HSA Multi-vendor Specification

Revision: Version 1.1 • Issue Date: 25 Feb 2016
© 2016 HSA Foundation. All rights reserved.

The contents of this document are provided in connection with the HSA Foundation specifications. This specification is protected by copyright laws and contains material proprietary to the HSA Foundation. It or any components may not be reproduced, republished, distributed, transmitted, displayed, broadcast or otherwise exploited in any manner without the express prior written permission of HSA Foundation. You may use this specification for implementing the functionality therein, without altering or removing any trademark, copyright or other notice from the specification, but the receipt or possession of this specification does not convey any rights to reproduce, disclose, or distribute its contents, or to manufacture, use, or sell anything that it may describe, in whole or in part.

HSA Foundation grants express permission to any current Founder, Promoter, Supporter Contributor, Academic or Associate member of HSA Foundation to copy and redistribute UNMODIFIED versions of this specification in any fashion, provided that NO CHARGE is made for the specification and the latest available update of the specification for any version of the API is used whenever possible. Such distributed specification may be re-formatted AS LONG AS the contents of the specification are not changed in any way. The specification may be incorporated into a product that is sold as long as such product includes significant independent work developed by the seller. A link to the current version of this specification on the HSA Foundation web-site should be included whenever possible with specification distributions.

HSA Foundation makes no, and expressly disclaims any, representations or warranties, express or implied, regarding this specification, including, without limitation, any implied warranties of merchantability or fitness for a particular purpose or non-infringement of any intellectual property. HSA Foundation makes no, and expressly disclaims any, warranties, express or implied, regarding the correctness, accuracy, completeness, timeliness, and reliability of the specification. Under no circumstances will the HSA Foundation, or any of its Founders, Promoters, Supporters, Academic, Contributors, and Associates members or their respective partners, officers, directors, employees, agents or representatives be liable for any damages, whether direct, indirect, special or consequential damages for lost revenues, lost profits, or otherwise, arising from or in connection with these materials.
Acknowledgments

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

**AMD**
- Paul Blinzer
- John Bridgman
- Mark Fowler
- Sean Keely
- Michael Mantor
- Rex McCravy
- Ben Sander
- Hari Thangirala
- Tony Tye

**ARM**
- Djordje Kovacevic
- Håkan Persson (Spec Editor)

**General Processor Technologies**
- John Glossner
- Stuart Stanley

**Imagination**
- James Aldis
- Jason Meredith
- Mark Rankilor

**Mediatek**
- Rahul Agarwal
- Richard Bagley
- Howard Chen
- Roy Ju (Workgroup Chair)
- James Lai
- Kenny Lee

**Texas Instruments**
- Abhijeet Chachad
- Filip Moerman

**Toshiba**
- Seiji Maeda
- Yoshio Masubuchi
- Hiroyuki Usui

**HSA Multi-vendor Specification, Version 1.1 | 3**
# Contents

**Acknowledgments** ............................................................................................................. 3

**Chapter 1. Introduction** ........................................................................................................ 8

1.1 Overview ................................................................................................................................. 8
1.1.1 HSA compatibility ................................................................................................................. 8
1.1.2 Rapid integration .................................................................................................................... 8
1.2 Audience ................................................................................................................................... 9
1.3 Platform scenarios ..................................................................................................................... 9
1.4 Multi-vendor architecture ........................................................................................................ 9
1.5 Specification organization ....................................................................................................... 11
1.5.1 Component interface .......................................................................................................... 11
1.5.2 Signals ................................................................................................................................. 11
1.5.3 User mode queues .............................................................................................................. 12
1.5.4 Shared virtual memory (SVM) ............................................................................................. 12

**Chapter 2. Component Interface** ............................................................................................. 13

2.1 Overview ................................................................................................................................. 13
2.2 Calling convention .................................................................................................................... 13
2.3 Error codes .............................................................................................................................. 14
2.4 Component interface summary ............................................................................................... 14
2.5 Configuration and initialization interface ................................................................................ 16
2.5.1 hsa_register_agent .............................................................................................................. 16
2.5.2 struct agent_driver_config ................................................................................................. 17
2.5.3 struct generic_driver_config .............................................................................................. 17
2.6 Client interface ....................................................................................................................... 18
2.6.1 client_new ............................................................................................................................ 18
2.6.2 client_remove ...................................................................................................................... 19
2.6.3 client_error .......................................................................................................................... 20
2.7 Signal interface ....................................................................................................................... 20
2.7.1 signal_new ............................................................................................................................ 20
2.7.2 signal_remove ...................................................................................................................... 21
2.7.3 destination_unreachable ....................................................................................................... 22
2.7.4 unblock_destination ............................................................................................................ 23
2.7.5 signal_group_new ............................................................................................................... 23
2.7.6 signal_group_remove .......................................................................................................... 24
2.8 User mode queue interface .................................................................................................... 25
2.8.1 queue_new .......................................................................................................................... 25
2.8.2 queue_remove .................................................................................................................... 27
2.9 Memory interface .................................................................................................................... 28
2.9.1 Structures .......................................................................................................................... 28
2.9.1.1 enum hsam_segment ....................................................................................................... 28
2.9.1.2 enum hsam_global_flag ................................................................................................. 28
2.9.1.3 emun hsam_access_permission ................................................................................... 28
2.9.2 memory_allocation ............................................................................................................ 29
2.9.3 memory_free ....................................................................................................................... 30
2.9.4 memory_acquire .................................................................................................................. 31
2.9.5 memory_release .................................................................................................................. 32
2.9.6 memory_copy ...................................................................................................................... 32
2.10 Agent memory interface ....................................................................................................... 33
2.10.1 agent_memory_reserve ...................................................................................................... 33
2.10.2 agent_memory_unreserve ............................................................... 34
2.10.3 agent_memory_allocate ............................................................... 35
2.10.4 agent_memory_allocate_safe_to_access ............................................. 35
2.10.5 agent_memory_free ................................................................. 36
2.10.6 agent_memory_map ............................................................... 37
2.10.7 agent_memory_unmap ............................................................. 37
2.10.8 agent_memory_allocate_physical .................................................. 38
2.10.9 agent_memory_free_physical ...................................................... 39
2.11 Miscellaneous interface .................................................................... 39
2.11.1 agent_reply ............................................................................ 39
2.11.2 launch_dll ............................................................................. 40

Chapter 3. Signals ................................................................................. 41
3.1 Overview .................................................................................... 41
  3.1.1 Multi-vendor considerations ......................................................... 41
  3.1.2 Multi-vendor signal strategy ......................................................... 42
3.2 Signal object ............................................................................. 43
  3.2.1 hsa_signal_object ..................................................................... 43
  3.2.2 Signal value location ................................................................ 44
  3.2.3 Array fields ........................................................................... 44
3.3 Signal group object ................................................................. 44
  3.3.1 hsa_signal_group_object .......................................................... 44
3.4 Root destination node ............................................................... 45
  3.4.1 hsa_root_dest_node ................................................................ 45
3.5 Destination node ......................................................................... 46
  3.5.1 hsa_destination_node ............................................................... 46
3.6 Group node ............................................................................... 47
  3.6.1 hsa_group_node ..................................................................... 47
3.7 Signal group destination ............................................................. 48
  3.7.1 hsa_signal_group_destination ................................................... 48
3.8 Destination nodes ........................................................................ 48
  3.8.1 Root destination node ............................................................... 49
  3.8.2 Destination node ................................................................... 50
  3.8.3 Group node ........................................................................... 51
3.9 Dynamic updates to signal object and nodes .................................... 53
  3.9.1 Dynamic signal object updates .................................................... 54
  3.9.2 Dynamic destination node updates ............................................. 56
  3.9.3 Dynamic group node updates ..................................................... 58
  3.9.4 Lifetime of node objects and doorbell locations .............................. 59
3.10 Object locations and memory attributes ........................................ 59
  3.10.1 HSA base profile .................................................................... 60
  3.10.2 HSA full profile .................................................................... 60
  3.10.3 Attributes ............................................................................ 61
3.11 Signal APIs ............................................................................. 61
  3.11.1 hsa_signal_wait ................................................................... 61
  3.11.2 hsa_signal_group_wait ........................................................... 64
  3.11.3 hsa_signal_send ................................................................... 67
3.12 Internal agent driver structures ..................................................... 69
  3.12.1 driver_group_internal_data_t ..................................................... 69
3.13 Internal agent driver functions ....................................................... 70
  3.13.1 barrier ................................................................................. 70
  3.13.2 get_my_dest_node .................................................................. 70
  3.13.3 get_my_root_node .................................................................. 71
  3.13.4 get_my_group_destination ....................................................... 71
Chapter 3. Component Interface APIs

3.13.5 wait_on_condition .................................................. 71
3.13.6 clear_signal_doorbell .............................................. 72
3.13.7 wait_on_signal_doorbell ........................................... 73
3.13.8 clear_group_doorbell ................................................ 73
3.13.9 wait_on_group_doorbell ............................................ 74
3.13.10 test_condition ...................................................... 74
3.13.11 yield_destination .................................................. 75
3.13.12 set_root_dest_inactive .......................................... 75
3.13.13 set_root_dest_active ............................................. 76
3.13.14 set_dest_node_in_root ............................................ 76

3.14 Component interface APIs

3.14.1 signal_new ............................................................ 77
3.14.2 signal_new_reply ................................................... 77
3.14.3 signal_remove ....................................................... 77
3.14.4 signal_remove_reply ................................................ 77
3.14.5 signal_group_new ................................................... 77
3.14.6 signal_group_new_reply .......................................... 78
3.14.7 signal_group_remove ............................................. 78
3.14.8 signal_group_remove_reply ..................................... 78
3.14.9 destination_unreachable ......................................... 78
3.14.10 unblock_destination ............................................. 78

Chapter 4. User Mode Queues ............................................. 79

4.1 Overview ................................................................. 79
4.2 User mode queue header ............................................. 79
4.3 Read and write indexes .............................................. 81
4.4 User mode queue payload .......................................... 81

Chapter 5. HSA Agent Drivers for Memory Management .......... 82

5.1 Overview ................................................................. 82
5.2 Multi-vendor memory management ................................ 83
5.3 APIs ........................................................................ 84
5.3.1 update_range_on_agent .......................................... 84
5.3.2 report_page_fault .................................................. 85
5.4 Use cases ............................................................... 85
5.4.1 Starting a client application ..................................... 85
5.4.2 Client application executes hsa_shut_down call ............ 86
5.4.3 Allocating fine-grained buffer from HSA allocator ........ 86
5.4.4 Allocating coarse-grained buffer to an agent for the first time 86
5.4.5 Changing ownership of coarse-grained buffer to a different agent 87
5.4.6 Register system buffer from OS allocator ................. 87
5.4.7 Memory copying ..................................................... 87
5.4.8 Page not allocated for agent implements demand paging 88
5.4.9 Allocation of private and group segment before kernel execution 88
5.4.10 Page fault at the host agent (CPU) ....................... 89
5.4.11 Kernel OS changes page tables of the client for shadow page tables 89
5.5 Full profile shadow page tables under Linux OS with heterogeneous memory management (HMM) .......... 89
5.5.1 HMM data structure .............................................. 89
5.5.1.1 hmm_mirror ..................................................... 89
5.5.1.2 hmm_event ..................................................... 90
5.5.1.3 hmm_device ................................................... 91
5.5.2 HMM APIs for agent drivers .................................... 91
5.5.2.1 hmm_device_register ....................................... 91
5.5.2.2 hmm_device_unregister ................................... 92
5.5.2.3 hmm_mirror_register .................................................. 92
5.5.2.4 hmm_mirror_unregister .............................................. 92
5.5.2.5 hmm_mirror_fault ....................................................... 92
5.5.3 Device supported callback functions for HMM .......................... 93
5.5.3.1 release ................................................................. 93
5.5.3.2 update ................................................................. 93

Appendix A. Glossary .......................................................... 95

Index ............................................................................. 101

Figures

Figure 1–1 Multi-vendor architecture ............................................ 10
Figure 2–1 Calling convention use cases ....................................... 14
Figure 3–1 Signal object destination node hierarchy ....................... 43
Figure 3–2 Root destination node example (without additional hierarchy) ................................................................. 49
Figure 3–3 Root destination node example (with single hierarchy) ................................................................. 50
Figure 3–4 Root destination node example (with additional hierarchy) ................................................................. 51
Figure 3–5 Signal group node example (with two signals) .................. 52
Figure 3–6 Signal group node example (with three signals and two signal groups) ................................................................. 53
Figure 3–7 Example of dynamic signal object update for destination moving to INACTIVE state within root node ................................................................. 55
Figure 3–8 Example of dynamic signal object update for destination moving to ACTIVE state within root node ................................................................. 55
Figure 3–9 Example of dynamic signal object update for destination node pointer update within root node ................................................................. 55
Figure 3–10 Example of dynamic node update for destination moving to INACTIVE state ................................................................. 57
Figure 3–11 Example of dynamic node update for destination exchanging nodes ................................................................. 57
Figure 3–12 Dynamic exchange of destination node in hierarchy ........ 58
Figure 3–13 Pseudo code definition of active signal wait ..................... 62
Figure 3–14 Pseudo code definition of wait_on_condition contained within user privilege ................................................................. 62
Figure 3–15 Pseudo code definition of wait_on_condition using agent driver ................................................................. 63
Figure 3–16 Pseudo code example of inactive wait .............................. 64
Figure 3–17 Pseudo code example of agent driver yield_destination ................................................................. 64
Figure 3–18 Pseudo code definition of active hsa_signal_group_wait ................................................................. 66
Figure 3–19 Pseudo code definition of inactive hsa_signal_group_wait ................................................................. 67
Figure 3–20 Pseudo code definition of signal send .............................. 69
Figure 5–1 Memory management related HSA software stack ............... 83
CHAPTER 1.
Introduction

1.1 Overview

HSA describes system-level support for powerful and efficient resource sharing and computation across devices for:

- **Data sharing** (coherent caches and shared virtual memory)
- **Intra-process communication** (signals)
- **Efficient and symmetric kernel dispatch** (user mode queues and an architected query language)

For multi-vendor platforms, this specification:

- Describes one way for SoC IP vendors to design their hardware and associated software for HSA compatibility (see 1.1.1 HSA compatibility (below))
- Permits rapid integration of HSA implementations using IP from multiple vendors (see 1.1.2 Rapid integration (below))
- Allows development of HSA system software that is portable between implementations

1.1.1 HSA compatibility

Certain HSA features need further elaboration in a multi-vendor context, including:

- Enabling system-wide signaling and signal semantics
- Enabling task submissions to user mode queues between agents from different vendors
- Standardizing communication between common and vendor specific software components

1.1.2 Rapid integration

Implementing many HSA requirements typically involves hardware, if only for reasons of performance. Ensuring interoperability between devices on an HSA platform is the responsibility of the system integration.

- When a platform integrator also produces the devices, all details needed for integration are known.
- When a platform integrator does not produce the devices but licenses the intellectual property of a device supplier, sufficient information is available to integrate parts on a platform. However, there may be limited flexibility to vary device specifications, either directly or via feedback from the supplier, to best integrate the parts. This limitation may be more apparent when integrating parts in support of the relatively advanced features of HSA.

With either scenario, different platform integrators will implement unique and typically proprietary solutions, a practice that deprives the ecosystem of the benefits of standardization and which could be even more of a liability with HSA platforms that involve greater varieties of parts and suppliers. The benefits of standardization may extend to issues with coexistence of components participating in HSA and those that do not.
This specification can be used to lessen the burden of integration and enable points of standardization across platform integrators. Definitions of HSA system features are provided in a way that is commonly accepted but can be implemented differently per device.

**1.2 Audience**

The audience for this specification includes:

- Platform integrators
- Component suppliers
- System software services (runtimes, verification)

**1.3 Platform scenarios**

Platforms are considered to have two or more agents. In one simple example, this consists of one CPU and one GPU. Enriching the platform for heterogeneous compute with additional devices of other types (e.g., VPU, DSP) makes integration more challenging.

Some devices may also make further demands on this specification. For example, integrating visual processing could require a mobile platform to move from coarse-grained coherency to the greater demands of fine-grained cache coherency.

This specification invites a multiplicity of devices to participate in an HSA implementation, but it is important to consider if the implementation can efficiently scale with increasing numbers of devices.

**1.4 Multi-vendor architecture**

Figure 1–1 (on the next page) shows the multi-vendor architecture. Architecture components are listed in Table 1–1 (on the next page).

This architecture:

- Provides a scalable solution for HSA implementation that doesn't limit the number of clients, signals, or user mode queues.
- Allows different implementations of signaling mechanisms to coexist within the same HSA implementation.
Chapter 1. Introduction

1.4 Multi-vendor architecture

Figure 1–1 Multi-vendor architecture

Table 1–1 Multi-vendor architecture components

<table>
<thead>
<tr>
<th>Component</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HSA client</td>
<td>The client is the user application that links against the HSA runtime user library. This application may queue packets for execution at agents’ execution units. The number of concurrent clients in the HSA implementation is unlimited.</td>
</tr>
<tr>
<td>Execution units</td>
<td>HSA agents provide execution units for clients to execute their packets.</td>
</tr>
<tr>
<td>Vendors</td>
<td>Each vendor provides one or more HSA agents.</td>
</tr>
<tr>
<td>HSA multi-vendor platform</td>
<td>This platform consists of the HSA runtime and generic HSA driver and is an integral part of any multi-vendor HSA implementation.</td>
</tr>
<tr>
<td>HSA runtime</td>
<td>The HSA runtime is the user library provided by the multi-vendor platform. This is the only user library needed by a client to link against in order to schedule packets on any of HSA agents in an HSA implementation.</td>
</tr>
<tr>
<td>Generic HSA driver</td>
<td>The generic HSA driver is a hardware-agnostic device driver provided by the multi-vendor platform. Its main task is to pass messages among agent drivers and the HSA runtime user library. The generic HSA driver implements the generic side of the multi-vendor component interface.</td>
</tr>
</tbody>
</table>

1Exception: Using vendor specific HSA extensions may require explicit links to user libraries provided by the vendor.
### Component Interface

**Agent driver**
An agent driver is implemented by vendors that provide HSA agents, which communicates with the generic HSA driver through the multi-vendor component interface and implements a packet processor and scheduler for their HSA agents. The multi-vendor architecture (see Figure 1-1 (on the previous page)) supports different types of packet processors; the two main types are described in Chapter 2 Component Interface (on page 13).

**Operating system with MMU driver**
The CPU operating system (OS) is expected to provide an interface for managing the virtual address space of the clients. See 5.5.2 HMM APIs for agent drivers (on page 91) for an example of the HMM library in the Linux OS.

<table>
<thead>
<tr>
<th>Component</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Agent driver</td>
<td>An agent driver is implemented by vendors that provide HSA agents, which communicates with the generic HSA driver through the multi-vendor component interface and implements a packet processor and scheduler for their HSA agents. The multi-vendor architecture (see Figure 1-1 (on the previous page)) supports different types of packet processors; the two main types are described in Chapter 2 Component Interface (on page 13).</td>
</tr>
<tr>
<td>Operating system with MMU driver</td>
<td>The CPU operating system (OS) is expected to provide an interface for managing the virtual address space of the clients. See 5.5.2 HMM APIs for agent drivers (on page 91) for an example of the HMM library in the Linux OS.</td>
</tr>
</tbody>
</table>

### 1.5 Specification organization

This specification acts as an appendix to the **HSA Platform System Architecture Specification Version 1.1**. Its purpose is to:

- Enable hardware design of multi-vendor compatible agents
- Propose a software architecture for implementing common HSA software components on Linux (HSA runtime and HSA kernel driver)

The multi-vendor architecture shown in Figure 1-1 (on the previous page) consists of:

- Generic HSA runtime (vendor independent)
- Generic HSA driver (vendor independent)
- Component interface (see 1.5.1 Component interface (below))

In addition, the following key implementation-defined aspects of the HSA system architecture are described:

- Signals (see 1.5.2 Signals (below))
- User mode queues (see 1.5.3 User mode queues (on the next page))
- Shared virtual memory (see 1.5.4 Shared virtual memory (SVM) (on the next page))

This specification does not cover extensions, including the finalizer and images extensions.

#### 1.5.1 Component interface

Chapter 2 Component Interface (on page 13) describes the component interface, which is an API for communication between the generic HSA driver and a vendor specific device driver.

The component interface API describes:

- Mechanisms for adding and removing clients
- Calling conventions
- Error codes
- Signals
- User mode queues
- Memory management

#### 1.5.2 Signals

Chapter 3 Signals (on page 41) describes the signaling mechanism in HSA.
An HSA signal is an opaque construct. In a multi-vendor scenario, the signal needs to be accessible, and there needs to be common agreement as to what a signal means on different platforms.

Signal support is commonly implemented partially or entirely in hardware. SOC integration is faced with the task of devising hardware or software to unify different signaling mechanisms. This specification aims to make this task more transparent and offer a common basis for solution.

1.5.3 User mode queues

Chapter 4 User Mode Queues (on page 79) describes user mode queues, which is part of a mechanism for fast, efficient, and symmetric dispatch. These queues are containers for packets defined as part of a query language (AQL). The packets encode necessary details for execution of a kernel.

A user mode queue also has properties including a read and write index. When populating the queue with packets for a kernel, the last known write index is incremented to allocate space for the one or more packets needed. When all packets are written to the queue, a notification doorbell signal is set that notifies a packet processor that additional packets are available for processing. As packets are processed, the read index is incremented.

In the specification, the read and write index are opaque handles, enabling a single vendor to determine implementation. In a multi-vendor setting, these values are exposed as generic addresses.

1.5.4 Shared virtual memory (SVM)

Chapter 5 HSA Agent Drivers for Memory Management (on page 82) defines management of page tables across vendors and provides details for either shared or shadow page table access. For the latter case, an HSA agent can maintain its own version of the page table and must take responsibility to keep its shadow page table aligned with that of the host.
CHAPTER 2.
Component Interface

2.1 Overview

This chapter describes the component interface. As shown in Figure 1-1 (on page 10), the component interface is the communication channel between the generic HSA driver on one side and vendor-specific agent drivers on the other side. Note that the architecture doesn't provide a direct communication channel among vendors' agent drivers.

The component interface accounts for different implementations of HSA agents (and their packet processors in particular), where two types of packet processor are:

- **CPU-based packet processor**: The packet processor functionality is provided by a vendor-specific agent driver on the host CPU. Some parts of the packet processor may execute in user virtual space of HSA clients.

- **Unit-based packet processor**: The functionality of the packet processor is provided on a unit other than the host CPU (e.g., a microcontroller tightly coupled with a GPU, a system-level microcontroller providing packet processor services to several accelerators, a DSP with a self-hosted packet processor or by some other arrangements).

These implementations of packet processors favor different designs of the component interface. While a CPU-based packet processor may usually acknowledge requests from the generic HSA driver promptly, the communication with a unit-based packet processor will likely require some sort of hardware resources, such as interrupts. For that reason, this specification provides both synchronous and asynchronous communication methods between the generic HSA driver and agent drivers.

2.2 Calling convention

Agent drivers may handle requests from the generic HSA driver either synchronously or asynchronously:

1. A request is synchronously acknowledged if an agent driver returns `HSAM_Done` error code.

2. A request is synchronously rejected if an agent driver returns any error code other than `HSAM_Done` and `HSAM_Async`.

3. A request is asynchronously acknowledged if:
   
   - An agent driver returns `HSAM_Async` error code and,
   
   - The agent driver sends an `agent_reply` message back to the generic HSA driver with the `HSAM_Done` error code.

4. A request is asynchronously rejected if:
   
   - An agent driver returns `HSAM_Async` error code and,
   
   - The agent driver sends an `agent_reply` message back to the generic HSA driver with any error code other than `HSAM_Done` and `HSAM_Async`.

Corresponding use cases are shown in Figure 2-1 (on the next page).
The requests from agent drivers are always handled by the generic HSA driver in a synchronous manner.

2.3 Error codes

The following enumeration defines the standard set of error codes in the component interface. The purpose of HSAMAsync error code is described in 2.2 Calling convention (on the previous page).

```c
enum HSAM_Error {
    HSAM_Done = 0,
    HSAM_Async = -1,
    HSAM_Not_Supported = -2,
    HSAM_OutOfResources = -3,
    //It is expected that the implementation of the generic HSA driver
    //provides the appropriate error codes
}
```

2.4 Component interface summary

Table 2–1 (on the facing page) summarizes the component interface:
- **Green rows** denote messages from the generic HSA driver to the agent driver.
- **Amber rows** denote messages from agent drivers to the generic HSA driver.

### Table 2–1 Component Interface

<table>
<thead>
<tr>
<th>Method</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Initialization Interface</strong></td>
<td></td>
</tr>
<tr>
<td><code>hsa_register_agent</code></td>
<td>Agent registration at boot time.</td>
</tr>
<tr>
<td><strong>Client Interface</strong></td>
<td></td>
</tr>
<tr>
<td><code>client_new</code></td>
<td>New application.</td>
</tr>
<tr>
<td><code>client_remove</code></td>
<td>Application shut down.</td>
</tr>
<tr>
<td><code>client_error</code></td>
<td>Client's fault that is not resolved by demand paging.</td>
</tr>
<tr>
<td><strong>Signal Interface</strong></td>
<td></td>
</tr>
<tr>
<td><code>signal_new</code></td>
<td>New signaling object.</td>
</tr>
<tr>
<td><code>signal_remove</code></td>
<td>Signal is destroyed.</td>
</tr>
<tr>
<td><code>destination_unreachable</code></td>
<td>Destination of a signal is inactive.</td>
</tr>
<tr>
<td><code>unblock_destination</code></td>
<td>Wake-up call for signal's destination.</td>
</tr>
<tr>
<td><code>signal_group_new</code></td>
<td>New signal group object.</td>
</tr>
<tr>
<td><code>signal_group_remove</code></td>
<td>Signal group destroyed.</td>
</tr>
<tr>
<td><strong>User Mode Queue Interface</strong></td>
<td></td>
</tr>
<tr>
<td><code>queue_new</code></td>
<td>New user mode queue.</td>
</tr>
<tr>
<td><code>queue_remove</code></td>
<td>User mode queue is destroyed.</td>
</tr>
<tr>
<td><strong>Memory Interface</strong></td>
<td></td>
</tr>
<tr>
<td><code>memory_allocation</code></td>
<td>HSA buffer allocation or registration of system memory.</td>
</tr>
<tr>
<td><code>memory_free</code></td>
<td>HSA buffer free or deregistration of system memory.</td>
</tr>
<tr>
<td><code>memory_acquire</code></td>
<td>Taking the ownership of a coarse-grained allocation.</td>
</tr>
<tr>
<td><code>memory_release</code></td>
<td>Giving up the ownership of a coarse-grained allocation.</td>
</tr>
<tr>
<td><code>memory_copy</code></td>
<td>Data copy.</td>
</tr>
<tr>
<td><strong>Agent Memory Interface</strong></td>
<td></td>
</tr>
<tr>
<td><code>agent_memory_reserve</code></td>
<td>Reserves virtual space in a client without physical mapping.</td>
</tr>
<tr>
<td><code>agent_memory_unreserve</code></td>
<td>Reverts <code>agent_memory_reserve</code>.</td>
</tr>
<tr>
<td><code>agent_memory_allocate</code></td>
<td>Allocates virtual memory in a client.</td>
</tr>
<tr>
<td><code>agent_memory.allocate_safe_to_access</code></td>
<td>Allocates safe-to-access virtual memory in a client.</td>
</tr>
<tr>
<td><code>agent_memory_free</code></td>
<td>Reverts <code>agent_memory.allocate_safe_to_access</code>.</td>
</tr>
<tr>
<td><code>agent_memory_map</code></td>
<td>Maps an arbitrary physical memory into a client’s space.</td>
</tr>
<tr>
<td><code>agent_memory_unmap</code></td>
<td>Reverts <code>agent_memory_map</code>.</td>
</tr>
<tr>
<td><code>agent_memory.allocate_physical</code></td>
<td>Allocates physical memory from the OS’s system pool.</td>
</tr>
<tr>
<td><code>agent_memory.free_physical</code></td>
<td>Reverts <code>agent_memory.allocate_physical</code>.</td>
</tr>
<tr>
<td><strong>Miscellaneous Interface</strong></td>
<td></td>
</tr>
<tr>
<td><code>agent_reply</code></td>
<td>Asynchronous reply from the agent driver.</td>
</tr>
<tr>
<td><code>launch_dll</code></td>
<td>Vendor’s request to launch a user DLL in client’s space.</td>
</tr>
</tbody>
</table>
2.5 Configuration and initialization interface

This section describes the handshake mechanism between the generic HSA driver and agent drivers at boot time. Each agent must register with the generic HSA driver and exchange configuration parameters.

2.5.1 hsa_register_agent

This function registers an HSA agent with the generic HSA driver and exchanges configuration parameters. It is called by each agent driver at initialization. All agents must register with the generic HSA driver through this interface before any client application is launched.

NOTE: The generic HSA driver must export this function in the library that all agent drivers link against.

Direction
From the agent driver to the generic HSA driver

Signature

```c
const hsa_generic_driver_interface* hsa_register_agent
    ( const hsa_agent_driver_interface* agent_driver_interface,
      const hsa_agent_driver_config* agent_driver_config,
      hsa_generic_driver_config* generic_driver_config
    )
```

Parameters

agent_driver_interface
Component interface of agent driver.

The structure `hsa_agent_driver_interface` holds a set of functions that represents the component interface of agent driver. The functions are labeled with the “From the generic HSA driver to the agent driver” direction in this section and are marked in green in Table 2–1 (on the previous page).

agent_driver_config

The agent's capabilities and configuration parameters provided by the agent (see 2.5.2 struct agent_driver_config (on the facing page)).

generic_driver_config

Configuration parameters provided by the generic HSA driver. It points to structure that is allocated by the agent driver and populated by the generic HSA driver during a `hsa_register_agent` call (see 2.5.3 struct generic_driver_config (on the facing page)).

Return values

Component interface of the generic HSA driver

The structure `hsa_generic_driver_interface` holds a set of functions that represents the component interface of the generic HSA driver. The functions are labeled with the “From the agent driver to the generic HSA driver” direction in this section and are marked in amber in Table 2–1 (on the previous page).
2.5.2 struct agent_driver_config

This data structure is passed to the generic HSA driver at the hsa_register_agent call when the HSA agent is registered. It contains the following information about the agent.

**General information**

Multi-vendor architecture major version
The major version of the multi-vendor architecture for which the agent is intended.

Multi-vendor architecture minor version
The minor version of the multi-vendor architecture for which the agent is intended.

Agent ID
Unique identification of the HSA agent that this agent driver represents. This ID is visible to clients and, for example, is used to specify destination agent of a user mode queue.

Name
Description of the agent.

Base or full profile
HSA profile that the agent supports.

**User mode queue**

Private field size
The size of destination_private field in the user mode queue.

Request for safe-to-access payload
If requested, the payload of all user queues that belong to this agent shall be placed in safe-to-access memory (see Appendix A Glossary (on page 95)).

Request for safe-to-access payload for all user mode queues
If requested, the payload of all user queues for all agents shall be placed in “safe to access” memory (see Appendix A Glossary (on page 95)).

Request for safe-to-access header of all user mode queues
If requested, the headers of user mode queues of all agents shall be placed in “safe to access” memory (see Appendix A Glossary (on page 95)).

**Memory**

Safe-to-access request for all memory allocations
If requested, the following memory shall be safe to access:

All coarse-grained and fine-grained memory allocated through the hsa_memory_allocate runtime API. (See HSA Runtime Programmer's Reference Manual Version 1.1, section 2.7.4.7 hsa_memory_allocate.)

System memory registered through the hsa_memory_register runtime API. (See HSA Runtime Programmer's Reference Manual Version 1.1, section 2.7.4.11 hsa_memory_register.)

2.5.3 struct generic_driver_config

The structure is allocated by the agent driver and populated by the generic HSA driver during hsa_register_agent call. It contains the following information.
**General information**

Generic HSA driver multi-vendor architecture major version
The major version of the multi-vendor architecture that the generic HSA driver supports.

The drivers are compatible if the major.minor version of the generic HSA driver is equal to or larger than the major.minor version of all agent drivers.

Generic HSA driver multi-vendor architecture minor version
The minor version of the multi-vendor architecture that the generic HSA driver supports.

Device driver ID
Unique identification of the caller. The generic HSA driver assigns a different device driver ID to each agent driver.

Device driver ID is used in a signal object to identify the signal's destination.

### 2.6 Client interface

The client interface is a set of messages for handling the creation and shutdown of HSA applications.

#### 2.6.1 client_new

This informs the HSA agent that an application is a client of the HSA system. Note that multiple applications may simultaneously be the clients of an HSA system.

The message is sent when one of the following events occurs in a client, whichever happens to be the first:

- The client creates the first user mode queue with a given HSA agent being specified as its destination point.
- The client creates the first signal with a given HSA agent specified as one of its destination points.

**Direction**

From the generic HSA driver to the agent driver

**Signature**

```c
HSAM_Cmd client_new
(
    int64_t client_handle,
    void* user_virtual_space,
    void* cookie,
    int64_t* client_handle_at_agent
)
```

**Parameters**

*client_handle*
Uniquely identifies the client across the HSA system.

*user_virtual_space*
Describes the client's virtual space. Its internal structure is specific to the OS. For example, the Linux structure is `mm_struct`.

*cookie*
An arbitrary value that must be returned in the corresponding `agent_reply` call.
This is used by the generic HSA driver to match the corresponding request and reply messages. Note that the \texttt{agent_reply} message from the agent driver shall not occur if the request is handled synchronously.

\textit{client\_handle\_at\_agent}

Pointer to an opaque handle that must be set by the callee (agent driver) before the request is acknowledged.

This is used to identify the client in all messages directed to a given agent driver. The agent driver should specify the appropriate value of the handle that would ease the task of object lookup.

The handle may be left uninitialized if the request is rejected.

\textbf{Return values}

\texttt{HSAM\_Done}

The request is successfully completed synchronously.

\texttt{HSAM\_Async}

The request will be asynchronously completed with the \texttt{agent\_reply} message.

\texttt{Otherwise}

The request is rejected for the reason specified in the error code.

\subsection*{2.6.2 client\_remove}

This message is sent when a client executes the \texttt{hsa\_shut\_down} runtime API or when it is about to terminate. (See \textit{HSA Runtime Programmer’s Reference Manual Version 1.1}, section 2.1.1.2 \texttt{hsa\_shut\_down}.)

\textbf{Direction}

From the generic HSA driver to the agent driver

\textbf{Signature}

\begin{verbatim}
HSAM_Error client_remove
   ( int64_t client_handle_at_agent,
     void* cookie
   )
\end{verbatim}

\textbf{Parameters}

\textit{client\_handle\_at\_agent}

Opaque handle of the client object (specified by the agent driver on a \texttt{client\_new} request).

\textit{cookie}

An arbitrary value that must be returned in the corresponding \texttt{agent\_reply} call.

This is used by the generic HSA driver to match the corresponding request and reply messages. Note that the \texttt{agent\_reply} message from the agent driver shall not occur if the request is handled synchronously.
Return values

HSAM_Done
The request is successfully completed synchronously.

HSAM_Async
The request will be asynchronously completed with the agent_reply message.

NOTE: Other returned error codes should be considered fatal. All error codes other than HSAM_Done in the corresponding agent_reply message are also considered fatal.

2.6.3 client_error

This message is sent when a client (that executes on an agent) experiences a fault that cannot be resolved by demand paging.

Direction
From the agent driver to the generic HSA driver

Signature

```c
void client_error
(
    int64_t client_handle,
    // other parameters are specific to particular CPU architecture
    // and they should be defined by the actual implementation
    // of the generic HSA driver.*1
)
```

Parameter

`client_handle`
Unique handle of the client.

Return value

void

2.7 Signal interface

The signal interface handles the creation and destruction of signals and provides the mechanism to alert signals’ destinations that are in passive state. The internal structure of signals in the multi-vendor architecture is described in Chapter 3 Signals (on page 41).

2.7.1 signal_new

This message is sent when:

- A client creates a signal with a given HSA agent being specified as one of its destination points. When a client creates a signal with more than one destination point, this message is sent to each corresponding agent driver.
- A client creates a user mode queue with a given HSA agent being specified as the destination point.

Direction
From the generic HSA driver to the agent driver
Signature

```c
HSAM_Error signal_new
{
    int64_t client_handle_at_agent,
    int64_t signal_handle,
    hsa_signal_object* signal_user_addr,
    hsa_root_dest_node* root_dest_node_kern_addr,
    void* cookie,
    int64_t* signal_handle_at_agent
}
```

Parameters

- `client_handle_at_agent`:
  Opaque handle of the client object (specified by the agent driver on a client_new request).

- `signal_handle`:
  Uniquely defines the signal across the HSA system. It is defined by the generic HSA driver.

- `signal_user_addr`:
  Address of the signaling object in the client's virtual space.

- `root_dest_node_kern_addr`:
  Address of the agent's root destination node in the OS address space.

- `cookie`:
  An arbitrary value that must be returned in the corresponding agent_reply call.
  This is used by the generic HSA driver to match the corresponding request and reply messages. Note that the agent_reply message from the agent driver shall not occur if the request is handled synchronously.

- `signal_handle_at_agent`:
  Pointer to opaque handle that must be set by callee (agent driver) before the request is acknowledged.
  This is used to identify the signal in all messages directed to a given agent driver. The agent driver should specify the appropriate value of the handle that would ease the task of object lookup.
  The handle may be left uninitialized if the request is rejected.

Return values

- `HSAM_Done`:
  The request is successfully completed synchronously.

- `HSAM_Async`:
  The request will be asynchronously completed with the agent_reply message.

- Otherwise:
  The request is rejected for the reason specified in the error code.

2.7.2 signal_remove

This message is sent when a client destroys the signaling object.

Direction

From the generic HSA driver to the agent driver
Signature

```c
HSAM_Error signal_remove
(
    int64_t client_handle_at_agent,
    int64_t signal_handle_at_agent,
    void* cookie
)
```

Parameters

- `client_handle_at_agent`
  Opaque handle of the client (specified by the agent driver when the client object is created).

- `signal_handle_at_agent`
  Opaque handle of the signaling object (specified by the agent driver during the `signal_new` request).

- `cookie`
  An arbitrary value that must be returned in the corresponding `agent_reply` call.

  This is used by the generic HSA driver to match the corresponding request and reply messages. Note that the `agent_reply` message from the agent driver shall not occur if the request is handled synchronously.

Return values

- `HSAM_Done`
  The request is successfully completed synchronously.

- `HSAM_Async`
  The request will be asynchronously completed with the `agent_reply` message.

  NOTE: Other returned error codes should be considered fatal. All error codes other than `HSAM_Done` in the corresponding `agent_reply` message are also considered fatal.

### 2.7.3 destination_unreachable

This message is sent when a destination of a signal that has been triggered is marked as passive.

Direction

From the agent driver to the generic HSA driver

Signature

```c
void destination_unreachable
(
    int64_t client_handle,
    hsam_signal* signal_user_addr,
    int64_t agent
)
```

Parameters

- `client_handle`
  Unique handle of the client.
signal_user_addr
   Pointer to the signaling object whose destination is inactive. Note that this argument must be verified by the callee (generic HSA driver).

agent
   Inactive destination of the signal. Note that the destination may be active (false alarm is possible).

Return value
   void

2.7.4 unblock_destination
   This is a request for the signal's destination to wake up. Note that the destination may be active (false alarm is possible).

Direction
   From the generic HSA driver to the agent driver

Signature
   void unblock_destination
   (
      int64_t client_handle_at_agent,
      hsam_signal* signal_handle_at_agent
   )

Parameters
   client_handle_at_agent
      Opaque handle of the client that was specified by the agent driver when the client object is created.

   signal_handle_at_agent
      Opaque handle of the signaling object whose destination has to be activated.

Return value
   void

2.7.5 signal_group_new
   This message is sent when a client creates a signal group with a given HSA agent being specified as one of the consumers. It is sent to all consumers of the signal group.

Direction
   From the generic HSA driver to the agent driver

Signature
   HSAM_Error signal_group_new
   (
      int64_t client_handle_at_agent,
      int64_t signal_group_handle,
      int32_t num_signals,
      const int64_t* signal_handle_at_agent,
   )
```c
hsa_signal_group_object* signal_group_user_addr,
hsa_signal_group_destination* dest_kern_addr,
void* cookie,
int64_t* signal_group_handle_at_agent
)
```

### Parameters

**client_handle_at_agent**
Opaque handle of the client object (specified by the agent driver on a `client_new` request).

**signal_group_handle**
Uniquely defines the signal group across the HSA system.

**num_signals**
Number of signals in the signal group.

**signal_handle_at_agent**
Pointer to an array of "signal handles at agent" of the signal group.

**signal_group_user_addr**
Address of the signal group object in the client's address space.

**dest_kern_addr**
Address of the agent's `hsa_signal_group_destination` object in the OS address space.

**cookie**
An arbitrary value that must be returned in the corresponding `agent_reply` call.

This is used by the generic HSA driver to match the corresponding request and reply messages. Note that the `agent_reply` message from the agent driver shall not occur if the request is handled synchronously.

**signal_group_handle_at_agent**
Pointer to opaque handle that must be set by the callee (agent driver) before the request is acknowledged.

This is used to identify the signal group in the messages directed to a given agent driver. The agent driver should specify the appropriate value of the handle that would ease the task of object lookup.

The handle may be left uninitialized if the request is rejected.

### Return values

- **HSAM.Done**
The request is successfully completed synchronously.

- **HSAM.Async**
The request will be asynchronously completed with the `agent_reply` message.

### 2.7.6 `signal_group_remove`

This message is sent when the client destroys the signal group object.

### Direction
From the generic HSA driver to the agent driver

**Signature**

```c
HSAM_Error signal_group_remove
(
    int64_t client_handle_at_agent,
    int64_t signal_group_handle_at_agent,
    void* cookie
)
```

**Parameters**

*client_handle_at_agent*

Opaque handle of the client (specified by the agent driver when the client object is created).

*signal_group_handle_at_agent*

Opaque handle of the signal group object (specified by the agent driver during `signal_group_new` request).

*cookie*

An arbitrary value that must be returned in the corresponding `agent_reply` call.

This is used by the generic HSA driver to match the corresponding request and reply messages. Note that the `agent_reply` message from the agent driver shall not occur if the request is handled synchronously.

**Return values**

*HSAM_Done*

The request is successfully completed synchronously.

*HSAM_Async*

The request will be asynchronously completed with the `agent_reply` message.

NOTE: Other returned error codes should be considered fatal. All error codes other than `HSAM_Done` in the corresponding `agent_reply` message are also considered fatal.

### 2.8 User mode queue interface

The user mode queue interface handles the creation and destruction of user mode queues.

#### 2.8.1 queue_new

This message is sent when a client creates a user mode queue where a given HSA agent is specified as its destination.

**Direction**

From the generic HSA driver to the agent driver

**Signature**

```c
HSAM_Error queue_new
(
    int64_t client_handle_at_agent,
```
int64_t signal_handle_at_agent,
int64_t queue_handle,
HSAM_queue_info_str queue_info,
void* cookie,
int64_t* queue_handle_at_agent
)

Parameters

client_handle_at_agent
Opaque handle of the client object (specified by the agent driver on a client_new request).

signal_handle_at_agent
Opaque handle of the signal object (specified by the agent driver on a signal_new request) that is attached to this user mode queue.

queue_handle
Uniquely defines the queue across the HSA system.

queue_info
Contains the following information about the user mode queue:

- **queue_header_user_addr**. The address of the user mode queue's header in the user's virtual space. This value shall be returned to the client when the hsa_queue_create runtime call completes. (See HSA Runtime Programmer's Reference Manual Version 1.1, section 2.5.4.4 hsa_queue_create.)

- **payload_user_addr**. The address of the user mode queue's payload in the user's virtual space.

- **size**. The number of packets the user mode queue may hold. This value is passed by the client when hsa_queue_create call is invoked.

- **read_index_ptr_kern_addr**. The address of the queue's read index in the OS address space.

- **write_index_ptr_kern_addr**. The address of the queue's write index in the OS address space.

- **type**. Type of the queue. The value is passed by the client when hsa_queue_create call is invoked.

- **private_segment_size**. The maximum expected private segment usage per work-item, in bytes. The value is passed by the client when the hsa_queue_create runtime call is invoked.

- **group_segment_size**. The maximum expected group segment usage per work-group, in bytes. The value is passed by the client when the hsa_queue_create runtime call is invoked.

- **callbacks_required_flag**. Indicates that the client has requested callback events related to this user mode queue.

cookie
An arbitrary value that must be returned in the corresponding agent_reply call.

This is used by the generic HSA driver to match the corresponding request and reply messages. Note that the agent_reply message from the agent driver shall not occur if the request is handled synchronously.

queue_handle_at_agent
Pointer to opaque handle that must be set by the callee (agent driver) before the request is acknowledged.
This is used to identify the queue in all messages directed to a given agent driver. The agent driver should specify the appropriate value of the handle that would ease the task of object lookup. The handle may be left uninitialized if the request is rejected.

**Return values**

**HSAM.Done**
- The request is successfully completed synchronously.

**HSAM.Async**
- The request will be asynchronously completed with the `agent_reply` message.

**Otherwise**
- The request is rejected for the reason specified in the error code.

### 2.8.2 queue_remove

This message is sent when a client destroys a user mode queue.

**Direction**
- From the generic HSA driver to the agent driver

**Signature**

```c
HSAM_Error queue_remove
(
    int64_t client_handle_at_agent,
    int64_t queue_handle_at_agent,
    void* cookie
)
```

**Parameters**

- **client_handle_at_agent**
  - Opaque handle of the client that was specified by the agent driver when the client object is created.

- **queue_handle_at_agent**
  - Opaque handle of the queue object that was specified by the agent driver when the object is created.

- **cookie**
  - An arbitrary value that must be returned in the corresponding `agent_reply` call.

  This is used by the generic HSA driver to match the corresponding request and reply messages. Note that the `agent_reply` message from the agent driver shall not occur if the request is handled synchronously.

**Return values**

**HSAM.Done**
- The request is successfully completed synchronously.

**HSAM.Async**
- The request will be asynchronously completed with the `agent_reply` message.
2.9 Memory interface

The memory interface API allows the generic HSA driver to propagate memory-related runtime calls to agent drivers.

The memory interface handles the following events and actions of the clients:

- Allocation of HSA memory through the `hsa_memory_allocate` runtime call. (See HSA Runtime Programmer's Reference Manual Version 1.1, section 2.7.4.7 `hsa_memory_allocate`.)
- Registration of system memory through the `hsa_memory_register` runtime call. (See HSA Runtime Programmer's Reference Manual Version 1.1, section 2.7.4.11 `hsa_memory_register`.)
- Memory copy through the `hsa_memory_copy` runtime call. (See HSA Runtime Programmer's Reference Manual Version 1.1, section 2.7.4.9 `hsa_memory_copy`.)
- Memory assignment to HSA agents through the `hsa_memory_assign_agent` runtime call. (See HSA Runtime Programmer's Reference Manual Version 1.1, section 2.7.4.10 `hsa_memory_assign_agent`.)

2.9.1 Structures

The following structures are defined by the memory interface.

2.9.1.1 `enum hsam_segment`

This is the segment type used by the `memory_allocation` call.

**Signature**

```c
typedef enum
{
    hsam_segment_global,
    hsam_segment_read_only,
    hsam_segment_private,
    hsam_segment_group,
    hsam_segment_kern_arg,
    hsam_segment_system
} hsam_segment;
```

2.9.1.2 `enum hsam_global_flag`

These are flags relating to the global segment used in the `memory_allocation` call.

**Signature**

```c
typedef enum
{
    hsam_gf_kern_arg =1,
    hsam_gf_fine_grained =2,
    hsam_gf_coarse_grained =4,
} hsam_global_flag;
```

2.9.1.3 `enum hsam_access_permission`

These represent access permissions used in the `memory_acquire` call.

**Signature**

```c
typedef enum
{
    // Permissions
}
```
Signature

typedef enum {
    hsam_ap_ro, // read only
    hsam_ap_wo, // write only
    hsam_ap_rw, // read and write
} hsam_memory_access_permission;

2.9.2 memory_allocation

This message is sent when:

- A client creates an allocation through the `hsa_memory_allocate` runtime call. (See HSA Runtime Programmer’s Reference Manual Version 1.1, section 2.7.4.7 `hsa_memory_allocate`.) In the case of coarse-grained global segment allocation, the call may be delayed until the allocation is assigned to the agent for the first time.

- A client registers system memory through the `hsa_memory_register` runtime call. (See HSA Runtime Programmer’s Reference Manual Version 1.1, section 2.7.4.11 `hsa_memory_register`.)

Direction

From the generic HSA driver to the agent driver

Signature

HSAM_Error memory_allocation(
    int64_t client_handle_at_agent,
    void* virtual_address,
    int32_t size,
    hsam_segment segment,
    int32_t global_flag_mask,
    void* cookie
)

Parameters

client_handle_at_agent
    Opaque handle of the client object (specified by the agent driver on a `client_new` request).

virtual_address
    Virtual address of the allocation.

size
    Size of the allocation in bytes.

segment
    Segment type of the allocation.

global_flag_mask
    Flags of the global mask with the values of `enum hsam_global_flag`, if segment is `hsam_segment_global`.
    The argument is unused otherwise and any value provided will be ignored.

cookie
    An arbitrary value that must be returned in the corresponding `agent_reply` call.
This is used by the generic HSA driver to match the corresponding request and reply messages. Note that the **agent_reply** message from the agent driver shall not occur if the request is handled synchronously.

### Return values

**HSAM_Done**
- The request is successfully completed synchronously.

**HSAM_Async**
- The request will be asynchronously completed with the **agent_reply** message.

**Otherwise**
- The request is rejected for the reason specified in the error code.

#### 2.9.3 memory_free

This message is sent when:

- A client frees an allocation through the **hsa_memory_free** runtime call. (See *HSA Runtime Programmer’s Reference Manual Version 1.1*, section 2.7.4.8 **hsa_memory_free**.)

- A client deregisters system memory through the **hsa_memory_deregister** runtime call. (See *HSA Runtime Programmer’s Reference Manual Version 1.1*, section 2.7.4.12 **hsa_memory_deregister**.)

### Direction

From the generic HSA driver to the agent driver

### Signature

```c
HSAM_Error memory_free
(
    int64_t client_handle_at_agent,
    void* virtual_address,
    void* cookie
)
```

### Parameters

**client_handle_at_agent**
- Opaque handle of the client object (specified by the agent driver on a **client_new** request).

**virtual_address**
- Virtual address of the allocation.

**cookie**
- An arbitrary value that must be returned in the corresponding **agent_reply** call.

This is used by the generic HSA driver to match the corresponding request and reply messages. Note that the **agent_reply** message from the agent driver shall not occur if the request is handled synchronously.
Return values

HSAM_Done
The request is successfully completed synchronously.

HSAM_Async
The request will be asynchronously completed with the agent_reply message.

Otherwise
The request is rejected for the reason specified in the error code.

2.9.4 memory_acquire

This message is sent when a client assigns coarse-grained allocation through the hsa_memory_assign_agent runtime API. (See HSA Runtime Programmer's Reference Manual Version 1.1, section 2.7.4.10 hsa_memory_assign_agent.) The message is sent to an agent to which the allocation is assigned.

Direction
From the generic HSA driver to the agent driver

Signature

```c
HSAM_Error memory_acquire
(
    int64_t client_handle_at_agent,
    void* virtual_address,
    hsam_access_permission access_permission,
    void* cookie
)
```

Parameters

client_handle_at_agent
Opaque handle of the client object (specified by the agent driver on a client_new request).

virtual_address
Virtual address of the allocation.

access_permission
Access permission of the allocation on the agent.

cookie
An arbitrary value that must be returned in the corresponding agent_reply call.

This is used by the generic HSA driver to match the corresponding request and reply messages. Note that the agent_reply message from the agent driver shall not occur if the request is handled synchronously.

Return values

HSAM_Done
The request is successfully completed synchronously.

HSAM_Async
The request will be asynchronously completed with the agent_reply message.
Otherwise
   The request is rejected for the reason specified in the error code.

2.9.5 memory_release
This message is sent when a client assigns coarse-grained allocation through the `hsa_memory_assign_agent` runtime API. (See HSA Runtime Programmer’s Reference Manual Version 1.1, section 2.7.4.10 `hsa_memory_assign_agent`.) The message is sent to an agent that the allocation was assigned to, before the runtime call had been made.

Direction
From the generic HSA driver to the agent driver

Signature

```c
HSAM_Error memory_release
(
   int64_t client_handle_at_agent,
   void* virtual_address,
   void* cookie
)
```

Parameters

`client_handle_at_agent`
   Opaque handle of the client object (specified by the agent driver on a `client_new` request).

`virtual_address`
   Virtual address of the allocation.

`cookie`
   An arbitrary value that must be returned in the corresponding `agent_reply` call.

This is used by the generic HSA driver to match the corresponding request and reply messages. Note that the `agent_reply` message from the agent driver shall not occur if the request is handled synchronously.

Return values

- **HSAM_Done**
  The request is successfully completed synchronously.

- **HSAM_Async**
  The request will be asynchronously completed with the `agent_reply` message.

- Otherwise
  The request is rejected for the reason specified in the error code.

2.9.6 memory_copy
This message is sent when a client copies a block of memory through the `hsa_memory_copy` runtime API. (See HSA Runtime Programmer’s Reference Manual Version 1.1, section 2.7.4.9 `hsa_memory_copy`.)

Direction
From the generic HSA driver to the agent driver
Signature

```c
HSAM_Error memory_copy(
    int64_t client_handle_at_agent,
    void* dest,
    const void* src,
    int32_t size,
    void* cookie)
```

Parameters

- `client_handle_at_agent`: Opaque handle of the client object (specified by the agent driver on a `client_new` request).
- `dest`: Virtual address of the buffer where data is to be copied.
- `src`: Virtual address of data to be copied.
- `size`: Number of bytes to copy.
- `cookie`: An arbitrary value that must be returned in the corresponding `agent_reply` call.

This is used by the generic HSA driver to match the corresponding request and reply messages. Note that the `agent_reply` message from the agent driver shall not occur if the request is handled synchronously.

Return values

- `HSAM.Done`: The request is successfully completed synchronously.
- `HSAM.Async`: The request will be asynchronously completed with the `agent_reply` message.
- Otherwise: The request is rejected for the reason specified in the error code.

2.10 Agent memory interface

The agent memory interface provides a mechanism for the agent drivers to modify the virtual space of clients. Contrary to the memory interface in 2.9 Memory interface (on page 28), this API provides communication in the opposite direction: from agent drivers to the generic HSA driver. For example, an agent should use this API to allocate private, group, or kernarg segments before the execution of packets.

The provided API supports both shared and shadow page tables for implementing shared virtual space.

2.10.1 agent_memory_reserve

This reserves a range of contiguous virtual memory in a client's virtual space. Physical memory is not mapped at the range.

Direction
From the agent driver to the generic HSA driver

**Signature**

```
HSAM_Error agent_memory_reserve
(  int64_t client_handle,
    int32_t size,
    void** virtual_address
)
```

**Parameters**

- **client_handle**
  - Client ID.

- **size**
  - Size of the range in bytes to be reserved, rounded up to the size of a memory page.

- **virtual_address (out)**
  - On return, this contains the address of the reserved virtual space.

**Return values**

- **HSAM_Done**
  - The request is successfully completed synchronously.

- Otherwise (except HSAM_Async)
  - The request is rejected for the reason specified in the error code.

### 2.10.2 agent_memory_unreserve

This cancels the reservation of a range of contiguous virtual memory. The range must have been previously reserved through the `agent_memory_reserve` interface.

**Direction**

From the agent driver to the generic HSA driver

**Signature**

```
HSAM_Error agent_memory_unreserve
(  int64_t client_handle,
    void* virtual_address,
    int32_t size
)
```

**Parameters**

- **client_handle**
  - Client ID.

- **virtual_address**
  - Address of the reserved virtual space.
size
Size of the reserved virtual space in bytes, rounded up to the size of a memory page.

Return values
HSAM.Done
The request is successfully completed synchronously.

Otherwise (except HSAM.Async)
The request is rejected for the reason specified in the error code.

2.10.3 agent_memory_allocate
This allocates a range of contiguous virtual memory in a client's virtual space. System physical memory (of
the OS) is mapped at the range.

Direction
From the agent driver to the generic HSA driver

Signature

```
HSAM_Error agent_memory_allocate
(
    int64_t client_handle,
    int32_t size,
    void** virtual_address
)
```

Parameters

client_handle
Client ID.

size
Size of the range to be allocated in bytes, rounded up to the size of a memory page.

virtual_address (out)
On return, this contains the address of the allocated virtual memory.

Return Values

HSAM.Done
The request is successfully completed synchronously.

Otherwise (except HSAM.Async)
The request is rejected for the reason specified in the error code.

2.10.4 agent_memory_allocate_safe_to_access
This allocates a range of contiguous virtual memory in a client's virtual space and makes it safe-to-access.
System physical memory (of the OS) is mapped at the range.

Direction
From the agent driver to the generic HSA driver
Chapter 2. Component Interface  2.10 Agent memory interface

Signature

HSAM_Error agent_memory_allocate_safe_to_access
(  
    int64_t client_handle,  
    int32_t size,  
    void** virtual_address  
  )

Parameters

client_handle
  Client ID.

size
  Size of the range to be allocated in bytes, rounded up to the size of a memory page.

virtual_address (out)
  On return, this contains the address of the allocated virtual memory.

Return Values

HSAM.Done
  The request is successfully completed synchronously.

Otherwise (except HSAM.Async)
  The request is rejected for the reason specified in the error code.

2.10.5 agent_memory_free

This frees virtual memory that had been previously allocated through either the agent_memory_allocate or agent_memory_allocate_safe_to_access interface.

Direction

From the agent driver to the generic HSA driver

Signature

HSAM_Error agent_memory_free
(  
    int64_t client_handle,  
    void* virtual_address,  
    int32_t size  
  )

Parameters

client_handle
  Client ID.

virtual_address
  Address of the virtual space to be freed.

size
  Size of the allocated virtual space in bytes, rounded up to the size of a memory page.
Return values

HSAM_Done
The request is successfully completed synchronously.

Otherwise (except HSAM_Async)
The request is rejected for the reason specified in the error code.

2.10.6 agent_memory_map

This allocates a range of contiguous virtual memory in a client's virtual space and maps the physical memory provided through the physical_address argument.

Direction
From the agent driver to the generic HSA driver

Signature

```c
HSAM_Error agent_memory_map(
    int64_t client_handle,
    int32_t size,
    void* physical_address,
    void** virtual_address
)
```

Parameters

client_handle
Client ID.

size
Size of the range to be allocated in bytes, rounded up to the size of a memory page.

physical_address
Address of physical memory to be mapped in the allocated virtual space.

virtual_address (out)
On return, this contains the address of the allocated virtual memory.

Return Values

HSAM_Done
The request is successfully completed synchronously.

Otherwise (except HSAM_Async)
The request is rejected for the reason specified in the error code.

2.10.7 agent_memory_unmap

This unmaps virtual memory that had been previously mapped through the agent_memory_map interface.

Direction
From the agent driver to the generic HSA driver
2.10 Agent memory interface

Signature

```c
HSAM_Error agent_memory_unmap(
    int64_t client_handle,
    void* virtual_address,
    int32_t size
)
```

Parameters

client_handle
   Client ID.

virtual_address
   Address of the virtual memory to be unmapped.

size
   Size of the virtual space to be unmapped in bytes, rounded up to the size of a memory page.

Return values

HSAM_Done
   The request is successfully completed synchronously.

Otherwise (except HSAM_Async)
   The request is rejected for the reason specified in the error code.

2.10.8 agent_memory_allocate_physical

This allocates contiguous physical memory from the system physical memory pool (of the OS).

Direction

From the agent driver to the generic HSA driver

Signature

```c
HSAM_Error agent_memory_allocate_physical(
    int64_t client_handle,
    int32_t size,
    void** physical_address
)
```

Parameters

client_handle
   Client ID.

size
   Size of physical memory to be allocated in bytes, rounded up to the size of a memory page.

physical_address (out)
   On return, this contains the address of the allocated physical memory.
Return values

HSAM_Done
   The request is successfully completed synchronously.

Otherwise (except HSAM_Async)
   The request is rejected for the reason specified in the error code.

2.10.9 agent_memory_free_physical

This returns contiguous physical memory to the system physical memory pool (of the OS).

Direction

From the agent driver to the generic HSA driver

Signature

```c
HSAM_Error agent_memory_free_physical
    (int64_t client_handle,
     void* physical_address
     int32_t size)
```

Parameters

client_handle
   Client ID.

physical_address
   Physical address of the memory to be freed.

size
   Size of physical memory in bytes to be freed, rounded up to the size of a memory page.

Return values

HSAM_Done
   The request is successfully completed synchronously.

Otherwise (except HSAM_Async)
   The request is rejected for the reason specified in the error code.

2.11 Miscellaneous interface

2.11.1 agent_reply

This is an asynchronous reply to a request from the generic HSA driver.

Any request from the generic HSA driver can be acknowledged by an agent driver either synchronously or asynchronously. The agent driver sends **agent_reply** to asynchronously acknowledge request. See 2.2 Calling convention (on page 13) for details.

Direction

From the agent driver to the generic HSA driver
2.11 Miscellaneous interface

**Signature**

```c
void agent_reply(
    HSAM_Error reply,
    void* cookie
)
```

**Parameters**

- **reply**
  - Any error code other than `HSAM_Async` or `HSAM.Done` indicates that the request has succeeded.

- **cookie**
  - Cookie value from the corresponding request from the generic HSA driver. This value is used by the generic HSA driver to match the request and reply message. See [2.2 Calling convention (on page 13)](#) for details.

**Return value**

`void`

### 2.11.2 launch_dll

This is a request from a vendor’s agent driver to launch a dynamic linked library (DLL) in the client's virtual space.

For example, an agent that implements a driver-based packet processor may want to execute a part of its processor in user virtual space. This API allows the HSA agent to launch the packet processor's DLL in virtual space of given client.

**Direction**

From the agent driver to the generic HSA driver

**Signature**

```c
void launch_dll(
    int64_t client_handle,
    char* dll_file,
)
```

**Parameters**

- **client_handle**
  - Identifies the client in which to launch the DLL.

- **dll_file**
  - File name of the dynamic linked library (DLL).

**Return Values**

`void`
CHAPTER 3.
Signals

3.1 Overview

This chapter describes multi-vendor signal requirements. These requirements extend the foundation provided by the HSA Platform System Architecture Specification Version 1.1 and the HSA Runtime Programmer’s Reference Manual Version 1.1 and should be read in conjunction with those documents.

Multi-vendor signals rely upon:

- HSA agent implementations sharing a common set of memory object definitions
- Standardization of the signal send process
- A set of requirements that must be met during the implementation of signal wait processes.

The key areas extended by this specification to provide for multi-vendor signals consist of:

- Extended definition of the signal object
- Definition of new object structures and their use:
  - Signal group object
  - Root destination node
  - Destination node
  - Group node
  - Signal group destination
- Extension of signal end requirements
- Extension of signal wait requirements
- Definition of related component interface APIs

Vendors who meet these requirements shall provide implementations that are compatible in a multi-vendor HSA system.

3.1.1 Multi-vendor considerations

This specification extends the definitions of HSA objects and APIs to allow a range of compatible hardware solutions that support HSA signals to be developed independently. The scalability and flexibility of multi-vendor systems that meet the requirements of this specification were important considerations during its development.

An important consideration for multi-vendor signals was the ability to produce event-driven notification methods. The following notification events are supported:

- Cache-line monitor
- Hardware register write
- Hardware FIFO buffer write
- Component interface APIs

HSA components waiting on a signal independently control their method selected for signal notification, which may be updated dynamically when following the rules in this specification.

It is a requirement of multi-vendor signals that components are provided with two classes of wait states with which they may implement their signal wait solutions:

- Active WAIT: An operating state where signal value changes are communicated to the receiving agent via signal value updates and any associated doorbells the agent has provided.
- Inactive WAIT: An operating state where the receiving agent requires the use of component interface APIs to communicate signal value events and provokes the agent to test the signal.

### 3.1.2 Multi-vendor signal strategy

The requirements for multi-vendor signals standardize the set of signal methods and introduce additional structures to the definition of the signal object for controlling these methods. The extensions of the signal object are described in section 3.2 Signal object (on the facing page), allowing all potential destinations of a signal in a multi-vendor system the option to register information for use during the signal send procedure.

The information provided in an HSA_root_dest_node struct is defined in section 3.4 Root destination node (on page 45). The signal object registers provide root node content for each destination that may be a recipient of the signal.

The hsa_root_dest_node struct allows the destination to create a further hierarchy of information using destination nodes (see 3.5 Destination node (on page 46)) and group nodes (see 3.6 Group node (on page 47)). The node hierarchy is used to direct the operations performed by an agent during the send of the signal to which the root node is registered.

**Figure 3-1 (on the facing page)** shows an example of a node hierarchy for the signal named SignalA. Two destinations are possible receivers of the signal in this example:

- Destination A has provided a root destination node (RootA) with no further hierarchy.
- Destination B has provided a root destination node (RootB) with one further level of destination node hierarchy.

Detailed definitions for these node structures are contained in section 3.4 Root destination node (on page 45), section 3.5 Destination node (on page 46), and section 3.6 Group node (on page 47). The use of these structures in node hierarchies is described in further detail in section 3.8 Destination nodes (on page 48).
3.2 Signal object

The signal object specification is extended here to define common structures required for multi-vendor interoperability.

The memory attributes related to signal objects are detailed in section 3.10 Object locations and memory attributes (on page 59).

3.2.1 hsa_signal_object

This is the signal object structure shared across all HSA components within the system.

Signature

```c
struct hsa_signal_object {
    // reference to signal value location
    uint64_t* signal_value_ptr;

    // Root destination node array
    uint32_t number_of_dests;
    struct hsa_root_dest_node root_dest_array[number_of_dests];
};
```

Data fields

- **signal_value_ptr**
  Virtual address location of the signal value.

- **number_of_dests**
  Number of HSA agents that may receive the signal.
3.3 Signal group object

The memory attributes related to signal group objects are detailed in section 3.10 Object locations and memory attributes (on page 59).

3.3.1 hsa_signal_group_object

This is the signal group object structure shared across all HSA components within the system.

Signal group objects are created and maintained by the generic HSA driver.

Signature

```c
struct hsa_signal_group_object {
    uint32_t number_of_signals;
    uint32_t number_of_dests;
    struct hsa_signal_object* signals[number_of_signals];
    struct hsa_signal_group_destination signal_destinations[number_of_dests];
}
```

Data fields

number_of_signals
  Number of signals contained in the group.

number_of_dests
  Number of destinations for the group.

signals
  Array of signals that comprise the group.

signal_destinations
  An hsa_signal_group_object array. The array size is set to the number of HSA agents that may wait on the signal group.
3.4 Root destination node

The root destination node is used by an HSA agent during the signal send process, as defined in section 3.11.3 hsa_signal_send (on page 67). This object is created, and its content controlled, by the destination agent.

Only HSA agents that may receive a signal are permitted to register a root destination node to that signal, and these HSA agents are required to register a root destination node during the creation of the signal object.

Root destination nodes owned by the HSA agent may be updated dynamically. For example, the state variable may be updated to indicate that a destination is no longer active. Dynamic updates to the content of the root destination node in the signal object must follow the requirements described in section 3.9.1 Dynamic signal object updates (on page 54).

The memory attributes related to root destination nodes are detailed in section 3.10 Object locations and memory attributes (on page 59).

3.4.1 hsa_root_dest_node

This is the root destination node object structure that is shared across all HSA components within the system.

**Signature**

```c
struct hsa_root_dest_node {
    uint64_t flags {
        bit[0]  state; // destination active state indication
        bit[4-1] barrier_type;
        bit[15-5] reserved;
        bit[31-16] device_driver_id;
        bit[63-32] doorbell_value;
    }
    void* internal_data_ptr;
    struct hsa_destination_node* next_dest_hier;
    struct hsa_group_node* next_group_hier;
}
```

**Data fields**

*flags.state*

Destination state value attributed to this level of the destination node hierarchy.

- 0 : ACTIVE (default value)
- 1 : INACTIVE

*flags.barrier_type*

Barrier type encoding.

barrier_type[0] classifies the barrier class:

- 0 : MEMORY
- 1 : PERIPHERAL
barrier_type[3-1] are architecture specific.

flags.device_driver_id
Unique device driver ID, as assigned to the destination agent by the HSA runtime.

flags.doorbell_value
32-bit value that will be written to all doorbell addresses in the destination and group node hierarchies attached from this root node.

internal_data_ptr
Virtual address of the location reserved for the destination agent's use (null is the default value).

next_dest_hier
Virtual address location of the next hsa_destination_node in the hierarchy; the list is terminated with null (null is the default value).

next_group_hier
Virtual address location of the next hsa_group_node in the hierarchy; the list is terminated with null (null is the default value).

3.5 Destination node

The destination node is used by an HSA agent during the signal send process as defined in section 3.11.3 hsa_signal_send (on page 67). This object is created, and its content controlled, by the destination agent.

Destination nodes owned by the HSA component may be updated dynamically. For example, the state variable may be updated to indicate that a destination is no longer active. Dynamic updates to the content of the destination node must follow the requirements described in section 3.9.2 Dynamic destination node updates (on page 56).

The memory attributes related to destination nodes are detailed in section 3.10 Object locations and memory attributes (on page 59).

3.5.1 hsa_destination_node

This is the destination node object structure that is shared across all HSA components within the system.

Signature

```c
struct hsa_destination_node {
    uint64_t flags 
    { 
        bit[0] state; // destination active state indication 
        bit[4-1] barrier_type; 
    }
    void* internal_data_ptr;
    uint64_t* doorbell_address;
    struct hsa_destination_node* next_dest_hier;
};
```

Data fields

flags.barrier_type
Barrier type encoding.
barrier_type[0] classifies the barrier class:

0 : MEMORY
1 : PERIPHERAL

barrier_type[3-1] are architecture specific.

flags.state
Destination state value attributed to this level of the destination node hierarchy.

0 : ACTIVE
1 : INACTIVE

internal_data_ptr
Virtual address reserved for the destination agent's use.

doorbell_address
Virtual address of the destination node's doorbell.

next_dest_hier
Virtual address location of the next hsa_destination_node in the hierarchy; the list is terminated with null.

3.6 Group node

The group node is used by an HSA agent during the signal send process as defined in section 3.11.3 hsa_signal_send (on page 67). This object is created, and its content controlled, by the destination agent.

The memory attributes related to destination nodes are detailed in section 3.10 Object locations and memory attributes (on page 59).

3.6.1 hsa_group_node

This is the group node object structure that is shared across all HSA components within the system.

Signature

```c
struct hsa_group_node {
    uint64_t flags {
        bit[4-1] barrier_type;
    }
    void* internal_data_ptr;
    uint64_t* doorbell_address;
    bit* state;
    struct hsa_group_node* next_group_hier;
};
```

Data fields

flags.barrier_type
Barrier type encoding.

barrier_type[0] classifies the barrier class:
0 : MEMORY
1 : PERIPHERAL

barrier_type[3-1] are architecture specific.

internal_data_ptr
Virtual address of the location reserved for the destination agent's use.

doorbell_address
Virtual address of the group node's doorbell.

state
Virtual address of the state value of the group.

next_group_hier
Virtual address location of the next hsa_group_node in the hierarchy; the list is terminated with null.

3.7 Signal group destination

The memory attributes related to signal group destination objects are detailed in section 3.10 Object locations and memory attributes (on page 59).

3.7.1 hsa_signal_group_destination

This is the signal group destination object structure that is shared across all HSA components within the system.

Signature

```
struct hsa_signal_group_destination {
    int device_driver_id;
    void* internal_data_ptr;
}
```

Data fields

device_driver_id
Unique device driver ID, as assigned to the destination agent by the HSA runtime.

internal_data_ptr
Virtual address reserved for the destination agent's use.

3.8 Destination nodes

Registering a destination node to a signal allows a destination agent to impart additional effects within the signal send process beyond the update of the signal value itself.

The destination node provides a device_driver_id for the destination (assigned by the generic HSA driver on signal creation), as well as the active state of the destination, and allows a destination node and group node to be referenced. The root destination node is located within the hsa_root_dest_node array of the signal object; each agent driver is informed of its location within this array at signal creation.

A destination node contains a set of flags and a doorbell address, which can be used to further influence the operation of the signal send process. An hsa_destination_node reference is also included to allow expansion of the node hierarchy.
A group node contains a set of flags, the doorbell address, and the state value address, which can be used to further influence the operation of the signal send process. An `hsa_group_node` reference is included to allow expansion of the node hierarchy.

This section introduces the support provided by the root and destination node references using examples of simple hierarchies that may be produced by implementations. The examples show signals with a single destination agent; this simplification is negligible in that each destination has its own destination root. The node hierarchies described here are certainly not the only node structures that could benefit from an HSA agent; however, the same concepts would apply to other structures.

NOTE: The `internal_data_ptr` reference in the destination node structure also allows the destination agent to link internal data structures that it may require for its own destination node manipulation routines. The definition and use of such internal data is not a part of this specification.

### 3.8.1 Root destination node

The root destination node in a signal allows a destination agent to cause additional effects within the signal send process beyond the update of the signal value itself. The root destination node provides the `device_driver_id` of the destination used to:

- Identify the agent to the generic HSA driver as well as the active state of the destination
- Allow a destination node to be referenced.

This section describes the support provided by the root and destination node references and the state value within the flags.

To introduce some of the benefits of using a hierarchy, a system shall be described that does not use additional node hierarchy beyond the root node. The example hierarchy in Figure 3–2 (below) shows a signal, S1, with root node values within its root destination array as shown here by `S1_RootNode`.

![Figure 3–2 Root destination node example (without additional hierarchy)](image)

When the signal S1 has its signal send process executed, the root node shall be parsed as described in section 3.11.3 `hsa_signal_send` (on page 67). This shall perform a test of the destination state flag value that controls the requirement to call the relevant component interface API functions as defined.
The state flag in the signal object allows the destination agent to maintain an independent thread execution model, whereby its threads waiting on different signals may do so as active and inactive independently. For example, a thread that is waiting on S1 is moved to an inactive state, while a different thread waiting on a different signal may retain an active state.

3.8.2 Destination node

Figure 3–3 (below) introduces a single layer of node hierarchy, independently created for two signals: S1 and S2. Destination nodes S1_Node and S2_Node are each referenced respectively from root destination nodes S1_RootNode and S2_RootNode.

Adding the destination node to the signal structure allows the destination agent to use the doorbell_address and doorbell_value support offered during the signal send process. The agent now also has multiple state flags available in the node hierarchies.

An HSA agent may choose to implement functionality that is stimulated by the doorbell operation performed during the signal send rather than being stimulated by the change in signal value.

![Figure 3–3 Root destination node example (with single hierarchy)](image)

An implementation could consider using a common level of node hierarchy as shown in Figure 3–4 (on the facing page) by introducing a common destination node, CommonNode. This common node is set as the hsa_destination_node referenced by the next_dest_hier pointer values within nodes S1_Node and S2_Node.

This structure allows the receiving HSA agent to specify independent doorbells for both S1 and S2 signals – within S1_Node and S2_Node – and further allows common doorbell and state flags to be provided within CommonNode.

This approach may be of benefit by allowing the destination to choose to be stimulated by the doorbell within the CommonNode only (reducing the active doorbells it is required to monitor). The additional doorbells within S1_Node and S2_Node in the example may optionally be used to determine more specifically which signals have been stimulated.
3.8.3 Group node

This section describes the group node hierarchies that a destination agent may choose to create.

The root destination node defined here includes the next_group_hier pointer. This is a pointer to a group node, and is provided for the use of destination agents in support of wait-on-signal group operations. The use of the information in the group node hierarchy is provided in section 3.11.3 hsa_signal_send (on page 67).

By using doorbells to support its implementation of signal detection, an HSA agent has a mechanism for monitoring a single doorbell operated when any signal within the signal group is sent.

The use of these structures is available for HSA agents that may receive a signal within a signal group and is provided in addition to the destination node hierarchy support described in section 3.8 Destination nodes (on page 48). Using these structures for the receiving agent is optional; however, a signal send process must perform all required functionality related to these structures as described in section 3.11.3 hsa_signal_send (on page 67). See section 3.11.2 hsa_signal_group_wait (on page 64) for additional information about the wait procedure for an agent using this mechanism.
Figure 3-5 (below) shows a signal group that contains two signals, S1 and S2, and a node structure that a destination agent has created in support of its signaling implementation. In this example, the two signals are considered as Group_A, and the destination has created a group node for each signal, each of which is attached to the hierarchy of their signal. A group node for each signal is required to allow independent node hierarchies to be maintained.

The group node provides a pointer to both the doorbell address and the state of the signal group. This allows the agent driver of the destination to maintain a single location to manage the values for the signal group.

In this example, when either signal is sent, the root node's doorbell value shall be written to the doorbell address of Group_A, allowing the destination node a common notification method spanning both signals. Where a destination moves a group wait to an inactive state, this may also be indicated to all signals in the group using the shared state value to which they all refer.

A destination agent may receive multiple concurrent wait operations covering different signal groups. Each group shall maintain its own group nodes, attached to each signal in their respective groups, referencing the doorbell and state values of the group. Such a structure is shown in Figure 3-6 (on the facing page).

In this example, two signal groups have been created. Group A contains signals S1 and S2, and Group B contains signals S2 and S3. This provides unique doorbell and state values for each group.
Updates to root destination nodes and group nodes for signal groups must follow the general destination node requirements in section 3.9.1 Dynamic signal object updates (on the next page). The lifetime of the group nodes is defined in section 3.9.4 Lifetime of node objects and doorbell locations (on page 59).

Figure 3–6 Signal group node example (with three signals and two signal groups)

3.9 Dynamic updates to signal object and nodes

Following the creation of a signal, an agent driver of an agent that may receive this signal may dynamically alter its root destination node within the signal object. It may also alter its destination nodes and group nodes within its node hierarchies.

The most common reasons for changing the content of the signal object, root node, destination node, or group node are:

- Signal creation
- Signal destruction
- Destination agent changing between active and inactive states
- Destination execution thread that is waiting on a signal changing between active and inactive states
- Signal group creation
- Signal group removal
- Destination execution thread that is waiting on a signal group changing between active and inactive states
- A user mode queue is added or removed, where the queue includes a signal containing a node hierarchy
These updates must be performed using the requirements in the following sub-sections.

### 3.9.1 Dynamic signal object updates

The signal object contains fields that agent drivers (of agents that may receive the signal) can change dynamically. This section describes the requirements of the agent drivers that are manipulating the signal object.

The fields in the signal object that may be changed by an agent driver allocated to node_index are limited only to this root node content:

\[ \text{root\_dest\_array[node\_index]} \]

No other locations may be altered by the agent driver.

The following rules apply to updates:

- Dynamic signal object updates must modify only fields related to agent driver.
- Dynamic signal object updates must be performed within a critical section of code.
- For the root node:
  - `device_driver_id` is a static value, as assigned on signal creation.
  - The observed barrier type must always be compatible for the observed doorbell address.
  - The destination node pointer must be null or point to a live destination node (see 3.9.2 Dynamic destination node updates (on page 56)).
  - The group node pointer must be null or point to a supported group node (see 3.9.3 Dynamic group node updates (on page 58)).
  - Updates to fields must not allow intermediate values to be visible. This is especially important for:
    - Destination node pointer
    - Group node pointer
  - When a group node is removed from the hierarchy, the downstream nodes must be processed for doorbell and state operations.
  - Active state cannot be visible when the destination is inactive. Inactive state may be visible when the destination is active.

- After dynamic signal object updates are complete, synchronization operations are required to guarantee visibility of the new values to all relevant HSA agents before this visibility is relied upon.

Example implementations are shown of agent driver functions that set the destination agent's root node state to:

- **INACTIVE** (Figure 3–7 (on the facing page))
- **ACTIVE** (Figure 3–8 (on the facing page))
Figure 3–7 Example of dynamic signal object update for destination moving to INACTIVE state within root node

```c
void set_root_dest_inactive(hsa_signal_object* signal) {
    struct hsa_root_dest_node* root_node;
    root_node = get_my_root_node(signal);
    if (root_node->flags.state == INACTIVE)
        return;
    mutex_lock(&hsa_signal_manager_lock);
    root_node->flags.state = INACTIVE;
    barrier((unsigned int)root_node->flags.barrier_type);
    mutex_unlock(&hsa_signal_manager_lock);
    return;
}
```

Figure 3–8 Example of dynamic signal object update for destination moving to ACTIVE state within root node

```c
void set_root_dest_active(hsa_signal_object* signal) {
    struct hsa_root_dest_node* root_node;
    root_node = get_my_root_node(signal);
    if (root_node->flags.state == ACTIVE)
        return;
    mutex_lock(&hsa_signal_manager_lock);
    barrier((unsigned int)root_node->flags.barrier_type);
    root_node->flags.state = ACTIVE;
    mutex_unlock(&hsa_signal_manager_lock);
    return;
}
```

Figure 3–9 (below) shows an example procedure for swapping in a new destination node to the root node. This process accesses the root node for the destination agent and stores the previous node pointer. If the new node matches the previous node, the function returns; otherwise, the signal manager lock for the agent is taken and the new node pointer is set to the root node.

```c
void set_dest_node_in_root(hsa_signal_object* signal,
                           hsa_destination_node* new_dest_node) {
    struct hsa_root_dest_node* root_node;
    struct hsa_destination_node* prev_dest_node;
    root_node = get_my_root_node(signal);
    prev_dest_node = root_node->next_dest_hier;
    if (prev_dest_node == new_dest_node)
        return;
    mutex_lock(&hsa_signal_manager_lock);
    root_node->next_dest_hier = new_dest_node;
    barrier((unsigned int)root_node->flags.barrier_type);
    mutex_unlock(&hsa_signal_manager_lock);
    return;
}
```
3.9.2 Dynamic destination node updates

When an agent driver dynamically changes the content of its own destination node structures, the following rules apply:

- An agent driver may change only the destination nodes it owns.

- Implementations that utilize state indication within an active destination node must ensure that:
  - Active state must never be visible when the destination is inactive
  - Active state cannot be visible when the destination is inactive. Inactive state may be visible when the destination is active.

- For destination nodes that are active:
  - The destination node pointer must be null or point to a live destination node
  - Updates to fields must not allow intermediate values to be visible. This is especially important for:
    - Destination node pointer
    - Doorbell address
    - Doorbell location type must indicate a compatible value for the doorbell address.

- After dynamic updates are complete, synchronization operations are required to guarantee visibility of the new values to all relevant HSA agents before this visibility is relied upon.

A live node is a destination node observable as referenced within any node hierarchy of any signal.

Pseudo code examples are provided in Figure 3–10 (on the facing page) to show the requirements for the agent driver manipulation of these nodes for both state value changes and doorbell changes.

Figure 3–10 (on the facing page) provides an implementation example for updating the state of the destination agent within its node hierarchy. This is performed in a critical section of code and takes the agent driver’s signal manager lock. In this example, it is assumed that the agent is using at least one level of destination node.

The method for determining the root_node, and the function get_my_dest_node that returns the dest_node that belongs to the destination for this signal may be performed more directly using the destination’s own internal data structures. The state is then updated within the destination node and a fence performed to ensure visibility prior to releasing the lock.
Exchanging destination nodes is a way to alter the content of the destination node hierarchies that are active. Figure 3–11 (below) shows an example process within a critical region for exchanging references of dest_node_old for dest_node_new. This may be desired if an agent is required to alter multiple data fields within the node structure.

Figure 3–12 (on the next page) shows a diagram of a node hierarchy to which the code example of Figure 3–11 (above) might be applied. In this diagram, CommonNodeA may be considered the dest_node_old input parameter to the function, and CommonNodeB the dest_node_new.

The implementation chosen by the agent has listed all nodes that reference CommonNodeA within its internal data structure referenced within the node; an alternative implementation may choose to retain this information within its own internal data structure. This function shall update the references contained within S1_RootNode and S2_RootNode to refer to CommonNodeB. A fence operation is performed prior to completion of the critical code region.
After destination nodes are exchanged, the agent driver shall detect if any send operation to this signal has been performed between the start of this exchange process and its completion. Any detection of a signal send shall be handled by the HSA agent as required.

3.9.3 Dynamic group node updates

An agent driver that dynamically changes the content of its own group node structures must adhere to the following rules:

- An agent driver may change only the group nodes it owns.
- Implementations that utilize state indication within an active group node must ensure that:
  - Active state cannot be visible when the destination is inactive. Inactive state may be visible when then destination is active.
- For group nodes that are active:
  - The group node pointer must be null or point to a live group node
  - Updates to fields must not allow intermediate values to be visible. This is especially important for:
    - Group node pointer
    - Doorbell address
    - Doorbell location type must indicate compatible values for doorbell and state address
3.9.4 Lifetime of node objects and doorbell locations

Root destination nodes are initialized during signal creation and are contained within the signal object. The following rules apply:

- Any destination node used by a signal at any time during its life must remain accessible until the signal is removed.
- If a destination node is used in multiple signals, it must remain accessible until all signals that have used it are removed. The agent driver is required to track which live signals have used each node to determine when it becomes safe to free a node.
- Destination nodes may be reused, with the implication that while previous signals that used them remain live, there may be spurious doorbell and unblock API operations received. This must not affect correct operation of an implementation's signal support.
- Any group node used by a signal at any time during its life must remain accessible until the signal is removed.
- Group nodes may be reused, with the implication that while previous signals that used them remain live, there may be spurious doorbell and unblock API operations received. This must not affect correct operation of an implementation's signal support.
- Doorbell locations that are used within any live destination node or group node are required to remain safe-to-access. This includes doorbell locations that may no longer be referenced within any node, having been replaced by a dynamic update of the destination node field; such doorbell locations must allow correct operation if operated spuriously.
- State locations that are used within any live group node are required to remain safe-to-access. This includes state locations that may no longer be referenced within any node, having been replaced by a dynamic update of the group node field; such state locations must allow correct operation if accessed spuriously.

3.10 Object locations and memory attributes

This section describes the requirements on the memory location of the following objects, structures, and pointer targets:

- Signal object
  - Signal value pointer target
- Group signal object
- Root destination node
  - Internal data pointer target
• Destination node
  ○ Internal data pointer target
  ○ Doorbell pointer target
• Group node
  ○ Internal data pointer target
  ○ Doorbell pointer target
  ○ State pointer target
• Signal group destination
  ○ Internal data pointer target

3.10.1 HSA base profile

The following must be located within shared virtual memory allocated at system runtime:

• Signal object
• Group signal object
• Destination node
• Group node
• Signal group destination
• value address

The following may be located in any memory type selected, and accessible, by the HSA agent responsible for the destination node:

• Target of internal_data_ptr within the signal group destination
• Target of internal_data_ptr within the destination node
• Target of internal_data_ptr within the group node

3.10.2 HSA full profile

The following must be located within shared virtual memory:

• Signal object
• Group signal object
• Destination node
• Group node
• Signal group destination
• Signal value address

The following may be located in any memory type selected, and accessible, by the HSA agent responsible for the destination node:
• Target of *internal_data_ptr* within the signal group destination
• Target of *internal_data_ptr* within the destination node
• Target of *internal_data_ptr* within the group node

### 3.10.3 Attributes

Table 3–1 (below) lists attribute requirements.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td>Target of signal value pointer</td>
<td>• Located in read-write space for kernel, kernel-driver, and user</td>
</tr>
<tr>
<td></td>
<td>• Safe-to-access</td>
</tr>
<tr>
<td>Signal objects</td>
<td>• Kernel read-write</td>
</tr>
<tr>
<td>Signal group objects</td>
<td>• Read-only for user</td>
</tr>
<tr>
<td>Signal group destination objects</td>
<td>• Safe-to-access</td>
</tr>
<tr>
<td>Root destination nodes</td>
<td></td>
</tr>
<tr>
<td>Destination nodes</td>
<td></td>
</tr>
<tr>
<td>Group nodes</td>
<td></td>
</tr>
<tr>
<td>Location of <em>internal_data_ptr</em> within</td>
<td>• May be restricted to kernel-driver read-write only</td>
</tr>
<tr>
<td>destination and group nodes</td>
<td>• User must assume no access rights to these locations</td>
</tr>
<tr>
<td></td>
<td>• Are not required to be safe-to-access</td>
</tr>
<tr>
<td>Doorbell address locations within destination</td>
<td>• Read-write for user</td>
</tr>
<tr>
<td>and group nodes</td>
<td>• Safe-to-access</td>
</tr>
<tr>
<td>State locations within group nodes</td>
<td></td>
</tr>
</tbody>
</table>

### 3.11 Signal APIs

All HSA agents that participate in a multi-vendor system must follow the standardized process objectives defined in this section.

Pseudo code descriptions that show acceptable implementations are provided, plus reference to examples that support agent driver functions. These supporting functions are also described in section 3.13 Internal agent driver functions (on page 70). An implementation is not required to replicate the pseudo code examples, but it is required to meet all requirements.

#### 3.11.1 hsa_signal_wait

The required operation of the signal wait is presented as two separate definitions: active and inactive. These are both accessible through the same *hsa_signal_wait* method and are logically a part of the same function.

Signal wait allows significant implementation-dependent functionality.

**Requirements**

A device that is waiting on a signal is required to:
• Detect a signal send if no further send operations to the signal are performed
• If using the state feature:
  o Never miss a doorbell (or signal change) in a race
• If using the doorbell feature:
  o Never miss a doorbell in a race

A device waiting on a signal is not required to see all intermediate values of this signal if multiple send operations are performed on the signal.

The wait function may return without satisfying the wait condition.

**Description**

Active wait is an HSA agent waiting on a signal being inherently reactive to either changes in the signal value or operation of its doorbells (as registered within any node hierarchy the destination agent has constructed within the signal object). An HSA agent may perform its signal wait functions in this class of wait only.

Pseudo code describing the active wait functionality is shown in Figure 3-13 (below). In this example, the wait_expectancy input field has not been used. This parameter may be used to adapt the method employed for detecting or reacting to changes in the signal value.

In the example, additional functions are used to handle each of the wait conditions. Section 3.13 Internal agent driver functions (on page 70) provides a brief description of the intent for these functions.

**Figure 3-13** Pseudo code definition of active signal wait

```c
hsa_signal_wait(hsa_signal_object* signal,
    hsa_wait_expectancy_t wait_expectancy,
    hsa_signal_condition_t condition,
    hsa_signal_value_t compare_value) {
    return wait_on_condition(signal, condition, compare_value);
}
```

As shown for one of the wait conditions in Figure 3-14 (below), an HSA agent may provide wait implementations that can be supported entirely within user privilege level.

**Figure 3-14** Pseudo code definition of wait_on_condition contained within user privilege

```c
wait_on_condition(hsa_signal_object* signal,
    hsa_signal_condition_t condition,
    hsa_signal_value_t compare_value) {
    int result = 0;
    while (result == 0) {
        tmp_reg = READ[signal.signal_value_ptr];
        tmp_reg = READ_EXCHANGE[tmp_reg];
        case (condition)
            HSA_EQ: if (tmp_reg == compare_value) result = 1;
            HSA_NE: if (tmp_reg != compare_value) result = 1;
            HSA_LT: if (tmp_reg < compare_value) result = 1;
            HSA_GTE: if (tmp_reg >= compare_value) result = 1;
        }
    return;
}
```
Alternately, for implementations that utilize the root destination node hierarchies, agent driver kernel privilege interaction may be included.

Figure 3–15 (below) shows a theoretical implementation that uses the agent driver calls `clear_signal_doorbell` and `wait_on_signal_doorbell`. Following an initial compare of the signal value, the user code calls into its agent driver. This allows the agent driver to utilize its privilege level and the additional signal behaviors provided in this specification, such as doorbells.

```
wait_on_condition(hsa_signal_object* signal,
    hsa_signal_condition_t condition,
    hsa_signal_value_t compare_value) {
  int result = 0;
  while (result == 0) {
    clear_signal_doorbell(signal);
    tmp_reg = READ[signal.signal_value_ptr];
    tmp_reg = READ_EXCHANGE[tmp_reg];
    case (condition)
      HSA_EQ: if (tmp_reg == compare_value) result = 1;
      HSA_NE: if (tmp_reg != compare_value) result = 1;
      HSA_LT: if (tmp_reg < compare_value) result = 1;
      HSA_GTE: if (tmp_reg >= compare_value) result = 1;
    wait_on_signal_doorbell(signal);
  }
  return;
}
```

Inactive wait is an HSA agent waiting on a signal being unresponsive to changes in signal value or operation of its doorbells (as registered in any node hierarchy the destination agent has constructed in the signal object).

Inactive wait requires interaction with the agent driver and uses the related component interface API calls.

An HSA agent may provide different implementations for wait, and it is expected that a common differentiation of the implementation performed would be based on the value of the wait_expectancy provided.

Figure 3–16 (on the next page) shows an example of pseudo code for an implementation introducing an inactive signal wait. In this figure, the inactive wait is performed when the wait_expectancy is other than `HSA_WAIT_EXPECTANCY_SHORT`.

An initial test of the signal value is performed prior to any agent driver interaction. If this does not meet the requested condition, the `yield_destination` agent driver call is made. This call allows the agent driver to move the client to an inactive state and update any root destination node and destination nodes it is maintaining for the signal as necessary.
The operation of the agent driver in Figure 3–16 (above) is demonstrated by the pseudo code in Figure 3–17 (below). A critical region encapsulates the core of the yield method. The agent driver first updates its state indicators in any root destination node or destination nodes it is maintaining and performs a barrier operation to ensure ordering. In Figure 3–17 (below), this is performed in the _set_root_dest_inactive_ function call.

The currently visible signal value is once again tested to ensure that it has not met the condition prior to the update of this destination state. If the signal value does not meet the condition, the destination yields the client process and becomes inactive.

The client process will become active again when the signal send process is performed by an HSA agent sending a value to this signal, and is reactivated during the operation of the component interface _unblock_destination_ call on this destination's agent driver.

### 3.11.2 hsa_signal_group_wait

Signal group wait allows the client to specify a set of signals using a signal group object, and a set of conditions and compare values. The indexing of the compare values and conditions is defined as matching the indexing of the signals listed in the signal group object.

Signal group wait allows significant implementation-dependent functionality.
Requirements
A device that is waiting on a signal group must:

- Detect a signal send if no further send operations for any signals in the group are performed
- If using the state feature:
  - Never miss a doorbell (or signal change) in a race
- If using the doorbell feature:
  - Never miss a doorbell in a race
- A pointer to the signal that has been judged to have met its condition is returned
- The observed signal value that was judged to meet the condition is returned
- Return HSA_STATUS_ERROR_OUT_OF_RESOURCES if the requested wait operation cannot be successfully handled
- Return HSA_STATUS_SUCCESS if the wait operation has successfully run
  - This response code does not indicate that the wait conditions were met

A device waiting on a signal group is not required to see all intermediate values of each signal in the group if multiple send operations are performed on the signal.

The group wait function may return without satisfying the wait condition.

Description
Active wait is an HSA agent waiting on a signal group being inherently reactive to either changes in any signal value or operation of the signal group doorbells (as registered in any node hierarchy the destination agent has constructed in the signal object). An HSA agent may perform its signal group wait functions in this class of wait only.

Figure 3-18 (on the next page) shows pseudo code that describes an active wait implementation. In this figure, the wait_expectancy input field has not been used; however, this parameter may be used to adapt the method employed for detecting or reacting to changes in any signal value.

A vendor's implementation may use more efficient mechanisms to wait on the signal group, such as waiting on the event of the group doorbell value being updated.
Signals

3.11 Signal APIs

Figure 3–18 Pseudo code definition of active hsa_signal_group_wait

```c
hsa_status_t hsa_signal_group_wait(hsa_signal_group_object* signal_group,
                                 hsa_wait_expectancy_t wait_expectancy,
                                 hsa_signal_condition_t *conditions,
                                 hsa_signal_value_t *compare_values,
                                 hsa_signal_object *signal,
                                 hsa_signal_value_t *value)
{
    int result = 0;
    hsa_signal_object local_signal;
    hsa_signal_value_t local_signal_value;
    while (result == 0) {
        clear_group_doorbell(signal_group);
        for (int index = 0; index < signal_group->number_of_signals; index++) {
            local_signal = signal_group->signals[index];
            local_signal_value = *local_signal->signal_value;
            if (test_condition(local_signal_value,
                               *condition[index],
                               *compare_values[index])) {
                *value = local_signal_value;
                signal = local_signal;
                result = 1;
            }
        }
        wait_on_group_doorbell(signal_group);
    }
    return;
}
```

Inactive wait is an HSA agent waiting on a signal group being unresponsive to either a change in any signal value or operation of the group doorbell of the agent.

Inactive wait requires interaction with the agent driver and uses related component interface API calls.

An HSA agent may provide different implementations for wait and select the implementation to be used based on the value of wait_expectancy when a wait is performed.

Figure 3–19 (on the facing page) shows an example of pseudo code for an implementation introducing an inactive signal wait. In this figure, the inactive wait is performed when the wait_expectancy is other than HSA_WAIT_EXPECTANCY_SHORT.

In Figure 3–19 (on the facing page), a critical region encapsulates the core of the inactive method. The agent driver first updates its state indicator for the signal group to display INACTIVE state to any signal being sent.

The currently visible signal values are tested to ensure that none has met their condition prior to the update of this destination state. If no signal value meets the condition, the destination yields the client process and becomes inactive.

The client process shall become active again when any signal in the group is sent a value by an HSA agent, and is reactivated during the operation of the component interface unblock_destination API call on this destination's agent driver.
3.11.3 hsa_signal_send

Multi-vendor components must follow a standardized procedure for sending signal values. This is performed using the information contained within the signal object, the registered root destination nodes, linked destination nodes, and group nodes within.

Requirements

A device that performs a store to a signal must:

- Perform a signal value update first

then:
• Parse the root destination node for each node entry as follows:
  o Perform the barrier defined in the root node
  o Test the destination state
  o Parse the destination node hierarchy and for each node entry as follows:
    o Perform the doorbell write operation
    o Perform the barrier defined in the destination node
    o Test the destination state
  o Parse the group node hierarchy and for each node entry as follows:
    o Perform the doorbell write operation
    o Perform the barrier defined in the group node
    o Test the destination state
  o Unblock the INACTIVE destination if detected in any processed node

The barrier requirements stated here create synchronization guarantees between the signal structures, maintained by the agent drivers to which they may be received, and the HSA agents which are sending signal values.

Additional requirements include:

• The signal value must be visible before any root node state or pointer value is observed
• A node’s doorbell value must be visible before its state or pointer value is observed

Description

Figure 3–20 (on the facing page) shows a pseudo code definition of the process that must be followed, including use of the component interface unblock_destination call.
Figure 3–20 Pseudo code definition of signal send

```c
hsa_signal_send(hsa_signal_object* signal,
                 hsa_signal_value_t value) {
    hsa_root_dest_node* root_dest = null;
    hsa_destination_node* dest_node = null;
    hsa_group_node* group_node = null;
    bool dest_inactive;

    *signal->signal_value_ptr = value;

    for (int idx = 0; idx < *signal->number_of_dests; idx++) {
        dest_inactive = false;
        root_dest = *signal->root_dest_array[idx];
        barrier((unsigned int)root_dest->flags.barrier_type);

        if (root_dest->flags.state == INACTIVE) dest_inactive = true;

        dest_node = root_desc.next_level_hier;
        while (dest_node != null) {
            *dest_node->doorbell_address = root_node->flags.doorbell_value;
            barrier((unsigned int)dest_node->flags.barrier_type);
            if (dest_node->flags.state == INACTIVE) dest_inactive = true;
            dest_node = dest_node->next_level_hier;
        }

        group_node = root_desc.next_group_hier;
        while (group_node != null) {
            *group_node->doorbell_address = root_node->flags.doorbell_value;
            barrier((unsigned int)group_node->flags.barrier_type);
            if ((group_node->state) == INACTIVE) dest_inactive = true;
            group_node = group_node->next_group_hier;
        }

        if (dest_inactive == true) {
            destination_unreachable(signal, root_dest->flags.driver_id);
        }
    }
}
```

3.12 Internal agent driver structures

Internal agent driver structure definitions are listed here as reference to describe required behaviors.

Vendor implementations are not required to reproduce these structs and may choose to create similar objects to support their implementations.

3.12.1 driver_group_internal_data_t

Example definition of agent driver internal object. This is provided to aid description; this structure is not exposed to other agents.

**Signature**

```c
struct driver_group_internal_data_t {
    uint64_t* barrier_type;
    uint64_t* state_address;
    uint64_t* doorbell_address;
}
```
Data fields

-barrier_type
Virtual address of the barrier type used by the agent for a signal group.

-state_address
Virtual address of the state indication used by the agent for a signal group.

doorbell_address
Virtual address of the doorbell used by the agent for a signal group.

3.13 Internal agent driver functions

Internal agent driver functions are listed here as reference to describe required behaviors.
Vendor implementations are not required to reproduce these functions and may choose to create similar functions to support their implementations.

3.13.1 barrier

This function as used in the examples explicitly indicates the critical synchronization points for the correct interaction of the HSA agents using signals.

The barrier function returns when all preceding writes have become visible to all other HSA agents. The barrier does not allow reads or writes later in program order to occur before the barrier function returns.

The synchronization required in the examples may be more relaxed than this barrier function creates.

Signature

void barrier
(  
   unsigned int barrier_type  
)

Parameter

 barrier type
Barrier type encoding, encoded in bits [3:0].

[0] classifies the barrier class:

 0 : MEMORY
 1 : PERIPHERAL

[3-1] are architecture specific.

3.13.2 get_my_dest_node

This function as used in the examples describes the agent driver returning the destination node that is suitable for manipulation of doorbell and state variables related to the signal.

Signature

hsa_root_dest_node* get_my_root_node
(  
   hsa_signal_object* signal  
)
Parameter

`signal`

Pointer to the signal object in the client's virtual space.

Return value

Pointer to the root destination node in the client's virtual space.

3.13.3 `get_my_root_node`

This function allows the agent to look up in its own record the root node reference to which it was assigned in the signal object by the generic HSA driver at signal creation.

Signature

```c
hsa_destination_node* get_my_dest_node
(   hsa_root_dest_node* root_node,
    hsa_signal_object* signal
)
```

Parameter

`signal`

Pointer to the signal object in the client's virtual space.

Return value

Pointer to the root destination node in the client's virtual space.

3.13.4 `get_my_group_destination`

This function allows the agent to look up in its own record the signal group object to which it was assigned in the signal group object by the generic HSA driver at signal group creation.

Signature

```c
hsa_signal_group_destination* get_my_group_destination
(   hsa_signal_group_object* signal_group
)
```

Parameter

`signal group`

Pointer to the signal group object in the client's virtual space.

Return value

Pointer to the signal group destination in the client's virtual space.

3.13.5 `wait_on_condition`

The function in this example returns when the signal value is detected as satisfying the condition to the compare_value.
Chapter 3. Signals  3.13 Internal agent driver functions

Signature
void wait_on_condition(
    hsa_signal_object* signal,
    hsa_signal_condition_t condition,
    hsa_signal_value_t compare_value
)

Alternative examples of pseudo code implementations are shown in the following illustrations:

- Figure 3–14 (on page 62) – Without agent driver support.
- Figure 3–15 (on page 63) – With agent driver support.

Parameters

signal
    Pointer to the signal object in the client's virtual space.

condition
    Condition with which to compare signal value and compare value.

compare_value
    Value that signal shall be compared against.

Return value
    none

3.13.6 clear_signal_doorbell

The example in this implementation uses an additional agent implementation function to obtain the agent's destination node, which contains the location of the doorbell and associated information necessary to perform its implementation of clearing the doorbell.

Signature
void clear_signal_doorbell(hsa_signal_object* signal) {
    hsa_destination_node* dest_node;
    struct hsa_root_dest_node* root_node;

    root_node = get_my_root_node(signal);
    dest_node = get_my_dest_node(root_node, signal);

    *dest_node->doorbell_address = 0;
    barrier(dest_node->flags.barrier_type);
    return;
}

Parameter

signal
    Pointer to the signal object in the client's virtual space.

Return value
    none
3.13.7 wait_on_signal_doorbell

The example in this implementation uses an additional agent implementation function to obtain the agent's destination node, which contains the location of the doorbell and associated information necessary to perform its implementation of waiting on the doorbell.

**Signature**

```c
void wait_on_signal_doorbell(hsa_signal_object* signal) {
    hsa_destination_node* dest_node;
    struct hsa_root_dest_node* root_node;
    int result = 0;

    root_node = get_my_root_node(signal);
    dest_node = get_my_dest_node(root_node, signal);

    while (result == 0) {
        result = READ[dest_node->doorbell_address];
    }
    return;
}
```

**Parameter**

*signal*

Pointer to the signal object in the client's virtual space.

**Return value**

none

3.13.8 clear_group_doorbell

This function clears the agent's signal group doorbell. The example in this implementation uses the agent internal data reference available in the signal group destination object. This agent has created its own struct during signal group creation, which allows it to efficiently discover the locations of the doorbell and associated information necessary to perform its implementation of clearing the doorbell.

**Signature**

```c
void clear_signal_group_doorbell(hsa_signal_group_object* signal_group) {
    hsa_signal_group_destination group_dest;
    driver_group_internal_data_t group_data;

    group_dest = get_my_group_destination(signal_group);
    group_data = *group_dest->internal_data_ptr;

    *group_data->doorbell_address = 0;
    barrier(group_data->barrier_type);
    return;
}
```

**Parameter**

*signal_group*

Pointer to the signal group in the client's virtual space.
### 3.13.9 wait_on_group_doorbell

The example in this implementation uses the agent internal data reference available in the signal group destination object. This agent has created its own struct during signal group creation, which allows it to efficiently discover the locations of the doorbell and associated information necessary to perform its implementation of waiting on the doorbell.

#### Signature

```c
void wait_on_group_doorbell(hsa_signal_group_object* signal_group) {
    hsa_signal_group_destination group_dest;
    driver_group_internal_data_t group_data;

    group_dest = get_my_group_destination(signal_group);
    group_data = *group_dest->internal_data_ptr;

    while (result == 0) {
        result = READ[group_data->doorbell_address];
    }
    return;
}
```

#### Parameter

- `signal_group`
  - Pointer to the signal group in the client's virtual space.

#### Return value

- none

### 3.13.10 test_condition

This function tests the current signal value with the compare value with the condition specified.

#### Signature

```c
bool test_condition(hsa_signal_value_t signal_value,
                    hsa_signal_condition_t condition,
                    hsa_signal_value_t compare_value) {
    case (condition)
        HSA_EQ: if (signal_value == compare_value) return true;
        HSA_NE: if (signal_value != compare_value) return true;
        HSA_LT: if (signal_value < compare_value) return true;
        HSA_GTE: if (signal_value >= compare_value) return true;
    return false;
}
```

#### Parameters

- `signal_value`
  - Current signal value.
condition
Condition with which to compare signal value and compare value.

compare_value
Value to test against signal value.

Return values

True
Condition satisfied.

False
Condition failed.

3.13.11 yield_destination
This function controls the yielding of the waiting thread, coordinating the state indications within the node hierarchy. In this example, the function is responsible for resolving any race conditions during the yield.

Signature

```c
void yield_destination
(   hsa_signal_object* signal,
    hsa_signal_condition_t condition,
    hsa_signal_value_t compare_value
)
```

Figure 3–17 (on page 64) shows an example of a pseudo code implementation.

Parameters

signal
   Pointer to the signal object in the client's virtual space.

c_condition
   Condition with which to compare signal value and compare value.

compare_value
   Compare value to test against signal value.

Return value
   none

3.13.12 set_root_dest_inactive
This function controls the update of the root node state value to INACTIVE.

Signature

```c
void set_root_dest_inactive
(   hsa_signal_object* signal
)
```

Figure 3–7 (on page 55) shows an example of a pseudo code implementation.
3.13 Internal agent driver functions

Parameter

signal
   Pointer to the signal object in the client’s virtual space.

Return value

none

3.13.13 set_root_dest_active

This function controls the update of the root node state value to ACTIVE.

Signature

void set_root_dest_active
(   hsa_signal_object* signal
)

Figure 3–8 (on page 55) shows an example of a pseudo code implementation.

Parameter

signal
   Pointer to the signal object in the client’s virtual space.

Return value

none

3.13.14 set_dest_node_in_root

This function controls the update of the pointer to the destination node hierarchy within the agent’s root node of the signal object.

Figure 3–9 (on page 55) shows an example of a pseudo code implementation.

Signature

void set_dest_node_in_root
(   hsa_signal_object* signal,
   hsa_destination_node* new_dest_node
)

Figure 3–8 (on page 55) shows an example of a pseudo code implementation.

Parameter

signal
   Pointer to the signal object in the client’s virtual space.

new_destination_node
   Pointer to the destination node in the client's virtual space.
3.14 Component interface APIs

Chapter 2 Component Interface (on page 13) lists full details of the component interface APIs.

The following sections describe requirements for the agent driver, which support the signaling mechanisms described in Chapter 3 Signals (on page 41) and apply to each applicable API.

3.14.1 signal_new

The generic HSA driver shall create a hsa_signal_object and initialize its content prior to calling this API (the initialized signal object must be visible).

The generic HSA driver provides the hsa_root_dest_node assigned to the agent driver in the root_dest_array (see section 3.2.3 Array fields (on page 44)) via a pointer to the root node object. The location assigned must be retained by the agent driver to correctly access this structure for any dynamic updates it performs after signal creation.

The agent driver must respond only after it has performed the required operations on signal and destination nodes, and these are visible. This may be an asynchronous response using signal_new_reply.

3.14.2 signal_new_reply

HSA agent may use this asynchronous response to indicate that it has completed and exposed its required operations.

3.14.3 signal_remove

The generic HSA driver shall call this API when it is in the process of removing a signal.

The removal of a signal with the use of this API passes a guarantee from the HSA runtime to the agent driver that the signal shall no longer be used. Any internal data, destination nodes, and group nodes that are not used by this agent for other live signals or signal groups may also be removed.

The agent driver may return a synchronous response or provide an asynchronous response using signal_remove_reply. When the final response (synchronous or asynchronous) is issued, the agent driver may not access the signal object.

3.14.4 signal_remove_reply

An HSA agent may use this asynchronous response to indicate that it has accepted the removal of the signal object and will not access it.

3.14.5 signal_group_new

The generic HSA driver shall create a hsa_signal_group_object and initialize its content prior to calling this API (the initialized signal group object must be visible).

The generic HSA driver provides the hsa_signal_group_destination assigned to the HSA agent within the signal_destinations array via a pointer to the signal group destination object. The location assigned should be retained by the agent driver to correctly access the structure following signal group creation.
The agent driver must respond only after it has performed any required operations it relies upon in its implementation and these are visible. This may be an asynchronous response using \texttt{signal\_group\_new\_reply}.

3.14.6 \texttt{signal\_group\_new\_reply}  
An HSA agent may use this asynchronous response to indicate that it has completed and exposed operations related to the creation of the signal group.

3.14.7 \texttt{signal\_group\_remove}  
The generic HSA driver shall call this API when it is in the process of removing a signal group. The removal of a signal group with the use of this API passes a guarantee from the HSA runtime to the agent driver that the signal group shall no longer be used. Any internal data, signal group destination, and group nodes that are not used by this agent for other live signals or signal groups may also be removed.

The agent driver may return a synchronous response or provide an asynchronous response using \texttt{signal\_group\_remove\_reply}. When the final response (synchronous or asynchronous) is issued, the agent driver may not access the signal group object.

3.14.8 \texttt{signal\_group\_remove\_reply}  
An HSA agent may use this asynchronous response to indicate that it has accepted the removal of the signal group object and will not access it.

3.14.9 \texttt{destination\_unreachable}  
This is an asynchronous API into the generic HSA driver, which passes the signal handle and the \texttt{device\_driver\_id} of the destination that has been detected as unreachable while sending a signal that it is a potential destination.

3.14.10 \texttt{unblock\_destination}  
This is an API into the agent driver and requires the agent driver to move the destination to an active state. The API may be operated by the generic HSA driver while the destination is in an active state.

All execution threads in the destination agent that are blocked on the identified signal will be moved to the active state.

In the examples shown in this specification, the active state of the HSA agent shall allow the \texttt{yield\_destination} process to return without any further interaction from the sending agent.

The implementation of this API is implementation dependent.
CHAPTER 4.
User Mode Queues

4.1 Overview

This chapter extends the description of user mode queues provided in the *HSA Platform System Architecture Specification* and the *HSA Runtime Programmer’s Reference Manual* and it should be read in conjunction with these two documents.

A user mode queue is created by the `hsa_queue_create` or `hsa_soft_queue_create` runtime API (see *HSA Runtime Programmer’s Reference Manual Version 1.1*, section 2.5.4.4 `hsa_queue_create` and section 2.5.4.5 `hsa_soft_queue_create`) and consists of:

- **Header**: The payload's base address and its read and write indexes.
- **Payload**: Contains packets.

The multi-vendor architecture adopts the definition of user mode queue from *HSA Platform System Architecture Specification* and *HSA Runtime Programmer’s Reference Manual*. It also defines the following aspects of user mode queues:

- **Positions of read and write indexes.**
  - The *HSA Runtime Programmer’s Reference Manual* provides the API for manipulating read and write indexes but doesn't specify where they actually reside.

- **Private space for the packet processor.**
  - An implementation of a packet processor may benefit from additional space in the user mode queue.

- **Memory mapping for user queue.**
  - Different agents in an HSA system may have different requirements for memory where user mode queues' headers and payloads reside.

The following terms are used frequently and are explained in Appendix A Glossary (on page 95):

- **Destination agent of a user mode queue**
- **System critical component**
- **Safe-to-access user memory**

4.2 User mode queue header

The header of the user mode queue shall be defined as follows:

```c
struct hsa_user_mode_queue {
    // The following data members are specified in the 'HSA Platform System Architecture Specification'
    // and hsa_queue_t in
    // HSA runtime library.
    int32_t type;
    int32_t features;
    int64_t base_address;
    int64_t doorbell_signal;
}
```
int32_t size;
int32_t reserved;
int64_t id;

// Reserved for future expansion of hsa_queue_t
int64_t reserved_2;

// Pointer to 64-bit write index of the queue.
int64_t write_index_ptr;

// Pointer to 64-bit read index of the queue.
int64_t read_index_ptr;

// Reserved for internal use of the destination agent.
// The actual size of array is defined by the destination
// agent of the user mode queue
int64_t destination_private[0];
};

- **The context of bits 511:0 of the header shall not change during the lifetime of the queue.**
  
  Clients and agents may keep their own copies of the header.

- **The size of the destination_private field (and, consequently, the size of the whole header) is defined by the queue's destination agent.**

  For details, see Private field size in 2.5.2 struct agent_driver_config (on page 17).
  
  - The destination_private field is reserved for the private use of the destination agent.
    
    This field can still be written to by the user and therefore it should not be relied upon by
    system-critical components of HSA.

  - If the user mode queue header is created via the hsa_soft_queue_create runtime API, the
    size of the destination_private field shall be zero, and therefore, the size of the header
    shall be 512 bits.
    
    See HSA Runtime Programmer’s Reference Manual Version 1.1, section 2.5.4.5 hsa_soft_queue_create.

- **The hsa_queue_create and hsa_soft_queue_create runtime APIs shall allocate the header in
  system memory on full-profile HSA systems. On base-profile HSA systems, the header is
  allocated in the fine-grain memory region.**

  Consequently, the content of the header may be overwritten by an application or paged out by the
  CPU OS. System-critical components must keep their own copies of the header.

  See HSA Runtime Programmer’s Reference Manual Version 1.1, section 2.5.4.4 hsa_queue_create.

  - The destination agent of the queue shall be able to create a copy of the header that is
    safe-to-access.
    
    As described in 2.8.1 queue_new (on page 25), all relevant information about the user mode
    queue is passed to the HSA agent when the queue is created.

  - An HSA agent may request that the headers of all user mode queues are placed in safe-
    to-access memory.
Such a request shall ensure that all components are free from memory access faults when they access the header. See Request for safe-to-access header of all user mode queues in 2.5.2 struct agent_driver_config (on page 17).

- **The destination agent of the signal that is created by the hsa_queue_create runtime API (and attached to the newly-created user mode queue) is specified by the “agent” input parameter. The signal has no other destination agents.**
  
The doorbell mechanism of this signal shall be defined by the destination agent (see Chapter 3 Signals (on page 41)).

4.3 Read and write indexes

- **Read and write indexes of the queues shall reside in safe-to-access memory.**
  
  Memory access to the queue’s indexes (pointed to by write_index_ptr and read_index_ptr) shall never cause memory access fault.

- **Read and write indexes shall be accessible from kernel virtual space.**
  
  An HSA agent may implement its packet processor in kernel virtual space.

- **HSA agents shall implement operations on the queue read and write indexes as 64-bit atomic memory transactions.**

4.4 User mode queue payload

- **The user mode queue payload is allocated in user (malloc) memory on full-profile HSA systems. Base-profile HSA systems allocate the payload in a fine-grain memory buffer.**

  A memory access to the payload may therefore generate a memory access fault.

  - **An HSA agent may request that the payloads of all its user mode queues are placed in safe-to-access memory.**

    An agent may make such a request to avoid memory access faults in its packet processor.

    For details, see Request for safe-to-access payload in 2.5.2 struct agent_driver_config (on page 17).

    This is also a debug option.

  - **An HSA agent may request that the payloads of all user mode queues are placed in safe-to-access memory.**

    Such request shall ensure that all components are free from memory access faults when either queuing or processing packets of user mode queues.

    For details, see Request for safe-to-access payload for all user mode queues in 2.5.2 struct agent_driver_config (on page 17).

    This is also a debug option.
CHAPTER 5.
HSA Agent Drivers for Memory Management

5.1 Overview

Shared virtual memory (SVM) enables:

- Different agents access to the same memory location with the same virtual address.
- Software developers to write code with extensive use of pointer-linked data structures such as linked list or trees that are shared between the host and devices.

Memory management plays a significant role in SVM and requires the cooperation of the host OS, generic HSA driver, and agent driver to make SVM work seamlessly under the hood. In certain implementations, the device has its own page tables that are different from the host.

The memory management function ensures that host and agents using the same virtual address see the same data. In the SVM environment:

- The host OS takes control of the system memory management function.
- The generic HSA driver and agent driver work with the HSA runtime and host OS to achieve the SVM with the agents.

This specification supports different types of memory management:

- The simple shared page tables where host and agents share the same page tables (page tables used by the host OS).
- The shadow (device) page tables where an agent has its own page tables that are different from the host page tables.

The agent driver is responsible for maintaining the shadow page tables and keeping them in sync with the host page tables.

For certain use cases, applications may want to lock the region of the buffer to prevent the physical space from swapping out, which is called safe-to-access memory in this specification. Locking a large amount of physical space may affect the performance of other contexts due to decreased physical space available for other contexts. Demand paging can be implemented, where the buffer region might be swapped out at the discretion of the host OS. In the demand-paging-based shadow page implementation, the host OS must notify the agent about the swapped-out memory region so it can update its shadow page tables to keep in sync with the host page tables.

A compliant HSA system will allow HSA agents to access shared system memory through the common HSA-unified virtual address space. For the HSA base profile, system memory allocated through the HSA runtime is shared. For the full profile, all system memory allocated to the HSA application is shared.
5.2 Multi-vendor memory management

Figure 5–1 (below) shows the multi-vendor architecture as it relates to memory management. If an application wants to create a coarse-grained buffer in either profile or a fine-grained buffer in the HSA base profile:

- The HSA runtime issues a system call to the generic HSA driver.
- The generic HSA driver requests a virtual range from the host OS in the shared/private range.
- The generic HSA driver also makes a function call to each agent driver participating in this region.

For a shared page tables implementation, the agent driver can be as simple as creating a context entry for this process and assigning a pointer to the host page tables for this process, since the host OS has taken care of the setting of the host page tables.

For a shadow (device) page tables implementation, each agent has its own shadow page tables (for each context), which is different from the host. This implies that an agent with shadow page tables has to have its own MMU to handle the page walk in case the virtual address does not match any of the agent TLB entries. In addition to what the shared page tables implementation has to perform, the agent driver for the shadow page tables implementation also has to copy the mapping of the host page tables to the shadow page tables.

Additional functions are required to maintain the consistency between the host page tables and the shadow page tables (including agent TLB). Any translation information cached in the local translation cache that is not coherent with the system memory must be maintained by the agent driver.
Not all the agents in the system support page faults. For those agents that do not support page fault, the vendor specific drivers must indicate this at boot time via `struct agent_driver_config`. All memory assigned to these agents must have its memory pinned to prevent taking page faults. Having too much physical memory pinned could lead to an out-of-memory issue. The system designer needs to verify that the agent without the page fault capability accesses only the safe-to-access memory.

For the agent implementing shadow page tables with demand paging, the agent must register the region of interest (shared memory) with the host OS. If the host decides to change the mapping information, the host OS can notify the agent that registered the memory with the host OS. The agent will sync its shadow page tables with the host page tables.

### 5.3 APIs

#### 5.3.1 update_range_on_agent

Update the page tables for the specified address range. The agent is notified when the OS modifies the page tables for the specified address range. The agent must update its page tables if shadow page tables are implemented. The associated TLB entries must be invalidated if no hardware TLB invalidation mechanism is implemented.

**Signature**

```c
int64_t update_range_on_agent
(
    uint64_t start,
    size_t len
)
```

**Direction**

From the host OS to the agent driver

**Parameters**

- `start`
  - Starting address of this virtual range.

- `len`
  - Length of the region to be released.

**Return values**

<table>
<thead>
<tr>
<th>Return code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Virtual range is successfully updated.</td>
</tr>
<tr>
<td>EINVAL</td>
<td>Invalid argument. This is used to indicate various kinds of problems with passing the wrong argument to a library function.</td>
</tr>
</tbody>
</table>
### 5.3.2 report_page_fault

The agent issues this API to request the service due to no page mapping information found or the correct privilege is not granted for the memory access by the agent. If the fault was successfully resolved by the OS (in the case of a shadow page implementation), the agent will copy the correct page information to the shadow page tables before the agent continues to execute its program.

**Signature**

```c
int64_t report_page_fault(
    uint64_t addr,
    vm_area_struct *vma
)
```

**Direction**

From the agent driver to the host OS

**Parameters**

- **Addr**
  Virtual address that caused the page fault.

- ***vma**
  Pointer to the VMA structure.

**Return values**

<table>
<thead>
<tr>
<th>Return code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Physical memory is successfully allocated for the agent.</td>
</tr>
<tr>
<td>EACCESS</td>
<td>Permission denied. File permissions do not allow the attempted operation.</td>
</tr>
<tr>
<td>EINVAL</td>
<td>Invalid argument. This is used to indicate various kinds of problems with passing the wrong argument to a library function.</td>
</tr>
<tr>
<td>ENOMEM</td>
<td>No memory available. The system cannot allocate more virtual memory.</td>
</tr>
<tr>
<td>EOVERFLOW</td>
<td>One or more fields of the data structure used to pass information out of the kernel are too small to hold the value.</td>
</tr>
</tbody>
</table>

### 5.4 Use cases

#### 5.4.1 Starting a client application

- The generic HSA driver should send a **client_new** message to the agent driver when:
  - A client creates a signal and the agent is specified as the destination (**hsa_signal_create**). (See HSA Runtime Programmer’s Reference Manual Version 1.1, section 2.4.1.3 hsa_signal_create.)
  - A client creates a user mode queue and the agent is specified as the destination (**hsa_queue_create**). (See HSA Runtime Programmer’s Reference Manual Version 1.1, section 2.5.4.4 hsa_queue_create.)
5.4.2 Client application executes hsa_shut_down call

- The client issues **hsa_shut_down**. (See *HSA Runtime Programmer’s Reference Manual Version 1.1*, section 2.1.1.2 *hsa_shut_down*.)
- The generic HSA driver issues a **client_remove** message to the vendors’ HSA drivers.
- The agent should remove any reference to the client. If the agent implements shadow page tables, the agent should remove the memory allocated for the shadow page tables.

5.4.3 Allocating fine-grained buffer from HSA allocator

- The client issues **hsa_memory_allocate** (see *HSA Runtime Programmer’s Reference Manual Version 1.1*, section 2.7.4.7 *hsa_memory_allocate*) to allocate memory in a global segment. The value of the attribute **HSA_REGION_INFO_SEGMENT** in this region is **HSA_REGION_SEGMENT_GLOBAL**, and the **HSA_REGION_GLOBAL_FLAG_FINE_GRAINED** flag must be set. **memory_allocation** is issued to all agents.
- The generic HSA driver should allocate virtual address in the host page tables (**agent_memory_reserve**). If any agent requires the memory be pinned, the generic HSA driver will also allocate the physical space for this region and pin the memory (**agent_memory_allocate_safe_to_access**).
- For shadow page tables implementations, the agent should update the mapping information in its table.
- For demand paging implementations, the agent should notify the host OS to keep track of this memory region in case the host decides to change the context of mapping in this region.

5.4.4 Allocating coarse-grained buffer to an agent for the first time

- The client issues **hsa_memory_allocate** to allocate memory in a global segment. (See *HSA Runtime Programmer’s Reference Manual Version 1.1*, section 2.7.4.7 *hsa_memory_allocate*) The value of the attribute **HSA_REGION_INFO_SEGMENT** in this region is **HSA_REGION_SEGMENT_GLOBAL**, and the **HSA_REGION_GLOBAL_FLAG_COARSE_GRAINED** flag must be set.
- **memory_allocation** is issued to the newly-assigned agent.
- **agent_memory_reserve** requests the generic HSA driver to allocate the virtual address in the host page tables. If any agent requires the memory to be pinned, **agent_memory_allocate_safe_to_access** is issued to the generic HSA driver to allocate the physical space for this region and pin the memory.
- **memory_acquire** is issued to the new agent owner.
- For shadow page tables implementations, the agent should update the mapping information in its
For demand paging implementations, the agent should notify the kernel OS to keep track of this memory region in case the host decides to change the context of mapping in this region.

5.4.5 Changing ownership of coarse-grained buffer to a different agent

- The client issues `hsa_memory_assign_agent` to assign a new agent to the coarse-grained buffer. (See HSA Runtime Programmer’s Reference Manual Version 1.1, section 2.7.4.10 hsa_memory_assign_agent.)

- `memory_release` is issued to the existing owner agent. The agent might have to flush its cache if its memory is not coherent with system memory (agent_memory_unmap).

- For shadow page tables implementations, the agent should update the mapping information in its table.

- For demand paging implementations, the agent should notify the host OS to keep track of this memory in case the host decides to change the context of mapping in this memory. If the agent requires safe-to-access memory, the agent should notify the host OS to lock the memory region.

- `memory_acquire` is issued to the new owner agent.

5.4.6 Register system buffer from OS allocator

- This memory must be already assigned by an OS allocator.

- The runtime call issuing `hsa_memory_register` to register a buffer serves as an indication to the runtime that the memory might be accessed from a component other than the host. (See HSA Runtime Programmer’s Reference Manual Version 1.1, section 2.7.4.11 hsa_memory_register.)

- `memory_allocation` is issued to all agents.

- If any agent requires the memory be pinned, the generic HSA driver will also allocate the physical space (if physical space has not been allocated by the OS allocator) for this region and pin the memory.

- For shadow page tables implementations, the agent should update the mapping information in its table.

- For demand paging implementations, the agent should notify the host OS to keep track of this memory in case the host decides to change the context of mapping in this memory.

5.4.7 Memory copying

The HSA runtime issues `hsa_memory_copy` to copy a block of data from source VA (virtual address) to destination VA. (See HSA Runtime Programmer’s Reference Manual Version 1.1, section 2.7.4.9 hsa_memory_copy.)

- If both source and destination buffers are not coarse-grained:
  - The generic HSA runtime or generic HSA driver performs the data copy.

- If either source or destination or both are coarse-grained buffers owned by the same agent:
  - `memory_copy` will be issued to the agent to perform the data copy.

- If both source and destination are coarse-grained buffers owned by different agents:
The generic HSA driver must create a temporary buffer (find-grained or system memory). Memory copy will be issued to the owner of the source buffer to copy the data to the previously created temporary buffer. Memory copy is issued to the owner of the destination buffer to copy the data from the temporary buffer to the destination. The temporary buffer does not have to be the size of the data. If the temporary buffer is smaller than the size of the data, multiple iterations will be required to move the data a block at a time until the specified range is copied.

- If both source and destination are in the fine-grained memory regions, the generic HSA driver will perform the data copying.

5.4.8 Page not allocated for agent implements demand paging

- The agent issues report_page_fault to request the host OS to commit the physical memory.
- If the physical memory is successfully allocated, the callback routine update_range_on_agent is called if the agent implements shadow page tables. The mapping information is copied to the shadow page tables. The instruction that caused the original page fault will be re-executed, and the mapping should be found in the replay path.
- If the physical memory allocation is unsuccessful, the agent sends a client_error message to the generic HSA driver.

5.4.9 Allocation of private and group segment before kernel execution

If shadow page tables with local memory are used for group and private segments:

- An agent issues agent_memory_map to request the allocation of a virtual range for group or private segments. The virtual range should be mapped to the local memory and information updated in the shadow page tables.

If shadow page tables with global memory are used for group and private segments:

- An agent issues agent_memory_allocate to allocate virtual and physical ranges for group and private segments. The agent needs to notify the OS to monitor the virtual range. If change occurs in the registered memory, the OS will notify the agent so that it may update its page tables.

If shared page tables with local memory are used for group and private segments:

- The agent issues agent_memory_map to request a virtual range and map the virtual range to the local memory.

If shared page tables with system memory are used for group and private segments with demand page supported:

- The agent issues agent_memory_reserve to request a virtual range. When the agent accesses this virtual range, a page fault is triggered, report_page_fault is issued, and the physical memory will be allocated by the host OS.

If shared page tables with system memory are used for group and private segments with memory pinned:

- An agent issues agent_memory_allocate_safe_to_access to request virtual range and physical memory with the memory pinned.
5.4.10 Page fault at the host agent (CPU)

- The fault should be handled by demand paging in the kernel OS and the memory should be paged in. If it succeeds, the application should continue.
- If the page-in fails, the kernel OS should terminate the application. The OS should notify the generic HSA driver that the allocation has terminated.
- The generic HSA driver issues a client_remove message to the agent drivers. The agent must remove any reference to the client.
- The client can't be removed until confirmation from all agents.

5.4.11 Kernel OS changes page tables of the client for shadow page tables

- The kernel OS should notify the vendor’s specific agent that the page has been changed only if the agent has registered the region of interest with the kernel OS.
- Registering the region of interest is strictly between the agent driver and kernel OS; the generic HSA driver is not involved.

5.5 Full profile shadow page tables under Linux OS with heterogeneous memory management (HMM)

Heterogeneous memory management (HMM) is software under GPL that handles the synchronization between the host and the shadow page tables. HMM provides a set of APIs to a device driver to mirror a process address space.

This set of APIs sits on top of the Linux memory management (MMU notifier API). Any change in a process address space is mirrored to the device page tables by the HMM code. To achieve this, HMM also requires each device driver to implement a set of callback functions to handle the device-specific functions unique to each device.

HMM allows the migration of a range of memory to the device local memory to take advantage of its low latency and higher bandwidth. This feature is not described in this specification.

The following sections list data structures used by the APIs. The list of HMM APIs and callback routines are identified for the implementation of the proposed APIs.

5.5.1 HMM data structure

This section describes the data structures used by the HMM APIs.

5.5.1.1 hmm_mirror

**Description**

Each device that mirrors a process has a unique hmm_mirror structure that associates the process address space with the device. The same process can be mirrored by several different devices at the same time. The hmm_mirror structure is per device and per process memory mapping HMM structure (mm HMM structure).

Each device that wants to mirror an address space must register one of these structures for each of the address spaces it wants to mirror. The same device can mirror several different address spaces. The same address space can be mirrored by different devices.
5.5 Full profile shadow page tables under Linux OS with heterogeneous memory management (HMM)

Signature

```c
struct hmm_mirror {
    struct hmm_device *device;
    struct hmm *hmm;
    struct kref kref;
    struct list_head dlist;
    struct hlist_node mlist;
    struct hmm_pt pt;
};
```

Members

device
- The `hmm_device` structure this `hmm_mirror` is associated with.

hmm
- The `hmm` structure this `hmm_mirror` is associated with.

kref
- Reference counter (private to HMM; do not use).

dlist
- List of all `hmm_mirror` for the same device.

mlist
- List of all `hmm_mirror` for the same process.

pt
- Mirror page tables.

5.5.1.2 hmm_event

Unique structure associated with an event. Typically, `hmm_event` is used to synchronize memory management events. For example, a process can use `hmm_event` to manage serialized events that affect overlapping address ranges.

Signature

```c
struct hmm_event {
    struct list_head list;
    unsigned long start;
    unsigned long end;
    dma_addr_t pte_mask;
    enum hmm_etype etype;
    bool backoff;
};
```

Each event is described by a type associated with a struct:

```c
enum hmm_etype {
    HMM_NONE = 0,
    HMM_FORK,
    HMM_MIGRATE,
    HMM_MUNMAP,
    HMM_DEVICE_RFAULT,
    HMMDEVICE_WFAULT,
    HMM_WRITE_PROTECT,
    HMM_COPY_FROM_DEVICE,
    HMM_COPY_TO_DEVICE,
};
```
Members

*list*
   Allows HMM to keep track of all active events.

*start*
   First address (inclusive).

*end*
   Last address (exclusive).

*pte_mask*
   HMM PTE update mask (bit[s] that are still valid).

*etype*
   Event type (munmap, migrate, truncate, etc.).

*backoff*
   Meaningful only for device page fault.

5.5.1.3 hmm_device

Each device that wants to mirror an address space must register one of these structures (only one per Linux device).

Signature

```
struct hmm_device {
    struct device *dev;
    const struct hmm_device_ops *ops;
    struct list_head mirrors;
    spinlock_t lock;
};
```

Members

*dev*
   Linux device structure pointer.

*ops*
   The HMM operations callback.

*mirrors*
   List of all active mirrors for the device.

*lock*
   Lock protecting mirrors list.

5.5.2 HMM APIs for agent drivers

5.5.2.1 hmm_device_register

Every device driver must register one and only one `hmm_device`, and the `hmm_device` is the link between HMM and each device driver.
Chapter 5. HSA Agent Drivers for Memory Management

5.5 Full profile shadow page tables under Linux OS with heterogeneous memory management (HMM)

**hmm_device_register()** is called when a device driver wants to register itself with HMM, and an `hmm_device` structure is set up for the device. During this process, the list of callback routines for the device will be stated in this `hmm_device` structure. If this function runs successfully, 0 is returned, otherwise –EINVAL.

**Signature**

```c
int hmm_device_register(struct hmm_device *device)
```

5.5.2.2 hmm_device_unregister

**hmm_device_unregister** is called when a device driver wants to unregister itself with HMM. This will check that there is not an active mirror and returns –EBUSY if so.

**Signature**

```c
int hmm_device_unregister(struct hmm_device *device)
```

5.5.2.3 hmm_mirror_register

**hmm_mirror_register** is called when a device driver wants to start mirroring a process address space. The HMM shim will register mmu_notifier and start monitoring process address space changes; hence callback to the device driver might happen even before this function returns. The task the device driver wants to mirror must be current. Only one mirror per mm (memory mapping) and `hmm_device` can be created. It will return zero if the `hmm_device` already has an `hmm_mirror` for the mm.

**Signature**

```c
int hmm_mirror_register(struct hmm_mirror *mirror)
```

5.5.2.4 hmm_mirror_unregister

Device unregister a process with HMM.

**Signature**

```c
void hmm_mirror_unregister(struct hmm_mirror *mirror)
```

**Description**

**hmm_mirror_unregister** is called when a device wants to stop mirroring a process. This will trigger a call to `release()` callback if it did not already happen. The caller must hold a reference on the mirror.

5.5.2.5 hmm_mirror_fault

Device generate a page fault for memory access in the mirrored region.

**Signature**

```c
int hmm_mirror_fault(struct hmm_mirror *mirror, struct hmm_event *event)
```

**Description**

**hmm_mirror_fault** is called for the following conditions:
- First, HMM_COPY_TO_DEVICE: a device wants to copy a memory region from main memory to device visible memory (not covered in this specification).
- Second, HMM_DEVICE_RFault: device page fault due to read.
- Third, HMMDEVICE_WFAULT: device page fault due to write.

For the page fault due to read and write, the host OS needs to walk the page tables and perform the requested page fault function.

After the page fault request is handled by the host OS, HMM will synchronize the device shadow page tables with the host CPU page tables. There must be a strong ordering between the device driver updating the shadow page tables and any invalidate/update of the same region by the host OS.

Pages that are exposed to the device driver must stay valid while the callback is in progress. To achieve this, HMM relies on a few steps:

- The mmap_sem ensures that any munmap() syscall will serialize with the callback (for update).
- Because the issue is with unmap_mapping_range(), HMM keeps track of the affected range of address and blocks the device page fault that hits the overlapping range.

5.5.3 Device supported callback functions for HMM

HMM requires a total of five callbacks, but the two callbacks relevant to this specification are described below.

5.5.3.1 release

Release all mirrored address spaces

**Signature**

```c
void (*release)(struct hmm_mirror *mirror);
```

**Description**

When release() is called, the device driver must kill all device threads using this mirror, and the mirror must stop using the address space. It is called from:

- process dying (all processes use this to exit)
- `hmm_mirror_unregister()` (if no other thread holds a reference)
- outcome of some device error reported by any of the device callbacks against that mirror

5.5.3.2 update

Update the shadow page table for the specified address range.

**Signature**

```c
int (*update)(struct hmm_mirror *mirror, struct hmm_event *event)
```
Description

When `update()` is called, the device driver updates the shadow page tables for a specified address range. The event type provides the nature of the update:

- Range is no longer valid (munmap)
- Range protection changes (mprotect, COW, ...)
- Range is unmapped (swap, reclaim, page migration, ...)
- Device page fault

The device driver must not update the HMM mirror page tables (except the dirty bit noted below). Core HMM will update the HMM page tables after the update is done. Note that the device must be cache-coherent with system memory (snooping in case of PCIe devices) so there should be no need for the device to flush its cache.

When write protection is turned on, the device driver must make sure the hardware will no longer be able to write to the page; otherwise, file system corruption may occur. The device must properly set the dirty bit using `hmm_pte_set_bit()` on each page entry for memory that was written by the device. If the device cannot properly account for write access, then the dirty bit must be set unconditionally so that proper write back of file-backed page can occur. The device driver must not fail in any way; any failure results in the device process being killed.

`Update()` returns 0 on success, otherwise -ENOMEN (out of memory) or -EIO (io error) in case of error.
APPENDIX A.
Glossary

acquire synchronizing operation
A memory operation that specifies an acquire memory ordering.

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

application global memory
Memory that is to be shared between all agents and the host CPUs for processing using the HSA. This corresponds to the global segment.

Architected Queuing Language (AQL)
An AQL packet is an HSA-standard packet format. AQL kernel dispatch packets are used to dispatch kernels on the kernel agent and specify the launch dimensions, kernel code handle, kernel arguments, completion detection, and more. Other AQL packets control aspects of a kernel agent such as when to execute AQL packets and making the results of memory operations visible. AQL packets are queued on user mode queues.

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

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

destination agent (of a user mode queue)
An HSA agent that executes packets from the queue, defined by the "agent" input parameter in the hsa_queue_create runtime API. It is also said that user mode queues “belong” to their destination agents. Note that a user queue that is created by hsa_soft_queue_create does not have its destination agent.

global segment
A memory segment in which memory is visible to all units of execution in all agents.

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

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

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

HSA application

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

HSA implementation

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

HSA memory management unit (HSA MMU)

A memory management unit used by kernel agents and other agents, designed to support page-granular virtual memory access with the same attributes provided by a host CPU MMU.

HSA Memory Node (HMN)

A node that delineates a set of system components (host CPUs and kernel agents) with “local” access to a set of memory resources attached to the node’s memory controller and appropriate HSA-compliant access attributes.

HSA platform

The framework described in the collected HSA specifications.

HSAIL

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

invalid address

An invalid address is a location in application global memory where an access from a kernel agent or other agent is violating system software policy established by the setup of the system page tables attributes. If a kernel agent accesses an invalid address, system software shall be notified.

kernarg segment

A memory segment used to pass arguments into a kernel.

kernel

A section of code executed in a data-parallel way by a kernel agent. Kernels are written in HSAIL and are translated by a finalizer to machine code.

kernel agent

An agent that supports the HSAIL instruction set and supports execution of AQL kernel dispatch packets. As an agent, a kernel agent can dispatch commands to any kernel agent (including itself) using memory operations to construct and enqueue AQL packets. A kernel agent is composed of one or more compute units.
memory type
A set of attributes defined by the translation tables, covering at least the following properties:

- Cacheability
- Data Coherency requirements
- Requirements for endpoint ordering, being the order of arrival of memory accesses at the endpoint
- Requirements for observation ordering, being restrictions on the observable ordering of memory accesses to different locations
- Requirements for multi-copy atomicity
- Permissions for speculative access

multi-vendor implementation
An implementation of the multi-vendor platform.

multi-vendor platform
The specialization of the HSA platform described in this specification.

natural alignment
Alignment in which a memory operation of size $n$ bytes has an address that is an integer multiple of $n$. For example, naturally aligned 8-byte stores can only be to addresses 0, 8, 16, 24, 32, 40, and so forth.

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

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

primary memory type
The memory type used by default for user processes. Agents shall have a common interpretation of the data coherency (excluding accesses to read-only image data) and cacheability attributes for this type.

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

process address space ID (PASID)
An ID used to identify the application address space within a host CPU or guest virtual machine. It is used on a device to isolate concurrent contexts residing in shared local memory.
queue ID
An identifier for a user mode queue in a process. Each queue ID is unique in the process. The combination of the queue ID and the packet ID is unique for a process.

readonly segment
A memory segment for read-only memory.

release synchronizing operation
A memory operation that specifies a release memory ordering.

safe-to-access memory
Memory that never causes access faults when it is read by or written to a client.
User memory that is allocated on user heap is the subject of the paging mechanism of the CPU OS, and the access to such memory may cause memory access fault. “Safe to access” memory is memory that may never cause memory page faults. Ways to implement such memory include:

- Pinning user heap memory (CPU OS paging mechanism is instructed not to page out certain set of pages)
- Mapping kernel memory into user address space (kernel memory is never paged out).

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

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

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

system critical component
A component that acts on behalf of multiple user clients bypassing process boundaries. If it fails, a system critical component may cause the failure of entire system. For example:

- A packet processor that executes at a privileged level and processes queues from multiple user clients is a system critical component.
- A packet processor that executes at user level and processes only the queues of a single client is not a system critical component.

unit of execution
A unit of execution is a program-ordered sequence of operations through a processing element. A unit of execution can be any thread of execution on an agent, a work-item, or any method of sending operations through a processing element in an HSA-compatible device.
user mode queue

A user mode queue is a memory data structure created by the HSA runtime on which AQL packets can be enqueued. The packets are processed by the packet processor associated with the user mode queue. For example, a user mode queue associated with the packet processor of a kernel agent can be used to execute kernels on that kernel agent.

wavefront

A group of work-items executing on a single program counter.

work-group

A work-group is a partitioning of the grid of work-items formed by a kernel dispatch. It is an instance of execution in a compute unit.

work-item

A single unit of execution of the grid formed by a kernel dispatch.
<table>
<thead>
<tr>
<th>Component</th>
<th>Page Numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td></td>
</tr>
<tr>
<td>acquire memory order</td>
<td>95</td>
</tr>
<tr>
<td>agent</td>
<td>8, 10, 95-96</td>
</tr>
<tr>
<td></td>
<td>registering an</td>
</tr>
<tr>
<td>agent driver</td>
<td>10-11, 13-16, 33, 53, 56, 77-78, 82, 85</td>
</tr>
<tr>
<td>agent driver internal object</td>
<td>69</td>
</tr>
<tr>
<td>agent memory interface</td>
<td>33</td>
</tr>
<tr>
<td>agent_memory_allocate</td>
<td>35</td>
</tr>
<tr>
<td>agent_memory_allocate_physical</td>
<td>38</td>
</tr>
<tr>
<td>agent_memory_allocate_safe_to_access</td>
<td>35</td>
</tr>
<tr>
<td>agent_memory_free</td>
<td>36</td>
</tr>
<tr>
<td>agent_memory_free_physical</td>
<td>39</td>
</tr>
<tr>
<td>agent_memory_map</td>
<td>37</td>
</tr>
<tr>
<td>agent_memory_reserve</td>
<td>33</td>
</tr>
<tr>
<td>agent_memory_unmap</td>
<td>37</td>
</tr>
<tr>
<td>agent_memory_unreserve</td>
<td>34</td>
</tr>
<tr>
<td>agent_reply</td>
<td>39</td>
</tr>
<tr>
<td>application creation</td>
<td>18</td>
</tr>
<tr>
<td>application global memory</td>
<td>95</td>
</tr>
<tr>
<td>application shutdown</td>
<td>18</td>
</tr>
<tr>
<td>Architected Queuing Language (AQL)</td>
<td>8, 12, 95-96</td>
</tr>
<tr>
<td>arg segment</td>
<td>95</td>
</tr>
<tr>
<td>array field</td>
<td>44</td>
</tr>
<tr>
<td>B</td>
<td></td>
</tr>
<tr>
<td>barrier</td>
<td>70</td>
</tr>
<tr>
<td>barrier function</td>
<td>70</td>
</tr>
<tr>
<td>base profile</td>
<td>60, 81-82</td>
</tr>
<tr>
<td>C</td>
<td></td>
</tr>
<tr>
<td>cache</td>
<td>8</td>
</tr>
<tr>
<td>callback</td>
<td>92-93</td>
</tr>
<tr>
<td>callback function</td>
<td></td>
</tr>
<tr>
<td></td>
<td>release</td>
</tr>
<tr>
<td></td>
<td>update</td>
</tr>
<tr>
<td>calling convention</td>
<td>11, 13</td>
</tr>
<tr>
<td>clear_group_doorbell</td>
<td>73</td>
</tr>
<tr>
<td>clear_signal_doorbell</td>
<td>72</td>
</tr>
<tr>
<td>client</td>
<td>9-11, 18-20</td>
</tr>
<tr>
<td>client interface</td>
<td>18</td>
</tr>
<tr>
<td>client_error</td>
<td>20</td>
</tr>
<tr>
<td>client_new</td>
<td>18</td>
</tr>
<tr>
<td>client_remove</td>
<td>19</td>
</tr>
<tr>
<td>coarse-grained buffer</td>
<td>87</td>
</tr>
<tr>
<td></td>
<td>allocating</td>
</tr>
<tr>
<td></td>
<td>changing ownership of</td>
</tr>
<tr>
<td>common destination node</td>
<td>50</td>
</tr>
<tr>
<td>component interface</td>
<td>11, 13-14</td>
</tr>
<tr>
<td></td>
<td>summary of methods in</td>
</tr>
</tbody>
</table>

---

**component interface APIs** | 11, 13, 44, 77 |
| agent_memory_allocate | 35 |
| agent_memoryAllocate_physical | 38 |
| agent_memoryAllocate_safe_to_access | 35 |
| agent_memory_free | 36 |
| agent_memory_free_physical | 39 |
| agent_memory_map | 37 |
| agent_memory_reserve | 33 |
| agent_memory_unmap | 37 |
| agent_memory_unreserve | 34 |
| agent_reply | 39 |
| client_error | 20 |
| client_new | 18 |
| client_remove | 19 |
| destination_unreachable | 22 |
| hsa_register_agent | 16 |
| memory_acquire | 31 |
| memory_allocation | 29 |
| memory_copy | 32 |
| memory_free | 30 |
| memory_release | 32 |
| queue_new | 25 |
| queue_remove | 27 |
| signal_group_new | 23 |
| signal_group_remove | 24 |
| signal_new | 20 |
| signal_remove | 21 |
| struct_agent_driver_config | 17 |
| struct_generic_driver_config | 17 |
| unblock_destination | 23 |

**component supplier** | 9 |

**compute unit** | 95-96 |

**configuration and initialization interface** | 16 |

**configuration parameter** | 16 |

**consumer** | 23 |

**core APIs** |
- signals API See signals API |

**CPU-based packet processor** | 13 |

**D** |
- data structure | 17 |
- destination agent | 45-53, 80, 95 |
- destination node | 41-42, 46-50, 52-53, 56-59, 61, 70 |
- destination node object | 46 |
- destination_unreachable | 22, 78 |
- device driver | 10-11, 91-94 |
- mirroring a process | 92 |
- stop mirroring a process | 92 |
- dispatch | 12, 95-96 |
doorbell 51-52, 59, 72-74
dooring clearance 72-73
waiting on 73-74
doorbell address 48-49, 52, 61
doorbell signal 12
driver_group_internal_data_t 69
dynamic destination node update 56
dynamic group node update 58
dynamic signal object update 54
E
error code 11, 13-14
   HSAM_Async 13-14
   HSAM_Done 13
execution unit 10
F
finalizer 96
fine-grained buffer
   allocating 86
fine-grained memory 88
full profile 60, 81-82, 89
G
generic HSA driver 10-11, 13-16, 44, 77-78, 82, 85
generic HSA runtime 11
get_my_dest_node 70
get_my_group_destination 71
global memory 88
global segment 44, 95
grid 95
group node 41-42, 47-49, 51-53, 58-59, 61
group node hierarchy 51
group segment 88, 95
group wait 52
get my group node 47
H
handshake mechanism 16
header 79-80
heterogeneous memory management (HMM) 89-94
   device supported callback functions 93
   registering a device driver with 91
   unregistering a device driver with 92
hmm_device 91
hmm_device_register 91
hmm_device_unregister 92
hmm_event 90
hmm_mirror 89
hmm_mirror_fault 92
hmm_mirror_register 92
hmm_mirror_unregister 92
HMN See HSA Memory Node (HMN)
host CPU 13, 89, 95-96
HSA application 96
HSA client 10
HSA compatibility 8
HSA driver 10
   generic 10, 13-15
HSA implementation 8-9, 96
HSA kernel driver 11
HSA memory allocation 28
HSA memory management unit (HSA MMU) 96
HSA Memory Node (HMN) 96
HSA platform 96
HSA runtime 10-11
hsa_destination_node 46
hsa_group_node 47
hsa_register_agent 16
hsa_root_dest_node 45
hsa_signal_group_destination 48
hsa_signal_group_object 44
hsa_signal_group_wait 64
hsa_signal_object 43
hsa_signal_send 67
hsa_signal_wait 61
HSA_Multi 96
I
implementation 61
initialization 16
internal agent driver
   functions 70
   structures 69
interoperability 8, 43
invalid address 96
K
kernarg segment 96
kern_sys 12, 96
kernel agent 95-96
kernel dispatch 8, 95, 97
kernel driver 11
kernel virtual space 81
L
live node 56, 59
M
malloc 81
memory
   copying 28, 32, 87
   memory allocation 28
      coarse-grained 31-32
      creating a 29
      freeing a 30
   memory assignment 28
   memory attribute 59
memory instructions 96
signal 98
memory interface 28
memory interface API 28
memory management 11, 82-83
memory model 95-96
memory segment 95-98
memory type 60, 97
memory_acquire 31
memory_allocation 29
memory_copy 32
memory_release 32
miscellaneous interface 39
MMU See HSA memory management unit (HSA MMU)
multi-vendor architecture 9-11, 79, 83
multi-vendor compatibility 11
multi-vendor component interface 10-11
multi-vendor implementation 10, 97
multi-vendor memory management 83
multi-vendor platform 8, 10, 97
multi-vendor signal 41-42
multi-vendor signal requirements 41

N
natural alignment 97
node hierarchy 42, 48-53
node object lifetime 59
notification 41

O
object location 59

P
packet 10, 12
packet ID 97
packet processor 11-13, 79, 97
CPU-based 13
unit-based 13
page fault 84-85, 88-89, 92-93
page table 12, 33, 82-84, 89
shadow 82-86, 88-89, 93-94
shared 82-83, 88
PASID See process address space ID (PASID)
payload 79, 81
physical memory 33, 35, 38, 88
allocating 38
freeing 39
mapping 37
platform 9
platform integrator 8-9
primary memory type 97
private segment 88, 97
process address space ID (PASID) 97
profile
base profile 60, 81-82
full profile 60, 81-82, 89

Q
queue 12
queue ID 98
queue_new 25
queue_remove 27

R
read index 12, 79, 81
readonly segment 98
release 93
release synchronizing operation 98
report_page_fault 85
root destination node 41-42, 45, 48-49, 51, 53-55, 59, 61
updating state value to ACTIVE 76
updating state value to INACTIVE 75
root destination node object 45
runtime 96
runtime API 19, 79

S
safe-to-access memory 79-82, 87, 98
segment 97-98
set_dest_node_in_root 76
set_root_dest_active 76
set_root_dest_inactive 75
shadow page table 82-86, 88-89, 93-94
shared page table 82-83, 88
shared virtual memory 12, 60, 82, 96
signal 8-9, 11-12, 18, 41
creating a 20, 53, 85
destroying a 20-21, 53, 77
doorbell 12
removing a 77
signal destination 20, 22-23
signal end 41
signal group 23
creating a 23, 53, 77-78
destroying a 24, 53
removing a 78
signal group destination 41, 48
signal group destination object 48, 61, 73-74
signal group doorbell
clearing 73
signal group object 41, 44, 61, 64
signal group wait 64-65
active 65
inactive 66
signal handle 78
signal interface 20
signal method 42
signal object 41-43, 50, 53, 61
signal semantics 8
signal send 45-50, 64
signal value 44, 48-49, 61-68
  sending a 67
  testing current value 74
signal wait 41-42, 61
  active 61-62
  inactive 61, 63
signal_group_new 23, 77
signal_group_new_reply 78
signal_group_remove 24
signal_group_remove_reply 78
signal_new 20, 77
signal_new_reply 77
signal_remove 21, 77
signal_remove_reply 77
signals API 61
spill segment 98
state flag 49-50
state location 59, 61
struct agent_driver_config 17
struct generic_driver_config 17
structure 28
system buffer
  registering 87
system critical component 79-80, 98
system memory
  deregistering 30
  registering 28-29

T
test_condition 74

U
unblock_destination 23, 78
unit-based packet processor 13
unit of execution 98
update 93
update_range_on_agent 84
user library 10
user mode queue 8-9, 11-12, 18, 79, 99
  creating a 20, 25, 53, 85
  destroying a 25, 27, 53
user mode queue header 79-80
user mode queue interface 25
user mode queue payload 81

V
vendor 10
virtual address space 11
virtual machine 96
virtual memory 8
  allocating 35, 37
  cancelling reservation of 34
  freeing 36
  mapping 37
  reserving 33

unmapping 37

W
wait condition 62, 65
wait operation 52
wait state 42
wait_on_condition 71
wait_on_group_doorbell 74
wait_on_signal_doorbell 73
wavefront 99
work-group 95, 99
work-item 97-99
write index 12, 79, 81

Y
yield_destination 75