with caller specified value. Any subsequent wait on the signal handle would be given a copy of this new signal value after the wait condition is met (and before the timeout expires). The signal infrastructure allows for multiple waiters on a single signal. A multi-threaded user application can have multiple threads sending and waiting on signals.

In addition to the update of signals using Send, the API for send signal must support other atomic operations as well. HSA defines *AND, OR, XOR, Exchange, Add, Subtract, Increment, Decrement, Maximum, Minimum* and *CAS*. Apart from the no synchronization case, which is referred to as *none* synchronization, there are three types of synchronization defined in the systems architecture requirements:

Acquire synchronization
: No memory operation listed after the acquire can be executed before the acquire-synchronized operation. Acquire synchronization can be applied to various operations including a load operation.

Release synchronization
: No memory operation listed before the release can be executed after the release-synchronized operation. Release synchronization can be applied to various operations including a store operation.

Acquire-Release synchronization
: This acts like a fence. No memory operation listed before the Acquire-Release synchronized operation can be move after it nor can any memory operation listed after the Acquire-Release synchronized operation can be executed before it.

Relaxed synchronization
: No synchronization is applied to the send or wait operation.

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.

Hence, the following represent the complete set of actions (with associated synchronization) that can be performed on a signal value: Send with release, Send with ~~acquire-release~~relaxed, AND

with release, AND with relaxed, OR with release, OR with relaxed, XOR with release, XOR with relaxed, Exchange with acquire-release, Exchange with relaxed, Add with release, Add with relaxed, Subtract with release, Subtract with relaxed, Increment with release, Increment with relaxed, Decrement with release, ~~Maximum with acquire release, Minimum with acquire release,~~ Decrement with relaxed, Maximum with acquire-release, Maximul with relaxed, Minimum with acquire-release, Minimum with relaxed, CAS release.

For efficiency, a unique signal API has been created for each of these actions. In the description of the API, for convenience, *value@signal_handle* is used to represent the value at a signal.

```
hsa_status_t hsa_signal_send_relaxed(hsa_signal_handle_t signal_handle,
                hsa_signal_value_t signal_value);
```

signal_handle input. Opaque handle of the signal object that is to be signaled.

signal_value input. Value of the signal, with relaxed semantics, signal_value is assigned to the value@signal_handle.

```
hsa_status_t hsa_signal_add_release(hsa_signal_handle_t signal_handle,
                hsa_signal_value_t add_value);
```

signal_handle input. Opaque handle of the signal object that is to be signaled.

add_value input. Value to add to the value@signal_handle (the addition is atomic).

```
hsa_status_t hsa_signal_add_relaxed(hsa_signal_handle_t signal_handle,
                hsa_signal_value_t add_value);
```

signal_handle input. Opaque handle of the signal object that is to be signaled.

add_value input. Value to add to the value@signal_handle (the addition is atomic).

```
hsa_status_t hsa_signal_and_release(hsa_signal_handle_t signal_handle,
                hsa_signal_value_t and_value);
```

signal_handle input. Opaque handle of the signal object that is to be signaled.

and_value input. Value to do an And with, value@signal_handle &= and_value.

```
hsa_status_t hsa_signal_and_relaxed(hsa_signal_handle_t signal_handle,
                hsa_signal_value_t and_value);
```

signal_handle input. Opaque handle of the signal object that is to be signaled.

and_value input. Value to do an And with, value@signal_handle &= and_value.

```
hsa_status_t hsa_signal_or_release(hsa_signal_handle_t signal_handle,
                hsa_signal_value_t or_value);
```

signal_handle input. Opaque handle of the signal object that is to be signaled.

or_value input. value@signal_handle OR= or_value.

```
hsa_status_t hsa_signal_or_relaxed(hsa_signal_handle_t signal_handle,
                hsa_signal_value_t or_value);
```

signal_handle input. Opaque handle of the signal object that is to be signaled.

or_value input. value@signal_handle OR= or_value.

```
hsa_status_t hsa_signal_xor_release(hsa_signal_handle_t signal_handle,
                hsa_signal_value_t xor_value);
```

signal_handle input. Opaque handle of the signal object that is to be signaled.

xor_value input. Value to do an XOR with, value@signal_handle XOR= xor_value.

```
hsa_status_t hsa_signal_xor_relaxed(hsa_signal_handle_t signal_handle,
                hsa_signal_value_t xor_value);
```

signal_handle input. Opaque handle of the signal object that is to be signaled.

xor_value input. Value to do an XOR with, value@signal_handle XOR= xor_value.

```
hsa_status_t hsa_signal_exchange_release(hsa_signal_handle_t
   signal_handle,
                hsa_signal_value_t exchange_value,
                hsa_signal_value_t *value_at_signal);
```

signal_handle input. Opaque handle of the signal object that is to be signaled.

exchange_value user allocated, input/output. Exchange value, the value to be placed at signal, value@signal_handle , after being stored in value_at_signal, is overwritten with exchange_value.

value_at_signal user allocated, output. value_at_signal = value@signal_handle; value@signal_handle = exchange_value.

```
hsa_status_t hsa_signal_exchange_relaxed(hsa_signal_handle_t
   signal_handle,
                hsa_signal_value_t exchange_value,
                hsa_signal_value_t *value_at_signal);
```

signal_handle input. Opaque handle of the signal object that is to be signaled.

exchange_value user allocated, input/output. Exchange value, the value to be placed at signal, value@signal_handle , after being stored in value_at_signal, is overwritten with exchange_value.

value_at_signal user allocated, output. value_at_signal = value@signal_handle; value@signal_handle = exchange_value.

```
hsa_status_t hsa_signal_decrement_release(hsa_signal_handle_t
   signal_handle,
                hsa_signal_value_t decrement_value);
```

signal_handle input. Opaque handle of the signal object that is to be signaled.

decrement_value input. Value the signal is to be decremented with, value@signal_handle -= decrement_value.

```
hsa_status_t hsa_signal_decrement_relaxed(hsa_signal_handle_t
   signal_handle,
               hsa_signal_value_t decrement_value);
```

signal_handle input. Opaque handle of the signal object that is to be signaled.

decrement_value input. Value the signal is to be decremented with, value@signal_handle -= decrement_value.

```
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 *value_at_signal);
```

signal_handle input. Opaque handle of the signal object that is to be signaled.

value_compare input. The value to compare value@signal_handle against (operator is equal to).

value_replace input. the value to replace the value@signal_handle with if signal.value was equal to value_compare.

value_at_signal user allocated, output. The value at the signal, prior to the atomic replace, if the comparision was successful.

```
hsa_status_t hsa_signal_max(hsa_signal_handle_t signal_handle,
               hsa_signal_value_t compare_value,
               hsa_signal_value_t *max_value);
```

signal_handle input. Opaque handle of the signal object that is to be signaled.

compare_value input. Compared with value@signal_handle to determine maximum.

max_value user allocated, output. MAX(compare_value, value@signal_handle). The signal value is updated with the max value.

```
hsa_status_t hsa_signal_min(hsa_signal_handle_t signal_handle,
                hsa_signal_value_t compare_value,
                hsa_signal_value_t *min_value);
```

signal_handle input. Opaque handle of the signal object that is to be signaled.

compare_value input. value@signal_handle is compared with this and the minimum of the two returned.

min_value user allocated, output. min(compare_value, value@signal_handle). The signal value is updated with the min value.

```
hsa_status_t hsa_signal_send_release(hsa_signal_handle_t signal_handle,
                hsa_signal_value_t signal_value);
```

signal_handle input. Opaque handle of the signal object that is to be signaled

signal_value input. Value of the signal

```
hsa_status_t hsa_signal_subtract_release(hsa_signal_handle_t
   signal_handle,
                hsa_signal_value_t subtract_value);
```

signal_handle input. Opaque handle of the signal object that is to be signaled

subtract_value input. Value to be subtracted from the value@signal_handle.

```
hsa_status_t hsa_signal_subtract_relaxed(hsa_signal_handle_t
   signal_handle,
               hsa_signal_value_t subtract_value);
```

signal_handle input. Opaque handle of the signal object that is to be signaled

subtract_value input. Value to be subtracted from the value@signal_handle.

```
hsa_status_t hsa_signal_increment_release(hsa_signal_handle_t
   signal_handle,
               hsa_signal_value_t increment_value);
```

signal_handle input. Opaque handle of the signal object that is to be signaled

increment_value input. Value to do the increment with

```
hsa_status_t hsa_signal_increment_relaxed(hsa_signal_handle_t
   signal_handle,
               hsa_signal_value_t increment_value);
```

signal_handle input. Opaque handle of the signal object that is to be signaled

increment_value input. Value to do the increment with

All of the **signal_send** API return HSA_STATUS_SUCCESS if the send is successful, any atomic operation that needed to be performed has been done successfully and any result value that needs to be returned has been copied into the user-given location. One of the following error values may be returned in case the send is not successful:

- ▷ HSA_STATUS_ERROR_INVALID_ARGUMENT if (a) the user is expecting an output but the pointer to the output signal value is invalid, (b) the *signal_value* doesn't represent a valid 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 defined here. Waiting on a signal returns the current value at the signal. The wait may return before the condition

is satisfied 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.

Wait *reads* the value, hence Acquire and Acquire-Release synchronizations may be applied to the read. The synchronization should only assumed to have been applied if the status returned by the wait API indicates a success (i.e. return type is `HSA_STATUS_SUCCESS`). The two wait APIs to support both the synchronizations are defined as follows:

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

signal_handle input. Opaque handle of the signal whose value is to be retrieved.

cond input. apply this condition to compare the wait_value with value@signal_handle and return the value@signal_handle only when the condition is met.

compare_value input. value to compare with.

return_value user allocated, output. Pointer to where the current value@signal_handle must be read into.

```
hsa_status_t hsa_signal_wait_acquire_release(hsa_signal_handle_t
   signal_handle,
                          hsa_signal_condition_t cond,
                          hsa_signal_value_t compare_value,
                          hsa_signal_value_t *return_value);
```

signal_handle input. Opaque handle of the signal whose value is to be retrieved.

cond input. apply this condition to compare the compare_value with value@signal_handle and return the value@signal_handle only when the condition is met.

compare_value input. value to compare with.

return_value user allocated, output. Pointer to where the current value@signal_handle must be read into.

The user must always check the return value of the wait before considering the *wait_value* as the wait may have returned due to a timeout. The wait API can return the following status:

- ▷ If an error is signaled on the signal the user is waiting on, the wait API returns `HSA_STATUS_ERROR` to indicate that an error has occurred. The API still returns the current value at the signal. The user may also inspect the value returned. when an error occurred (see Section 2.4.1).
- ▷ `HSA_STATUS_ERROR_INVALID_ARGUMENT` if (a) the user is expecting an output but the pointer to the output signal value is invalid, (b) the *signal_value* doesn't represent a valid signal.
- ▷ `HSA_STATUS_INFO_SIGNAL_TIMEOUT` the signal wait has timedout.

The `hsa_wait_condition_t` is defined as follows:

```
typedef enum hsa_signal_condition_t{
    HSA_EQUALS,
    HSA_NOTEQUALS,
    HSA_GREATER,
    HSA_GREATER_EQUALS,
    HSA_LESSER,
    HSA_LESSER_EQUALS
}hsa_signal_condition_t;
```

HSA_EQUALS the return from the wait API will be either when signal.value == wait_value or the max timeout has been reached.

HSA_NOTEQUALS the return from the wait API will be either when signal.value != wait_value or the max timeout has been reached.

HSA_GREATER the return from the wait API will be either when signal.value ¿ wait_value or the max timeout has been reached.

HSA_GREATER_EQUALS the return from the wait API will be either when signal.value ¿= wait_value or the max timeout has been reached.

HSA_LESSER the return from the wait API will be either when signal.value ¡ wait_value or the max timeout has been reached.

HSA_LESSER_EQUALS the return from the wait API will be either when signal.value ¡= wait_value or the max timeout has been reached.

The runtime also defines an API to query the current signal value. 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. Queried value may be used to check progress of a kernel, if the kernel were updating the signal at various stages of its execution. Query is a non-blocking API and does not take `hsa_wait_condition_t` as input. It merely obtains the current value at the signal.

The `hsa_signal_query_acquire` API is defined as follows:

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

signal_handle input. Opaque handle of the signal whose value is to be retrieved.

value user allocated, output. Pointer to where the current value@signal_handle must be read into.

The `hsa_signal_query_acquire` API returns `HSA_STATUS_SUCCESS` when the value at the signal has been successfully returned. Otherwise, it returns one of the following errors:

- `HSA_STATUS_ERROR_INVALID_ARGUMENT` if *signal_handle* is invalid.

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 kernel.

Signals cannot be used for Inter-Process Communication (IPC).

2.4.1 Indicating Errors with Signals

~~When an error is detected during the execution of a kernel, the completion signal (if present) will be signaled with an error indication value. Specifically~~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 to see if an error has occurred by looking at the return code of the `hsa_signal_wait<acquire_release/Acquire>` API. 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.

2.4.2 Usage Example

2.5 Architected Queue in HSA

HSA hardware supports kernel dispatch through user mode queues. A queue in HSA is associated with a specific HSA component. Conceptually, user mode queues are ring buffers that expose separate memory locations defining the current read and write state of the queue. In a HSA system, agents write AQL packets to the queue to enqueue work on to the HSA components. The queue memory is processed by HSA packet processor(s) 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.

Details on queue mechanics are discussed in the SAR [**?**] document.

A queue in HSA is defined with the following structure:

```
typedef struct hsa_queue_s{
  uint32_t queue_type;
  uint32_t queue_features;
  uint64_t base_address;
  hsa_signal_handle_t doorbell_signal;
  uint32_t size;
  uint32_t queue_id;
  uint32_t queue_active_group_count_global;
  const hsa_group_execution_info_t *mailbox_ptr;
  hsa_signal_handle_t mailbox_signal;
} hsa_queue_t;
```

queue_type used for dynamic queue protocol determination. Currently, 0, the default queue type, is the only type supported.

queue_features bitfield to indicate specific features supported by queue. On a queue creation, if user observes that some unknown bits are set, then the user should ignore them.

base_address A 64-bit pointer to the base of the virtual memory which holds the AQL packets for the queue. At the time of queue creation, the address passed in by the user as queue memory is copied here. This address must be 64-byte aligned.

doorbell_signal After writing a packet to the queue, user must signal this signal object with the most recent write_offset. The packet may already have been processed by the packet processor by the time this doorbell is signaled, however, it may not be processed at all if the doorbellSignal is not signaled.

size A 32-bit unsigned integer which specifies the maximum size of the queue in the number of packets. The size of the queue is always aligned with a power of two number of AQL packets.

queue_id A 32-bit ID for a queue which is unique-per-process.

queue_active_group_count_global maximum number of concurrent workgroups that can run out of this queue

hsa_group_execution_info_t A pointer to the mailbox

mailbox_signal the signal that user can wait on to get an indication that mailbox needs processing

`base_address` is the starting address of the buffer where the packets will be written. `size` is simply the size of the queue in bytes. The `queue_id` member is the unique (per-process) identifier for a queue and helps identify a queue when more than one queue is present in the system.

Internally, the queue structure contains read index and write index. These are not exposed to the

user directly. The user can access them via `hsa_queue_get/cas/add_write_index` and `hsa_queue_get_read_index` API.

The API is defined as follows:

```
uint64_t hsa_queue_get_read_index(hsa_queue_t *queue);
```

queue input. The HSA queue structure.

```
uint64_t hsa_queue_get_write_index(hsa_queue_t *queue);
```

queue input. The HSA queue structure.

```
uint64_t hsa_queue_set_write_index(hsa_queue_t *q,
   uint64_t val);
```

queue input. The HSA queue structure.

val input. The new value of the write index.

```
uint64_t hsa_queue_cas_write_index(hsa_queue_t *q,
   uint64_t old_val,
   uint64_t new_val);
```

queue input. The HSA queue structure.

old_val input. The value to compare with

new_val input. If a match is determined, the write index is updated with new_val

```
uint64_t hsa_queue_add_write_index(hsa_queue_t *q,
   uint64_t val);
```

queue input. The HSA queue structure

val input. The value to add to the write index

These API are all setter/getter APIs and hence do not return `hsa_status_t`. If the queue structure passed to the API is invalid, the behavior of the API is undefined. All the API return the value of the corresponding index. The CAS, ADD and WRITE API on the write index return the value of the write index prior to the update.

The write index is a unique identifier for AQL packets in the queue. The read index indicates the next AQL packet that will be consumed by the HSA packet processor. The write index memory is updated by the agents via. the runtime defined API, while the read index memory location is updated by the HSA Component and can be read by the agent a runtime specified API call or the kernel via HSAIL operation. The read index is automatically advanced when a packet is read by the HSA packet processor. When the agent observes read index matches write index, at that instance, 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, an asynchronous error is generated by the packet processor and queue is put in error state.

The *mailbox_ptr* and *mailbox_signal* are for getting execution information when either a **`debugtrap_u32`** instruction or **`syscall`** 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 2.12. The *queue_active_group_count* is the count of maximum number of ork-groups that can be executed in parallel for dispatches executed on this queue.

The `doorbell_signal` is a signal from the agent writing the AQL packet to the HSA packet processor indicating that it has work to do. The value which the `doorbell_signal` must be signaled with shall be be the latest write index at which an AQL packet has been written into. The purpose of this signal is to *inform* the HSA packet processor that it has packets that need to be processed. However, packets may be processed by the HSA packet processor even before the `doorbell_signal` has been signaled by the agent writing the AQL packet. This is because when write index is advanced by the agent there are two scenarios that could arise:

- the HSA packet processor is in some low-powered state awaiting work and requires the `doorbell_signal` signal to *wake* it to continue reading packets.
- the HSA packet processor is already actively processing a packet and observers the write index be updated by the agent and continues to process the new packets written – even before the agent has signaled the `doorbell_signal`.

Hence, despite the fact that the AQL packet for which the agent is signaling the doorbell may already have been processed, the agent must ring the doorbell for every batch of AQL packets written.

The hsa_queue structure is the output of hsa_queue_create function, which is defined as follows:

```
hsa_status_t hsa_queue_create(const hsa_agent_t *component,
                              size_t size,
                              uint32_t queue_type,
                              hsa_runtime_context_t *context,
                              hsa_queue_t **queue);
```

component input. The component on which this queue is to be created.

size input. Size of the queue memory in number of packets in is expected to hold.

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

context input. The context in which this queue is being created. Any errors/notifications will be reported via callbacks registered in the same context.

queue runtime allocated, output. The queue structure, filled up and returned by the runtime.

The hsa_queue_create API allocates the memory for the queue. Space for size number of packets is allocated by the implementation. The size is required to be aligned with a power of two number of AQL packets. The pointer to the beginning of the memory allocated can be obtained from the queue structure in the field base_address. No memory shall be allocated by an implementation if the queue creation fails. An implementation may or many not initialize the hsa_queue structure if queue creation fails. Hence the user should rely on the error code to determine if the hsa_queue structure is valid. The hsa_queue_create API returns HSA_STATUS_SUCCESS when the queue is successfully created. Otherwise, it can return one of the following status messages:

- HSA_STATUS_ERROR_INVALID_ARGUMENT error code is returned when 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 *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.
- status > HSA_STATUS_OTHER_BEGIN Any implementation specific error has a error value >HSA_STATUS_OTHER_BEGIN (see 2.1 for details).

The first ratified version of the SAR specification does not define the queue_type and queue_feature – they have been marked as fields for future expansion.

The API to destroy a queue is defined as follows:

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

queue input. The queue structure that points to the queue that needs to be destroyed, after destruction base_address or the rest of the queue structure, even if cached by the user, are no longer valid.

After queue destruction, it is considered undefined to access the memory pointed to be the `base_address`. An implementation may choose to do internal memory management and reuse this memory for another queue.

~~Once a queue is created, it is monitored and scheduled to be processed by the HSA packet processor(s). The `hsa` structure returned after the queue is created is to be considered read-only and should not be modified by the user. If the user wishes to change something in the queue, the runtime provides an API, `hsa` to accomplish it. However, at this time, only the `queue` and `size` may be altered once a queue is created. The implementation ensures a safe copy of pending packets into the resized queue. The `hsa` is defined as follows:~~

2.5.1 Deactivating a Queue

The queue can forcefully be ~~deactivated~~ inactivated by the user. This will kill any pending executions and prevent any new packets to be processed. Any more packets written to the queue once it is ~~deactivated~~ inactivated will be ignored by the packet processor.

The API for deactivating the queue is defined as follows:

```
hsa_status_t hsa_queue_force_inactivate(hsa_queue_t *queue);
```

queue input. The queue that needs to be inactivated.

2.5.2 Queue Error Reporting, Deactivation and Queue State

The HSA queue structure includes an error message queue, *message_queue_handle*, that the user must initialize and pass as an argument at queue creation. The error message queue may either be created by the user using `hsa_error_message_queue_create` API. The user may also use the default error message queue generated by the `hsa_initialize` API.

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

- 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*.

Active Once a queue is successfully created using the `hsa_queue_create` API, it enters an active state, packets can be put on the queue and when the write-index is updated and the doorbell is updated, the packet processor processes the packets. The actual initiation of dispatch may depend on the resources available for the dispatch. Only in the *active* state, writing packets to the queue, updating the write index or ringing the doorbell has any effect. The queue is no longer being monitored by a queue packet processor for new packets 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 right after the queue is marked inactive. The user may choose to

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_force_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. Invocation of `hsa_queue_force_inactivate` API when the queue 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.

A state diagram showing the various states and transitions is shown in Figure 2.2.

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 ~~deactivate~~ inactivate the queue the first error it encounters.

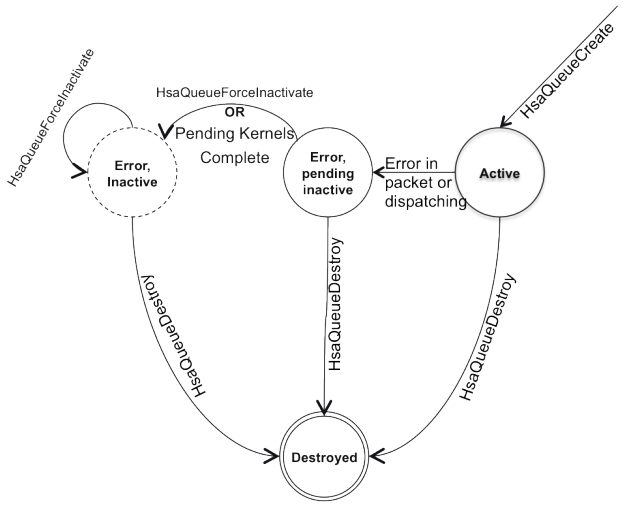

Figure 2.2: Once the queue is created and is active, any error in packet processing takes the queue into pending inactive state where the queue is performing tasks to get to inactive state. Failure during the attempt to ~~deactivate~~ inactivate results in queue reaching an error state. A queue that is in active, error or inactive state may be destroyed by the using using the `hsa_queue_destroy` API Provided by the HSA runtime.

2.5.3 Multi-Threaded Queue Access

HSA Core API does not provide explicit API for synchronized access to the queues – the architected queue data structure and read/write index update API are sufficient to allow users to implement thread-safe packet insertion into the queue. Users can use several techniques to support multiple concurrent writers writing AQL packets to the queue. The following example code illustrates one such technique – several other techniques that allow concurrent writes to the queue can be utilized in a similar way.

The sample code below demonstrates a simple reader and writer logic to do a multi-threaded queue access using the queue structure above.

```
// Read the current queue write offset via intrinsic
tmp_write_index = hsa_queue_get_write_index(q);

// wait until the queue is no longer full.
while(tmp_write_index == read_index + size) {}

// Atomically bump the WriteOffset via intrinsic
if (hsa_queue_cas_write_index(q, tmp_write_index , tmp_write_index +
    1) ==
tmp_write_index)
{
  // calculate index
  uint32_t index = tmp_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 Core Runtime Support for AQL

AQL is a command-interface for describing a dispatch or a dependency in a standard format for the queue packet processor. To match with and support the AQL packet definitions in the HSA SAR, HSA core base runtime includes structures for different types of AQL packets. SAR defines three different kinds of AQL packets: invalid, dispatch and barrier. There is a common packet header across these three packet types and is defined by the following structure:

```
typedef struct hsa_aql_packet_header_s {
  uint32_t format:8;
  uint32_t barrier:1;
  uint32_t acquire_fence_scope:2;
  uint32_t release_fence_scope:2;
  uint32_t invalidate_instruction_cache:1;
  uint32_t invalidate_ro_image_cache:1;
  uint32_t dimensions:2;
  uint32_t reserved:15;
} hsa_aql_packet_header_t;
```

format:8 8 bits for describing the packet type, 0 for INVALID, 1 for DISPATCH, 2 BARRIER. All other values are reserverd.

barrier:1 If set then processing of packet will only begin when all preceding packets are complete.

acquire_fence_scope:2 defines the scope and type of the memory fence operation applied before the packet enters the active state.

release_fence_scope:2 defines the scope and type of the memory fence operation applied after kernel completion but before the packet is marked completed.

invalidate_instruction_cache:1 setting this bit indicates that fence additionally applies to any instruction cache(s).

invalidate_ro_image_cache:1 setting this bit indicates that fence additionally applies to any read-only image cache(s).

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

reserved:15 reserved

Acquire Fence Scope	Description
0	None – no fence is applied.
1	Make memory operations made by this HSA component prior to launch of this packet, visible to this packet operation.
2	Make memory operations made by HSA agents prior to launch of this packet, visible to this packet operation.
3	Make memory operations made by HSA agents prior to launch of this packet, visible to this packet operation, and affected caches in the HSA Component are invalidated.

Table 2.1: Acquire Fence Scope Values and Actions

Release Fence Scope	Description
0	None – no fence is applied.
1	The release fence is applied to the HSA Component only.
2	The release fence is applied globally to the HSA System.
3	The release fence is applied globally to the HSA System, and additionally affected caches in the HSA Component are cleaned and invalidated.

Table 2.2: Release Fence Scope Values and Actions

The `acquire_fence_scope` is used to control the ordering of memory operations before the packet enters the active state. There are 4 possible values for acquire fence scope. Each of the values defines a particular action by HSA agents and components. The details are described in Table 2.1.

Similarly, the release fence scope, which is also 2 bits, can be used to define the desired memory fence and cache actions at the end of kernel execution, but prior to the packet being marked as complete. Table 2.2 describes the different controls.

The `format` field in the header is used to specify the packet type. Beyond the three packet types defined, all the other packet types are reserved for implementation use. In addition to this, the last 15 bits in the packet header are also reserved for future or implementation specific use. The format field indicates the type of the packet. Of the three packet types, the dispatch and the barrier packet have individual packet-state diagrams that are discussed along with their description.

Invalid AQL packet Indicates that the packet not ready to be processed by the packet processor.

2.6.1 Dispatch AQL Packet

Dispatch packet type is used for dispatching a kernel on to a HSA component. The dispatch AQL packet can have five different states: *on queue*, *processing*, *error*, *active* or *complete*. Figure 2.3 shows the different states of a packet and transitions leading to those states.

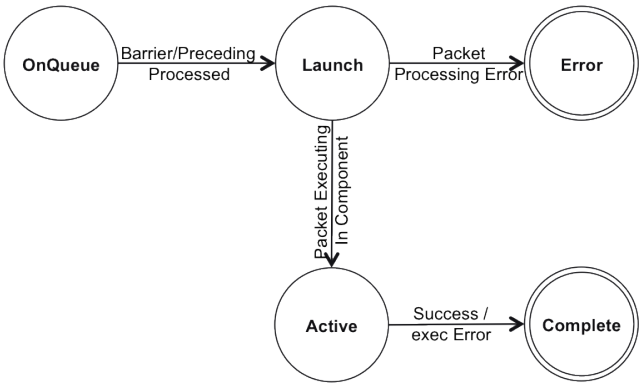

Figure 2.3: Dispatch Packet State Diagram

On queue state A packet is considered to 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. Any other value for format puts the packet and the queue in error state.

Processing state 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 processing state. From the launch state, two states are possible: error or active.

Error state the packet processor encountered an error processing this packet. This results in a queue error (see Figure 2.2) 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_force_deactivate`** API or the **`hsa_queue_destroy`** API while the packet is this state, the completion object will be signaled with error.

Active state 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, a user action (see 2.5.1) can also take the packet out of active state into complete state. When the user invokes the `hsa_queue_force_deactivate` API or the `hsa_queue_destroy` API while the packet is this state, the completion object will be signaled with error.

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

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.

The structure for the dispatch AQL packet is shown below:

```
typedef struct hsa_aql_dispatch_packet_s{
  hsa_aql_packet_header_t header;
  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;
```

header packet header packet header structure

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

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 Total size in bytes of private memory allocation request (per work-item).

group_segment_size_bytes Total 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 used to indicate completion of the job.

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 AQL 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 described in Section 2.8.

Of the other HSA segments, the kernarg segment is also a part of the AQL packet, but as a pointer. This is because kernarg segment carries the arguments required to execute the kernel being dispatched and must be setup by the user (as specified in Section 2.7) prior to writing the AQL packet to the queue (unlike the group and private segments, whose lifespan spans only the active state of the AQL dispatch packet).

2.6.2 Barrier AQL packet

The barrier packet allows the user to specify up to 5 dependencies as `hsa_signal` objects and require 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. The HSA core base runtime structure for the AQL barrier packet is shown below:

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

header packet header packet header structure

reserved2 reserved

dep_signal0 The first dependency signal, a negative value means dependency not met and the completion signal for this packet will be set to

dep_signal1 The first dependency signal, a negative value means dependency not met and the completion signal for this packet will be set to

dep_signal2 The first dependency signal, a negative value means dependency not met and the completion signal for this packet will be set to

dep_signal3 The first dependency signal, a negative value means dependency not met and the completion signal for this packet will be set to

dep_signal4 The first dependency signal, a negative value means dependency not met and the completion signal for this packet will be set to

reserved3 reserved

completion_signal HSA signaling object used to indicate completion of the dependency resolution, success of failure

If any of 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 2.4.1.

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 HSA 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.

Alike the dispatch packet, the barrier packet can also be in one of the following states: *on queue*, *processing*, *completed, error* or *completed, success*.

On queue state A packet is considered to 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. Any other value for format puts the packet and the queue in error state.

Processing state 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 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-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_force_deactivate` API or the `hsa_queue_destroy` API while the packet is in processing state (the completion object will be appropriately signaled with an error).

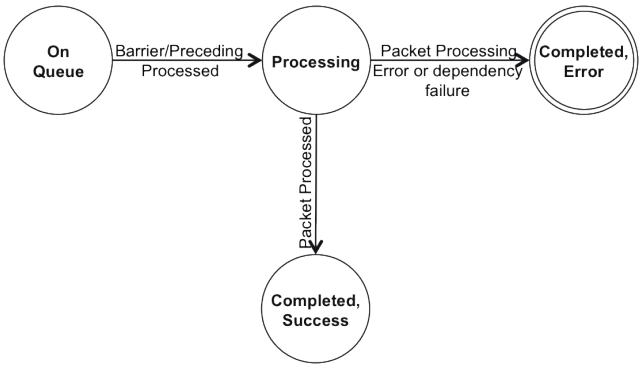

Figure 2.4: Barrier Packet State Diagram

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

A state diagram in Figure 2.4 shows these transitions.

~~A malformed packet on the queue can put the queue in error state. The runtime provides API to check an AQL packet for errors. The error checking for a dispatch packet validates the `format`, `dimensions`, `reserved`, `kernel`, `kernarg` and the completion signal. The validation for a barrier packet includes the format, the reserved field, and all the signals.~~

~~Both of the validate API return `HSA` when the packet is valid. Otherwise, one of the following errors is returned:~~

~~`HSA` if a NULL value is passed for the argument.~~

~~`HSA` based on which structure element has an invalid value.~~

2.6.3 AQL Setup Example

2.7 HSAIL Application Binary Interface

The HSAIL ABI specifies the in-memory format for the kernarg segments. The ABI architects a simple set of rules for the size and alignment of the kernel arguments and ensures that all HSA vendors use a common argument format. Users and tools can determine the offset of all kernel arguments merely by examining or knowing the kernel signature (without any runtime feedback or metadata from the finalizer). For more information on the HSAIL ABI, see the HSA Programmer's Reference Manual, Sections 4.22 (Kernarg Segment) and 4.19 (Declaring and Defining Identifiers).

32-bit vs 64-bit HSAIL does not contain a pointer type with an implementation-defined size. Instead, HSAIL is specifically compiled for either small-mode (32-bit) or large-mode (64-bit). Pointers in the small mode use 32-bit types; pointers in the large mode use 64-bit types. Because the pointer size is explicit, the ABI does not need to address the issue of pointer sizing. Let us consider a simple kernel signature and the associated code required to set up the kernarg segment. The first example shows the setup for a simple vector copy kernel.

```
kernel &__vector_copy_kernel(kernarg_u32 %arg_p0,
    kernarg_u32 %arg_p1,
    kernarg_u32 %arg_p2)
{
  workitemaid $s0, 0;
  shl_u32 $s0, $s0, 2;
  ld_kernarg_u32 $s1, [%arg_p1];
  add_u32 $s1, $s1, $s0;
  ld_kernarg_u32 $s2, [%arg_p0];
  add_u32 $s0, $s2, $s0;
  ld_global_u32 $s0, [$s0];
  st_global_u32 $s0, [$s1];
  ret;
};

//Create an AQL packet.
hsa_aql_dispatch_packet_t aql_packet;
uint64_t* kernel_arguments =
reinterpret_cast<uint64_t *>(malloc(argument_size));

memset(kernel_arguments, 0, argument_size);

//The ABI dictates that all arguments must be aligned to a 64bit
   boundary,
//hence the typecast.
kernel_arguments[0] = (uint64_t)a;
kernel_arguments[1] = (uint64_t)b;
kernel_arguments[2] = (uint64_t)size;

//Set the address for kernel argument within the AQL packet.
aql_packet.kernarg_address = (uint64_t)kernel_arguments;
```

2.8 Support for Finalization And Code Object Generation Kernel

Compilation support in the HSA core runtime comprises of two parts. The first is the finalization step, the primary functionality of this step is to translate the HSAIL to component specific

instruction set and produce a relocatable code object. The second part is support for binding, it resolves user defined symbols that are not already bound.

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 required despite a HSA components' native support of HSAIL. This is because of two primary reasons:

1. SAR [**?**] specifies that the HSA code object (this takes the form of a structure, `hsa_code_object_t` in the HSA core runtime) is 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 enqueues from the HSA component, kernel code object must reside in a memory layout specification since dispatches initiated from components are able use memory operations to get the information necessary for a dispatch – HSA core runtime does not define or handle a file format container.

A code object in memory 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 kernel code objects (see Section 2.3).

The core runtime accepts HSAIL programs coded in the BRIG binary format, as defined in the HSA PRM, for its finalization process. BRIG is a binary format defined by the HSA PRM and includes 5 different section, *.string*, *.directive*, *.code*, *.operand*, *.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`.

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 represent the kernel to be executed in the HSA component and binding of unresolved symbols associated with the code object generated. In addition to the generation of the code object, debug information (`hsa_kernel_debug_info_t` and symbol information`hsa_kernel_symbol_map_t` are also generated by the finalization process. A symbol can be a variable, a function, image or a sampler. When finalization occurs, a `hsa_code_object_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 have been resolved at the time of its finalization. For example, the kernel may be invoking a function call (whose symbol is not present in the current compilation step) or using memory from a segment such as a group segment, the size of which is dispatch-dimension dependent (see Section 2.6) and hence can only be resolved at dispatch time. Similarly, debug information that associates a kernel object with BRIG is also required to map symbols and instructions in the kernel object back to the appropriate BRIG sections/offsets.

Hence, the **`hsa_finalize_brig`** API accepts `hsa_brig_t` as an input and produces relocatable code object in which global and group symbols are not bound to actual, component recognizable, addresses. A pointer to an output argument of type `hsa_symbol_map_t` that can provide such mapping, and, pointer to an output argument of type `hsa_debug_info_t` that provides a mapping

from the code object back to the BRIG. The `hsa_finalize_brig` API also takes in a argument of type `hsa_symbol_map_t` as an input in order to skip an explicit `hsa_bind_kernel_code_object` step when symbol information is potentially already available. The API for **`hsa_finalize_brig`** is described.

```
typedef struct hsa_brig_s{
    void *string_section;
    void *directive_section;
    void *code_section;
    void *operand_section;
    void *debug_section;
    uint32_t string_section_size_bytes;
    uint32_t directive_section_size_bytes;
    uint32_t code_section_size_bytes;
    uint32_t operand_section_size_bytes;
    uint32_t debug_section_size_bytes;
} hsa_brig_t;
```

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.

debug_section From PRM: the debug information generated by the high-level compiler. The finalizer does not modify the .debug section.

string_section_size_bytes size of string section in bytes

directive_section_size_bytes size of directive section in bytes

code_section_size_bytes size of code section in bytes

operand_section_size_bytes size of the operand section in bytes

debug_section_size_bytes tbd

```
typedef struct hsa_symbol_s{
    uint64_t current_binding;
    uint32_t directive_section_offset;
} hsa_symbol_t;
```

current_binding current binding of that symbol

directive_section_offset directive section offset in BRIG

```
typedef struct hsa_symbol_map_s{
    uint32_t number_of_symbols;
    hsa_symbol_t *symbol;
    uint64_t reserved;
} hsa_symbol_map_t;
```

number_of_symbols the count of the total number of symbols in this map

symbol the array of symbols

reserved for implementations to store other information and allow for queries

```
hsa_status_t hsa_finalize_brig(const hsa_agent_t *component,
    hsa_brig_t *brig,
    uint32_t kernel_directive,
    const char *options,
    hsa_symbol_map_t *input_symbols,
    hsa_code_object_t **kernel,
    hsa_debug_info_t **debug_info,
    hsa_symbol_map_t **symbol_map);
```

component input. A valid pointer to the hsa_agent_t.

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

kernel_directive input. A valid kernel directive

options input. Compilation options (a pointer), in ISO/IEC 646 encoded english language string.

input_symbols input. If it is know to how some of the symbols can be resolved.

kernel runtime allocated, output. A pointer to the location to which the output AQL kernel object is to be copied.

symbol_map runtime allocated, output. Hsail symbol map infomation.

debug_info runtime allocated, output. A pointer to the debug information structure.

The **hsa_finalize_brig** API returns HSA_STATUS_SUCCESS when the finalization is successful. Otherwise, it returns one of the following errors:

- 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.
- status > HSA_STATUS_OTHER_BEGIN Any implementation specific error has a error value >HSA_STATUS_OTHER_BEGIN (see 2.1 for details).

The hsa_code_object_t is defined as follows:

```
typedef struct hsa_control_directives_s {
    uint64_t reserved[16];
}hsa_control_directives_t;
```

reserved[16] directives are yet to be defined

```
typedef struct hsa_code_object_s{
    uint64_t size_bytes;
    char vendor[16];
    char name[16];
    uint32_t hsail_version_major;
    uint32_t hsail_version_minor;
    bool hsail_profile_base;
    bool hsail_machine_model_small;
    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;
    uint8_t wavefront_size;
    uint8_t reserved1[6];
    hsa_control_directives_t control_directive;
} hsa_code_object_t;
```

size_bytes Size of this object in bytes.

vendor[16] The vendor of the device 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. This value matches that in the component structure.

name[16] The vendor's name of the device 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. This value matches that in the component structure.

hsail_version_major The HSAIL major version. This information is from the HSAIL version directive.

hsail_version_minor The HSAIL minor version. This information is from the HSAIL version directive.

hsail_profile_base The HSAIL profile defines which features are used. This information is from the HSAIL version directive.

hsail_machine_model_small The HSAIL machine model gives the address sizes used by the code. This information is from the HSAIL version directive.

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.

wavefront_size Wavefront size. Must be a power of 2 in range 1..64 inclusive.

reserved1[6] Reserved. Must be 0.

control_directive the actual control directive used in generating this code object.

The user must ensure that the BRIG is valid, failing which the finalize API will return an error. The desired kernel to finalize is specified as a location in the BRIG. The location is described as an offset into the BRIG .directive section.

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 core runtime also provides a corresponding destruction API that destroys the code object. In addition to that, 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 of the **hsa_code_object_destroy** API is as follows:

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

code_object input. a pointer to the kernel code object that needs to be destroyed.

Note that destroying does not impact any memory segments that may have been allocated for this object. It merely release resources used to build the object.

The **hsa_code_object_destroy** API destroys the code object. It returns HSA_STATUS_SUCCESS if the destruction was successful. Otherwise, it can return one of the following errors:

- 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 *code_object* or is NULL or does not point to a valid code object.

```
hsa_status_t hsa_code_object_serialize(hsa_code_object_t *code_object,
   uint64_t *size,
   void *serialized_object);
```

code_object input. The code object to serialize.

size output. Size of the serialized object that is generated.

serialized_object Output. Pointer to the serialized object.

```
hsa_status_t hsa_code_object_deserialize( void *serialized_object,
   uint64_t size,
   hsa_code_object_t **code_object);
```

serialized_object input. Pointer to the serialized object.

size input. Size of the serialized object that is generated.

code_object runtime allocated, output. The code object generated as a part of serialization.

HSA_STATUS_SUCCESS is returned if serialize/deserialize API finish successfully. Success of serialize API means the *code_object* has been successfully serialized and the copied into the

location in memory (*serialized_object*). A successful deserialization recreates the code object, allocates memory for it and returns it. One of the following errors may be returned:

- HSA_STATUS_ERROR_INVALID_ARGUMENT if *code_object* or is NULL or does not point to a valid code object in the serialize API. For the deserialize API, this means the *serialized_object* is either null or is not valid or the size is 0.

- HSA_STATUS_ERROR_OUT_OF_RESOURCES If the serialize API cannot allocate memory for *serialized_object* or the deserialize cannot allocate memory for *code_object*.

As discussed above, the generated kernel object may not have all of its symbols bound to addresses. In order to support late binding of symbols, the HSA core runtime supports a bind API defined as follows:

```
hsa_status_t hsa_symbol_bind_code_object( hsa_code_object_t *kernel,
    hsa_symbol_map_t *input_symbol_map,
    hsa_code_object_t *output_kernel);
```

kernel input. A pointer to the kernel code object.

symbol_map input. A pointer to the symbol information structure.

code_object runtime allocated, output. A pointer to the kernel code object.

The **hsa_symbol_bind_code_object** API returns HSA_STATUS_SUCCESS when one or more symbols were successfully bound based on the *input_symbol_map*. Otherwise, it can return one of the following status values:

- HSA_STATUS_ERROR_INVALID_ARGUMENT if *code_object* or is NULL or does not point to a valid code object or the *input_symbol_map* is invalid or NULL.

  HSA_STATUS_INFO_NO_SYMBOLS_BOUNDED if no unbounded symbols in *kernel* were bound using the *input_symbol_map*.

  - HSA_STATUS_ERROR_OUT_OF_RESOURCES if it cannot allocate memory for *output_kernel*.

A Bind method that takes a code object and a map, and produces a new code object with any symbols not already bound that are used and defined in the symbol map generated when this code object was produced. Binding is also a step where relocation occurs. If an address is specified in the input symbol map for the kernel or function it is relocated to that address.

The bind API takes in the code object as input, attempts to resolve unresolved symbols and produces a new kernel object as output. Since binding is is mostly specific to compilation instance and the HSA component itself, creation of a new kernel object is necessary. The bind API does not destroy the input kernel. A runtime provided API, hsa_code_object_destroy, can be used by the user to destroy a code object. The bind API may partially resolve a subset of the symbols and

can be invoked multiple times and each time produce a relocatable kernel object. The kernel object is required to track which symbols have already been resolved. Bind function does not generate an error when the symbol map passed to it as input already has symbols resolved for that kernel object. By default, it does not attempt overwrite the previous resolution and does not generate an error. The bind API instead produces an output symbol map of previously unresolved symbols that have been resolved in the current bind attempt and the addresses they have been resolved with.

2.9 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 AQL dispatch packet's `hsa_aql_dispatch_packet_t`.*group_segment_size_bytes* field.
- 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 `hsa_code_object_t`.*workgroup_group_segment_size_byte*
- After setting the offset for each group memory argument, the user must set the QL 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
        ...
```

2.10 Memory Registration and Deregistration

2.10.1 Overview

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

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.

The API for registering and deregistering is defined as follows:

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

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

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

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

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

.

This API returns HSA_STATUS_SUCCESS to indicate that registration/deregistration has been successfully performed. Otherwise, it returns one of the following for status:

- ▷ HSA_STATUS_ERROR_INVALID_ARGUMENT if a NULL value is passed for *address* or 0 for size.
- ▷ HSA_STATUS_INFO_NOT_REGISTERED this is applicable to **hsa_memory_deregister** API and indicates that a degistration is attempted on an address of memory that has not been registered or a part of prior registered ranges.
- ▷ HSA_STATUS_INFO_OUT_OF_RESOURCES if there is a failure in allocating necessary resources to perform registration. Note that this just an info, since registration doesn't impact functionality, the user can still continue considering this an info.
- ▷ status > HSA_STATUS_OTHER_BEGIN Any implementation specific error has a error value >HSA_STATUS_OTHER_BEGIN (see 2.1 for details). One of causes for this status could be that registration is not supported on a particular platform.

2.10.2 Usage

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 counterintuitive, 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.

2.11 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 provides host interface to allocate/deallocate and access component local memory. The result of the allocation is a pointer to an address in application processes address space, which can be accessed directly by the component during kernel execution.

The API is defined as follows:

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

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

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

address output. A valid pointer to the location of where to return the pointer to the base of the allocated region of memory.

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

address input. A pointer to the address to be deallocated. If the pointer is NULL, no operation is performed.

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

dst user allocated, output. A valid pointer to the destination array where the content is to be copied.

src input. A valid pointer to the source of data to be copied.

size input. Number of bytes to copy.

All the three API return HSA_STATUS_SUCCESS when the allocate/free/copy is successful. Otherwise, one of the following status values is returned:

- 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. The **hsa_memory_allocate_component_local** and **hsa_memory_copy_component_local_to_system** API can return this error.

▷ HSA_STATUS_ERROR_INVALID_ARGUMENT if *component*, *address*, *src* or *dst* are not valid and if the *address* pointer in **hsa_memory_allocate_component_local** is NULL. Allocation of size 0 is allowed.

2.11.1 Usage

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.

2.12 Execution Control At the Core Level

As per the systems architecture specification, HSA system must support debugging of a HSAIL kernel. The HSA Programmers Reference Manual (PRM) describes that the ”block” section could hold debug data and such a section can be placed even 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 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.

```
typedef enum hsa_interrupt_condition_t{
    HSA_DEBUGTRAP=1,
    HSA_SYSCALL=4,
    HSA_OTHER_INTERRUPT=8
} hsa_interrupt_condition_t;
```

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;
```

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

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

condition what caused this execution to stop.

workitem_id the flattend workitem IDs, array[workgroup_size].

compute_unit_id the 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 the 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, etc.
    }
    hsa_signal_send(queue_ptr->mailbox[i].full_flag, 0);
```

```
    }
  }
```

2.13 Syscall Support at the Core Level

The mailbox described in the Debugging At the Core Level section is also utilized in supporting the HSAIL syscall instruction.

The syscall operation in HSAIL is used to call functions that are outsize of the HSAIL Program.

The responsibility of the HSA core runtime is to carry this information and provide it to the agent so appropriate action can be taken. Syscall is unique in that once invoked, unless the function is executed on the host, the control is not returned back to the HSAIL program. Hence, syscall requires a synchronous communication with the agent.

An example of syscall invocation is shown below, for more information, refer to the HSA programmers reference manual.

```
syscall_u32 $s1, 3, $s2, $s3, $s4;
syscall_u64 $d1, 10, $d2, 100, $d4;
```

Consider the example shown in Core API Documentation. The code to wait for completion: Here is a simple example that shows how syscall handler may be written:

```
    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++) {
      if(HSA_STATUS_SUCCESS ==
          hsa_signal_poll(queue->mailbox[i].full_flag,HSA_EQUALS,
             1))
        for(j=0; j < queue->mailbox[i].workgroup_size; j++) {
          process_syscall_args(queue->mailbox[i].args,
             queue->mailbox[i].workgroup_size);
        }
    }
```

Support for syscall and tie-in of the execution information to an actual kernel, etc, is a good candidate for a service level API – it has use across implementors.

Appendices

A

Component Initiated Dispatches

Core API Documentation.

A.1 Component-Initiated Dispatch

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 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 dispatchs 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 dispatchd.
- 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.

B

Error Status Structure and Defined Values

Bibliography

[1] HSA Foundation. The Heterogeneous Systems Architecture Programmers Reference Manual. Technical report, HSA Foundation, 2013.

[2] HSA Foundation. The Heterogeneous Systems Architecture Systems Architecture Requirements Specification . Technical report, HSA Foundation, 2013.